Practical Secure Evaluation of Semi-Private Functions

Annika Paus, Ahmad-Reza Sadeghi*, and Thomas Schneider**

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

Abstract. Two-party Secure Function Evaluation (SFE) is a very useful cryptographic tool which allows two parties to evaluate a function known to both parties on their private (secret) inputs. Some applications with sophisticated privacy needs require the function to be known only to one party and kept private (hidden) from the other one. However, existing solutions for SFE of private functions (PF-SFE) deploy Universal Circuits (UC) and are still very inefficient in practice.
In this paper we bridge the gap between SFE and PF-SFE with SFE of what we call *semi-private functions* (SPF-SFE), i.e., one function out of a given class of functions is evaluated without revealing which one.
We present a general framework for SPF-SFE allowing a fine-grained trade-off and tuning between SFE and PF-SFE covering both extremes. In our framework, semi-private functions can be composed from several privately programmable blocks (PPB) which can be programmed with one function out of a class of functions. The framework allows efficient and secure embedding of constants into the resulting circuit to improve performance. To demonstrate practicability of the framework we have implemented a compiler for SPF-SFE based on the Fairplay SFE framework.
SPF-SFE is sufficient for many practically relevant privacy-preserving applications, such as privacy-preserving credit checking which can be implemented using our framework and compiler as described in the paper.

Key words: SFE of semi-private functions, Yao's protocol, topology, optimization, compiler, privacy

1 Introduction

Two-party Secure Function Evaluation (SFE) is an important and wide area of cryptographic research (see, e.g., [Yao86,LP04,MNPS04,AHL05,Kol05,LP07,KS08a,LPS08]). It allows two parties to securely evaluate a common function on their private inputs without involving a trusted third party. The function is represented as a boolean circuit and evaluated based on a garbled version of the circuit which is created by one party (constructor Bob) and evaluated by the other party (evaluator Alice). Usually SFE hides the intermediate results but - as the function is known to both parties - not the structure (topology) of the function.

In practice, however, a variety of business models require privacy properties beyond the secrecy of parties' input data to additionally keep the evaluated function private. The underlying business motivations vary from commercial incentives (e.g., protection of intellectual property) to pure security requirements to reduce the probability of credential forgery or to make insider attacks obsolete. Typical use cases are client-server applications where a user Alice inputs her private data x (hidden to Bob), the server Bob inputs his private function f (hidden to Alice), and the protocol outputs $f(x)$ to both parties such that neither party gain any information about the other party's input. Prominent examples are privacy-preserving trust negotiation schemes [FAL04,FLA06,FAL06], credit checking [FAZ05], or data classification using neural networks [SS08,OPB07,PCB+08].

* The second author was supported by the European Union under FP6 project SPEED.
** The third author was supported by the European Union under FP7 project CACE.

To allow SFE of a private function, called PF-SFE [KS08b], a universal circuit (UC) [Val76,KS08b,SS08] is evaluated that simulates the function, and entirely hides the structure of their circuit representation. UCs require a huge overhead of $O(k \log k)$ [Val76], $O(k \log^2 k)$ [KS08b], respectively $O(k^2)$ [SS08] additional gates, where k is the number of gates of the simulated circuit.

Fairplay [MNPS04], a state-of-the art implementation of SFE, can evaluate functions consisting of millions of gates whereas in FairplayPF [KS08b], a recent implementation for PF-SFE, functions are restricted to a few thousand gates only due to the huge overhead for evaluating UC. Hence, a better trade-off between maximal performance (SFE) and maximal privacy of the evaluated function (PF-SFE) is desired. For many practically relevant applications (e.g., those mentioned above) it is sufficient that functions are only partly private, what we call *semi-private functions* (SPF). Basically, these applications reflect the following scenario: A user Alice has private data x, and a service provider Bob has a semi-private function $f \in \mathcal{F}$ as input, where $\mathcal{F}$ represents a given class of functions. At the end of the protocol, Alice obtains $f(x)$ but not which specific f was evaluated and Bob obtains no information on x. This problem, called *secure function evaluation of semi-private functions* (SPF-SFE), can be reduced to Yao's protocol where circuit's topology is revealed to the evaluator and only the functionality of the gates of the circuit is hidden. The evaluator sees the circuit topology but can only guess which functionality each part of the circuit might evaluate. We concentrate on *relaxed-security* model, i.e., security against malicious evaluator Alice and semi-honest (honest-but-curious) constructor Bob. This model is widely used in current cryptographic literature [NP99,AIR01,LL07] and well-justified in many practical applications where performance is crucial and constructor Bob can be assumed to behave semi-honestly by means of legal contracts or possible loss of reputation.

While SPF-SFE based on Yao's protocol has been proposed as building block in many applications (e.g., [FAL04,FAZ05,FLA06,FAL06,SS08]), we give the first unified theory for SPF-SFE. Extending and improving previously known techniques we present a general theoretical framework for SPF-SFE together with a compiler to automatically generate SPF-SFE protocols for practical applications.

Related Work. The idea of constructing circuits for a special class of functions and evaluating them efficiently with Yao's protocol in the relaxed security model have been used in several sub-protocols [FAL04,FAZ05,FLA06,FAL06,KS08b,SS08]. Frikken et al. call the respective building blocks oblivious gates/circuits where evaluator does not know the function that each gate/circuit computes. However, they only mention the existence of several useful topologies like binary trees, comparison circuits, or universal circuits together with their asymptotic size, but do not give explicit constructions. We extend their basic ideas into a generic framework and provide a wide class of functional blocks, each with a concrete efficient implementation (topology, programming, and exact size), that can be arbitrarily combined to represent semi-private functions in many practical applications.

Existing frameworks for secure computation based on Yao's protocol are the Fairplay SFE system [MNPS04] with a proposed extension to the malicious model [LPS08] and another extension to private functions with UCs (PF-SFE), called FairplayPF [KS08b]. The Fairplay compiler includes an optimizer that optimizes on the basis of the high-level Secure Function Description Language (SFDL) using peek-hole optimization, duplicate code removal, and dead code elimination. In contrast to this, our proposed optimization algorithm for constant inputs optimizes on the lower abstraction level of circuits and can also be applied to further optimize the output of circuits generated with the Fairplay compiler.

Our Contribution & Outline. We propose a general framework together with a compiler for efficient secure function evaluation of semi-private functions (SPF-SFE) in the relaxed-security model. Some of our contributions are of independent interest:

- In §2 we describe how common SFE can be extended with building blocks that we call *Privately Programmable Blocks* (PPB) to allow practical secure evaluation of semi-private functions (SPF-SFE). A privately programmable block (§4) consists of a fixed topology of several programmable gates (with a small number of inputs) and can be programmed to evaluate different functions out of a class of functions. The evaluator learns how the blocks are connected (topology) but *not* with which of the functions of their corresponding class of functions the blocks are programmed. Hence parts of the function are hidden from the evaluator while the topology is still revealed. In §5 we show how to design efficient constructions for PPBs that also allow to securely incorporate private constants into PPBs and give concrete constructions that are of special interest for practical applications. In particular we present efficient PPB constructions for arithmetic operations, i.e., adding or subtracting two numbers/a number and a private constant, compare two numbers/a number and a private constant, multiply a number with a private constant, as well as boolean operations (§B). Also switching functions, e.g., permutation and selection blocks, as well as universal circuits from [KS08b] fit into this concept. The resulting SPF-SFE protocol is as efficient as plain SFE (both in the relaxed-security model) while providing function privacy at the same time.
- In §8 we present an optimization algorithm that incorporates constant inputs into the circuit resulting in a circuit with less inputs and smaller size having a topology which is *independent* of the values of the constant inputs. Besides the well known propagation of constant inputs, our algorithm additionally eliminates resulting gates with one input by incorporating them into surrounding gates which results in smaller circuit size. The proposed optimization algorithm applies no cryptographic modification of circuits and hence is of independent interest. This optimization can be used in combination with Yao's SFE protocol in the relaxed-security scenario where constant inputs might be public values known to both parties as well as the inputs of circuit constructor Bob.
- In order to allow usage of SPF-SFE in many practical applications we present a general compiler framework for secure evaluation of semi-private functions, called FairplaySPF, based on the well known Fairplay SFE system [MNPS04] as described in §6. Our new Secure Programmable Block Description Language (SPBDL) allows to specify the topology of interconnected programmable blocks together with their corresponding private programming. A compiler automatically compiles SPBDL descriptions to circuits described in Fairplay's Secure Hardware Description Language (SHDL). After incorporating Bob's inputs into the circuit with the optimization algorithm presented in §8, the circuit can securely be evaluated with the SPF-SFE protocol while hiding the programming. Also a universal circuit (UC) that is evaluated in PF-SFE (cf. [KS08b]) can be seen as a privately programmable block that is programmed with a private circuit (specified in SHDL). By incorporating UCs as programmable blocks into SPBDL, our framework becomes a general purpose framework capable of expressing SFE, SPF-SFE as well as PF-SFE and also arbitrary combinations of them where only sensitive parts of the function's structure are hidden as shown in the example in §7. This allows a fine-grained trade-off between performance and privacy of the evaluated function.
- Our framework and compiler can be applied (combining SPF-SFE and PF-SFE) to implement and improve efficiency of several applications such as privacy-preserving credit checking [FAZ05], blinded policy evaluation [FAL04,FLA06,FAL06], or secure data classification [SS08]. In §7 of this paper we concentrate on privacy-preserving credit checking. Usually, before getting a loan from a bank a person has to reveal a substantial amount of private information. This information has to satisfy certain criteria that are defined by

the bank. We show how SPF-SFE can be used to securely evaluate the trustworthiness of a borrower while ensuring that (i) the privacy of his input is preserved and (ii) nothing is revealed about the criteria of the bank used for credit checking. Instead of using a UC for the whole function as in PF-SFE we reveal the topology of the trivial part of the function (e.g., comparing attributes with thresholds) and only hide the sensitive part in a UC, which is much more efficient. The description of the function in SPBDL can automatically be compiled into SHDL code with our compiler. This can be obliviously evaluated in a one-round protocol.

2 Yao's Protocol and Semi-Private Functions

Yao's Protocol. In the following, we concentrate on Yao's protocol [Yao86] for SFE. Yao's protocol is often called *garbled circuit* protocol as a garbled version of the (boolean) circuit representing the function is created by one party (constructor Bob) and evaluated by the other party (evaluator Alice) as described in the following. For each wire of the circuit, Bob uses two random bit strings (garbled values) that are assigned to the corresponding values 0 and 1, respectively. Note, that the garbled values do not reveal to which value they correspond as they are chosen randomly. Bob sends *only* the garbled values corresponding to his inputs (garbled inputs) to Alice. For Alice's inputs, Bob uses 1-out-of-2 oblivious transfer (OT) to send Alice *only* the garbled values corresponding to her inputs without Bob learning which strings she gets. Additionally, for each gate G_i of the circuit, Bob creates and sends to Alice a *garbled table* T_i with the following property: given garbled values for G_i's inputs, T_i allows to recover *only* the garbled value of the corresponding output of G_i and nothing else. Afterwards, Alice uses the received garbled values of the input wires and garbled tables T_i to evaluate the garbled circuit gate by gate. The output wires of the circuit are not garbled (or the mappings from garbled values to values 0 and 1 are published by Bob), thus Alice learns (only) the output of the circuit, but no plain values of internal wires (only garbled values). Correctness and security against semi-honest adversaries of Yao's protocol are proven in [LP04]. It is easy to show that Yao's protocol is even secure against malicious Alice, i.e. relaxed secure, as the only message Alice sends to Bob is within OT protocol where Alice is unable to cheat assuming security of OT protocol against malicious adversaries [FAL06, Appendix A]. An efficient relaxed secure OT protocol is given for example in [AIR01].

Yao's protocol is the kernel of existing implementations of SFE protocols [MNPS04,LPS08] which also extend it to be secure against malicious constructor Bob via cut-and-choose, e.g., multiple circuits are garbled, correctness of some of them is verified by revealing all garbled input values (called open) and the remaining ones are evaluated. As justified in the introduction, we concentrate on the plain Yao's protocol (secure against semi-honest Bob and potentially malicious Alice) where only *one* circuit is evaluated and no circuits are opened.

Yao's Protocol for Semi-Private Functions. Observe, in Yao's protocol the garbled tables T_i consist of symmetric encryptions of the garbled output value using the corresponding garbled input values as keys. Alice can use these garbled input values to decrypt exactly the one garbled output value corresponding to these keys. All other garbled output values, i.e., entries of the garbled function table remain hidden from Alice and hence she cannot determine the type of the gate. The only information Alice learns about the function in Yao's protocol is the *topology* of the circuit, i.e., the way the different gates are connected and how many inputs each gate has.

When Alice obtains a garbled circuit from Bob, she can guess from its topology what functionality the circuit evaluates, e.g., chains of 3-input gates might be an integer comparison circuit. This can be exploited constructively by Bob to keep parts of the function

private, we call this a *semi-private function*, as follows. Bob composes his intended functionality from blocks with a fixed topology that can evaluate different functionalities each, called *privately programmable blocks* (PPBs) as explained in §4. The maximum amount of information Alice can gain from the topology of each PPB is the set of functionalities the PPB might compute but not the specific functionality of this PPB chosen privately by Bob.

From combining these two arguments follows that evaluation of a circuit, composed out of several PPBs representing the semi-honest function, with Yao's protocol is a secure protocol for SPF-SFE.

Additionally, (semi-honest) Bob can incorporate his input values into the circuit before garbling the circuit if they are already known at that time. In §8 we give an algorithm for efficient optimization of circuits for Bob's (constant) inputs together with an example. The optimization *only* depends on the topology of the original circuit but not on Bob's input values and hence the optimized circuit does not reveal more information on Bob's input values than the original circuit. After this optimization, Bob no longer needs to transfer the garbled values corresponding to his input values and also the size of the circuit is reduced (resulting in less communication and computation). The bottom part of Fig. 8 contains the high-level overview of SPF-SFE protocol including this optimization.

3 Definitions and Preliminaries

Let $x \in [0, 2^\ell)$ be an unsigned ℓ-bit integer value and $\boldsymbol{x} = (x_1, .., x_\ell)$, $x_i \in \{0,1\}$ its corresponding representation as bit vector, i.e., $x = \sum_{i=1}^{\ell} x_i 2^{i-1}$. The *length* of $\boldsymbol{x}$ is $|\boldsymbol{x}| = \ell$.

We draw a (single) *wire* with one-bit value as $\longrightarrow$. As usual, *multi wire* X with ℓ-bit value x is drawn as $\overset{\ell}{\longrightarrow}$ and consists of ℓ wires indexed by $X[i]$, $i = 1, .., \ell$ with values x_i.

A *gate* G with degree d has d *inputs* and one *output*. It is the implementation of a boolean function $g : \{0,1\}^d \to \{0,1\}$. As special case, a *constant gate* has no inputs ($d = 0$) and outputs a constant value. The *size* of a gate G, denoted by $|G|$, is the number of function table entries needed to implement the gate, namely $|G| = 2^d$. A gate with $e > 0$ outputs can easily be combined from e gates with one output resulting in size $e \cdot 2^d$.

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. The *size* of a circuit, denoted by $|C|$, is the sum of the sizes of its gates. Note, communication and computation complexity of efficient SFE protocols is linear in the size of the circuit.

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 function $f_B : \{0,1\}^u \to \{0,1\}^v$ mapping input values to output values. Blocks consist of connected gates and other sub-blocks. *Size* of block B, denoted by $|B|$, is the sum of the sizes of its sub-elements.

A *programmable gate* (PG) is a gate with an unspecified function table. *Programming* it is done by providing a specific function table with 2^d entries (one entry for each input combination). The concept of PGs corresponds to a universal circuit for simulating a single gate in Valiant's UC construction [Val76]. As described in the previous section, in SPF-SFE evaluator Alice is not able to extract the corresponding function table (program) from PG.

Analogously, a *programmable block* (PB) is a block consisting of programmable gates or programmable sub-blocks. It is programmed by programming each of its sub-elements. As described before, in SPF-SFE evaluator Alice is unable to extract the program from PB.

4 Privately Programmable Blocks

In this section we present our new concept of *Privately Programmable Blocks* (PPB) for constructing semi-private functions. Using our efficient PPB constructions given in §5 with

the SPF-SFE protocol of §2 allows to preserve the privacy of the function while the protocol remains as efficient as SFE protocol.

Definition. A *Privately Programmable Block* (PPB) is a programmable block which can be programmed to compute any function f of a given class of functions $\mathcal{F}$ (e.g., $\mathcal{F} = \{ADD, SUB\}$) with a corresponding program p_f (e.g., $f = ADD$). We write PPB^f for a PPB which is programmed to compute f.

$$\forall f \in \mathcal{F}, \forall (in_1, .., in_u) \in \{0,1\}^u : PPB^f(in_1, .., in_u) = f(in_1, .., in_u).$$

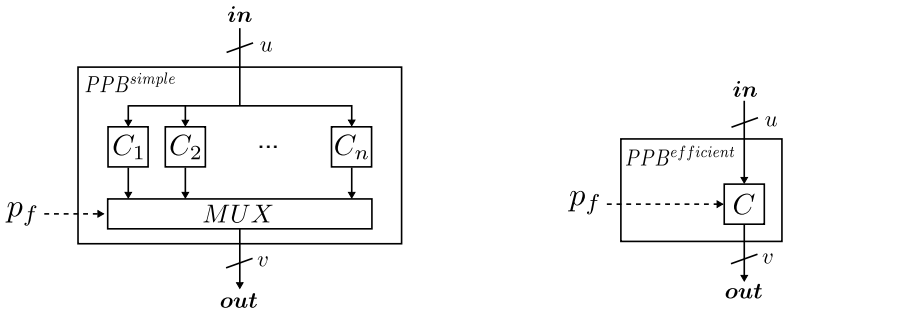
As explained in §2 before, in SPF-SFE the function to be evaluated is composed of several PPBs. Evaluator Alice learns how the PPBs are connected (topology), but the programming of the PPBs remains to be private information of constructor Bob (that's why PPBs are called *privately* programmable). Alice can infer from the topology of a PPB at most the class of possible functionalities $\mathcal{F}$ but not the specific functionality f chosen by Bob. Hence, from Alice's point of view the PPB can compute one functionality from $\mathcal{F}$ and the amount of information hidden inside the PPB is $\log_2 |\mathcal{F}|$ bits. For a semi-private function which is composed from multiple programmable blocks $PPB_1, .., PPB_n$, the program of each PPB can be combined with any programming of the other PPBs and hence the maximum (as some combinations might not make sense depending on the application) amount of information hidden in the semi-private function is $\log_2(|\mathcal{F}_1| \cdot .. \cdot |\mathcal{F}_n|) = \sum_{i=1}^{n} \log_2 |\mathcal{F}_i|$ bits. Clearly, if this is not large enough (i.e., if number of PPBs n or number of possible functionalities of PPBs $|\mathcal{F}_i|$ is small), evaluator Alice might just guess the correct function with high probability or probe the system via exhaustive search which must be prohibited by other means.

Universal Circuits (UC) indeed are special PPBs that can be programmed to compute an arbitrary function. UC_k is capable of simulating *any* function corresponding to a circuit with up to k gates with two inputs each. UCs provide *full privacy* of the evaluated function as the topology is hidden entirely. However, they cause a huge overhead by increasing the size of the evaluated circuit by $O(k \log k)$ [Val76], $O(k \log^2 k)$ [KS08b], or $O(k^2)$ [SS08] additional gates which is often intolerable in practice. Evaluating a UC programmed with a private function known by constructor Bob with a SFE protocol is called Secure Evaluation of Private Functions (PF-SFE). By combining the PPBs presented in this paper with UCs, users can find a fine-grained trade-off between efficient PPB constructions for semi-private functions (SPF-SFE) and less efficient UC constructions for completely private functions (PF-SFE) as explained in §7.

Simple PPB Construction. A straight-forward implementation of a PPB for a class of n arbitrary functionalities $\mathcal{F} = \{f_1, f_2, .., f_n\}$ can directly be derived from the definition of PPB as shown in Fig. 1(a). Each functionality f_i is computed by circuit C_i and an $n:1$ multiplexer (MUX) which is programmed to select the intended output. The MUX block can be constructed from v parallel selection blocks S_1^n (as defined in [KS08b]) for each of the v outputs that can be programmed to select any of their n inputs as outputs.

If the program p_f is known by Bob beforehand it can directly be incorporated into the circuit as described in §8. After optimization, each of the v selection blocks consists of a chain of $n-1$ programmable 2-input gates programmable to select either their left or right input as output each [KS08b]. Size of this construction is $\left|PPB^{simple}\right| = 4v(n-1) + \sum_{i=1}^{n} |C_i|$.

Efficient PPB Constructions. *Efficient PPB constructions* can be obtained by choosing special classes of functionalities having circuits with exactly the same topology. This allows to re-use the same circuit C for the different functionalities f_i as shown in Fig. 1(b). For



(a) Simple PPB construction (b) Efficient PPB construction

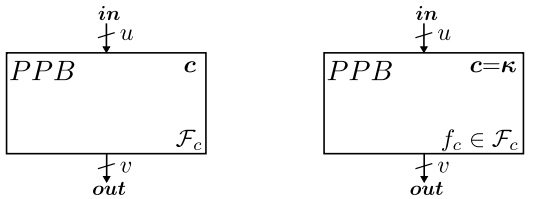
Fig. 1. PPB constructions

instance, the topology of an adder is the same as that of a subtractor and hence for $\mathcal{F} = \{ADD, SUB\}$ the same topology can be used (cf. §5.1 for a detailed description of this PPB). Based on the intended functionality $f \in \mathcal{F}$, the sub-elements of C are programmed differently while the topology is the same. The size is $\left|PPB^{efficient}\right| = |C| = |C_i| \ll \left|PPB^{simple}\right|$.

When incorporating a private constant $\boldsymbol{c}$ into a PPB, the *value of the constant can not be extracted from PPB's topology* and hence is hidden from evaluator in SPF-SFE protocol, e.g., circuits to add/subtract an input with a s-bit constant $\boldsymbol{c}$ have the same topology. To simplify notation, we parametrize the class of possible functionalities with parameter c and write $\mathcal{F}_c = \{f_{1_c}, .., f_{n_c}\}$ for $\mathcal{F} = \{f_1|_{c=0}, .., f_1|_{c=2^s-1}, f_2|_{c=0}, .., f_2|_{c=2^s-1}, \ldots, f_n|_{c=0}, .., f_n|_{c=2^s-1}\}$, e.g., $\mathcal{F}_c = \{ADD_c, SUB_c\}$ in the example given above (a detailed description of this PPB is given in §5.2). The amount of information hidden inside a PPB is

$$log_2 |\mathcal{F}| = log_2 |\mathcal{F}_c| + |c| = \log_2(n) + s\text{bits}. \tag{1}$$

Graphical Notation. In the following we use the uniform graphical notation shown in Fig. 2 to specify the interface of PPBs. For a *not programmed PPB* (Fig. 2(a)), the lower right corner contains the class $\mathcal{F}_c$ of functions with which it can be programmed and the upper right corner contains an (optional) $\boldsymbol{c}$ to denote whether the block also hides an s-bit constant $\boldsymbol{c}$. A *programmed PPB* (Fig. 2(b)) additionally contains the specific functionality f_c in the lower right corner ($f_c \in \mathcal{F}_c$) and the specific value $\boldsymbol{\kappa}$ of the constant $\boldsymbol{c}$ in the upper right corner ($\boldsymbol{c} = \boldsymbol{\kappa}$). If the PPB does not hide a constant $\boldsymbol{c}$, we write f respectively $f \in \mathcal{F}$ and leave the upper right corner empty. For PPBs that compute arithmetic expressions, the input is grouped into inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ and the output is called $\boldsymbol{z}$.



(a) not programmed PPB (b) programmed PPB

Fig. 2. Uniform graphical notation for PPBs

5 Practical Efficient PPB Constructions

In this section we show how to construct several efficient PPBs that are useful in practical applications (cf. §7). All these building blocks are implemented in our framework for practical SPF-SFE described in §6. We concentrate on PPBs for *arithmetic operations* here, i.e., addition or subtraction §5.1 (with private constant §5.2), comparison §5.3 (with private constant §5.4), multiplication with private constant §5.5. PPBs for *boolean operations* on bitvectors §B.1 (with private constant §B.2) are given in §B. Our SPF-SFE framework also provides PPBs for *Switching Functions* (i.e., permutation and selection blocks) and *Universal Circuits* for which we refer to the definitions, descriptions, and constructions in [KS08b]. A list of efficient PPB constructions implemented in our framework is given in §A.

The main idea underlying the efficient PPB constructions presented in this paper is to combine functionalities that have structurally equivalent recursive definitions that directly translate into programmable gates of equivalent topologies. For instance, to compare whether two m-bit numbers $\boldsymbol{x}$ and $\boldsymbol{y}$ of bitlength m are greater or equal is defined recursively as

$$(x \geq y) \Leftrightarrow \Big((x_m > y_m) \vee \big((x_m = y_m) \wedge ((x_{m-1}, .., x_1) \geq (y_{m-1}, .., y_1))\big)\Big).$$

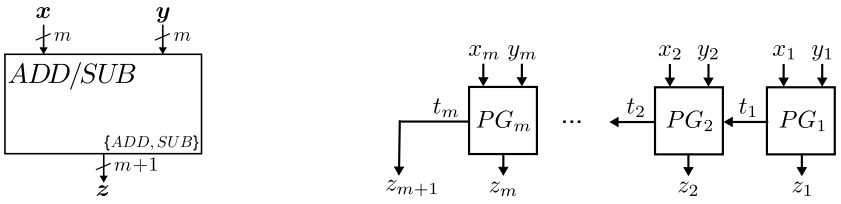
Whether two numbers are less or equal is defined recursively as

$$(x \leq y) \Leftrightarrow \Big((x_m < y_m) \vee \big((x_m = y_m) \wedge ((x_{m-1}, .., x_1) \leq (y_{m-1}, .., y_1))\big)\Big)$$

which is structurally equivalent and hence translates into the same topology (cf. Fig. 5(b)).

For each PPB we give the *Interface* specifying the functionality of the block, its number of outputs and the different possibilities for programming. The *Implementation* describes the topology of the corresponding efficient PPB construction, how to program it, and its size. The inputs are called $\boldsymbol{x}$, $\boldsymbol{y}$ and the potential private constant $\boldsymbol{c}$, where $|\boldsymbol{x}| = m$, $|\boldsymbol{y}| = n$, and $|\boldsymbol{c}| = s$. To simplify presentation we assume w.l.o.g. $m = n$, respectively $m = s$ in the following descriptions. The other cases can easily be derived from these by padding the shorter input with zeros and optimizing constant inputs afterwards as described in §8. Recall, that evaluator Alice can neither extract the chosen function $f_{(c)} \in \mathcal{F}_{(c)}$, nor the value of the possibly embedded private constant $c \in \{0,1\}^s$, from the topology of any PPB. Recall, the amount of information hidden inside the PPB is given by equation (1).

5.1 PPB:ADD/SUB - add or subtract two numbers



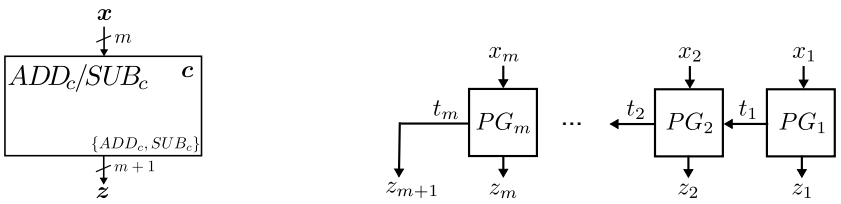
(a) Interface (b) Topology

Fig. 3. PPB:ADD/SUB

Interface (Fig. 3(a)). $PPB_{ADD/SUB}$ implements $z = f(x, y) = x \pm y$, where $|\boldsymbol{z}| = m + 1$. The class of functions is $\mathcal{F} = \{ADD, SUB\}$.

Implementation (Fig. 3(b)). The topology of $PPB_{ADD/SUB}$ is the well known structure for adders consisting of a chain of m programmable gates PG_i (full adders) with inputs carry-in t_{i-1}, x_i, y_i and outputs carry-out t_i, sum z_i. The constant t_0 can be directly incorporated into block PG_1. In case $f = ADD$, PG_i is programmed to compute $z_i = x_i \oplus y_i \oplus t_{i-1}$ and $t_i = (x_i \wedge y_i) \vee (x_i \wedge t_{i-1}) \vee (y_i \wedge t_{i-1})$ with $t_0 = 0$. In case $f = SUB$, y is converted to two's complement - PG_i is programmed with $z_i = x_i \oplus \overline{y_i} \oplus t_{i-1}$ and $t_i = (x_i \wedge \overline{y_i}) \vee (x_i \wedge t_{i-1}) \vee (\overline{y_i} \wedge t_{i-1})$ with $t_0 = 1$. The size is $\left|PPB_{ADD/SUB}\right| = 2 \cdot ((m-1) \cdot 2^3 + 2^2) = 16m - 8$.

5.2 PPB:ADDc/SUBc - add or subtract number with private constant



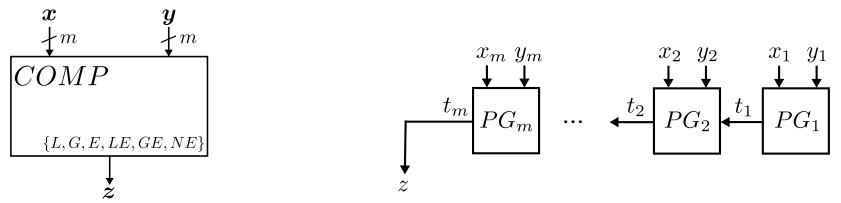
(a) Interface (b) Topology

Fig. 4. PPB:ADDc/SUBc

Interface (Fig. 4(a)). $PPB_{ADDc/SUBc}$ implements $z = f_c(x) = x \pm c$, where $\boldsymbol{c}$ is a private constant hidden inside PPB and $|\boldsymbol{z}| = m + 1$. Class of functions is $\mathcal{F}_c = \{ADDc, SUBc\}$.

Implementation (Fig. 4(b)). Topology of $PPB_{ADDc/SUBc}$ is exactly the same as that of $PPB_{ADD/SUB}$ described in the previous section, however, each programmable gate PG_i has no input for y_i which is replaced by the private constant c_i. Also the programming is exactly the same as for $PPB_{ADD/SUB}$ with private constant c_i instead of input y_i. This block has size $\left|PPB_{ADDc/SUBc}\right| = 2 \cdot ((m-1) \cdot 2^2 + 2^1) = 8m - 4$.

5.3 PPB:COMP - compare two numbers



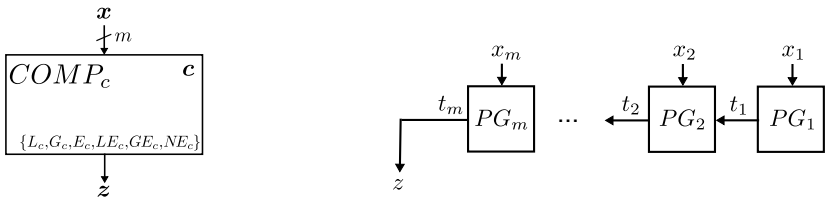
(a) Interface (b) Topology

Fig. 5. PPB:COMP

Interface (Fig. 5(a)). PPB_{COMP} implements $z = f(x, y) = x \bowtie y$, where $\bowtie \in \{<, >, =, \leq, \geq, \neq\}$ and $|\boldsymbol{z}| = 1$. The corresponding class of functions is $\mathcal{F} = \{L, G, E, LE, GE, NE\}$.

Implementation (Fig. 5(b)). Topology of PPB_{COMP} consists of a chain of m programmable gates PG_i (full comparers) with input bits x_i, y_i, and carry-in t_{i-1} and output carry-out t_i. The output of PPB_{COMP} is $z = t_m$ and the first carry $t_0 = 1$ can be directly incorporated into PG_1. The carry t_i propagates whether for the i least significant bits $x_{<i} = x \ mod \ 2^i$ and $y_{<i} = y \ mod \ 2^i$ the corresponding relation is fulfilled ($t_i = 1$) or not ($t_i = 0$). In the following we describe the programming for the cases $=$, $\leq$, and $\geq$; the corresponding cases $\neq$, $>$, and $<$ can be easily derived from this by negating output t_m in PG_m. In case $f = E$, PG_i is programmed to compute $t_i = (x_i = y_i) \wedge (x_{<i} = y_{<i}) = (x_i = y_i) \wedge t_{i-1}$. Analogously, in case $f = LE$, PG_i computes $t_i = (x_i < y_i) \vee [(x_i = y_i) \wedge t_{i-1}]$ and in case $f = GE$, PG_i computes $t_i = (x_i > y_i) \vee [(x_i = y_i) \wedge t_{i-1}]$. This block has size $|PPB_{COMP}| = (m-1) \cdot 2^3 + 2^2 = 8m - 4$.

5.4 PPB:COMPc - compare number with private constant



(a) Interface (b) Topology

Fig. 6. PPB:COMPc

Interface (Fig. 6(a)). PPB_{COMPc} implements $z = f_c(x) = x \bowtie c$, where $\bowtie \in \{<, >, =, \leq, \geq, \neq\}$, $\boldsymbol{c}$ is a private constant hidden inside PPB, and $|\boldsymbol{z}| = 1$. The corresponding class of functions is $\mathcal{F}_c = \{L_c, G_c, E_c, LE_c, GE_c, NE_c\}$.

Implementation (Fig. 6(b)). Topology of PPB_{COMPc} is exactly the same as that of PPB_{COMP} described in the previous section, however, each programmable gate PG_i has no input for y_i which is replaced by the internal constant c_i. Also the programming is exactly the same as for PPB_{COMP} with constant c_i instead of input y_i. This block has size $|PPB_{COMPc}| = (m-1) \cdot 2^2 + 2^1 = 4m - 2$.

5.5 PPB:MULc - multiply number with private constant

Interface (Fig. 7(a)). PPB_{MULc} multiplies input $\boldsymbol{x}$ with private constant $\boldsymbol{c}$ hidden inside PPB, i.e., $z = f_c(x) = x \cdot c$, where $|\boldsymbol{z}| = s + m$. The class of functions is $\mathcal{F}_c = \{MULc\}$.

Implementation (Fig. 7(c)). PPB_{MULc} is implemented according to the "school method" for multiplication, i.e., adding up the bitwise multiplications of c_i and x shifted corresponding to the position: $\boldsymbol{x} \cdot \boldsymbol{c} = \sum_{i=1}^{s} 2^{i-1}(c_i \cdot \boldsymbol{x})$. This results in the topology shown in Fig. 7(c), a matrix of s rows and m columns of programmable gates $PG_{i,j}$ (Fig. 7(b)), where the carry inputs in the first row and last column are set to zero and built into the corresponding outer gates $PG_{i,j}$: $t_{0,j} = d_{i,0} = 0$. The programmable gates in each row i have exactly the same

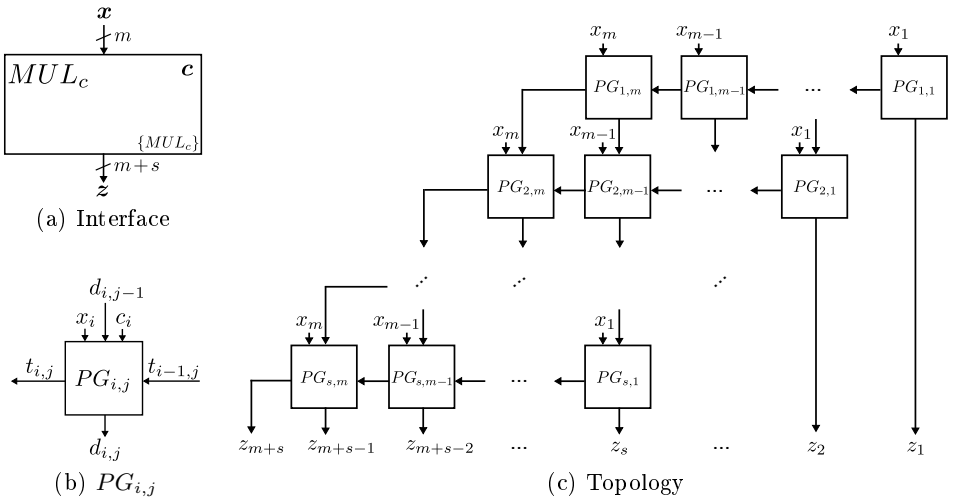


Fig. 7. PPB:MULc

topology as an adder (cf. Fig. 3(b)) that can be programmed to add to the shifted sum of the previous rows $d_{i-1} = \sum_{i=1}^{i-1} 2^{i-1}(c_i \cdot x)$ either the value of x in case $c_i = 1$ or zero in case $c_i = 0$, i.e., $d_i = 2d_{i-1} + c_i \cdot x$. Each programmable gate $PG_{i,j}$ is programmed to compute $t_{i,j} = (c_i x_i \wedge d_{i,j-1}) \vee (c_i x_i \wedge t_{i-1,j}) \vee (d_{i,j-1} \wedge t_{i-1,j})$ and $d_{i,j} = c_i x_i \oplus d_{i,j-1} \oplus t_{i-1,j}$. This block has size $|PPB_{MULc}| = 2 \cdot (2^3(m-1)(s-1) + 2^2(m+s-2) + 2^1) = 16ms - 8(m+s) + 4$.

6 FairplaySPF - a General Framework for SPF-SFE

We have implemented a general framework for secure evaluation of semi-private functions (SPF-SFE) called *FairplaySPF* by extending Fairplay SFE framework [MNPS04], both written in JAVA. [1] Fairplay provides two languages: The high-level Secure Function Description Language (SFDL) allows users to specify the functionality to be computed with elements known from other high-level hardware description languages like VHDL or Verilog (e.g., variables, arrays, procedures, arithmetic- and logic expressions, control structures, etc.). Fairplay optimizes the function described in SFDL and automatically transforms it into a boolean circuit described in Fairplay's low-level Secure Hardware Description Language (SHDL). This language consists of wires, input wires, gates, and output gates only. Using the SHDL circuit as input for both parties, Alice and Bob invoke their respective programs of the Fairplay runtime environment to execute the two-party SFE protocol. These programs evaluate the function on their respective private inputs over a TCP connection.

FairplaySPF Framework. In FairplaySPF, we extend the Fairplay framework [MNPS04] to secure evaluation of semi-private functions that are known to Bob only. The workflow of FairplaySPF framework described in the following is visualized in Fig. 8.

Bob composes his semi-private function from several available privately programmable blocks (as described in §5) in our newly designed *Secure Programmable Block Description Language (SPBDL)* explained later in this section. Our FairplaySPF compiler automatically translates this SPBDL program into an SHDL circuit. Alternatively, SHDL circuits

[1] The FairplaySPF framework will be available for download soon.

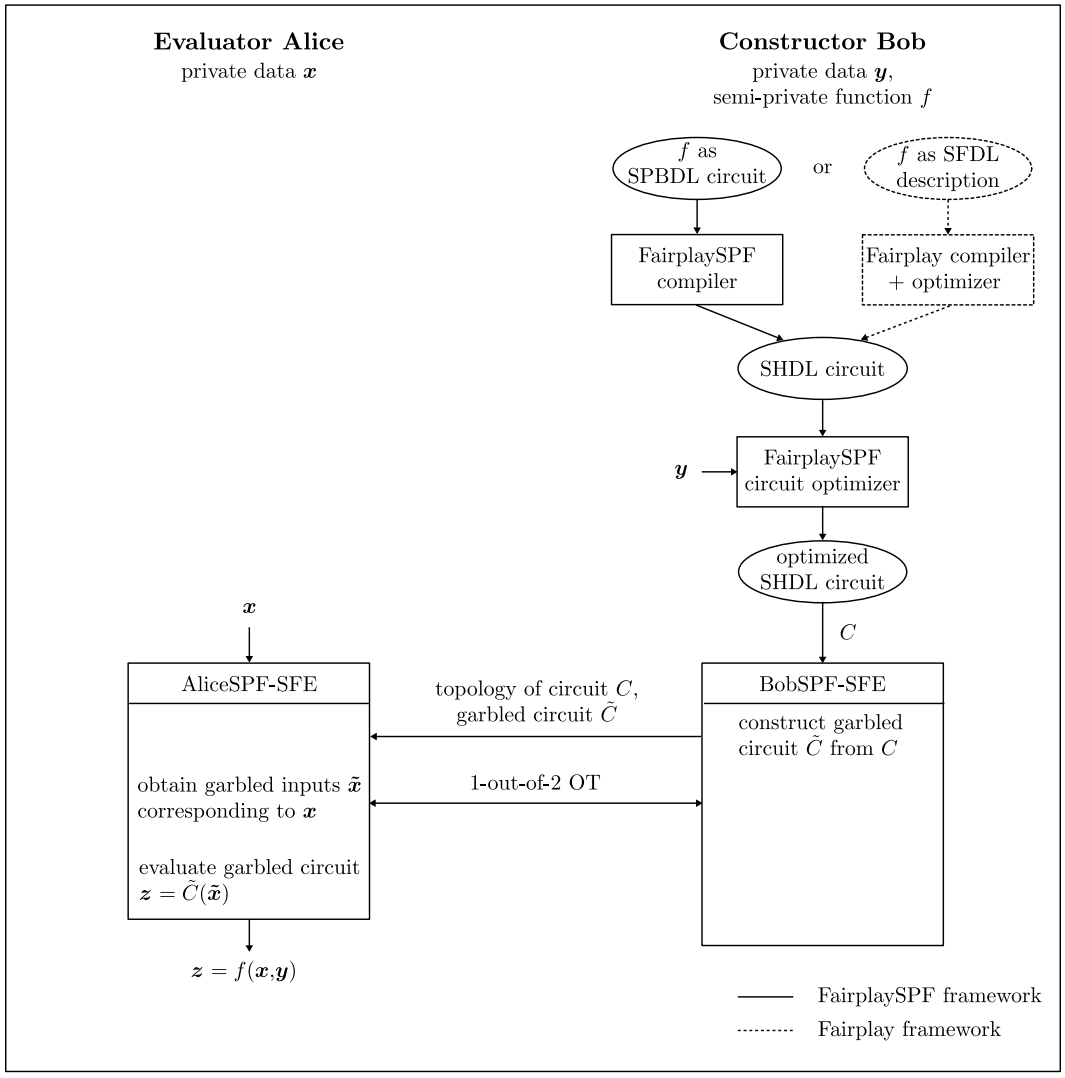


Fig. 8. Architecture of FairplaySPF Framework

that are generated by the original Fairplay compiler from SFDL descriptions can be used. Bob's private input data is automatically incorporated into the SHDL circuit and optimized afterwards by FairplaySPF circuit optimizer as described in §8 resulting in an optimized SHDL circuit. This optimized SHDL circuit (containing the combination of Bob's semi-private function and his private data) is evaluated by FairplaySPF runtime environment (RE) which is only a slight modification of Fairplay RE for semi-private functions: In FairplaySPF RE *only Bob* inputs the SHDL circuit but not Alice. The topology of the circuit (but without the types of the gates) is sent to Alice and afterwards the SPF-SFE protocol (as described in §2) is executed between Alice and Bob over a TCP connection.

Secure Programmable Block Description Language (SPBDL). Our new SPBDL language allows to easily specify semi-private functions by combining different PPBs. SPBDL extends the basic functionality of SHDL to input wires (`input`), multi-wires (`vector`), privately programmable blocks (`block`), programmable gates (`gate`), and output wires (`output`). The formal *syntax* specification of SPBDL in Extended Backus-Naur Form (EBNF) is given in §C. In the following, we briefly describe the *semantics* of SPBDL. Please see Fig. 9 for an example SPBDL description of a semi-private function. As in SHDL, each line of a SPBDL program starts with a line number beginning with 0. In following lines, this number refers to the output of the element defined in this line. Line comments start with `//`.

In the beginning of a SPBDL program, inputs are defined as `input Player [w]`, where `Player` defines from which party the input is given (`alice` or `bob`). The optional parameter `[w]` specifies that the input consists of w bits (default is $w = 1$ if omitted).

Afterwards, three kinds of elements can be specified - `gate`, `vector`, and `block`:
A programmable gate is defined as `gate in [ Wires ] p [ Bits ]`, where `Wires` is its (space-separated) list of inputs and `Bits` is the programming of its function table.
A list of `Wires` can be grouped into a vector with `vector [ Wires ]`. The single wires of a vector can be accessed via `Vector.Index`, e.g., `4.2` denotes the second wire of vector 4.
A PPB is defined as `block [Btype] out v in [ Vects ] p [ Bprog ]`, where `Btype` is the type of the PPB (e.g., `addsub` for $PPB_{ADD/SUB}$ described in §5.1), `v` specifies that its output is a vector of v bits, and `Vects` is the list of input vectors. The programming of the PPB specified in `Bprog` depends on the type of the PPB `Btype`. All types of PPBs `Btype` and corresponding programming parameters `Bprog` available in SPBDL are given in §C.

Finally, outputs are defined as `output Player Vect`, where `Player` defines which party obtains the output (`alice` or `bob`) and `Vect` is the vector to be output.

7 Applications

Many applications can be reduced to Secure Evaluation of Semi-Private Functions (SPF-SFE) for which our general framework presented in this paper can be used. Examples are *Blinded Policy Evaluation* [FAL04,FLA06,FAL06], *Privacy-Preserving Credit Checking* [FAZ05], or provably secure evaluation of *private Neural Networks* [SS08,OPB07,PCB+08].

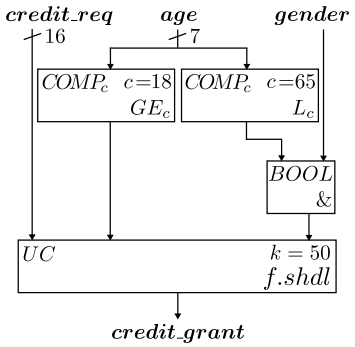
In the following we concentrate on privacy-preserving credit checking [FAZ05] which demonstrates how the evaluated function can be partitioned into semi-private and private parts which are both supported by our framework.

Privacy-Preserving Credit Checking. Typically, before granting a loan from a lender (Bob), the credit worthiness of the borrower (Alice) is checked to have the confidence that she will be able to pay it back later. The borrower is asked for her credit report that contains a large amount of private information including for example gender, age, income, salary, or

other sensitive information like how many trade lines she owns, the number of overdrafts, or the number of late payments. However, revealing this data should be avoided as the lender may not always be a credible organization or, even worse, dishonest employees (so called insiders) could sell such private information on customers to third parties.

Additionally, the evaluation criteria of the lender are highly sensitive information that must be protected as revelation of these may cause loss of intellectually property or loss of repudiation for the lender.

As suggested by Frikken et al. [FAZ05], this scenario can be reduced to SPF-SFE, where Alice inputs her private credit report and Bob evaluates his semi-private function that checks if the credit report fulfills his criteria. To ensure that Alice inputs correct data into the SPF-SFE protocol, the authors describe how to replace the oblivious transfer step by a Credit Report Agency, i.e., a trusted third party, that checks and accredits Alice's inputs instead.




```
0 input alice [7]  // age
1 input alice      // gender
2 input alice [16] // credit_req
3 block [compc] out 1 in [0] p [GE 18]
4 block [compc] out 1 in [0] p [L 65]
5 block [bool] out 1 in [4 1] p [AND]
6 vector [2 3 5]
7 block [uc] out 1 in [6] p [50 f.shdl]
8 output alice 7 // credit_grant
9 output bob   7 // credit_grant
```

Fig. 9. Example for Privacy-Preserving Credit Checking

Bob's semi-private credit checking function can be expressed in our framework for SPF-SFE as shown in the tiny example of Fig. 9 which is due to space limitations not intended to give the complete solution but merely to show the main concepts. The upper part of the circuit performs some obvious computation on Alice's data, e.g., compare her *age* with a private constant, or combine this result with her *gender*. The sensitive information in this part of the function are the private constants, e.g., grant credit only to female persons ($gender = 1$) that are younger than 65 ($age < 65$), which are hidden from Alice, whereas the obvious topology can safely be revealed.

The highly sensitive part of the functionality that combines these results depending on the amount of credit requested ($credit\_req$) is hidden entirely from Alice within the universal circuit UC. This PPB can be programmed to compute *any* functionality computable by a circuit of up to $k = 50$ gates with arbitrary topology. The specific functionality intended by Bob is the SHDL circuit described in `f.shdl`, which can automatically be generated from a high-level description in SFDL with Fairplay compiler.

This example shows how our framework for SPF-SFE can be used to implement an application-specific, reasonable tradeoff between efficiency while revealing irrelevant information (SPF-SFE with PPBs) and complete function privacy (PF-SFE with UC).

Comparison of SPF-SFE and PF-SFE. Revealing the topology of obvious parts of the functionality while hiding the sensitive parts in a UC results in a smaller circuit as UC overhead can be substantially reduced due to less simulated gates k and less inputs into UC.

This reduced size of the evaluated circuit directly translates into corresponding speedups in *any implementation* of the underlying SPF-SFE protocol.

A) Gates hidden in UC, k	25	50	100	200
B) Gates extracted, $14/(k+14)$	35.9%	21.9%	12.3%	6.5%
C) UC overhead in PF-SFE (UC type)	1,861 (M3)	3,720 (M3)	8,264 (M3)	19,419 (M3)
D) UC overhead in SPF-SFE (UC type)	850 (M1)	2,571 (M3)	6,797 (M3)	17,542 (M3)
E) Improvement SPF-SFE vs. PF-SFE	1,011 (54.3%)	1,149 (30.9%)	1,467 (17.8%)	1,877 (9.7%)

Table 1. Improved UC Overhead in the Example of Fig. 9

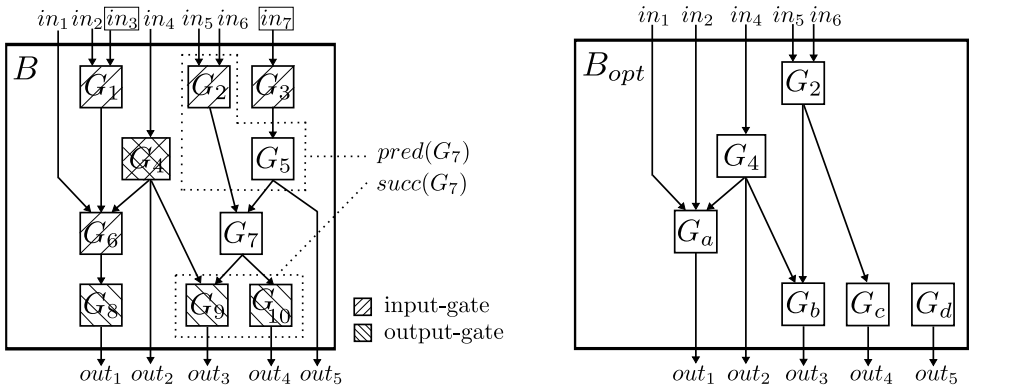
As concrete example, Table 1 shows the number of gates that can be saved in the privacy-preserving credit checking example of Fig. 9 compared to hiding the functionality entirely in a UC in PF-SFE. For different maximum size k (row A) of the part of the functionality which is hidden in UC we give the achieved performance improvements when extracting the obvious part of the functionality into the upper part of the circuit ($COMP_c$ blocks and $BOOL$ block in Fig. 9). In our example, these blocks consist of 14 gates, i.e., row B contains the fraction of the functionality which is revealed: $14/(k+14)$. Row C shows how many gates are needed to hide the whole functionality of $14+k$ gates in a UC with 24 inputs (for *credit_req*, *age*, and *gender*) using the most efficient UC construction of [SS08] which is denoted in parentheses. Row D shows how many gates are needed to implement the UC in our mixed approach as shown in Fig. 9, where UC has 18 inputs and simulates k gates. The resulting improvements compared to the PF-SFE solution (row E) supersedes the fraction of the gates extracted (row B) as the number of inputs into UC is also reduced.

8 Optimization of Circuits with Constant Inputs

We describe a general optimization algorithm that incorporates constant inputs into a block (sub-circuit) B. The topology of the resulting optimized block B_{opt} is *independent* of the values of the constant inputs and its number of inputs and size are smaller, i.e., the number of gates respectively their degree is reduced as shown in Fig. 10. Besides the well known propagation of constant inputs (step 1), our algorithm additionally eliminates resulting gates with one input by incorporating them into surrounding gates (steps 2 and 3) which results in smaller circuit size. The optimization algorithm is a non-cryptographic transformation of circuits and hence of independent interest. As outlined in §2, one possible application is to use this optimization to improve Yao's protocol. In this application, constant inputs might be public constant values known to both parties as well as the private inputs of (semi-honest) circuit constructor Bob (if known at the time of construction of the garbled circuit).

Terminology. The following terminology is visualized in Fig. 10(a). Assume the gates G_i, $i = 1, .., n$ of a block B are numbered in topological order, i.e., gate G_i has no inputs that are outputs of gates with larger index $G_{j>i}$. Otherwise, this order can be obtained efficiently via topological sorting in $O(n)$.

An *output gate* is a gate whose output is also an output of B. Similarly, an *input gate* is a gate, which has at least one input that is also an input of B. For gate G_i, $pred(G_i)$ denotes the set of its predecessors, i.e., gates whose output is an input into G_i. Analogously, $succ(G_i)$ denotes the set of G_i's successors, i.e., gates having the output of G_i as input. The fan-out of a gate G_i is the number of its successors, i.e., $fanout(G_i) = \#succ(G_i)$.



(a) block with constant inputs in_3 and in_7 (b) optimized block produced by Algorithm 1

Fig. 10. Example for circuit optimization

Optimization. We refer to the running example of Fig. 10 that optimizes a block B with constant inputs in_3 and in_7 in the following description of Algorithm 1.

Algorithm 1: Optimize block B with constant inputs

Input: Block B of gates $G_1, .., G_n$ in topological order
Output: Optimized block B_{opt}
begin

```
1   # Eliminate constant inputs
    forall constant inputs c_j with constant value v_j that are not outputs of B do
        forall gates G_i having c_j as k_i-th input do
            eliminateConstInput(G_i, k_i, v_j)
2   # Eliminate non-output gates with one input
    forall non-output gates G_i with d_i = 1 do
        integrateInSucc(G_i)
3   # Eliminate output gates with one input
    forall output gates G_i with d_i = 1 do
        let {G_p} = pred(G_i)
        if G_i is not input gate and fanout(G_p) = 1 then
            integrateInPred(G_i,G_p)
```

end

Step 1 - Eliminate constant inputs. The first step of Algorithm 1 eliminates all constant inputs c_j, $j = 1, .., c$ of block B with respective constant value $v_j \in \{0, 1\}$. For all gates G_i with degree d_i having c_j as k_i-th input, **eliminateConstInput**$(G_i,\ k_i,\ v_j)$ is called that eliminates the corresponding input of G_i. Only the lines of the function table of G_i with value v_j in the k_i-th position are used while the other entries are eliminated, i.e., the modified gate G_i' computes $g_i'(in_1, .., in_{k_i-1}, in_{k_i+1}, .., in_{d_i}) = g_i(in_1, .., in_{k_i-1}, v_j, in_{k_i+1}, .., in_{d_i})$. $|G_i|$ shrinks by a factor of two for each of its constant inputs. Let $\#c_i$ denote the number of constants of the d_i inputs of G_i, then $|G_i'| = 2^{d_i - \#c_i}$ after Step 1 of Algorithm 1 has eliminated all constant inputs. To obtain an efficient implementation of Algorithm 1 it is

crucial that **eliminateConstInput()** does not copy the entire function table of a gate G_i for each elimination of a constant input as this would result in runtime $O(\#c_i \cdot |G_i|)$ for each gate. Instead, the constant inputs are marked first in runtime $O(\#c_i)$ and afterwards all constant inputs are eliminated simultaneously in runtime $O(|G_i|)$ by copying the corresponding elements of the function table. This results in runtime $O(|G_i|)$ per gate.

Possibly resulting constant gates G'_i with $d'_i = 0$ are propagated into its successors by recursively calling **eliminateConstInput(**G_s**,** k_s**,** $g_i(v_j)$**)** for all $G_s \in succ(G'_i)$ having G'_i as k_s-th input. If constant gate G'_i is not an output gate it is eliminated afterwards.

In the running example of Fig. 10, constant input in_3 is input into gate G_1 whose size is reduced by half when eliminating the second input ($k_1 = 2$). The resulting gate G'_1 has one non-constant input in_2 and hence no further optimization is performed. The other constant input in_7 is input into G_3 which is optimized into a constant gate G'_3 by eliminating the constant input. Hence, **eliminateConstInput()** is called recursively for successor G_5 and G'_3 is eliminated. Similarly to G_3, gate G_5 is also reduced to a constant gate G'_5 and **eliminateConstInput()** is called for successor G_7 which eliminates its second input. As the output of G'_5 is also output of B it is not eliminated and remains as constant gate G_d.

After the optimizations in Step 1, there might be gates G_i with only one input. The next two steps of Algorithm 1 try to remove these gates by replacing them with wires and incorporating their functionalities into their successors (Step 2) or predecessors (Step 3).

Step 2 - Eliminate non-output gates with one input. The second step of Algorithm 1 eliminates non-output gates with $d = 1$. The functionality of each one-input gate G_i which is not an output gate is incorporated into its successors $G_s \in succ(G_i)$ by the function **integrateInSucc(**G_i**)**. This function eliminates G_i by replacing it with a wire and incorporating the functionality of g_i into the function tables of all its successors $G_s \in succ(G_i)$: Let the output of G_i be the k-th input of G_s and d the degree of G_s. Then, the modified gate G'_s computes $g'_s(in_1, .., in_k, .., in_d) = g_s(in_1, .., g_i(in_k), .., in_d)$. Note that, independent of the functionality g_i, the resulting gate G'_s has the same size as G_s but additionally incorporates the functionality of g_i while not revealing any additional information on it. As in Step 1, the function tables of gates are not directly modified but first all needed modifications are marked and then done simultaneously to get runtime in $O(|G_i|)$ per gate.

In the running example of Fig. 10, Step 2 eliminates G_1 by replacing it with a wire and modifying the function table of G_6 correspondingly. Analogously, gate G'_7 which only has one input from G_2 left after the optimizations performed in Step 1 is replaced by a wire. The function tables of its successors $G_9 \rightarrow G_b$ and $G_{10} \rightarrow G_c$ are modified correspondingly.

Step 3 - Eliminate output gates with one input. The third step of Algorithm 1 tries to eliminate output gates with $d = 1$. The functionality of each output gate G_i with one input is incorporated into its predecessor G_p. This is only possible if G_i is the only successor of G_p, i.e., $fanout(G_p) = 1$. In this case, function **integrateInPred(**G_i**,**G_p**)** is called which eliminates gate G_i by replacing it with a wire and incorporates its functionality into gate G_p with d inputs. The modified gate G'_p computes $g'_p(in_1, .., in_d) = g_i(g_p(in_1, .., in_d))$. As in Step 2, this optimization step is independent of the functionality g_i and the resulting gate G'_p has the same size as G_p but additionally incorporates the functionality of g_i while not revealing any additional information on it.

In the running example of Fig. 10, Step 3 eliminates G_8 by replacing it with a wire and modifying the function table of $G_6 \rightarrow G_a$ correspondingly. In contrast to this, gate G_c cannot be incorporated into its predecessor G_2 as G_c is not its only successor ($fanout(G_2) = 2$). The optimized block B_{opt} produced by Algorithm 1 is shown in Fig. 10(b). It has size $|B_{opt}| = 21$ which is less than 62% of the size of the original block $|B| = 34$.

Correctness, efficiency and security of Algorithm 1 are summarized in the following theorem.

Theorem 1. *Algorithm 1 efficiently eliminates all $c > 0$ constant inputs that are not outputs of block B in runtime $O(|B|)$. The optimized block B_{opt} has smaller size and computes the same functionality as B. The topology of B_{opt} does not reveal anything about the values of the constant inputs.*

Proof. Let n denote the number of gates of B in the following.

1. *Termination*: Algorithm 1 always terminates as all loops are upper bounded and the recursive call of **eliminateConstInput()** in Step 1 terminates if G_i has no successors.
2. *Efficiency*: Step 1 of Algorithm 1 first marks all gates with constant inputs in runtime $O(c \cdot n) \subseteq O(|B|)$. Afterwards, the marked constant inputs are eliminated in $O(|B|)$. Step 2 also needs at most $O(|B|)$ operations for elimination of gate G_i, marking and incorporating the functionality of G_i into the succeeding gates and analogously Step 3 runs in $O(|B|)$ as well. Hence, the overall runtime of Algorithm 1 is in $O(|B|)$.
3. *All constant inputs that are not outputs are eliminated*: Step 1 of Algorithm 1 eliminates all constant inputs that are not outputs of B by incorporating them into the input gates G_i in **eliminateConstInput()**.
4. *Size is reduced*: As $c > 0$ there is at least one constant input which is not output of B and therefore must be input of at least one gate G (otherwise the circuit would not be connected). The size of G is reduced by **eliminateConstInput()** in Step 1. As all other optimizations never increase the size of the block its size is strictly reduced: $|B_{opt}| \lneqq |B|$.
5. *Functional equivalence*: None of the optimizations performed in Algorithm 1 changes the functionality of B as they incorporate the values of constant inputs that are not outputs (Step 1) respectively gates with one input (Step 2 and Step 3) into the functionality of surrounding gates. The functionality b_{opt} computed by optimized block B_{opt} with u' non-constant inputs is identical to the functionality b' computed by original block B with constant input values set: $\forall(in'_1, .., in'_{u'}) \in \{0,1\}^{u'} : b_{opt}(in'_1, .., in'_{u'}) = b'(in'_1, .., in'_{u'})$.
6. *Topology does not reveal values of constant inputs*: All optimizations performed in Algorithm 1 change the topology (i.e., remove gates or reduce the size of function tables by reducing the number of inputs) *independently* of the values v_j of the constant inputs. Only the contents of the modified function tables depend on the values v_j. Hence, the resulting topology of the optimized block B_{opt} does not depend on the values v_j and therefore B_{opt} does not reveal anything about v_j.

This concludes the proof of Theorem 1. □

9 Conclusion

Exploiting Yao's protocol to additionally hide parts of the evaluated function f as described in this paper is as efficient as Yao's plain SFE protocol. Hence, SPF-SFE is much more efficient than PF-SFE whose practicability is restricted (due to large overhead caused by UC). In many practical applications full hiding of f is not necessary at all as shown in §7: Obvious parts of the topology can safely be revealed while the sensitive parts are still hidden in a (much smaller) UC.

Our FairplaySPF compiler extends the well-known Fairplay SFE framework [MNPS04] with capabilities to easily describe semi-private functions with our new Secure Programmable Block Description Language SPDL, compile and optimize them into circuits which can efficiently be evaluated with an SPF-SFE protocol based on Yao's protocol.

In principle, the ideas of SPF-SFE using PPBs can be extended to malicious, respectively covert constructor Bob as well by using SFE protocols that apply cut-and-choose technique [MNPS04,LP07,LPS08], respectively [AL07,GMS08] instead. The opening phase reveals the gate types and hence Bob's inputs can not be incorporated into the circuit and PPBs require additional inputs by Bob to select the intended functionality which results in larger circuits.

Acknowledgements. We would like to thank Vladimir Kolesnikov and anonymous reviewers of ACNS'09 for helpful comments on the paper.

References

AHL05. L. v. Ahn, N. J. Hopper, and J. Langford. Covert two-party computation. In *37th ACM Symposium on Theory of Computing (STOC '05)*, pages 513–522. ACM, 2005.

AIR01. W. Aiello, Y. Ishai, and O. Reingold. Priced oblivious transfer: How to sell digital goods. In *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 119–135. Springer, 2001.

AL07. Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *Theory of Cryptography, TCC 2007*, volume 4392 of *LNCS*, pages 137–156. Springer, 2007.

FAL04. K. B. Frikken, M. J. Atallah, and J. Li. Hidden access control policies with hidden credentials. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society (WPES '04)*, pages 27–27. ACM, 2004.

FAL06. K. B. Frikken, M. J. Atallah, and J. Li. Attribute-based access control with hidden policies and hidden credentials. *IEEE Trans. Comput.*, 55(10):1259–1270, 2006.

FAZ05. K. B. Frikken, M. J. Atallah, and C. 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.

FLA06. K. B. Frikken, J. Li, and M. J. Atallah. Trust negotiation with hidden credentials, hidden policies, and policy cycles. In *Network and Distributed System Security Symposium (NDSS 06)*, 2006.

GMS08. V. Goyal, P. Mohassel, and A. Smith. Efficient two party and multi party computation against covert adversaries. In *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 289–306. Springer, 2008.

Kol05. V. Kolesnikov. Gate evaluation secret sharing and secure one-round two-party computation. In *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 136–155. Springer, 2005.

KS08a. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP '08 - 35th Int. Colloquium on Automata, Languages and Programming*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.

KS08b. V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In *Financial Cryptography and Data Security, FC '08*, volume 5143 of *LNCS*, pages 83–97. Springer, 2008. `http://thomaschneider.de/FairplayPF`.

LL07. Sven Laur and Helger Lipmaa. A new protocol for conditional disclosure of secrets and its applications. In *Applied Cryptography and Network Security (ACNS '07)*, volume 4521 of *LNCS*, pages 207–225. Springer, 2007.

LP04. Y. Lindell and B. Pinkas. A proof of Yao's protocol for secure two-party computation. ECCC Report TR04-063, Electronic Colloquium on Computational Complexity (ECCC), 2004.

LP07. Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, 2007.

LPS08. Y. Lindell, B. Pinkas, and N. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *SCN 2008*, volume 5229 of *LNCS*, pages 2–20. Springer, 2008.

MNPS04. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *USENIX*, 2004. `http://www.cs.huji.ac.il/project/Fairplay/fairplay.html`.

NP99. M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *ACM Symposium on Theory of Computing (STOC '99)*, pages 245–254. ACM, 1999.

OPB07. C. Orlandi, A. Piva, and M. Barni. Oblivious neural network computing via homomorphic encryption. *EURASIP: European Journal of Information Systems*, 2007(1):1–10, 2007.

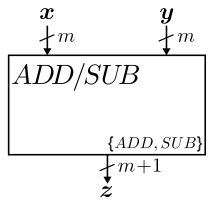
PCB+08. A. Piva, M. Caini, T. Bianchi, C. Orlandi, and M. Barni. Enhancing privacy in remote data classification. *SEC '08: New Approaches for Security, Privacy and Trust in Complex Environments*, 2008.

SS08. A.-R. Sadeghi and T. Schneider. Generalized universal circuits for secure evaluation of private functions with application to data classification. In *11th International Conference on Information Security and Cryptology (ICISC '08)*, 2008.

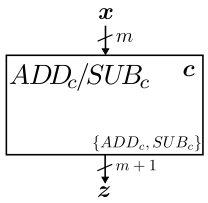
Val76. L. G. Valiant. Universal circuits (preliminary report). In *Proc. 8th ACM Symp. on Theory of Computing, STOC '76*, pages 196–203. ACM Press, 1976.

Yao86. A. C. Yao. How to generate and exchange secrets. In *Proc. 27th IEEE Symp. on Foundations of Comp. Science, FOCS '86*, pages 162–167, Toronto, 1986. IEEE.

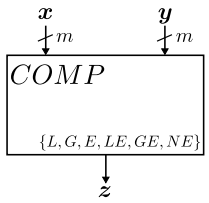
A Privately Programmable Blocks Provided by SPBDL



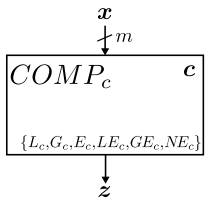
(a) addsub (5.1)



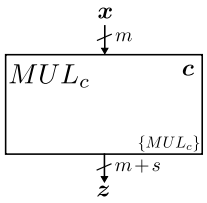
(b) addsubc (5.2)



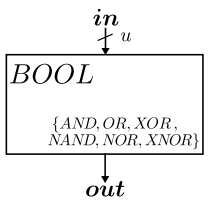
(c) comp (5.3)



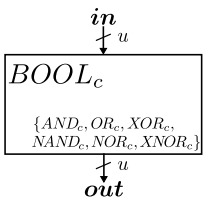
(d) compc (5.4)



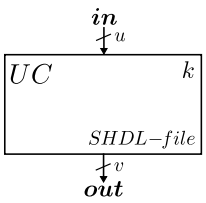
(e) mulc (5.5)



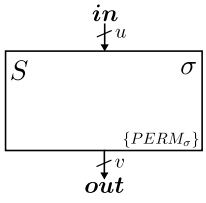
(f) bool (B.1)



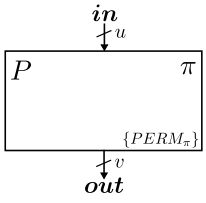
(g) boolc (B.2)



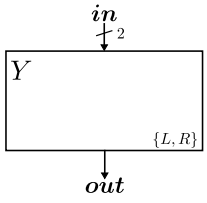
(h) uc [KS08b]



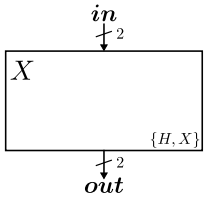
(i) sel [KS08b]



(j) perm [KS08b]



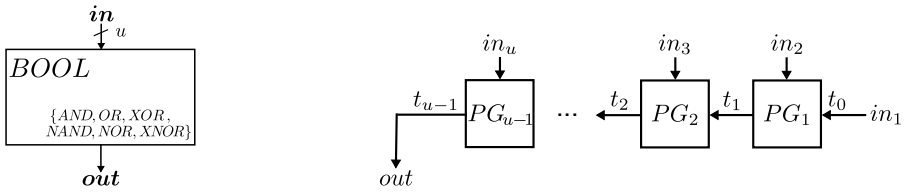
(k) yblock [KS08b]



(l) xblock [KS08b]

B Efficient Privately Programmable Blocks for Boolean Functions

B.1 PPB:BOOL - boolean combination of input bits



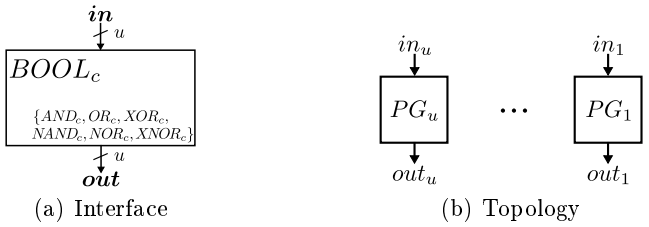
(m) Interface (n) Topology

Fig. 11. PPB:BOOL

Interface (Fig. 11(m)). PPB_{BOOL} combines u boolean inputs $in_1, .., in_u$ with function $out = f(in_1, .., in_u) = \bigodot_{i=1}^{u} in_i$, where $\bigodot \in \{\bigwedge, \bigvee, \bigoplus, \overline{\bigwedge}, \overline{\bigvee}, \overline{\bigoplus}\}$ and $|\boldsymbol{out}| = 1$. The corresponding class of functions is $\mathcal{F} = \{AND, OR, XOR, NAND, NOR, XNOR\}$.

Implementation (Fig. 11(n)). Topology of PPB_{BOOL} is a chain of $u-1$ programmable gates PG_i, $i = 1, .., u-1$ with input bits in_{i+1}, carry-in t_{i-1}, and output t_i. The output of PPB_{BOOL} is $out = t_{u-1}$ and the first carry-in is $t_0 = in_1$. The carry t_i contains the intermediate result from combining the lower $i+1$ input bits $in_{<i+1} = in \bmod 2^{i+1}$ with the corresponding operation. In the following we describe the programming for the cases AND, OR, and XOR; the corresponding cases $NAND$, NOR, and $XNOR$ can be easily derived from this by negating output t_{u-1} in PG_{u-1}. In case $f = AND$, PG_i is programmed to compute an AND gate $t_i = in_{i+1} \wedge t_{i-1}$. The cases $f = OR$ and $f = XOR$ are constructed analogously. This block has size $|PPB_{BOOL}| = (u-1) \cdot 2^2 = 4u - 4$.

B.2 PPB:BOOLc - boolean combination of input and private constant



(a) Interface (b) Topology

Fig. 12. PPB:BOOLc

Interface (Fig. 12(a)). PPB_{BOOLc} computes bitwise boolean combination of input $\boldsymbol{in}$ with private constant $\boldsymbol{c}$, i.e., $out = f_c(in) = in \odot c$, where $\odot \in \{\wedge, \vee, \oplus, \overline{\wedge}, \overline{\vee}, \overline{\oplus}\}$ and $|\boldsymbol{out}| = u$. The class of functions is $\mathcal{F}_c = \{AND_c, OR_c, XOR_c, NAND_c, NOR_c, XNOR_c\}$.

Implementation (Fig. 12(b)). Topology of PPB_{BOOLc} consists of u programmable gates PG_i in parallel with input bit in_i and output out_i that combine the corresponding input bit in_i and the private constant c_i. In case $f_c = AND_c$, PG_i is programmed to compute $out_i = in_i \wedge c_i$. The other cases are similar. This block has size $|PPB_{BOOLc}| = u \cdot 2^1 = 2u$.

C EBNF of Secure Programmable Block Definition Language (SPBDL)

```
Program ::= {Input} {Element} {Output};

Input   ::= Line 'input' Player [ '[' Positive ']' ];
Element ::= Gate | Block | Vector;
Gate    ::= Line 'gate' 'in' '[' {Wire} ']' 'p' '[' {Bit} ']';
Block   ::= Line 'block' '[' Btype ']' 'out' Positive 'in' '[' {Vect} ']'
              'p' '[' Bprog ']';
Vector  ::= Line 'vector' '[' {Wire} ']';
Output  ::= Line 'output' Player Vect;

Btype ::= 'addsub' | 'addsubc' | 'mulc' | 'comp' | 'compc' |
          'bool' | 'boolc' | 'yblock' | 'xblock' | 'sel' | 'perm' | 'uc';
Bprog ::= Paddsub | Paddubc | Pmulc | Pcomp | Pcompc |
          Pbool | Pboolc | Pyblock | Pxblock | Psel | Pperm | Puc;

Paddsub  ::= 'ADD' | 'SUB';
Paddsubc ::= Paddsub Const [ ConstLen ];
Pmulc    ::= Const [ ConstLen ];
Pcomp    ::= 'L' | 'G' | 'E' | 'LE' | 'GE' | 'NE';
Pcompc   ::= Pcomp Const [ ConstLen ];
Pbool    ::= 'AND' |'OR' | 'XOR' | 'NAND'| 'NOR'| 'XNOR';
Pboolc   ::= Pbool Const;
Pyblock  ::= 'L' | 'R';
Pxblock  ::= 'H' | 'X';
Psel     ::= Unsigned {Unsigned};
Pperm    ::= Unsigned {Unsigned};
Puc      ::= Unsigned <File>;

Line ::= Unsigned;
Const := Unsigned;
ConstLen ::= Positive;
Vect  ::= Line;
Wire ::= Line | Line '.' Unsigned;
Player ::= 'alice' | 'bob'

Bit      ::= '0'|'1';
Digit    ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
Unsigned ::= Digit {Digit};
Positive ::= ('1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9') {Digit};
```