Generalized Universal Circuits for Secure Evaluation of Private Functions with Application to Data Classification

Ahmad-Reza Sadeghi[*] and Thomas Schneider[**]

Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
{ahmad.sadeghi,thomas.schneider}@trust.rub.de

Abstract. Secure Evaluation of Private Functions (PF-SFE) allows two parties to compute a private function which is known by one party only on private data of both. It is known that PF-SFE can be reduced to Secure Function Evaluation (SFE) of a Universal Circuit (UC). Previous UC constructions only simulated circuits with gates of $d = 2$ inputs while gates with $d > 2$ inputs were decomposed into many gates with 2 inputs which is inefficient for large d as the size of UC heavily depends on the number of gates.

We present generalized UC constructions to efficiently simulate any circuit with gates of $d \geq 2$ inputs having efficient circuit representation. Our constructions are non-trivial generalizations of previously known UC constructions.

As application we show how to securely evaluate private functions such as neural networks (NN) which are increasingly used in commercial applications. Our provably secure PF-SFE protocol needs only one round in the semi-honest model (or even no online communication at all using non-interactive oblivious transfer) and evaluates a generalized UC that entirely hides the structure of the private NN. This enables applications like privacy-preserving data classification based on private NNs without trusted third party while simultaneously protecting user's data and NN owner's intellectual property.

Keywords: universal circuits, secure evaluation of private functions, neural networks, private data classification, privacy.

1 Introduction

Today, a variety of new business models can be provided as electronic services where customers post their requests to a remote provider who performs specific knowledge based operations on their data and provides customers with the results. Examples are expert systems for heath diagnostics, remote data bases, multimedia data processing, or data classification tools (e.g., for spam). From

[*] The first author was supported by the European Union under FP6 project SPEED.

[**] The second author was supported by the European Union under FP7 project CACE.

P.J. Lee and J.H. Cheon (Eds.): ICISC 2008, LNCS 5461, pp. 336–353, 2009.

security targets point of view customers may send sensitive and security critical data and hence require the protection (confidentiality and integrity) of their data, while the service providers may require the protection of their Intellectual Property (IP), i.e., their expertise embedded in their system.

Hence, the problem can be formulated as follows: two parties, a service requester R (client) and a service provider P (server), are involved in the computation of a function f (belonging to P) on data x (input by R) where P should not obtain any information about x and R should not get any useful information about f besides the result $f(x)$[1].

For arbitrary functions f, this is tackled by *Secure Function Evaluation* (SFE) of *Private Functions* (PF-SFE). SFE protocols [27,12,14,13,9,5] allow two parties to securely evaluate a common function on private data. Based on this, PF-SFE [20,16,10] evaluates a *Universal Circuit* (UC) [25,10] as common function which is programmed with the private function f. As UC can be programmed to simulate any function, it entirely hides f while SFE ensures privacy of data x.

Previous UC constructions can simulate circuits with gates of $d = 2$ inputs only. For example, circuits performing arithmetic operations like addition, number comparison, or multiplication are most efficiently implemented from chains of $d = 3$ input gates (full adders, and full comparers). When these types of circuits need to be simulated with known UC constructions (to hide which arithmetic operations are performed), the large gates must be decomposed into many $d = 2$ input gates, e.g., five gates per 3-input gate using Shannon's expansion theorem: $f(a, b, c) = (c \wedge f(a, b)|_{c=1}) \vee (\bar{c} \wedge f(a, b)|_{c=0}))$.

To overcome this overhead, we generalize previous UC constructions to $d \geq 2$ as non-trivial extensions of previous work together with new constructions that are especially suited to simulate small circuits. Our constructions are much more efficient than using the straight-forward solution of evaluating $\lceil d/2 \rceil$ known UCs (where $d = 2$) in parallel. The overhead of our best Generalized UC (GUC) construction is by a factor of two smaller than the best known UC construction for the practical example given in §5.4.

As application of our GUC constructions we show how to securely evaluate private Neural Networks (NN) where each neuron has d inputs. Amongst others, NNs are very useful tools for data classification including pattern/sequence recognition, and sequential decision making. NNs are increasingly becoming important for deployment in commercial applications like spam filtering [3], speech recognition [24], and many more [23] where the "expertise" of the provider is embedded in NN. Neural networks allow to model any function [7], and are robust against noise. Previous work [15,17] is based on straight forward use of homomorphic encryption in multiple rounds that cannot hide NN's structure completely and is not provably secure. In contrast we show that our solution (i) is efficient w.r.t. the size of the circuit needed for a reasonable NN, (ii) requires no or only one round to evaluate a GUC that (iii) hides the underlying NN and its topology entirely and (iv) is provably secure in the semi-honest model.

[1] Clearly, evaluating f on different inputs x_i allows R to learn some information on f. Thus, P should restrict the maximum number of evaluations of f by other means.

1.1 Related Work

Two-Party Secure Function Evaluation (SFE) protocols [27,12,9] allow two parties to evaluate any function represented as boolean circuit which is known to both of them on their respective private inputs. All of these protocols are provably secure against semi-honest adversaries and can be extended to be secure against malicious adversaries via cut-and-choose [14,13,5]. SFE in semi-honest model is based on Oblivious Transfer (OT) requiring one round of communication [19,1,8] or a non-interactive implementation based on an extension of trusted computing modules [6].

NNs can be represented as boolean circuits - threshold NNs with back-propagation [21] or evolutionary learning algorithms [18] were implemented in hardware, but were not considered to be evaluated within SFE protocols yet.

SFE can be extended to private functions (PF-SFE) where the private function is known to one party only and hidden entirely from the other party. PF-SFE is reduced to SFE of a Universal Circuit (UC) that is programmed with the private function and entirely hides its structure [20,16,10]. The SFE protocol of [9] which allows very efficient evaluation of UCs can also be used to improve evaluation of GUCs of this paper (by a factor between two- and fourfold).

Currently known UC constructions [25,10] can simulate gates with $d = 2$ inputs only. Gates with more than two inputs can be simulated by decomposing them into multiple gates with two inputs.

Oblivious training and evaluation of NNs was studied in the context of Oblivious Polynomial Evaluation (OPE) [2]. The OPE protocol reduces oblivious polynomial evaluation to OT. Based on this protocol, they show how to train and evaluate NNs in multiple rounds without hiding the structure of the NN. Non-linear activation functions are either evaluated using a circuit based SFE protocol or piece-wise approximated as polynomials evaluated via OPE protocol.

Oblivious NN evaluation using homomorphic encryption [15,17] requires multiple rounds as well - one per layer of the NN. The protocol allows evaluation of NNs with several activation functions like threshold or sigmoid function. To hide the structure of the NN, dummy neurons are introduced but this does not entirely hide the structure (e.g., maximal number of neurons per layer and maximal number of layers are revealed as outlined in their paper). Also, these protocols are not provably secure as blinding an additively homomorphic encrypted value with a randomly chosen factor reveals information on the magnitude of the value.

1.2 Our Contributions

In §4 we present practical *Generalized Universal Circuit (GUC) constructions* to efficiently simulate circuits of gates with $d \geq 2$ inputs having efficient circuit representations. Former UCs are restricted to $d = 2$ and decompose larger gates into multiple gates with two inputs resulting in much more overhead than our GUC constructions (cf. Table 1 in §5.4). Our constructions are non-trivial generalizations of known UC constructions that are special cases of ours for $d = 2$.

Based on these GUC constructions we present *protocols to securely evaluate private Neural Networks* (NN) with activation functions implemented as small circuits (such as threshold function) in §5. Unlike previous work, our protocols are provably secure, need a constant number of rounds (one round in the semi-honest model or even no online communication at all using non-interactive OT), hide NN's structure entirely (besides number of in- and outputs, maximum degree d and number of neurons k) and are still practical.

1.3 Basic Idea and Outline

A *Generalized Universal Circuit* (GUC) is a circuit which can be programmed to simulate an arbitrary circuit that consists of gates with d inputs each. These gates are required to have an efficient circuit representation which is the case in our example for neurons in §5 or if the size of their function table 2^d is small.

A GUC can be thought of as a kind of processor (here: programmable circuit) that takes as input some data x and a program p_f corresponding to a function (here: input circuit) and evaluates the program on the data (here: input circuit on data): $UC(p_f, x) = f(x)$. As GUC can be programmed with *any* function it does not reveal anything about the function. This allows to evaluate arbitrary functions privately. In contrast to previous UC constructions we relax the restriction that input circuits need to consist of gates of $d = 2$ inputs only.

In §4 we give different methods to construct GUCs in an iterative (§4.1), modular (§4.2), or graph based (§4.3) way and compare them in §4.4. Definitions are given in §2, new building blocks in §3.

As practical application where simulation of d input gates is advantageous, we show how neural networks (NN), described in §5.1, can be expressed as circuits consisting of d input gates in §5.2. Their structure can be entirely hidden inside a GUC which allows secure evaluation of private NNs as shown in §5.3. Finally, we compare our GUC construction and its application to securely evaluate private NNs to previously known constructions and protocols in §5.4.

2 Definitions and Preliminaries

The following definitions generalize those of [10] to gates with multiple inputs.

A *gate* G is the implementation of a boolean function $\{0,1\}^d \to \{0,1\}$ with d *inputs* and one *output*. The *size* of a gate G, denoted by $|G|$, is the multiple of function table entries needed to implement the gate w.r.t. a 2 input gate, namely $|G| = 2^{d-2}$ (e.g., $|B|_{d=1} = 0.5$, $|B|_{d=2} = 1$, etc.).

We consider acyclic *circuits* consisting of connected gates with arbitrary fan-out, i.e., the output of each gate can be used as input to arbitrary many gates. Further, each output of circuit C is the output of a gate and not a redirected input of C. The *size* of a circuit is the sum of the sizes of its gates. Communication and computation complexity of SFE protocols is linear in the size of the circuit.

A *programmable gate* is a gate with an unspecified function table. To *program* it, a specific function table with 2^d entries for each input combination is given.

To simplify presentation we group gates into functional *blocks* as follows:

A *block* B_v^u is a sub-circuit with u inputs $in_1, .., in_u$ and v outputs $out_1, .., out_v$. B_v^u computes a function $f_B : \{0,1\}^u \to \{0,1\}^v$ that maps the input values to the output values. For simplicity, we identify B_v^u with f_B and write: $B(in_1,..,in_u) = (out_1,..,out_v)$. Blocks consist of connected gates and other sub-blocks. The size of block B, denoted by $|B|$, is the sum of the sizes of its sub-elements.

A *programmable block* is a block consisting of programmable gates or programmable blocks. It is programmed by programming each of its sub-elements.

A *generalized Universal Circuit* (GUC) $\mathrm{UC}_{k,u,v\times d}$ is a programmable block with u inputs and v outputs that can be programmed (denoted by $\mathrm{UC}^C_{k,u,v\times d}$) to simulate any circuit C with up to u inputs, v outputs and k gates with d inputs, i.e., $\forall (in_1,..,in_u) \in \{0,1\}^u : \mathrm{UC}^C_{k,u,v\times d}(in_1,..,in_u) = C(in_1,..,in_u)$.

We use programmable block constructions from [10] with the given number of in- and outputs and the following informal functionalities (see [10] for exact definitions and constructions): Y switching block (Y_1^2 programmable as left: $out_1 = in_1$ or right: $out_1 = in_2$), X switching block (X_2^2 programmable as pass: $(out_1, out_2) = (in_1, in_2)$ or cross: $(out_1, out_2) = (in_2, in_1)$), S_v^u selection block (programmable to select for each of the v outputs any of the u inputs including duplicates). Their sizes and properties are summarized in Table 3 in §B.

Our constructions use different compositions of wires:
A (single) *wire* has value either 0 or 1 and is drawn as thin arrow (⟶).

A *multi wire* W consist of ω wires with fixed ordering. The single wires can be indexed by $W[1],..,W[\omega]$. The value of W is the unsigned integer value $w = \sum_{i=1}^{\omega} 2^{i-1} W[i]$. Multi wires are drawn as filled thick arrows (➡).

A *bundle* consists of wires with irrelevant ordering and no duplicates (no two wires are the output of the same gate). $u \times d$ denotes u bundles of d wires each.

Exact calculations of the sizes of our constructions and building blocks with all intermediate steps are given in the full version of this paper.[2]

3 Bundle Blocks for GUC Constructions

The main difference of our efficient GUC constructions compared to previous UC constructions is to switch bundles of d wires instead of single wires only. To do this, we construct efficient bundle blocks that are used as the fundamental building blocks of the GUC constructions described in §4.

C_d^u Choice Block. is a programmable block that can be programmed to choose from the u inputs a bundle of d distinct values as outputs (without recurrence) where the order of the outputs does not matter. More formally, given a subset $\Gamma \subseteq \{1,..,u\}, |\Gamma| = d$, the choice block computes $C(in_1,..,in_u) = (in_{\gamma_1},..,in_{\gamma_d})$ where $\Gamma = \{\gamma_1,..,\gamma_d\}$ (note, the set equality implies irrelevance of ordering and no duplicate inputs). Of course the definition of a choice block makes only sense for $u \geq d$ as Γ is undefined for $u < d$.

A simple implementation of a C_d^u choice block, $C_d^{u,\mathbf{simple}}$, is to use d selection blocks S_1^{u-d+1} in parallel: $out_i = S_1^{u-d+1}(in_i,..,in_{u-d+i})$; $i = 1,..,d$, i.e., the first selection block can be programmed to select any of the inputs

[2] The full version of this paper is available at `http://eprint.iacr.org/2008/453`.

$in_1, .., in_{u-d+1}$, the second any of $in_2, .., in_{u-d+2}$ and so on. This results in $|C_d^{u,\mathbf{simple}}| = d \cdot |S_1^{u-d+1}| = du - d^2$. The equation also holds true for $u = d$ where the choice block consists of wires only and its size is 0. The straight-forward programming algorithm works as follows: sort Γ ascendingly; for $i = 1, .., d$ do: let k be the smallest element in Γ; program the i-th selection block to select in_k; remove k from Γ; next i. Correctness and efficiency are easy to verify.

Alternatively, a choice block, $C_d^{u,\mathbf{sublin}}$, which is much more efficient for larger d can be derived as a special case of bundle permutation block described next.

$\boldsymbol{BP_{v\times d}^{u\geq vd}}$ Bundle Permutation Block. is a programmable block that can be programmed to permute the u inputs to v bundled outputs of d wires each (without duplicates). We define bundle permutation blocks to have at least as many inputs as outputs: $u \geq vd$. More formally, let $S \subseteq 1, .., u; |S| = vd$ be the subset of inputs that are chosen as outputs and $\Phi = (\Phi_1, .., \Phi_v)$ be an ordered partition of S with $\bigcup\limits_{i=1}^{v} \Phi_i = S; |\Phi_i| = d$, the block computes $BP(in_1, .., in_u) = (in_{\varphi_{1,1}}, .., in_{\varphi_{1,d}}, .., in_{\varphi_{v,1}}, .., in_{\varphi_{v,d}})$, with $\Phi_i = \{\varphi_{i,1}, .., \varphi_{i,d}\}; i = 1, .., v$.

Our efficient implementation of $BP_{v\times d}^{u\geq vd}$ bundle permutation block is based on the *truncated permutation block* construction of [10]. A $TP_v^{u\geq v}$ truncated permutation block is a programmable block that can be programmed to permute its $u \geq v$ inputs to its v outputs (u and v need not be equal or powers of two as in [26]). Remaining $u - v$ inputs are discarded (truncated permutation).

TP_v^u block is constructed recursively from two $TP_{v/2}^{u/2}$ sub-blocks and $u/2 - 1$ X switching blocks on top that distribute the inputs to the sub-blocks and $v/2$ X switching blocks on bottom to distribute their outputs to the outputs of TP_v^u block as shown in Fig. 1(a). 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). Note, that our construction is upside-down compared to the original construction of [10] to have the saved X blocks on top instead (the programming algorithm remains the same using the inverse permutation instead). This modification implies that our GUC construction $M3$ in special case $d = 2$ is more efficient than the original UC construction of [10].

To obtain an efficient $BP_{v\times d}^{u\geq vd}$ bundle permutation block, a $TP_{vd}^{u\geq vd}$ truncated permutation block is constructed without the lowest $\log d$ layers of X switching blocks which can be replaced with wires as the order within each bundle of d wires is irrelevant. Hence, $|BP_{v\times d}^u| = |TP_{vd}^{u\geq vd}| - |X| \cdot \log d \cdot dv/2 = (u + dv)\log v + (\log d + 1)u - 3dv + 2$. An efficient programming algorithm for bundle permutation blocks can easily be derived from the one given in [10].

Our efficient construction of the bundle permutation block is indeed a generalization of the truncated permutation block of [10] which is a special case for $d = 1$ of our construction ($BP_{v\times 1}^{u\geq v} \equiv TP_v^{u\geq v}$) with exactly the same size $|BP_{v\times 1}^{u\geq v}| = |TP_v^{u\geq v}| = (u + v)\log v + u - 3v + 2$.

By fixing the other parameter $v = 1$, we obtain a more efficient (sub-linear in the number of outputs d) construction for choice blocks, $C_d^{u,\mathbf{sublin}} := BP_{1\times d}^{u\geq d} \equiv C_d^u$, with size $|C_d^{u,\mathbf{sublin}}| = |BP_{1\times d}^{u\geq d}| = (\log d + 1)u - 3d + 2$.

4 Generalized Universal Circuits

A *generalized universal circuit* (GUC) $UC_{k,u,v\times d}$ is a boolean circuit that can be programmed to simulate any circuit with u inputs, v outputs and k gates with $d \geq 2$ inputs each. Existing UC constructions [25,10] can simulate gates with two inputs only and can be seen as a special case of our corresponding generalized constructions for $d = 2$. $UC_{k,u,v\times d}$ has exactly u inputs and v outputs.

Each d input gate of the simulated circuit is simulated within a *gate simulation block*, i.e., a programmable block which can be programmed to simulate the functionality of the gate and has d inputs and 1 output. Gates are simulated in topologic order which can be computed efficiently by topologic sorting in $O(k)$.

In the following, we assume that the order of the inputs of a gate simulation block is irrelevant and no inputs are duplicated. This is the case for gate simulation blocks implemented as a d input programmable gate that is programmed with the same function table as the simulated d input gate of exponential size $|G| = 2^{d-2}$. The entries of the function table can be swapped according to an arbitrary input ordering and duplicate inputs can easily be eliminated. Also for gate simulation blocks that implement neurons as a circuit the order of inputs is irrelevant and duplicate inputs can be eliminated as we will explain in §5. The irrelevance of the input ordering without duplicates is reflected by bundles of d wires as input into a gate simulation block.

If the order of inputs of the simulated gates is relevant or duplicate inputs are needed, the following GUC constructions can be extended by replacing the bundle blocks from §3 with their corresponding non-bundled counterparts where the order of outputs does matter at the cost of a small overhead: $C_d^{u,\mathbf{simple}} \mapsto S_d^u, C_d^{u,\mathbf{sublin}} \mapsto S_d^{u\geq d}, BP_{v\times d}^{u\geq vd} \mapsto TP_{vd}^{u\geq vd}$.

We stress that the sizes of all building blocks and the GUCs presented in the remainder of this section only depend on the parameters u, v, k, d but neither on the input data nor the simulated circuit. Hence, dynamic choice of the smallest implementation for each building block in so called *combined constructions* respectively choosing smallest GUC construction reveals nothing about the input data nor the simulated circuit.

4.1 Iterative GUC Constructions

A simple GUC is constructed iteratively by choosing for the i-th gate simulation block G_i any of the u inputs of the circuit or the output of a previous gate simulation block $G_1, .., G_{i-1}$ with C_d^{u+i-1} choice block. The v outputs of the GUC can be selected to be any of the outputs of the k gate simulation blocks using $S_v^{k\geq v}$ selection block.

Using simple $C_d^{u,\mathbf{simple}}$ choice blocks results in a total size of

$$|UC_{k,u,v\times d}^{\mathbf{I1}}| = 0.5dk^2 + (du - d^2 - 0.5d + |G| + 1)k + (k + 3v)\log v - 4v + 3$$
$$\sim 0.5d \cdot k^2 + d \cdot uk.$$

With sub-linear $C_d^{u,\textbf{sublin}}$ choice blocks instead the construction has size

$$|UC^{\textbf{I2}}_{k,u,v\times d}| = (0.5\log d + 0.5)k^2 + (u\log d + u - 0.5\log d - 3d + |G| + 2.5)k \\ + (k+3v)\log v - 4v + 3 \sim 0.5\log d \cdot k^2 + \log d \cdot uk.$$

A combination of both approaches chooses the smallest implementation for each choice block (simple or sub-linear) dynamically.

$$|UC^{\textbf{I}}_{k,u,v\times d}| \leq min(|UC^{\textbf{I1}}_{k,u,v\times d}|, |UC^{\textbf{I2}}_{k,u,v\times d}|).$$

All iterative GUC constructions are only practical for a small number of simulated gates k and few inputs u.

4.2 Modular GUC Constructions

Another approach to construct a GUC is to separate the inputs and the outputs from the simulation of the gates. This results in modular GUC constructions which are a generalization of the modular UC construction of [10]. The *modular GUC* is composed out of three programmable blocks as shown in Fig. 1(b). The *generalized Universal Block (UB)*, $U_{k\times d}$, simulates the k gates, the *input selection block* chooses the corresponding inputs and the *output selection block* chooses the outputs of the simulated gates as outputs of the modular GUC. The construction has size

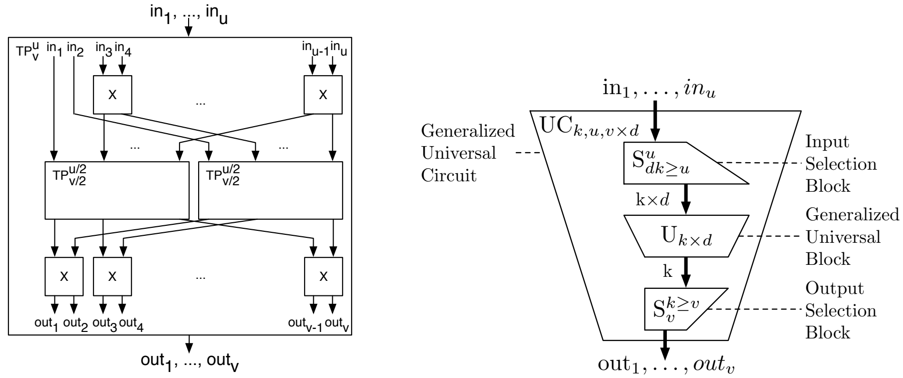
$$|UC^{\textbf{Mi}}_{k,u,v\times d}| = |S^u_{dk\geq u}| + |U^{\textbf{Mi}}_{k\times d}| + |S^{k\geq v}_v| \sim 2dk\log k + dk\log u + |U^{\textbf{Mi}}_{k\times d}|.$$

The overall complexity is determined by the complexity of the generalized UB $U_{k\times d}$ that only depends on the number of simulated gates k and no longer on the number of inputs u or outputs v. The generalized UB $U_{k\times d}$ has k bundles of d inputs where the i-th bundle, $in_{di}, .., in_{di+d-1}$, can be switched to the i-th gate simulation block G_i; the output of G_i is connected to out_i of $U_{k\times d}$.

Next, we give different constructions for generalized UBs that can be plugged into the modular GUC construction. The iterative constructions (generalized from [22, Section 5.3.1]) grow like $d{\cdot}k^2$ and $\log d{\cdot}k^2$ and the recursive construction (generalized from [10]) grows like $dk\log^2 k$.

Iterative Generalized Universal Block Construction. An iterative construction for a generalized UB is similar to the iterative GUC construction described in §4.1 but without the dependancy on the inputs that are handled efficiently by the input selection block of the modular GUC construction. For each gate simulation block G_i, a C_d^{d+i-1} choice block can be programmed to choose any of the d wires of the i-th input bundle of the generalized UB and the $i-1$ outputs of the previous gate simulation blocks $G_1, .., G_{i-1}$ as input to G_i. Using simple $C_d^{u,\textbf{simple}}$ choice blocks this results in a total size of

$$|U^{\textbf{M1}}_{k\times d}| = 0.5dk^2 - 0.5dk + k\cdot|G| \sim 0.5d\cdot k^2.$$



(a) Truncated Permutation Block

(b) Modular GUC Construction

Fig. 1. Building blocks for GUCs

With sub-linear $C_d^{u,\mathbf{sublin}}$ choice blocks instead the size is

$$|U_{k\times d}^{\mathbf{M2}}| = (0.5\log d + 0.5)k^2 + (d\log d - 0.5\log d - 2d + |G| + 1.5)k$$
$$\sim 0.5\log d \cdot k^2.$$

Both modular iterative constructions still grow like k^2 but are more efficient than the iterative GUC constructions from §4.1 for circuits with many inputs due to the efficient handling of inputs with the input selection block.

Recursive Generalized Universal Block Construction. A generalization of the recursive UB construction of [10] yields a generalized UB of size

$$|U_{k\times d}^{\mathbf{M3}}| = (0.625d + 0.25)k\log^2 k + (0.5d\log d - 0.625d - 1.25)k\log k$$
$$+ (|G| + 3)k - 3 \sim 0.625dk\log^2 k.$$

The detailed description of the construction is in the full version of this paper. Compared to the constructions presented before, the recursive construction grows like $k\log^2 k$ instead of k^2 which is clearly much slower for larger circuits.

Combined Generalized Universal Block Construction. A combination of these generalized UB constructions uses the smallest generalized UB implementation (M1, M2 or M3) dynamically. *Dynamic Programming* avoids recalculation of the smallest construction for given parameters by caching it in a table.

$$|U_{k\times d}^{\mathbf{M}}| \le min(|U_{k\times d}^{\mathbf{M1}}|, |U_{k\times d}^{\mathbf{M2}}|, |U_{k\times d}^{\mathbf{M3}}|)$$

4.3 Universal Graph Based GUC Construction

A generalization of Valiant's UC construction [25] which is based on Universal Graphs results in a GUC construction of size

$$|UC_{k,u,v\times d}^{\mathbf{UG}}| \sim 4.75d(2k + u + \frac{v}{d-1})\log k.$$

This GUC construction is asymptotically better than those shown before for large circuits and is based on the following theorem.

Theorem 1. *Each circuit of arbitrary fan-out with v outputs and k gates of d inputs each can be converted into an equivalent circuit with fan-out $\leq d$ by adding at most $k + \frac{v}{d-1}$ gates.*

For lack of space, the detailed description of this construction and the proof of the theorem are given in §A.

4.4 Comparison of GUC Constructions

The sizes of the different GUC constructions for several practical parameters are shown in Fig. 4 of §B. Which construction is the smallest (and hence should be used for least overhead) depends on the parameters d, k, u, and v only. Quadratic constructions ($I1, I2, M1, M2$) are suitable for small, whereas recursive construction ($M3$) is better for mid-size and universal graph based construction (UG) for large circuits.

5 Secure Evaluation of Private NNs with GUCs

5.1 Structure of NNs

A *neural network* (NN) is an acyclic directed graph of several neurons. The neurons are arranged in multiple layers (Fig. 2) in topologic order. Each neuron has d input bits, one output bit, internal precision of s bits and is structured as shown in Fig. 3(a). Each input bit in_i of a neuron is multiplied with a s-bit constant weight w_i. The Σ block computes the sum σ of these weighted inputs. The threshold function τ compares σ with a threshold value t and sets the output: if $\sigma := \sum_{i=1}^{d} w_i \cdot in_i < t$, then $out = 0$, else $out = 1$.

The input neurons of the first layer in the NN have only 1 input with multiple bits m. They can also be realized with a neuron as shown in Fig. 3(a) by setting the weights correspondingly.

Neurons fulfill the restrictions for d input gates of §4: Inputs can be permuted arbitrarily by permuting their weights in the same way. Duplicate inputs into a neuron with the same source can be merged into one input by adding their weights. Implementation of neurons presented next guarantees that the weights of neurons remain hidden from requester and hence these modifications are not detectable for him.

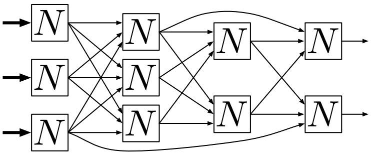
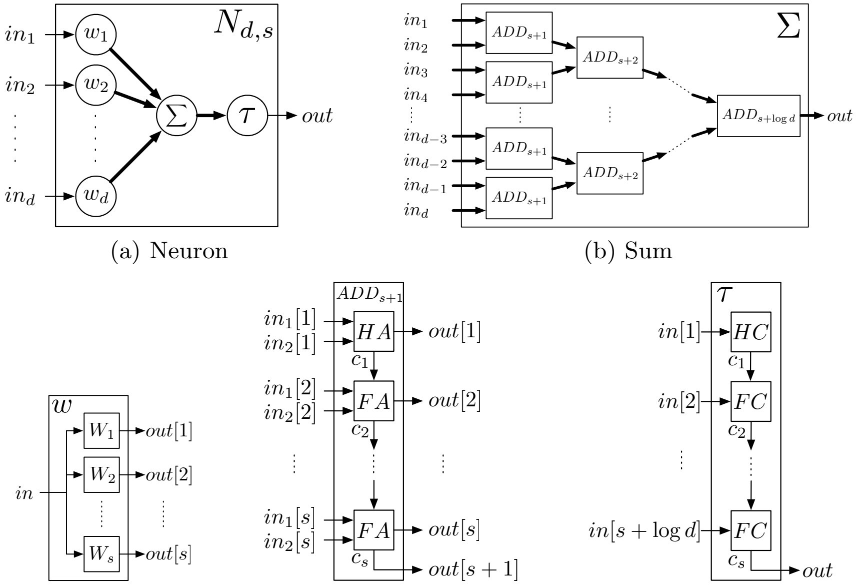


Fig. 2. NN with 4 layers of $k = 10$ neurons with degree $d = 3$ including $u = 3$ input neurons and $v = 2$ outputs

5.2 Circuit Implementation of Neurons

If the number of inputs d is small, each neuron can be implemented as programmable d input gate of size $|N_{d,s}^{\mathbf{gate}}| = 2^{d-2}$ (for arbitrary activation function), otherwise as programmable d input block (Fig. 3(a)). The neuron can be programmed depending on the weights $\omega_1, .., \omega_d$ and the threshold value t.



(a) Neuron

(b) Sum

(c) Weights

(d) Adder

(e) Threshold

Fig. 3. Circuit implementation of a neuron

As before we give directly the total size of the building blocks. Exact calculations with all intermediate steps are given in the full version of this paper.

The w block multiplies its input bit with constant ω of s bits (Fig. 3(c)). The bits $\omega[i]$, $i = 1, .., s$, determine the programming of the programmable gate W_i: if $\omega[i] = 0$, then $W_i = 0$, else $W_i = in$. The size is $|W_i| = 0.5$, $|w| = s \cdot |W_i| = 0.5s$.

The Σ block sums up the d input values of s bits each to a $s + \log d$ bit value by pairwise adding them in a tree (Fig. 3(b)). An adder ADD_{s+1} to add two s bit values to an $s+1$ bit value is composed as usual from a half adder HA and $s-1$ full adders FA (Fig. 3(d)). The size is $|HA| = 2$, $|FA| = 4$, $|ADD_{s+1}| = 4(s+1) - 6$, $|\Sigma| < 4ds + 2d - 4s + 6$.

The τ block compares the $s + \log d$ bit input with an $s + \log d$ bit constant t (Fig. 3(e)). The carry $c_i = 0$, $i = 1, .., s + \log d$, tells that the i least significant bits of in are less than the i least significant bits of t: if $((in \mod 2^i) < (t \mod 2^i))$, then $c_i = 0$, else $c_i = 1$. Depending on $t[1]$, the programmable half comparer block HC is programmed: if $t[1] = 0$, then $HC = 1$, else $HC = in_1$. The remaining bits $t[i]$, $i = 2, .., s + \log d$, determine the program of the programmable full comparer blocks FC_i: if $t[i] = 0$, then $FC_i = in_i \vee c_{i-1}$, else $FC_i = in_i \wedge c_{i-1}$. The size is $|HC| = 0.5$, $|FC| = 1$, $|\tau| = s + \log d - 0.5$.

The total size of a neuron implemented as programmable block is $|N^{\mathbf{block}}_{d,s}| < 4.5ds + 2d - 3s + \log d + 5.5$.

5.3 Protocol for Oblivious Evaluation of NNs Using GUCs

Oblivious evaluation of NNs is reduced to SFE of GUCs similar to the reduction for PF-SFE [10]. A GUC is programmed to simulate the structure of the NN. Each gate simulation block G is instantiated with a programmable circuit for a neuron $N_{d,s}$ programmed with the coefficients of the neuron it simulates: $UC_{NN} = UC_{k,u,v \times d}|_{G=N_{d,s}}$. Programmed GUC simulates the NN and entirely hides its structure (besides size and number of inputs and outputs): $\forall (in_1, .., in_u) \in \{0,1\}^u : \mathrm{UC}_{NN}(in_1, .., in_u) = NN(in_1, .., in_u)$.

When requester evaluates the programmed UC_{NN} with a SFE protocol, he learns no more about NN than the maximal number of neurons k, maximal degree of neurons d, internal precision s, inputs u and outputs v.

The protocol needs one round in the semi-honest model (using interactive OT such as [19,1,8]) or is non-interactive (using non-interactive OT of [6]). Recall, the reduction from PF-SFE to SFE using UC (resp. GUC in our case) is non-cryptographic and the security of the PF-SFE protocol is exactly that of the underlying SFE protocol which is provably secure against semi-honest adversaries (e.g., [27,12,9]). This can be extended to be provably secure against malicious adversaries by using correspondingly secure SFE protocols (e.g., [14,13,5]) which need more than one but still a constant small number of rounds.

5.4 Comparison with Previous Work

We compare our GUC constructions with existing UC constructions before comparing our protocol for secure evaluation of NNs with existing protocols. As example for both comparisons we use the practical NN to classify sonar targets from [4] for which performance results are given in [15]. However, we use threshold instead of sigmoid as activation function as performance of [15] is almost the same independent of the used activation function. Besides this we use exactly the same parameters for the neural network, namely $u = 60$ inputs, $v = 2$ outputs, $k = 12$ hidden neurons with an internal resolution $s = 20$ (corresponding to their quantization factor $Q = 10^{-6}$). In order to obfuscate the structure of NN they propose to embed NN into a grid of 5 layers with 15 neurons each. Hence, the in-degree of each neuron is $d = 15$ while the total number of neurons is hidden to be less than $k = 75$. This results in neurons of size $|N| = |N^{\mathbf{block}}_{d=15,s=20}| \leq 1,330 < |N^{\mathbf{gate}}_{d=15,s=20}| = 8,192$.

We compare three possible alternatives that protect the internal weights and thresholds of the neurons and incrementally protect the structure of the NN:

(A) Embed NN into a 5 × 15 grid to obfuscate its structure (same as [15]): evaluate $75 \cdot |N| \leq 99,750$ gates.

(B) Hide the structure of the NN entirely:
simulate NN of $k = 12$ neurons ($12 \cdot |N| \leq 15,960$ gates) in GUC ($|UC^{\mathbf{M1}}_{k=12,u=60,v=2\times d=15}| \leq 2,304$ gates, cf. Fig. 4(g) in §B).

(C) Additionally hide the number of neurons to be less than 75:
simulate NN of $k = 75$ neurons ($75 \cdot |N| \leq 99,750$ gates) in GUC ($|UC^{\mathbf{M2}}_{k=75,u=60,v=2\times d=15}| \leq 27,371$ gates, cf. Fig. 4(h) in §B).

Comparison of GUC Construction. As shown in Table 1, the overhead of our GUC introduced in case (B) and (C) is very moderate compared to known UC constructions. Applying *UC* directly results in an overhead which is by a factor of $> 10^3$ times bigger, while using $\lceil d/2 \rceil = 8$ parallel UCs still is by a factor of more than two times bigger than our solution.

Table 1. Comparison of UC overhead (in number of gates)

	UC		Parallel UCs		GUC
	[25]	[10]	[25]	[10]	§4
(B): $k = 12$	$4,242,114$	$5,556,431$	$23,432$	$6,577$	$2,304$
(C): $k = 75$	$31,482,358$	$47,478,158$	$100,359$	$58,618$	$27,371$

Comparison of Protocol for Secure Evaluation of NNs. We used Fairplay SFE system [14] implemented in Java as well without cut and choose step to evaluate a circuit with $u = 60$ inputs, $v = 2$ outputs and the given number of gates within two processes on a notebook with 2.16 GHz Intel Core 2 processor and 2 GB memory. As [15] do not specify the exact hardware used ("two mid-level notebooks, connected on a LAN network"), the results of the comparison shown in Table 2 are qualitative but not necessarily quantitative:

Unlike the protocol in [15], ours are provably secure and approaches (B) and (C) hide the structure of NN better than just obfuscating it.

The total amount of communication overhead of [15] is by a factor of at least 10 times better than our solutions, however they need multiple rounds (for each layer of the NN) whereas our solutions need only one round in the semi-honest model or even no online-communication at all (using non-interactive OT). On analyzing the communication complexity separately for server and client we see that in our solutions the amount of data sent by the client is much smaller than that of the server and only depends on the number of inputs but not on the size of NN. The amount of data sent by client is by a factor of five less than that in [15] (as the client in their symmetric protocol sends approximately half of the total data). This asymmetry in the communication exactly corresponds to modern communication networks such as mobile networks or the internet, where the upstream of the client is much slower than its downstream. Downloading the maximum amount of 5.4 MB in (C) is realistic for today's mobile networks.

The total time for executing the protocol of (A) and (C) is almost the same as that of [15] while (B) is almost three times faster. While in [15] client has to do only 20% of the work, in our protocols, server and client need approximately the same amount of computation for creating and evaluating the garbled

Table 2. Comparison of protocols for secure evaluation of NNs

Protocol	[15]	(A)	(B)	(C)
Level of privacy	obfuscate	obfuscate	hide structure	hide structure&size
Provably secure	no	yes		
Communication (Total)	76 kB	4.2 MB	0.79 MB	5.4 MB
Server (send)	≈ 38 kB	4.2 MB	0.78 MB	5.4 MB
Client (send)	≈ 38 kB	7.5 kB		
Rounds	5	1 (0)		
Computation (Total)	11.7 s	11.3 s	4.0 s	13.4 s
Server	9.3 s	≈ 5.7 s	≈ 2.0 s	≈ 6.7 s
Client	2.4 s	≈ 5.7 s	≈ 2.0 s	≈ 6.7 s

circuit. Online-computation of our server can be avoided almost completely by constructing the garbled circuit in advance while server is idle. This reduces total execution time of our protocols by half.

Using [9] as underlying SFE protocol or a high-speed SFE implementation such as [11] written in C with elliptic curve based OT would further improve communication and computation complexity of our protocols.

Acknowledgments. We thank Vladimir Kolesnikov and anonymous reviewers for their helpful comments.

References

1. Aiello, W., Ishai, Y., Reingold, O.: Priced oblivious transfer: How to sell digital goods. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 119–135. Springer, Heidelberg (2001)
2. Chang, Y.-C., Lu, C.-J.: Oblivious polynomial evaluation and oblivious neural learning. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 369–384. Springer, Heidelberg (2001)
3. Drewes, R.: An artifical neural network spam classifier (August 2002), `http://www.interstice.com/drewes/cs676/spam-nn/`
4. Gorman, R.P., Sejnowski, T.J.: Analysis of hidden units in a layered network trained to classify sonar targets. Neural Networks 1(1), 75–89 (1988)
5. Goyal, V., Mohassel, P., Smith, A.: Efficient two party and multi party computation against covert adversaries. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 289–306. Springer, Heidelberg (2008)
6. Gunupudi, V., Tate, S.R.: Generalized non-interactive oblivious transfer using count-limited objects with applications to secure mobile agents. In: Tsudik, G. (ed.) FC 2008. LNCS, vol. 5143, pp. 98–112. Springer, Heidelberg (2008)
7. Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural Networks 2(5), 359–366 (1989)
8. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending Oblivious Transfers Efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer, Heidelberg (2003)

9. Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free XOR gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008)
10. Kolesnikov, V., Schneider, T.: A practical universal circuit construction and secure evaluation of private functions. In: Tsudik, G. (ed.) FC 2008. LNCS, vol. 5143, pp. 83–97. Springer, Heidelberg (2008), `http://thomaschneider.de/FairplayPF`
11. Lindell, Y., Pinkas, B., Smart, N.: Implementing two-party computation efficiently with security against malicious adversaries. In: Ostrovsky, R., De Prisco, R., Visconti, I. (eds.) SCN 2008. LNCS, vol. 5229, pp. 2–20. Springer, Heidelberg (2008)
12. Lindell, Y., Pinkas, B.: A proof of Yao's protocol for secure two-party computation. Cryptology ePrint Archive, Report 2004/175 (2004)
13. Lindell, Y., Pinkas, B.: An efficient protocol for secure two-party computation in the presence of malicious adversaries. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 52–78. Springer, Heidelberg (2007)
14. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay — a secure two-party computation system. In: USENIX (2004), `http://www.cs.huji.ac.il/project/Fairplay/fairplay.html`
15. Orlandi, C., Piva, A., Barni, M.: Oblivious neural network computing via homomorphic encryption. European Journal of Information Systems (EURASIP) 2007(1), 1–10 (2007)
16. Pinkas, B.: Cryptographic techniques for privacy-preserving data mining. SIGKDD Explor. Newsl. 4(2), 12–19 (2002)
17. Piva, A., Caini, M., Bianchi, T., Orlandi, C., Barni, M.: Enhancing privacy in remote data classification. In: New Approaches for Security, Privacy and Trust in Complex Environments (SEC 2008) (2008)
18. Plagianakos, V.P., Vrahatis, M.N.: Parallel evolutionary training algorithms for hardware-friendly neural networks. Natural Computing 1(2-3), 307–322 (2002)
19. Rabin, M.O.: How to exchange secrets with oblivious transfer. Technical report, Harward University, Available at Cryptology ePrint Archive, Report 2005/187 (1981)
20. Sander, T., Young, A., Yung, M.: Non-interactive cryptocomputing for NC^1. In: Proc. 40th IEEE Symp. on Foundations of Comp. Science, FOCS 1999, New York, pp. 554–566. IEEE, Los Alamitos (1999)
21. Sato, K., Hikawa, H.: Implementation of multilayer neural network with threshold neurons and its analysis. Artificial Life and Robotics 3(3), 170–175 (1999)
22. Schneider, T.: Practical secure function evaluation. Master's thesis, University of Erlangen-Nuremberg (2008), `http://thomaschneider.de/theses/da/`
23. StatSoft, Inc. STATISTICA Automated Neural Networks (2008), `http://www.statsoft.com/products/stat_nn.html`
24. Tebelskis, J.: Speech Recognition using Neural Networks. PhD thesis, School of Computer Science, Pittsburgh (1995)
25. Valiant, L.G.: Universal circuits (preliminary report). In: STOC 1976, pp. 196–203. ACM Press, New York (1976)
26. Waksman, A.: A permutation network. J. ACM 15(1), 159–163 (1968)
27. Yao, A.C.: How to generate and exchange secrets. In: FOCS 1986, Toronto, pp. 162–167. IEEE, Los Alamitos (1986)

A Universal Graph Based GUC Construction

To construct an asymptotically better UC than the practical constructions presented before we generalize Valiant's UC construction [25]. Valiant shows how to construct a UC by embedding the given circuit into a universal graph $\Gamma_d(k')$ with $|\Gamma_d(k')| \sim 4.75dk' \log k'$, where d is the fan-in and fan-out of the simulated graph and k' is the number of simulated nodes. A circuit with u inputs and k gates having fan-in and fan-out maximum d can be represented as such a simulateable graph with $k' = k + u$ nodes and embedded into this universal graph. We generalize Valiant's construction (that uses $d = 2$ and can simulate circuits with gates having 2 inputs only) to a GUC that simulates circuits with arbitrary fixed in-degree d and arbitrary fan-out of size

$$|UC^{\mathbf{UG}}_{k,u,v\times d}| \sim 4.75d(2k + u + \frac{v}{d-1}) \log k.$$

The circuit is converted from arbitrary fan-out to a circuit with fan-out at most d which can be embedded into the universal graph $\Gamma_d(k')$. This is done by replacing each gate with fan-out $x > d$ by a binary tree of $\lceil \frac{x}{d-1} \rceil + 1$ gates with fan-out $\leq d$. At most $e = k + \frac{v}{d-1}$ extra gates are needed as described in §A.1. Setting $k' = k + u + e$ results in the stated complexity for the generalized Valiant's UC construction. By setting $d = 2$ we obtain exactly the asymptotic complexity of Valiant's original UC construction.

A.1 Converting Circuit to Fan-Out $\leq d$

As described in §A, a circuit with s gates of arbitrary fan-out and m outputs can be converted into one having gates with fan-out $\leq d$ only, by replacing each gate with fan-out $x > d$ by a binary tree of $\lceil \frac{x}{d-1} \rceil + 1$ gates with fan-out $\leq d$ each.

Theorem 1 in §4.3 gives an upper bound for the maximal number of extra gates e added which we prove similar to [25, Fact 3.1 and Corollary 3.1]:

Fact: Suppose that a graph G with fan-in d has among the nodes of in-degree zero, n' nodes of nonzero out-degree, and amongst the rest v' nodes of out-degree zero, f_i of out-degree i, $i = 1, .., d-1$, and g of out-degree greater than $d-1$. Suppose also that $e' = \sum(out - degree - d) = \sum x$ over the set of nodes with out-degree greater than d. Then $e' \leq \sum_{i=1}^{d-1}(d-i)f_i + dv' - n'$.

Proof. The total of the out-degrees must equal the total of the in-degrees. The former is $\geq n' + \sum_{i=1}^{d-1}(if_i) + dg + e'$, and the latter is $\leq d(v' + \sum_{i=1}^{d-1} f_i + g)$. □

Corollary: $e \leq k + \frac{v}{d-1}$

Proof. Any gate with fan-out $x + d$, $x > 0$, can be replaced by a binary tree with $\lceil \frac{x}{d-1} \rceil + 1 \leq \frac{x}{d-1} + 2$ gates. Hence for any circuit there is an equivalent one having fan-out d and at most $e \leq \frac{e'}{d-1} + g$ more gates. But the total number of gates $k = \sum_{i=1}^{d-1} f_i + g + v'$ and in any minimal circuit $v' \leq v$. Hence at most

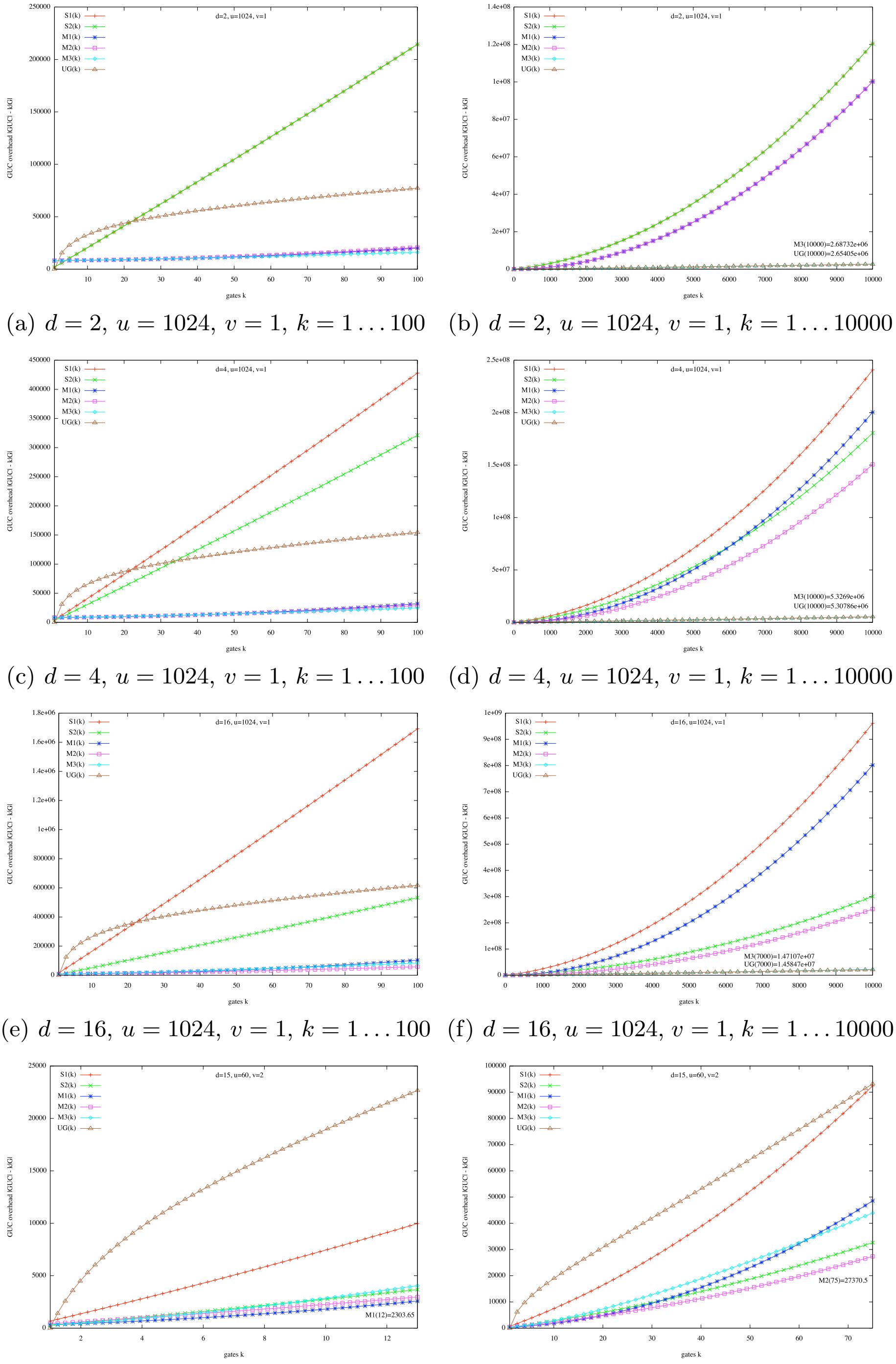
$$e \leq \frac{e'}{d-1} + g \leq \frac{d}{d-1}v' + \sum_{i=1}^{d-1} \frac{d-i}{d-1} f_i + g \leq s + \frac{d}{d-1}v' - v' \leq k + \frac{v}{d-1}$$

extra gates are needed which completes the proof of the corollary and Theorem 1. □

B Tables and Figures

Table 3. Programmable switching blocks

Block	Name of Block	Size	Duplicates	Order
S_1^u [10]	Selection	$u-1$	X	X
S_v^u [10]	Selection (simple)	$v(u-1)$	X	X
$S_{v\geq u}^u$ [10]	Selection (efficient)	$(u+v)\log u + 2v \log v - 2u - v + 3$	X	X
$S_v^{u\geq v}$ [10]	Selection (efficient)	$(u+3v)\log v + u - 4v + 3$	X	X
S_{2u}^u [10]	Selection (improved)	$6u \log u + 3$	X	X
P_u^u [26]	Permutation	$2u \log u - 2u + 2$	-	X
$EP_{v\geq u}^u$ [10]	Permutation (expanded)	$(u+v)\log u - 2u + 2$	-	X
$C_d^{u,\mathbf{simple}}$ (§3)	Choice (simple)	$du - d^2$	-	-
$C_d^{u,\mathbf{sublin}}$ (§3)	Choice (sub-linear)	$(\log d + 1)u - 3d + 2$	-	-
$BP_{v\times d}^{u\geq vd}$ (§3)	Bundle Permutation	$(u+dv)\log v + (\log d + 1)u - 3dv + 2$	-	-



(a) $d = 2,\ u = 1024,\ v = 1,\ k = 1 \ldots 100$ (b) $d = 2,\ u = 1024,\ v = 1,\ k = 1 \ldots 10000$

(c) $d = 4,\ u = 1024,\ v = 1,\ k = 1 \ldots 100$ (d) $d = 4,\ u = 1024,\ v = 1,\ k = 1 \ldots 10000$

(e) $d = 16,\ u = 1024,\ v = 1,\ k = 1 \ldots 100$ (f) $d = 16,\ u = 1024,\ v = 1,\ k = 1 \ldots 10000$

(g) $d = 15,\ u = 60,\ v = 2,\ k = 1 \ldots 13$ (h) $d = 15,\ u = 60,\ v = 2,\ k = 1 \ldots 75$

Fig. 4. Comparison of GUC constructions for different parameters