A State-Separating Proof for Yao's Garbling Scheme

Chris Brzuska[1] and Sabine Oechsner[2⋆]

[1] Aalto University, Finland, chris.brzuska@aalto.fi
[2] University of Edinburgh, UK, s.oechsner@ed.ac.uk

Abstract. State-separating proofs (SSPs) are a recent proof and definition style for cryptographic security games in pseudo-code. SSPs allow to carry out computational security reductions for cryptography such that "irrelevant code" can be dealt with syntactically and does not require reasoning about execution semantics. Real-world protocols have notoriously long specifications, and the SSP style of breaking down security games and identifying subgames enables the analysis of such protocols. Indeed, SSPs have been used to analyze the key schedules of TLS (ePrint 2021/467) and MLS (S&P 2022).
Similarly, secure multi-party computation (MPC) protocols tend to have lengthy specifications. In this work, we explore how to use SSP techniques in the MPC context and for simulation-based security. On the example of Yao's circuit garbling scheme, we adapt the definitional style of SSPs and show that structuring the circuit and security proof in a *layered* way allows for a brief, compelling, syntactic construction of the reductions required in the hybrid proof of Yao's garbling scheme.

1 Introduction

Computational security of cryptographic constructions requires reduction arguments to underlying assumptions on computational hardness and thus are notoriously difficult to analyze. The approaches for tackling the complexity of such reductions can be roughly divided into three categories: (semi-)automated or interactive symbolic provers with computational guarantees, e.g. [Bla06],[BDJ+21]; proof assistants for computational security proofs, most prominently EasyCrypt [BGHZ11]; and finally techniques for handwritten proofs which are our focus.

For handwritten proofs of complex protocols, composability and abstraction are chief techniques for managing complexity. Notably, universal composability [Can01] and its variants, e.g. the IITM model [KTR20], GNUC [HS15], iUC [CKKR19]; as well as reactive systems [BPW04] and constructive cryptography [Mau10] have provided frameworks which help with composable *definitions* of security. Orthogonally, code-based game-playing by Bellare and Rogaway [BR06] allows to write security *proofs* as a sequence of small code-based game-hops, highlighting the small changes between subsequent games and verifying locally that these changes are admissible. In a more recent addition to techniques, Brzuska, Delignat-Lavaud, Fournet, Kohbrok and Kohlweiss (BDFKK [BDF+18]) introduced *state-separating proofs* (SSPs) which propose to structure the pseudocode of cryptographic games into stateful code pieces (*packages*) that call each other. The program structure is represented by graphs, where the nodes represent packages with disjoint state. One of the advantages of SSPs is that reductions can be defined *syntactically*: When describing a cryptographic game as graph, one can identify a subgraph which matches a cryptographic assumption. The cryptographic assumption states then that these subgraphs can be replaced, and the remaining part of the graph of the bigger cryptographic game becomes the reduction. Syntactic arguments, thus, allow cryptographers to avoid to reason about conceptually "irrelevant" code in a reduction proof.

SSPs have been used to show reduction-based security proofs for the key schedules of TLS [BDE+21] and MLS [BCK22], two real-life protocols with complex design. Security notions for key exchange protocols are typically game-based, and each game hop idealizes only a single small primitive so that most of the protocol code is "irrelevant" to the reduction argument, making them an ideal target for SSPs. In this work, we ask

Are SSPs equally useful for the modular analysis of secure multiparty computation protocols?

Secure multiparty computation (MPC) enables mutually distrusting parties to compute a public function f of their secret inputs. While MPC security definitions are usually simulation-based rather than game-based, the constructions and proofs follow an inherent structure: The function f is typically represented as a circuit and both construction and security proof proceed in a gate-by-gate fashion. In this paper, we study the security of

⋆ Part of work done while author was at Aarhus University, Denmark.

Yao's garbling scheme, one of the central building blocks for constructing MPC protocols. A garbling scheme allows one party, the garbler, to *garble* a circuit and a secret input such that another party, the *evaluator*, can evaluate the garbled circuit and learn the result of the computation but not the input. The scheme is secure if the garbling can be simulated given only the circuit evaluation, but without knowledge of the secret input. Garbling schemes typically garble circuits garbled gate by gate, and each reduction step in the security proof only treats a single gate within the circuit. Thus the reduction contains a large amount of "irrelevant" code, namely the description of the garbling for the remaining circuit. We show that, indeed, SSPs can help to syntactically define and visually identify code which us not relevant to the reduction step at hand.

Yao's Garbling Scheme. Yao's garbling scheme associates each gate of a circuit with 4 input keys and 2 output keys and uses double encryption to encrypt the output keys under suitable input keys. The existing security proofs due to Lindell and Pinkas [LP09] and Bellare, Hoang and Rogaway (BHR) [BHR12b] reduce the security of the garbling scheme to the security of the underlying encryption scheme, performing a gate-wise hybrid argument following a suitable topological sorting on the gates. See Section 2.2 for the full description.

Our Approach. Our approach is two-fold. The first part concerns the proof structure. We start with an observation: Expressing circuits as composition of layers can greatly simplify our proof effort by adding additional structure.[3] In a first step, we can now reduce circuit garbling security to layer garbling security. We then implement a hybrid argument that reduces the security of each garbling layer to IND-CPA security of the encryption scheme. In this reduction, we can focus on a single layer and ignore all other circuit levels, i.e., SSPs allow us to remove this code via a syntactic argument. These two hybrid arguments allow us to obtain a game for circuit garbling in which all primitives are idealized. In a final game hop, we split this game so that part of it acts as simulator while the rest evaluates the circuit as desired. This step can again be performed by identifying a subgraph, in this case the circuit evaluation.

The second important ingredient is recasting the BHR [BHR12b] garbling scheme security notion to better align computation model and security notion. We describe security notions *and* garbling scheme syntax as composition of packages as opposed to algorithms and games, respectively. This allows for a structured description of specific garbling schemes and reduces some of the proof overhead which stems from "inlining" the construction into a security game.

Our proof of Yao's garbling scheme can be seen as the next logical step in a sequence of works: Lindell and Pinkas [LP09] published the first security proof for Yao's garbled circuits; BHR [BHR12b] abstracted Yao's garbled circuit construction to a general notion of garbling schemes and stated the reductions in clean pseudo-code; the soundness of their reductions was later proved in EasyCrypt by Almeida, Barbosa, Barthe, Dupressoir, Grégoire, Laporte and Pereira [ABB+17] via rather complex invariant proofs. Our contribution is a manually easily verifiable version of the proof, including the soundness of the reductions. We remark that the recent work of Dupressoir, Kohbrok and Oechsner [DKO21] as well as discussions with the authors of [ABB+17] give us hope that our proof can be mechanized. However, formal verification–whether in existing general-purpose or SSP-specific tools such as SSProve [AHR+21]–is a goal that is orthogonal to the scope of this work and we thus leave this question as future work.

Outline. Section 2 introduces garbling schemes, Section 3 introduces state-separa-ting proofs (SSPs), Section 4 formulates garbling schemes in terms of state-separated packages. Section 5 defines Yao's garbling schemes in state-separated packages and provides its security proof. Finally, we provide additional perspective and discussion in Section 8.

2 Garbling Schemes

2.1 Garbling schemes

Bellare, Hoang and Rogaway (BHR) [BHR12b] introduce the notion of a *garbling scheme* as an abstraction of the primitive underlying the garbled circuits approach.

Definition 1 (Garbling scheme [BHR12b]). *A* (circuit) garbling scheme *consists of* 5 *probabilistic, polynomial-time algorithms* $\mathsf{gs} = (gb, en, de, ev, gev)$ *for circuit garbling, input encoding and output decoding, circuit evaluation and garbled circuit evaluation, respectively.*

[3] Strictly speaking, layers are not necessary and gates would work as well, but layered circuits provide a visually more appealing version of the argument, and are common anyway, e.g., in AES.

A garbling scheme is *projective* if input encoding is of the following shape: the circuit garbling gb outputs input encoding information e that consists of two tokens per input bit, and input encoding en selects for each input bit the corresponding token. We add the notion of an *inverse projective* garbling scheme as an analogy for output decoding: A garbling scheme is inverse projective if the output decoding information d of gb consists of two tokens per output bit, and de selects for each output token the corresponding bit. Furthermore, we assume for the rest of this work that the circuit evaluation algorithm ev is fixed and hence omit it in descriptions and write $C(x)$ instead of $ev(C, x)$. $\Phi(C)$ is defined as the information the circuit garbling leaks about a circuit C (e.g. the circuit topology for Yao's garbling scheme). For simplicity, throughout this article, $\Phi(C)$ will be equal to C.

Definition 2 (Garbling scheme correctness [BHR12b]). *Let $\kappa \in \mathbb{N}$. A garbling scheme $\mathsf{gs} = (gb, en, de, gev)$ is* perfectly correct *if for all circuits C and inputs x,*

$$\Pr_{(\tilde{C},e,d) \leftarrow\!\$\, gb(1^\kappa, C)} \Big[C(x) = de(d, gev(\tilde{C}, en(e, x))) \Big] = 1$$

Garbling scheme gs *is* statistically correct *if the above equality holds with overwhelming probability in κ.*

Security is formulated using the *simulation paradigm*: A real execution of the garbling scheme on real input x is compared to a simulated (idealized) execution without access to x, but only to $\Phi(C)$ and the result $C(x)$ of the computation. If both executions are indistinguishable, then the real execution leaks as much information about x as the ideal information. Since the ideal execution is independent of x, no information about x is leaked. More formally, the executions are formulated as games $\mathtt{PRVSIM}^b_{\mathsf{gs},\Phi,\mathcal{S}}$ for $b \in \{0, 1\}$ (Figure 1).

$\mathtt{PRVSIM}^0_{\mathsf{gs},\Phi,\mathcal{S}}$	$\mathtt{PRVSIM}^1_{\mathsf{gs},\Phi,\mathcal{S}}$
$\mathsf{GARBLE}(C, x)$	$\mathsf{GARBLE}(C, x)$
$(\tilde{C}, e, d) \leftarrow gb(1^\kappa, C)$	$y \leftarrow C(x)$
$\tilde{x} \leftarrow en(e, x)$	$(\tilde{C}, \tilde{x}, d) \leftarrow \mathcal{S}(1^\kappa, y, \Phi(C))$
return $(\tilde{C}, \tilde{x}, d)$	**return** $(\tilde{C}, \tilde{x}, d)$

Fig. 1: Garbling scheme security games $\mathtt{PRVSIM}^b_{\mathsf{gs},\Phi,\mathcal{S}}$.

Definition 3 (Garbling scheme security [BHR12b]). *Let $\kappa \in \mathbb{N}$. A garbling scheme $\mathsf{gs} = (gb, en, de, gev)$ is* secure wrt. leakage function Φ *if for all PPT adversaries $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that the distinguishing advantage*

$$\left| \Pr\left[1 \leftarrow\!\$\, \mathcal{A} \rightarrow \mathtt{PRVSIM}^0_{\mathsf{gs},\Phi,\mathcal{S}}\right] - \Pr\left[1 \leftarrow\!\$\, \mathcal{A} \rightarrow \mathtt{PRVSIM}^1_{\mathsf{gs},\Phi,\mathcal{S}}\right] \right|$$

of $\mathcal{A}$ interacting with $\mathtt{PRVSIM}^0_{\mathsf{gs},\Phi,\mathcal{S}}$ and $\mathtt{PRVSIM}^1_{\mathsf{gs},\Phi,\mathcal{S}}$ is negligible in κ.

The above security notion is also referred to as *selective security* because an adversary needs to choose both circuit C and input x to be garbled together. *Adaptive security* [BHR12a] on the other hand allows the adversary to obtain a circuit garbling and only then adaptively choose the input x. This notion is notoriously hard to achieve. This work focuses on the standard notion of selective security.

2.2 Yao's garbling scheme

Two-party computation Assume that two mutually distrusting parties P_1 and P_2 with secret inputs x_1 and x_2, respectively, wish to compute the public function $f(x_1, x_2)$ of their inputs and learn the result $y = f(x_1, x_2)$. To achieve this task, they can run the following protocol template proposed by Yao in his seminal work on two-party computation [Yao86].

1. Represent the function f to be computed as Boolean circuit C with gates of fan-in 2.
2. P_1 *garbles* the circuit C into $\tilde{C}$ and *encodes* their input x_1 into $\tilde{x}_1$, and sends $\tilde{C}$ and $\tilde{x}_1$ to P_2.
3. P_1 and P_2 run a protocol to encode P_2's input x_2.
4. P_2 *evaluates* the garbled circuit $\tilde{C}$ on $\tilde{x}_1 || \tilde{x}_2$, *decodes* the output $\tilde{y}$ to y, and sends y to P_1.

The garbled circuits approach is one of the main design paradigms for secure two-party and multiparty computation protocols. Yao's construction is in fact only the first in a long sequence of works on successively more optimized constructions, see [NPS99,KS08,PSSW09,ZRE15,GLNP15,RR21] and references therein. We focus on Yao's original construction since is the most studied and we are mainly interested in a comparison of *proofs*. Apart from secure multiparty computation, applications of the garbled circuits idea include but are not limited to one-time programs [GKR08] and verifiable computation [GGP10].

Yao's construction We now describe how Yao's garbling scheme construction matches the above template. The construction uses an IND-CPA secure symmetric encryption scheme $(kgen, enc, dec)$ where $kgen$ selects uniformly random bitstrings as keys. The encryption scheme is assumed to return a special error symbol $\perp$ if decryption fails due to the use of an incorrect decryption key[4].

Circuit garbling. To garble a circuit C with q gates, n input and m output wires, start by choosing two random bitstrings per circuit wire j using $kgen$, and assign them semantics 0 and 1, respectively. Denote the wire key map by Z_j. Then for each gate g_j with operation op_j, do: Let Z_ℓ, Z_r, Z_j be the left and right input wire key maps and the output wire key maps, respectively. Encrypt the output wire keys under the input wire keys according to op_j (see code of the right). The garbled gate $\tilde{g}_j$ consists of the ciphertexts $c_0, \ldots, c_3$ in randomly permuted order, and the garbled circuit $\tilde{C}$ consists of the q garbled gates $\tilde{g}_1, \ldots, \tilde{g}_q$ as well as input encoding information (the key maps of the n input wires) and output decoding information (the key maps of the m output wires).

$$
\begin{aligned}
c_0 &= enc_{Z_r(0)}(enc_{Z_\ell(0)}(Z_j(op_j(0,0)))),\\
c_1 &= enc_{Z_r(1)}(enc_{Z_\ell(0)}(Z_j(op_j(0,1)))),\\
c_2 &= enc_{Z_r(0)}(enc_{Z_\ell(1)}(Z_j(op_j(1,0)))),\\
c_3 &= enc_{Z_r(1)}(enc_{Z_\ell(1)}(Z_j(op_j(1,1)))).
\end{aligned}
$$

Fig. 2: Yao's gate garbling

Input encoding. For each bit x_i of input x, select the corresponding input wire key on the ith input wire.

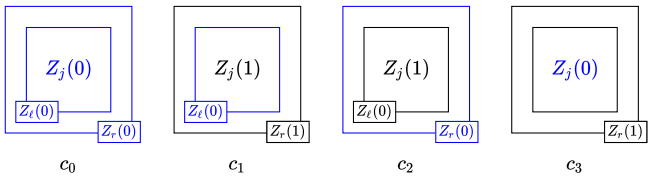
Circuit evaluation. Given garbled circuit $\tilde{C}$ and encoded input $\tilde{x}$, the garbled circuit is evaluated as follows: For each gate $\tilde{g}_j$, let z_ℓ and z_r be the wire keys corresponding to left and right input wire (either obtained from $\tilde{x}$ or a previous gate evaluation). Then attempt to decrypt each of the four gate ciphertexts with z_ℓ and z_r. If the circuit was garbled correctly, then exactly one will decrypt to the desired output wire key z_j without error.

Output decoding. For each key z_i on output wire i, select the corresponding output bit on the ith output wire.

Oblivious transfer protocol To encode P_2's input, an oblivious transfer (OT) protocol can be used. An OT protocol consists of two parties, a sender and a receiver. The sender will input two bitstrings s_0, s_1; the receiver inputs a choice bit b. The goal is for the receiver to learn s_b without the sender learning b.

Security The protocol described is secure against semi-honest parties that follow that protocol description but might try to learn additional information. Intuitively, security means here that no party learns anything about the input of the other party other than what the output reveals. For security against corrupt P_2, this reduces to security of the OT protocol against receiver corruption. For corrupt P_1, security reduces to OT security against sender corruption and the following property of circuit garbling and input encoding: Given C, $\tilde{C}$ and $\tilde{x}$, no information about x other than what $C(x)$ leaks is revealed.

In order to prove this, we make the following observation: For each wire, the adversary only knows *one* of the two wire keys, usually referred to as the *active* key. Moreover, gate garbling in (Fig. 2) is defined such that when the adversary only knows the two active keys of the input wires of the gate, then the adversary also only learns the active output key of the gate. To see this, let us consider the xor operation as an example, and let us say that for the left input wire, the 0-key is active (known to the adversary) and for the right input wire, also the 0-key is active. The 4 ciphertexts can be illustrated as follows:



The adversary only knows the blue key and thus can only recover the blue key. This observation generalizes to arbitrary operations, since there are four ways[5] to combine left active/inactive and right active/inactive key so that the adversary always only learns one ciphertext—there is only one ciphertext which can be represented by two nested blue squares. The adversary always learns the active output key, because if b_ℓ and b_r are the active bits, then we encrypt $Z_j(op(b_\ell, b_r)$ under $Z(b_\ell)$ and $Z(b_r)$—which is the active key. Applying this argument recursively yields the desired security statement. The goal of the main example of this work will be to make this informal security intuition precise.

[4] This can be achieved, e.g., by padding the message with zeros before encryption and checking this condition during decryption.

[5] active/active, active/inactive, inactive/active, inactive/inactive

2.3 Conventions

We omit the security parameter κ whenever it is clear from context. We make the simplifying assumption that each circuit is *layered*. A layered circuit is a circuit whose gates can be partitioned into layers $1, \ldots, d$ such that each wire connects gates in adjacent layers. Circuits such as AES are naturally layered, and transforming an arbitrary circuit of size s into a layered one results in at most a quadratic increase in size [Weg87]. All our results can be modified to non-layered circuits by formulating appropriate gate assumptions instead of layer assumptions. We focus in layered circuits due to their nicer visual representation. Namely, if dependencies between gates can be arbitrary, they can neither be drawn nor described in concise algebraic terms. Finding a suitable notation is an interesting open problem.

3 State-separating proofs

Security games such as $\mathtt{PRVSIM}^0_{\mathsf{gs},\Phi,\mathcal{S}}$ described in the previous section are not known to come with a natural way of composition such as Universal Composability [Can01,Mau12]. However, Brzuska, Delignat-Lavaud, Fournet, Kohbrok, and Kohlweiss (BDFKK [BDF+18]) observe that by splitting a game into multiple parts while carefully preserving dependencies, one can indeed achieve compositionality and modularity. This section provides a brief overview over the key concepts of their proposal, state-separating proofs (SSPs), which are relevant for this work.

3.1 Packages

The central object of SSPs are packages, a collection of oracles with shared state that are described in pseudocode.

Definition 4 (Package). *A package* M *provides a set of oracles* $[\rightarrow M]$ *which operate on a shared state and make calls to a set of oracles* $[M \rightarrow]$*, which we call the dependencies of* M.

Packages M, N can be composed sequentially along matching oracle names and dependencies, i.e., $[M \rightarrow] \cap [\rightarrow N]$, or in parallel if their oracle names are disjoint, i.e., $[\rightarrow M] \cap [\rightarrow N] = \emptyset$. The result is again a package.

A simple example of a package is what we call a *monolithic* security game such as $\mathtt{PRVSIM}^0_{\mathsf{gs},\Phi,\mathcal{S}}$. Splitting the block of pseudocode that describes a security game into individual packages of code, each with their own state, yields a modular game description. We can re-use code packages, and, in particular, games can share packages and be composed via their shared code packages. A schematic example of such a modular game is given in Fig. 3. Following BDFKK, we represent games by *call graphs*. Boxes represent packages and arrows labelled by oracle names represent oracles. An arrow from package M to N stands for the fact that M is allowed to call the oracles of N noted on the arrow. Any oracle that has no start or end package is open for composition.

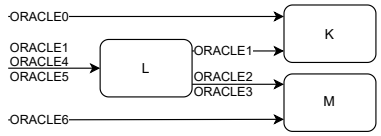


Fig. 3: Example call graph with packages K, L and M.

Note that a package M cannot call its own oracles. We restrict how packages can call each other by requiring to arrange them in an *acyclic*, directed graph, avoiding some known issues of interactive Turing machines (ITM) previously discussed in the context of Universal Composability such as scheduling [HUM09,MT13]. This restriction is in fact a functional style of oracles, i.e. after a caller M calls a callee N, the package N might make further oracle calls to other packages, but eventually returns control to M. Interaction between packages is restricted to an explicit call graph specifying which package is allowed to call which oracle(s) of which other packages. Going back to Fig. 4, we can see that package L provides oracles ORACLE1, ORACLE4 and ORACLE5 and whose code includes calls to oracles ORACLE1, ORACLE2 and ORACLE3. This is meaningful due to the aforementioned self-calling restriction. I.e., if the code of ORACLE1 of L contains the line $a \leftarrow \mathsf{ORACLE1}(b)$, then this means that ORACLE1 of package K is called.

Notation. For a package M, we denote by $[\rightarrow M]$ the oracle(name)s which M provides and by $[M \rightarrow]$ the dependencies of M, namely the names of the oracles to which the oracles of M make calls. We use pseudo-code notation. Sets and tables are denoted by capital letters: S, $T[x]$. The notation $x \leftarrow y$ means that the value of variable y is stored in variable x, and $x \leftarrow\!\!\$\, S$ means that x is sampled uniformly at random from S. Finally $x \leftarrow\!\!\$\, \mathsf{algo}(a)$ means that the randomized algorithm algo is executed on argument a and the result is stored in variable x. We use the following abbreviation in our code for error handling:

$$\textbf{assert}\ \text{cond} := \textbf{if}\ \neg\text{cond}\ \textbf{then return}\ \text{error symbol}.$$

We assume that a system cannot be called anymore after an **assert** was violated. However, the adversary will still be allowed to produce an output.

Note that we use a visually more suggestive notation than BDFKK: We use notation $[\to M]$ and $[M \to]$ instead of $\mathsf{out}(\mathtt{M})$ and $\mathsf{in}(\mathtt{M})$, and we allow parallel composition of packages $\mathtt{M}$ and $\mathtt{N}$ with overlapping $[\to M]$ and $[\to N]$. We allow copies of the same package to appear several times in a graph. A subscript to the package and its oracles is added when disambiguation requires it. The latter allows for a leaner notation with fewer parameters and subscripts. The notation remains unambigious, since the call graph clarifies all oracle calls.

3.2 Games

The term *game* refers to any (composition of) package(s) which has no open oracle calls, that is:

Definition 5 (Game). *A package* $\mathtt{G}$ *is a* game *if* $[\mathtt{G} \to] = \emptyset$.

Note that the definition of a game $\mathtt{G}$ applies to single packages and compositions of packages alike, since a composition of packages is syntactically a package. An example of such a game is the composition of packages $\mathtt{K}$, $\mathtt{L}$ and $\mathtt{M}$ in Fig. 3. We will sometimes write $\mathtt{G}(\mathtt{M})$ to highlight that game $\mathtt{G}$ is parametrized by a package $\mathtt{M}$, i.e. all of $\mathtt{G}$ is fixed except for $\mathtt{M}$. An adversary is a special package that does not provide any oracles itself:

Definition 6 (Adversary). *An* adversary *is a package* $\mathtt{A}$ *with* $[\to \mathtt{A}] = \emptyset$.

Conveniently, we can now compose an adversary with a game (provided their interfaces match), as shown in Fig. 4 on our example from before: The adversary $\mathtt{A}$ can interact with the game by making calls to the oracles visible to it. As control is always returned to the adversary, the adversary eventually terminates by outputting a bit. In security statements, the adversary is typically an abstract package and we quantify over all possible implementations. We will omit the adversary from call graphs whenever it is clear from context how it would look like. We will return to experiments with adversaries shortly after defining (polynomial) runtime.

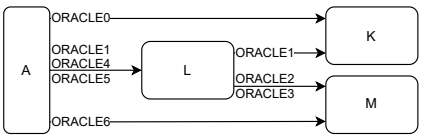


Fig. 4: Adversary $\mathtt{A}$ composed with game from Fig. 3.

Definition 7 (Polynomial Runtime). *A* game $\mathtt{G}$ *runs in* probabilistic polynomial time *(PPT) if the number of steps it makes are polynomial in the number of oracle queries made to it and the accumulated length of the arguments of these queries. A* package $\mathtt{P}$ *runs in PPT if for all PPT games* $\mathtt{G}$ *with* $[P \to] = [\to G]$*, we have that the game* $\mathtt{P} \to \mathtt{G}$ *runs in PPT.*

3.3 Game equivalences

We express properties of a game by comparing it to another game. The most stringent equivalence between two games is *code equivalence*.

Definition 8 (Code Equivalence). *Two games* $\mathtt{G}^0$ *and* $\mathtt{G}^1$ *are* code equivalent *if* $\mathtt{G}^0$ *can be transformed into* $\mathtt{G}^1$ *via a sequence of code transformations which do not affect input-output behaviour, we write* $\mathtt{G}^0 \overset{code}{\equiv} \mathtt{G}^1$*. Similarly, two packages* $\mathtt{P}^0$ *and* $\mathtt{P}^1$ *are* code equivalent *if* $\mathtt{P}^0$ *can be transformed into* $\mathtt{P}^1$ *via sequence of code transformations which do not affect the input-output behaviour regardless of the context in which* $\mathtt{P}^0$ *and* $\mathtt{P}^1$ *are used. We also write* $\mathtt{P}^0 \overset{code}{\equiv} \mathtt{P}^1$.

We will define two weaker notions of equivalence between games which refer to the view of an adversary onto the games.

Definition 9 (Advantage). *Let* G^0 *and* G^1 *be two games and let* $\mathcal{A}$ *be an adversary. Then the distinguishing* advantage $\mathsf{Adv}(\mathcal{A}; G^0, G^1)$ *of* $\mathcal{A}$ *is defined as*

$$\left|\Pr\left[1 \leftarrow_{\$} \mathcal{A} \to G^0\right] - \Pr\left[1 \leftarrow_{\$} \mathcal{A} \to G^1\right]\right|.$$

Definition 10 (Statistical Indistinguishability). *Two games* G^0 *and* G^1 *are* statistically indistinguishable*, denoted* $\mathtt{G}^0 \overset{stat}{\approx} \mathtt{G}^1$*, if for all (computationally unbounded) adversaries* $\mathcal{A}$ *making at most a polynomial number of queries to* $\mathtt{G}^b$*, the advantage* $\mathsf{Adv}(\mathcal{A}; G^0, G^1)$ *is negligible.*

Definition 11 (Computational Indistinguishability). *Two games G^0 and G^1 are* computationally indistinguishable *if for all PPT adversaries $\mathcal{A}$, the advantage $\mathsf{Adv}(\mathcal{A}; G^0, G^1)$ is negligible, we write $\mathtt{G}^0 \overset{comp}{\approx} \mathtt{G}^1$.*

Claim 1 (Indistinguishability) *For two games G^0 and G^1,*

$$\mathtt{G}^0 \overset{code}{\equiv} \mathtt{G}^1 \Rightarrow \mathtt{G}^0 \overset{stat}{\approx} \mathtt{G}^1 \text{ and } \mathtt{G}^0 \overset{stat}{\approx} \mathtt{G}^1 \Rightarrow \mathtt{G}^0 \overset{comp}{\approx} \mathtt{G}^1.$$

3.4 Reductions

Computational analysis reduces the security of a cryptographic protocol to one or several computational hardness assumptions. The security of a cryptographic protocol is expressed as indistinguishability of two games G^0 and G^1 (such as $\mathrm{G}^0 = \mathtt{SEC}^0(\mathtt{GB})$ and $\mathrm{G}^1 = \mathtt{SEC}^1(\mathtt{SIM})$), and for the proof of their indistinguishibility, we define a sequence of hybrid games $\mathtt{H}_0$,..,$\mathtt{H}_t$ such that $\mathrm{G}^0 \overset{\text{code}}{\equiv} \mathtt{H}_0$ and $\mathrm{G}^1 \overset{\text{code}}{\equiv} \mathtt{H}_t$. We then need to argue that for all $i \in \{1, .., t\}$, we have that $\mathtt{H}_{i-1} \overset{\text{comp}}{\approx} \mathtt{H}_i$. The easiest way to prove such an argument is via a *perfect* reduction to a computational assumption which is also expressed via indistinguishability of two games $(\mathtt{A}^0, \mathtt{A}^1)$.

Definition 12 (Perfect Reduction). *Let $\mathtt{H}_{i-1}$ and $\mathtt{H}_i$ and $\mathtt{A}^0$ and $\mathtt{A}^1$ be game pairs with $[\to \mathtt{H}_{i-1}] = [\to \mathtt{H}_i]$ and $[\to \mathtt{A}^0] = [\to \mathtt{A}^1]$. A* perfect reduction *from $(\mathtt{H}_{i-1}, \mathtt{H}_i)$ to $(\mathtt{A}^0, \mathtt{A}^1)$ is a PPT package $\mathcal{R}$ such that*

$$\mathtt{H}_{i-1} \overset{code}{\equiv} \mathcal{R} \to \mathtt{A}^0 \text{ and } \mathtt{H}_i \overset{code}{\equiv} \mathcal{R} \to \mathtt{A}^1.$$

From Definition 12, it follows that for all PPT adversaries, $\mathsf{Adv}(\mathcal{A}; \mathtt{H}_{i-1}, \mathtt{H}_i) \leq \mathsf{Adv}(\mathcal{A} \to \mathcal{R}; \mathtt{A}^0, \mathtt{A}^1)$ and thus, in particular, a perfect reduction implies that

$$\mathtt{A}^0 \overset{\text{comp}}{\approx} \mathtt{A}^1 \Rightarrow \mathtt{H}_{i-1} \overset{\text{comp}}{\approx} \mathtt{H}_i.$$

The SSP style is particularly useful for perfect reductions. In particular, often, $\mathtt{H}_i$ and $\mathcal{R} \to \mathtt{A}^1$ will be identical graphs, i.e., putting the graph of packages defining $\mathcal{R}$ together with the graph of packages defining $\mathtt{A}^1$, we obtain the graph of packages $\mathtt{H}_i$. We will see several such cases in this article.

4 State-Separated Garbling Schemes

We now apply the SSP ideas from Section 3 to garbling schemes. We first revisit the syntax, correctness and security of a garbling scheme. We then discuss how to express and prove further properties.

4.1 Syntax and Correctness.

Traditionally (including the SSP literature), cryptographic constructions are viewed as a tuple of algorithms, or alternatively Turing machines. Security and correctness are then described as games which invoke the different algorithms. In this work, we take a different approach: The syntax of a garbling scheme will be defined as a tuple of packages to align computation model and security model as closely as possible. As a result, we can eliminate the proof overhead caused by incompatibilities.

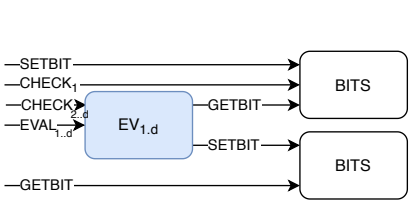
Since garbling scheme security as introduced in Section 2 follows the simulation paradigm, we start by defining our ideal functionality for circuit evaluation. The ideal functionality $\mathtt{F}_d^{\text{gbl}}$ provides oracles SETBIT for setting input bits, d oracles EVAL_i for evaluating circuit layers 1 to d, GETBIT to obtain the result, as well as oracles $\mathsf{CHECK}_{1..d}$ that allows to check whether a circuit wire is already assigned a bit value. The purpose of the last oracle is to allow the switch between different key semantics during the security proof of Yao's garbling scheme. The semantic switch will be stated as Claim 2. We will discuss the implications in more detail in Section 8.

Definition 13 (Ideal Functionality of a Garbling Scheme). *The ideal garbling functionality $\mathtt{F}_d^{gbl}$ for circuits of depth d is defined in Fig. 5 and has*

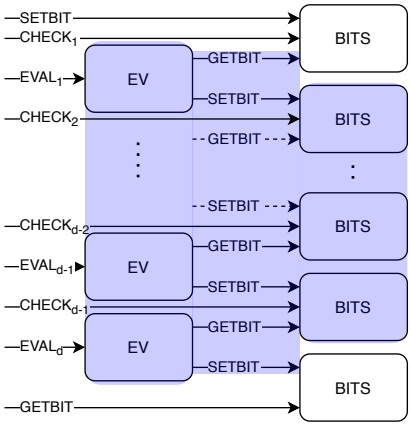
$$[\to \mathtt{F}_d^{gbl}] : \mathsf{SETBIT}, \mathsf{EVAL}_{1..d}, \mathsf{GETBIT}, \mathsf{CHECK}_{1..d}.$$

Fig. 5a shows $\mathtt{F}_d^{\text{gbl}}$ as composition of three packages: Two BITS packages that model the input bits and output bits, respectively, and a circuit evaluation package $\mathtt{EV}_{1,d}$ that performs the actual computation. Note that for convenience, each BITS package is a *multi-instance* package, i.e. can be seen as containing multiple instances of the same package, each storing a single bit, combined into one package. The queries to individual instances are identified by their index j. We will see several examples of these multi-instance packages throughout this paper. In the spirit of modularity, we further refine $\mathtt{EV}_{1,d}$ in Fig. 5b: Remember that our circuit is layered, i.e. the computation can be split into d layers where each layer only depends on the previous layer (or on the input in case of the first layer). We therefore define $\mathtt{EV}_{1,d}$ as composition of gate evaluation packages EV with further BITS packages that store the intermediate result between layers. The pseudocode for packages EV and BITS is shown in Fig. 5c.

$\mathtt{F}_d^{\text{gbl}}$ models circuit evaluation. The adversary can set input bits via SETBIT and use the EVAL_1 query to evaluate the first layer of the circuit, then the EVAL_2 query to evaluate the second layer of the circuit until the last EVAL_d query. In between, the adversary may perform CHECK queries to check whether bits in a the BITS package have been set (this feature will later be used by the simulator). In the end, the adversary can retrieve the result of the evaluation via the GETBIT query to the lowest BIT package. In the definition of $\mathtt{F}_d^{\text{gbl}}$, we already use the feature that the composition of several packages is again a package: We define $\mathsf{EV}_{1..d}$ as composition of EV_1 to EV_d with BITS packages in between (the graph highlighted in blue in Fig. 5), and we define $\mathtt{F}_d^{\text{gbl}}$ as the composition of all of the packages in the graph.



(a) Ideal functionality $\mathtt{F}_d^{\text{gbl}}$.



(b) $\mathtt{EV}_{1..d}$ is marked in blue.

EV

$\mathsf{EVAL}_j(\ell, r, op)$
$z_\ell^{\text{in}} \leftarrow \mathsf{GETBIT}(\ell)$
$z_r^{\text{in}} \leftarrow \mathsf{GETBIT}(r)$
$z_j^{\text{out}} \leftarrow op(z_\ell^{\text{in}}, z_r^{\text{in}})$
$\mathsf{SETBIT}(j, z_j^{\text{out}})$
return ()

BITS

$\mathsf{SETBIT}(j, z)$
assert $z_j = \bot$
$z_j \leftarrow z$
return ()

$\mathsf{GETBIT}(j)$
assert $z_j \neq \bot$
return z_j

$\mathsf{CHECK}(j)$
assert $z_j \neq \bot$
return ()

(c)

Fig. 5: Graph and code of $\mathtt{F}_d^{\text{gbl}}$.

For convenience, we define $\mathtt{F'}_d^{\text{gbl}}$ of to be the simplified ideal functionality $\mathtt{F}_d^{\text{gbl}}$ without CHECK queries. We use $\mathtt{F}_d^{\text{gbl}}$ to define correctness and elaborate on the difference between $\mathtt{F'}_d^{\text{gbl}}$ and $\mathtt{F}_d^{\text{gbl}}$ in Section 8. Running/Composing the algorithms of a garbling scheme adequately should yield a program which behaves as $\mathtt{F}_d^{\text{gbl}}$. In order to make such as statement, let us express the BHR modeling of garbling schemes in packages. Recall that BHR describe a garbling scheme as five algorithms (gb, en, gev, de, ev), where gb garbles a circuit C, en garbles an input x, gev evaluates a garbled circuit on a garbled input, de provides the output of the garbled circuit using decoding information obtained from the garbled evaluation (Def. 2) and ev is a simple circuit evaluation algorithm. Also recall that, in this paper, we consider fixed en, de and ev since we focus on projective garbling schemes.

We start by replacing the five algorithms by five packages GB, KEYS, DE, GEV and EV. The call graph in Fig. 6 shows the intended use of the packages: Garbling a circuit and input, then evaluating the garbled circuit on the garbled input and decoding the result. Each package is meant to capture the algorithm of the same name, except for KEYS which models en with additional state (more discussion on this naming convention later). As KEYS, DE, and EV are fixed, a garbling scheme is defined only by the two packages $(\mathtt{GB}, \mathtt{GEV})$. We further define a real garbling scheme correctness game $\mathtt{GCORR}(\mathtt{GB}, \mathtt{GEV})$, that is a correctness game parameterized by the at this point abstract garbling scheme packages GB, GEV, as the package composition in Fig. 5. Then a projective garbling scheme is defined as follows:

Definition 14 (Projective Garbling Scheme). *Let d be a function of the security parameter. A pair of packages* $(\mathtt{GB}, \mathtt{GEV})$ *is a* projective garbling scheme *with inverse projective decoding if the games* $\mathtt{GCORR}(\mathtt{GB}, \mathtt{GEV})$ *(Fig. 6a) and* $\mathtt{F'}_d^{gbl}$ *(Fig. 5) are statistically indistinguishable, i.e.,* $\mathtt{GCORR}(\mathtt{GB}, \mathtt{GEV}) \overset{stat}{\approx} \mathtt{F'}_d^{gbl}$, *and*

$$[\to \mathtt{GB}] \ : \ \mathsf{GBL}_i : i \in \{1..d\} \qquad [\to \mathtt{GEV}] : \ \mathsf{EVAL}_i : i \in \{1..d\}$$
$$[\mathtt{GB} \to] \ : \ \mathsf{GETKEYS}^{out}, \mathsf{GETKEYS}^{in} \qquad [\mathtt{GEV} \to] : \ \mathsf{GETA}^{out}, \mathsf{GETA}^{in}, \mathsf{GBL}_i : i \in \{1..d\}$$

In our formalization of garbling scheme, GB and GEV share keys via the KEYS package (Fig. 6). KEYS samples keys uniformly at random. GB sees both keys per wire (modeled via $\mathsf{GETKEYS}^{\text{in}}$) and GEV only sees the active key (modeled via $\mathsf{GETA}^{\text{out}}$). Oracles O with superscript O^{out} need to be called before oracles with superscript O^{in}—this enforces (via asserts, see Fig. 6) that the input is garbled before the circuit is garbled which is required for selective security.

4.2 Security

We expressed correctness of a garbling scheme by requiring that it behaves (up to negligible probability) like the ideal functionality. We express security of a garbling scheme by stating that it does not leak more information to the adversary than the ideal functionality. As is standard, we capture this concept by demanding the existence of an efficient simulator which obtains information from the ideal functionality (namely the circuit C and the output value $y = C(x)$, but not x) and needs to simulate the garbling in a way that is indistinguishable to the adversary. We define a *real game* which corresponds to the real execution of a garbling scheme and is captured by the code on the right of the dashed line in Figure 6a which we reproduce in Figure 6c for convenience. This is the information which is transmitted over the network in a real execution of a garbling scheme. Additionally, we also define an *ideal game*, parametrized by a simulator which corresponds to the ideal execution of a garbling scheme. Security then demands that the real and ideal game are indistinguishable for every efficient adversary.

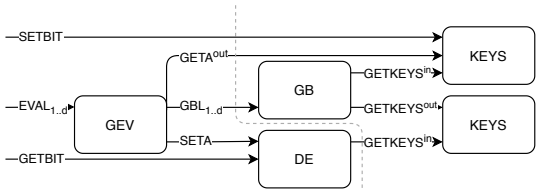
Definition 15 (Garbling scheme security). *Let d be a polynomial in the security parameter, and let $\mathsf{gs} = (\mathtt{GB}, \mathtt{GEV})$ be a garbling scheme.* gs *is secure if there is a PPT simulator* SIM *such that for all PPT adversaries $\mathcal{A}$, the advantage*

$$\mathsf{Adv}(\mathcal{A}; \mathtt{SEC}^0(\mathtt{GB}), \mathtt{SEC}^1(\mathtt{SIM}))$$

for games $\mathtt{SEC}^0(\mathtt{GB})$ and $\mathtt{SEC}^1(\mathtt{SIM})$ in Fig. 6 is negligible.

4.3 Further properties

We have modeled correctness and security of garbling schemes as indististinguishability of games, and we will now express *separable state* via indistinguishability of games. In the proof of Yao's garbling scheme, we will need to show that the KEYS package can be split into AKEYS and a BITS package, so that AKEYS only has an active and inactive key, but does not know the semantics of which one is the 0 key and which one is the 1 key, see Fig. 7b, while BITS has the information of whether the 0 key or the 1 key is active. This is true, because in Fig. 7b, there is no $\mathsf{GETKEYS}^{\text{in}}$ or $\mathsf{GETKEYS}^{\text{out}}$ query on KEYS, but only the queries for active/inactive keys $\mathsf{GETA}^{\text{out}}$, $\mathsf{GETA}^{\text{in}}$ and $\mathsf{GETINA}^{\text{in}}$ (see Fig. 6 for their code).



(a) Real correctness game $\mathtt{GCORR}(\mathtt{GB}, \mathtt{GEV})$.

DE

$\mathsf{SETA}(j, k)$
assert $k_j = \bot$
$k_j \leftarrow k$
return ()

$\mathsf{GETBIT}(j)$
assert $k_j \neq \bot$
$Z \leftarrow \mathsf{GETKEYS}(j)$
if $Z(0) = k_j$:
  return 0
if $Z(1) = k_j$:
  return 1
return ().

KEYS

$\mathsf{SETBIT}(j, z)$
assert $z_j = \bot$
$z_j \leftarrow z$
return ()

$\mathsf{GETKEYS}^{\text{out}}(j)$
$\mathtt{bflag}_j \leftarrow 1$
if $Z_j = \bot$ **then**
  $Z_j(0) \leftarrow\!\!\$\ \{0,1\}^\lambda$
  $Z_j(1) \leftarrow\!\!\$\ \{0,1\}^\lambda$
return $Z_j(z_j)$

$\mathsf{GETKEYS}^{\text{in}}(j)$
assert $z_j \neq \bot$
assert $\mathtt{aflag}_j = 1$
  $\vee\ \mathtt{bflag}_j = 1$
return z_j

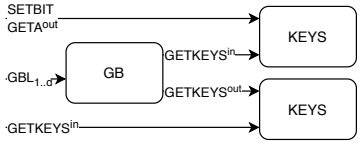
$\mathsf{GETA}^{\text{in}}(j)$
assert $z_j \neq \bot$
assert $\mathtt{aflag}_j = 1$
assert $Z_j \neq \bot$
return $Z_j(z_j)$

$\mathsf{GETBIT}(j)$
assert $z_j \neq \bot$
return z_j

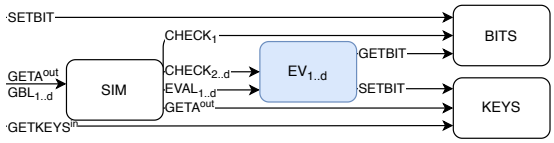
$\mathsf{GETA}^{\text{out}}(j)$
assert $z_j \neq \bot$
$\mathtt{aflag}_j \leftarrow 1$
if $Z_j = \bot$ **then**
  $Z_j(0) \leftarrow\!\!\$\ \{0,1\}^\lambda$
  $Z_j(1) \leftarrow\!\!\$\ \{0,1\}^\lambda$
return $Z_j(z_j)$

$\mathsf{GETINA}^{\text{in}}(j)$
assert $z_j \neq \bot$
assert $\mathtt{aflag}_j = 1$
assert $Z_j \neq \bot$
return $Z_j(1\text{-}z_j)$

(b) Code of DE and KEYS packages



(c) Real security game $\mathtt{SEC}^0(\mathtt{GB})$.



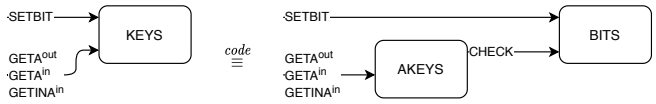
(d) Ideal security game $\mathtt{SEC}^1(\mathtt{SIM})$.

Fig. 6: Correctness game $\mathtt{GCORR}(\mathtt{GB}, \mathtt{GEV})$, code for DE and KEYS and security games $\mathtt{SEC}^0(\mathtt{GB})$ and $\mathtt{SEC}^1(\mathtt{SIM})$. Note that the oracles in the right column of KEYS are not called in $\mathtt{GCORR}(\mathtt{GB}, \mathtt{GEV})$.

Claim 2 *The left and right game in Fig. 7a are perfectly indistinguishable.*

Proof. Column 1 of Figure 7b depicts the oracles of KEYS, and column 4 depicts AKEYS. Column 3 shows BITS inlined into AKEYS as well as the SETBIT oracle of BITS. From column 3 to column 2, we encode zero differently, namely as $z_j \oplus z_j$, and in the output of the $\mathsf{GETINA}^{\text{in}}$ oracle, we add the zero bit $z_j \oplus z_j$ to 1. Moreover, in the $\mathsf{GETINA}^{\text{in}}$ oracle, if $z_j = 1$, then we sample $Z_j(0)$ and $Z_j(1)$ in different order, but neither of the two variables is not read before the other variable is used, their sampling order does not affect the input-output behaviour. Finally, from column 1 to column 2, we perform a permutation on the indices under which we store key values by replacing each bit b by $b \oplus z_j$ (This bijection works both, from column 1 to column 2 and conversely.). Hence, KEYS is code-equivalence to the composition of AKEYS and BITS.

Claim 2, in essence, states that if the garbling (of the simulator) only relies on active/inactive key semantics, then indeed, the 0/1 key semantics of the keys does not need to be available to any algorithm. This argument is part of the aforementioned semantic switch step in the proof of Yao's garbling scheme. We make two observations: (1) This particular code equivalence step is conceptually relevant in the proof of Yao's garbling scheme and cannot be omitted. (2) This code equivalence step is small and easy to verify.



(a) Key semantics switch games

KEYS

SETBIT(j, z)
assert $z_j = \bot$
$z_j \leftarrow z$
return ()

GETA$^{\text{out}}(j)$
assert $z_j \neq \bot$
$\texttt{aflag}_j \leftarrow 1$
if $Z_j = \bot$ **then**
$Z_j(0) \leftarrow_{\$} \{0,1\}^\lambda$
$Z_j(1) \leftarrow_{\$} \{0,1\}^\lambda$
return $Z_j(z_j)$

GETA$^{\text{in}}(j)$
assert $z_j \neq \bot$
assert $\texttt{aflag}_j = 1$
assert $Z_j \neq \bot$
return $Z_j(z_j)$

GETINA$^{\text{in}}(j)$
assert $z_j \neq \bot$
assert $\texttt{aflag}_j = 1$
assert $Z_j \neq \bot$
return $Z_j(1 \oplus z_j)$

KEYS

SETBIT(j, z)
assert $z_j = \bot$
$z_j \leftarrow z$
return ()

GETA$^{\text{out}}(j)$
assert $z_j \neq \bot$
$\texttt{aflag}_j \leftarrow 1$
if $Z_j = \bot$ **then**
$Z_j(0 \oplus z_j) \leftarrow_{\$} \{0,1\}^\lambda$
$Z_j(1 \oplus z_j) \leftarrow_{\$} \{0,1\}^\lambda$
return $Z_j(z_j \oplus z_j)$

GETA$^{\text{in}}(j)$
assert $z_j \neq \bot$
assert $\texttt{aflag}_j = 1$
assert $Z_j \neq \bot$
return $Z_j(z_j \oplus z_j)$

GETINA$^{\text{in}}(j)$
assert $z_j \neq \bot$
assert $\texttt{aflag}_j = 1$
assert $Z_j \neq \bot$
return $Z_j(1 \oplus z_j \oplus z_j)$

$\frac{\text{ID}_{\text{SETBIT}}}{\text{AKEYS}} \rightarrow$ BITS

SETBIT(j, z)
assert $z_j = \bot$
$z_j \leftarrow z$
return ()

GETA$^{\text{out}}(j)$
assert $z_j \neq \bot$
$\texttt{aflag}_j \leftarrow 1$
if $Z_j = \bot$ **then**
$Z_j(0) \leftarrow_{\$} \{0,1\}^\lambda$
$Z_j(1) \leftarrow_{\$} \{0,1\}^\lambda$
return $Z_j(0)$

GETA$^{\text{in}}(j)$
assert $z_j \neq \bot$
assert $\texttt{aflag}_j = 1$
assert $Z_j \neq \bot$
return $Z_j(0)$

GETINA$^{\text{in}}(j)$
assert $z_j \neq \bot$
assert $\texttt{aflag}_j = 1$
assert $Z_j \neq \bot$
return $Z_j(1)$

AKEYS

GETA$^{\text{out}}(j)$
assert CHECK(j)
$\texttt{aflag}_j \leftarrow 1$
if $Z_j = \bot$ **then**
$Z_j(0) \leftarrow_{\$} \{0,1\}^\lambda$
$Z_j(1) \leftarrow_{\$} \{0,1\}^\lambda$
return $Z_j(0)$

GETA$^{\text{in}}(j)$
assert CHECK(j)
assert $\texttt{aflag}_j = 1$
assert $Z_j \neq \bot$
return $Z_j(0)$

GETINA$^{\text{in}}(j)$
assert CHECK(j)
assert $\texttt{aflag}_j = 1$
assert $Z_j \neq \bot$
return $Z_j(1)$

(b) Inlining argument

Fig. 7: Code equivalence proof

5 Yao's Garbling Scheme

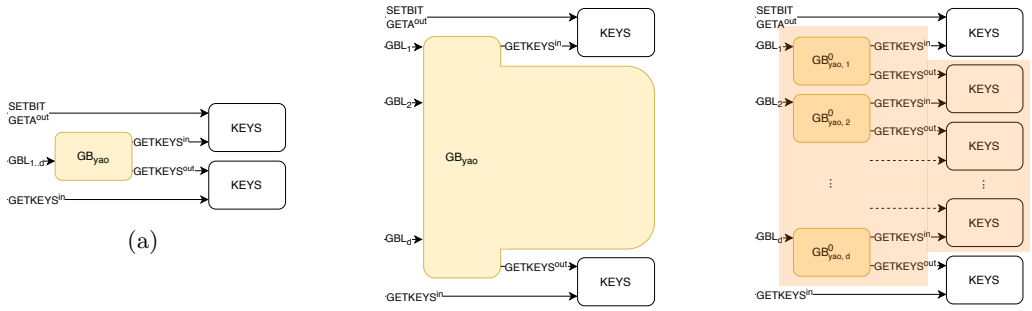
This section introduces our implementation of Yao's garbling scheme. We focus on garbling scheme *security* and the package $\texttt{GB}_{\text{yao}}$, and omit the garbled circuit evaluation package $\texttt{GEV}_{\text{yao}}$.

5.1 Definition of Yao's Circuit and Layer Garbling

Recall that in the traditional description of Yao's garbling scheme (Section 2.2), each gate is garbled individually. Grouping the garbling of multiple gates together into a layer yields a layer garbling. Given a layer garbling, one can build a circuit garbling as follows. Assume for a moment that we already have a layer garbling package $\texttt{GB}^0_{\text{yao},i}$ for garbling the ith layer of circuit C. The garbling process of a layer only shares state with the previous and the subsequent layer of the circuit but not beyond. This shared state consists of *key pairs* (a 0-key and a 1-key for each circuit wire) which we can store in the previously introduced KEYS packages. Hence $\texttt{GB}_{\text{yao}}$ can be defined as composition of layer garbling packages $\texttt{GB}^0_{\text{yao},i}$ with KEYS packages between them.

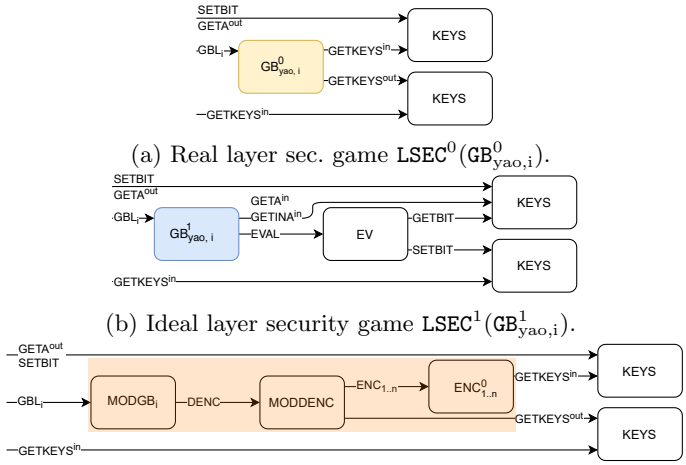
Definition 16 (Yao's Circuit Garbling). *We define the circuit layer garbling* $\texttt{GB}_{yao}$ *as composition of layer garbling packages* $\texttt{GB}^0_{yao,i}$ *with* KEYS *packages as in Figure 8.*

We now define the layer garbling also as a composition of smaller packages: $\texttt{MODGB}_i$ provides a modular description of garbling and transforms gate operations into double encryption queries which $\texttt{MODGB}_i$ sends to



(a) (b) (c)

Fig. 8: Circuit garbling package $\mathtt{GB}_{\mathrm{yao}}$ and game $\mathtt{SEC}^0(\mathtt{GB}_{\mathrm{yao}})$ in different representations.



(a) Real layer sec. game $\mathtt{LSEC}^0(\mathtt{GB}^0_{\mathrm{yao,i}})$.

(b) Ideal layer security game $\mathtt{LSEC}^1(\mathtt{GB}^1_{\mathrm{yao,i}})$.

(c) Real layer security game $\mathtt{LSEC}^0_i(\mathtt{GB}^0_{\mathrm{yao},i})$ with $\mathtt{GB}^0_{\mathrm{yao},i}$ highlighted in orange.

Fig. 9: Layer garbling package $\mathtt{GB}_{\mathrm{yao},i}$ and layer security games $\mathtt{LSEC}^0(\mathtt{GB}^0_{\mathrm{yao,i}})$ and $\mathtt{LSEC}^1(\mathtt{GB}^1_{\mathrm{yao,i}})$.

MODDENC which provides a modular description of double encryption and makes (simple) encryption queries to $\mathtt{ENC}^0_{1\ldots n}$ (see Fig. 9c for the call-graph).

Definition 17 (Yao's Layer Garbling). *Let $i \in \mathbb{N}$. We define the circuit layer garbling package $\mathtt{GB}^0_{yao,i}$ as*

$$\mathtt{GB}^0_{yao,i} := \mathtt{MODGB}_i \to \mathtt{MODDENC} \to \mathtt{ENC}^0_{1\ldots n}$$

for packages $\mathtt{MODGB}_i$, MODDENC, $\mathtt{ENC}^0_{1\ldots n}$ defined in Fig. 12 and composed in Fig. 9c.

We now define the Yao-specific layer security games which turn out useful in the proof.

Definition 18 (Layer Security Games). *Let $i \in \mathbb{N}$. We define the real circuit layer security game $\mathtt{LSEC}^0(\mathtt{GB}^0_{yao,i})$ by Fig. 9a and the ideal circuit layer security game $\mathtt{LSEC}^1(\mathtt{GB}^1_{yao,i})$, parametrized by the layer simulator $\mathtt{GB}^1_{yao,i}$, by Fig. 9b.*

5.2 The Simulator

Analogously to the garbling scheme, we now also define the simulator in a layered way and thus reduce the problem of finding a circuit garbling simulator to finding a layer garbling simulator. Assume for a moment that we have a layer garbling simulator package $\mathtt{GB}^1_{\mathrm{yao,i}}$ as well as a special key package AKEYS that stores keys with active/inactive semantics instead of 0/1 semantics.

Definition 19 (Yao's Circuit Simulator). *We define the circuit garbling simulator $\mathtt{SIM}_{yao}$ as composition of layer garbling simulator packages $\mathtt{GB}^1_{yao,i}$ with AKEYS packages in Figure 10c.*

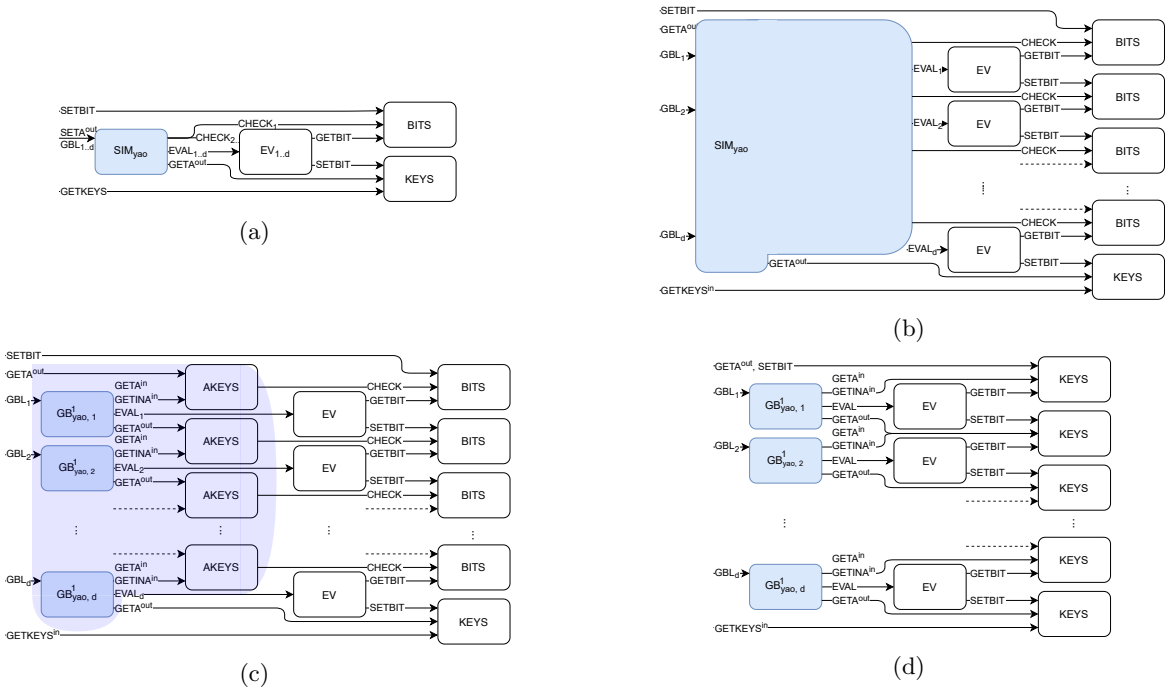


Fig. 10: Circuit garbling simulator $\mathtt{SIM}_{\mathrm{yao}}$ and ideal circuit garbling game $\mathtt{SEC}^1(\mathtt{SIM}_{\mathrm{yao}})$ in different representations.

Figure 10c depicts the definition of the circuit simulator package $\mathtt{GB}^1_{\mathrm{yao}}$, marked in blue in the graph at the bottom of the figure. It consists of the layer simulator $\mathtt{GB}^1_{\mathrm{yao,\,i}}$, together with the AKEYS packages which store the keys shared between two subsequent layers. The AKEYS packages (cf. Figure 7b) contains pairs of keys with active/inactive key semantics and no information about bits as is required in the ideal security definition. Thinking ahead about the security proof we want to perform, we get the ideal security game $\mathtt{SEC}^1(\mathtt{SIM}_{\mathrm{yao}})$ in Figure 10c syntactically closer to the real security game $\mathtt{SEC}^0(\mathtt{GB}_{\mathrm{yao}})$ by merging the AKEYS package and the BITS package via inlining into a KEYS package which exposes only the oracles SETBIT, GETBIT, $\mathsf{GETA}^{\mathrm{out}}$, $\mathsf{GETA}^{\mathrm{in}}$, $\mathsf{GETINA}^{\mathrm{in}}$, as we saw in Section 4.3, Claim 2. The result of replacing AKEYS and BITS into KEYS is shown in Figure 10d.

Finally, we define the previously assumed layer garbling simulator $\mathtt{GB}^1_{\mathrm{yao,i}}$ as the composition of a double encryption simulator $\mathtt{SIM}_{denc}$ composed with the wrapper $\mathtt{MODGB}_i$ from before.

Definition 20 (Yao's Layer Simulator). *Let $i \in \mathbb{N}$. We define the layer garbling simulator $\mathtt{GB}^1_{yao,i}$ as*

$$\mathtt{GB}^1_{yao,i} := \mathtt{MODGB}_i \rightarrow \mathtt{SIM}_{denc}$$

for packages $\mathtt{MODGB}_i$ and $\mathtt{SIM}_{denc}$ defined in Fig. 12.

We use the simulator $\mathtt{GB}^1_{\mathrm{yao,i}}$ to parametrize our previously introduced ideal layer garbling security notion $\mathtt{GB}^1_{\mathrm{yao,i}}$ (Definition 18, Fig. 9b).

5.3 Main Theorem for Yao's Garbling Scheme

The main theorem for Yao's Garbling scheme states that its security reduces to the IND-CPA security of the underlying symmetric encryption scheme.

Theorem 1. *Let $\mathcal{A}$ be a PPT adversary, let d be a polynomial upper bound on the depth of the circuit which $\mathcal{A}$ chooses, let n denote the width of the circuit and let se denote the symmetric encryption scheme used within $\mathtt{GB}_{yao}$. Then, there exists a PPT reduction $\mathcal{R}$ such that*

$$\mathsf{Adv}(\mathcal{A}; \mathtt{SEC}^0(\mathtt{GB}_{yao}), \mathtt{SEC}^1(\mathtt{SIM}_{yao})) \leq d \cdot n \cdot \mathsf{Adv}(\mathcal{A} \rightarrow \mathcal{R}; \mathtt{IND\text{-}CPA}^0(se), \mathtt{IND\text{-}CPA}^1(se)). \tag{1}$$

In particular, $\mathcal{R}$ will be defined as sampling a uniformly random $i \leftarrow_{\$} \{1, .., d\}$ and running $\mathcal{R}^i := \mathcal{R}^i_{circ} \rightarrow \mathcal{R}^i_{layer} \rightarrow \mathcal{R}_{se}$ where reduction $\mathcal{R}^i_{circ}$ is in Lemma 1, reduction $\mathcal{R}^i_{layer}$ is defined in Lemma 2 and reduction $\mathcal{R}_{se}$ is defined in Lemma 3.

We first establish that circuit security reduces to layer security.

Lemma 1 (Circuit Security). *Let d be a polynomial upper bound on the depth of the circuit which $\mathcal{A}$ chooses. Then, for each $1 \leq i \leq d$, there exist a PPT reductions $\mathcal{R}^i_{circ}$ such that*

$$\mathsf{Adv}(\mathcal{A}; \mathtt{SEC}^0(\mathtt{GB}_{yao}), \mathtt{SEC}^1(\mathtt{SIM}_{yao})) \leq \sum_{i=1}^{d} \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^i_{circ}; \mathtt{LSEC}^0(\mathtt{GB}_{yao,i}), \mathtt{LSEC}^1(\mathtt{SIM}_{yao,i})).$$

Lemma 1 will be shown in Section 6. We then show that layer security reduces to multi-instance IND-CPA security.

Lemma 2 (Layer Security). *Let $\mathcal{R}_{layer,i}$ be the reduction defined in Figure 12b. We have*

$$\mathsf{Adv}(\mathcal{A}; \mathtt{LSEC}^0(\mathtt{GB}_{yao,i}), \mathtt{LSEC}^1(\mathtt{SIM}_{yao,i})) \leq \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^i_{layer}; \mathtt{IND\text{-}CPA}^0_{1..n}(se), \mathtt{IND\text{-}CPA}^1_{1..n}(se)).$$

The proof of Lemma 2 can be found in Section 7. Finally, we invoke the hybrid self-composition lemma by BDFKK [BDF+18] to show that multi-instance security of IND-CPA encryption reduces to single-instance IND-CPA encryption.

Lemma 3 (Self-composition [BDF+18]). *There exists a PPT reduction $\mathcal{R}_{se}$ such that for all PPT $\mathcal{A}$, we have that*

$$\mathsf{Adv}(\mathcal{A}; \mathtt{IND\text{-}CPA}^0_{1..n}(se), \mathtt{IND\text{-}CPA}^1_{1..n}(se)) \leq n \cdot \mathsf{Adv}(\mathcal{A} \to \mathcal{R}_{se}; \mathtt{IND\text{-}CPA}^0(se), \mathtt{IND\text{-}CPA}^1(se)).$$

Theorem 1 follows by a standard argument which we provide for completeness below.

Proof (Theorem 1). Let $\mathcal{A}$ be a PPT adversary, then

$$\begin{aligned}
&\mathsf{Adv}(\mathcal{A}; \mathtt{SEC}^0(\mathtt{GB}_{\mathrm{yao}}), \mathtt{SEC}^1(\mathtt{SIM}_{\mathrm{yao}})) \\
&\overset{\text{Lem 1}}{\leq} \sum_{i=1}^{d} \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^i_{\mathrm{circ}}; \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},i}), \mathtt{LSEC}^1(\mathtt{SIM}_{\mathrm{yao},i})) \\
&\overset{\text{Lem 2}}{\leq} \sum_{i=1}^{d} \mathsf{Adv}(\mathcal{B}^i \to \mathcal{R}^i_{\mathrm{lyr}}; \mathtt{IND\text{-}CPA}^0_{1..n}(se), \mathtt{IND\text{-}CPA}^1_{1..n}(se)) \qquad \text{with } \mathcal{B}^i := \mathcal{A} \to \mathcal{R}^i_{\mathrm{circ}} \\
&\overset{\text{Lem 3}}{\leq} \sum_{i=1}^{d} n \cdot \mathsf{Adv}(\mathcal{C}^i \to \mathcal{R}_{\mathrm{se}}; \mathtt{IND\text{-}CPA}^0(se), \mathtt{IND\text{-}CPA}^1(se)) \qquad \text{with } \mathcal{C}^i := \mathcal{A} \to \mathcal{R}^i_{\mathrm{circ}} \to \mathcal{R}^i_{\mathrm{lyr}} \qquad (2)
\end{aligned}$$

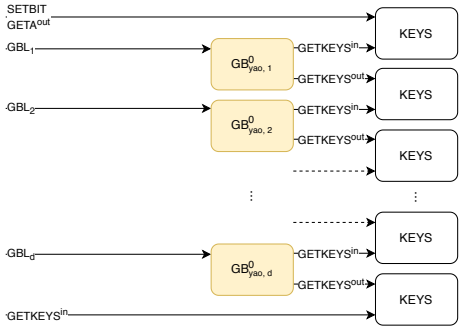
Reduction $\mathcal{R}$ draws a random $i \leftarrow_{\$} \{1, .., d\}$ and then runs $\mathcal{R}^i = \mathcal{R}^i_{\mathrm{circ}} \to \mathcal{R}^i_{\mathrm{layer}} \to \mathcal{R}_{\mathrm{se}}$ (see Theorem 1). Note that by definition

$$\mathcal{C}^i \to \mathcal{R}^i_{\mathrm{se}} = \mathcal{A} \to \mathcal{R}^i \qquad (3)$$

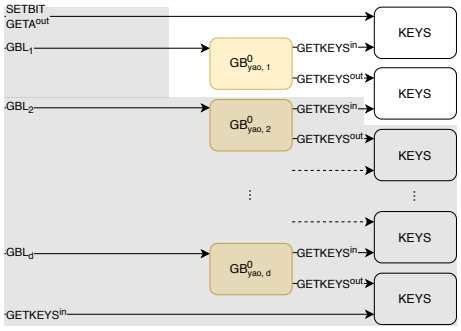
Thus, we have

$$\begin{aligned}
&\mathsf{Adv}(\mathcal{A} \to \mathcal{R}; \mathtt{IND\text{-}CPA}^0(se), \mathtt{IND\text{-}CPA}^1(se)) \qquad (4) \\
&= \left|\Pr\left[1 = \mathcal{A} \to \mathcal{R} \to \mathtt{IND\text{-}CPA}^0(se)\right] - \Pr\left[1 = \mathcal{A} \to \mathcal{R} \to \mathtt{IND\text{-}CPA}^1(se)\right]\right| \\
&= \left|\sum_{i=1}^{d} \frac{1}{d} \Pr\left[1 = \mathcal{A} \to \mathcal{R}^i \to \mathtt{IND\text{-}CPA}^0(se)\right] - \frac{1}{d} \Pr\left[1 = \mathcal{A} \to \mathcal{R}^i \to \mathtt{IND\text{-}CPA}^1(se)\right]\right| \\
&\overset{\text{Eq. 3}}{=} \frac{1}{d}\left|\sum_{i=1}^{d} \Pr\left[1 = \mathcal{C}^i \to \mathcal{R}_{\mathrm{se}} \to \mathtt{IND\text{-}CPA}^0(se)\right] - \Pr\left[1 = \mathcal{C}^i \to \mathcal{R}_{\mathrm{se}} \to \mathtt{IND\text{-}CPA}^1(se)\right]\right| \\
&= \frac{1}{d}\left|\sum_{i=1}^{d} \mathsf{Adv}(\mathcal{C}^i \to \mathcal{R}_{\mathrm{se}}; \mathtt{IND\text{-}CPA}^0(se), \mathtt{IND\text{-}CPA}^1(se))\right| \\
&\overset{\text{Eq. 2}}{\geq} \frac{1}{dn} \mathsf{Adv}(\mathcal{A}; \mathtt{SEC}^0(\mathtt{GB}_{\mathrm{yao}}), \mathtt{SEC}^1(\mathtt{SIM}_{\mathrm{yao}})). \qquad (5)
\end{aligned}$$

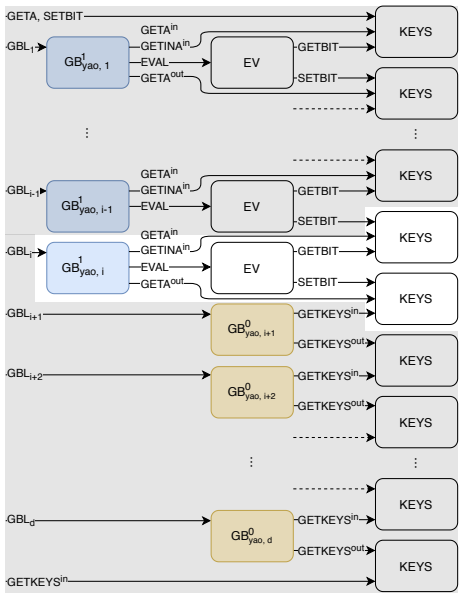
Comparing (4) and (5) establishes Inequality 1 in Theorem 1.



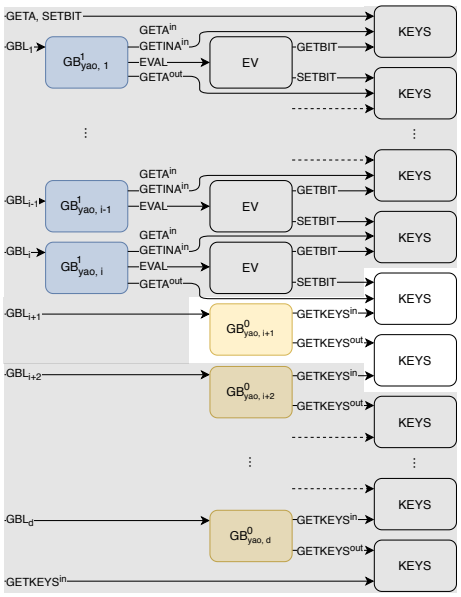
(a) Game $\texttt{SEC}^0(\texttt{GB}_{\text{yao}})$, cf. Fig. 8c.



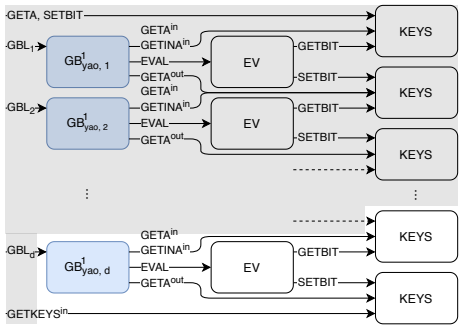
(b) Game $\mathcal{R}^1_{\text{circ}} \to \texttt{LSEC}^0(\texttt{GB}_{\text{yao},1})$, reduction $\mathcal{R}^1_{\text{circ}}$ is marked in grey.



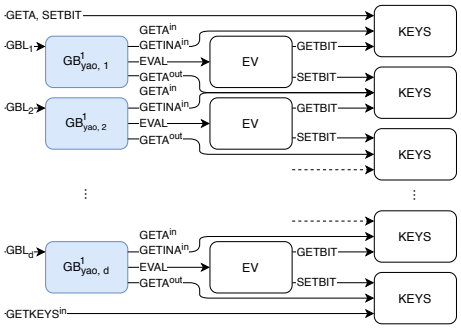
(c) Game $\mathcal{R}^i_{\text{circ}} \to \texttt{LSEC}^1(\texttt{GB}_{\text{yao},i})$, reduction $\mathcal{R}^i_{\text{circ}}$ is marked in grey.



(d) Game $\mathcal{R}^{i+1}_{\text{circ}} \to \texttt{LSEC}^0(\texttt{GB}_{\text{yao},i+1})$, reduction $\mathcal{R}^{i+1}_{\text{circ}}$ is marked in grey.



(e) Game $\mathcal{R}^d_{\text{circ}} \to \texttt{LSEC}^1(\texttt{GB}_{\text{yao},d})$, reduction $\mathcal{R}^d_{\text{circ}}$ is marked in grey.



(f) Game $\texttt{SEC}^1(\texttt{SIM}_{\text{yao},d})$, cf. Fig. 10d.

Fig. 11: Reductions for the hybrid argument for Lemma 1.

6 Proof of Lemma 1

Lemma 1 reduces circuit garbling security to layer garbling security via a hybrid argument over the d layers of the circuit. I.e., our game-hopping argument starts with hybrid 0 which is $\mathtt{SEC}^0(\mathtt{GB}_{\mathrm{yao}})$ and gradually makes modifications until reaching hybrid d which is $\mathtt{SEC}^1(\mathtt{SIM}_{\mathrm{yao}})$. Between these d+1 hybrid games, we have d reductions $\mathcal{R}^i_{\mathrm{circ}}$ with $1 \le i \le d$ (cf. Fig. 11b) so that $\mathcal{R}^i_{\mathrm{circ}}$ is the reduction used to prove the indistinguishability between hybrid i and hybrid $i+1$. For convenience of proof, we define our i-th hybrid as composition between reduction $\mathcal{R}^i_{\mathrm{circ}}$ and a layer security game and observe that, by definition of the first reduction $\mathcal{R}^0_{\mathrm{circ}}$ and the last reduction $\mathcal{R}^d_{\mathrm{circ}}$, we have

$$\mathtt{SEC}^0(\mathtt{GB}_{\mathrm{yao}}) \overset{\mathrm{code}}{\equiv} \mathcal{R}^1_{\mathrm{circ}} \to \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},1}) \tag{6}$$

$$\mathtt{SEC}^1(\mathtt{SIM}_{\mathrm{yao}}) \overset{\mathrm{code}}{\equiv} \mathcal{R}^d_{\mathrm{circ}} \to \mathtt{LSEC}^1(\mathtt{GB}_{\mathrm{yao},d}), \tag{7}$$

which translates $\mathtt{SEC}^0(\mathtt{GB}_{\mathrm{yao}})$ into $\mathcal{R}^1_{\mathrm{circ}} \to \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},1})$ and $\mathtt{SEC}^1(\mathtt{SIM}_{\mathrm{yao}})$ into $\mathcal{R}^d_{\mathrm{circ}} \to \mathtt{LSEC}^1(\mathtt{GB}_{\mathrm{yao},d})$. Now, for $i \in \{1,..,d\text{-}1\}$, we can describe our hybrid games either as $\mathcal{R}^i_{\mathrm{circ}} \to \mathtt{LSEC}^1(\mathtt{GB}_{\mathrm{yao},i})$ or, equivalently, $\mathcal{R}^{i+1}_{\mathrm{circ}} \to \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},i+1})$, i.e.,

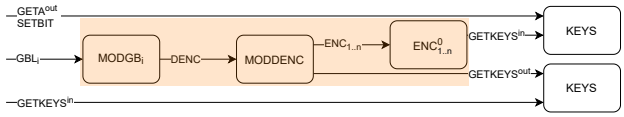
$$\forall\, i \in \{1,..,d\text{-}1\} \colon \mathcal{R}^i_{\mathrm{circ}} \to \mathtt{LSEC}^1(\mathtt{GB}_{\mathrm{yao},i}) \overset{\mathrm{code}}{\equiv} \mathcal{R}^{i+1}_{\mathrm{circ}} \to \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},i+1}). \tag{8}$$

To verify Equation 6-8, note that for $i \in \{1,..,d\}$, Fig. 11b defines $\mathcal{R}^i$, Fig. 9a defines the real layer game $\mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},i})$, and Fig. 9b defines the ideal layer game $\mathtt{LSEC}^1(\mathtt{GB}_{\mathrm{yao},i})$.

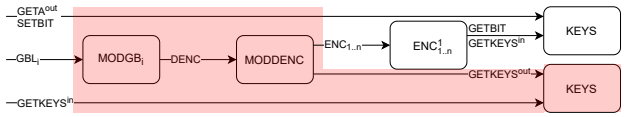
For Equation 6, we provide $\mathtt{SEC}^0(\mathtt{GB}_{\mathrm{yao}})$ in Fig. 11a (copied here for convenience from Fig. 8) and $\mathcal{R}^1_{\mathrm{circ}} \to \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},1})$ in Fig. 11b. Seeing the graphs side-by-side, we observe that they are, indeed, equal. This equivalence proof (by graph equality) is a good example of the SSP technique which takes a single graph and either views it as a single security game (Fig. 11a) or a composition of two things, reduction $\mathcal{R}^1_{\mathrm{circ}}$ and the (smaller) game $\mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},1})$. Analogously, for Equation 7, Fig. 11f describes $\mathtt{LSEC}^1(\mathtt{SIM}_{\mathrm{yao},i})$ (copied from Fig. 9b for convenience), and Fig. 11e contains game $\mathcal{R}_d \to \mathtt{LSEC}^1(\mathtt{GB}_{\mathrm{yao},d})$. Indeed, also these two graphs are equal. Finally, for Equation 8, we provide $\mathcal{R}^i_{\mathrm{circ}} \to \mathtt{LSEC}^1(\mathtt{GB}_{\mathrm{yao},i})$ in Fig. 11c and $\mathcal{R}^{i+1}_{\mathrm{circ}} \to \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},i+1})$ in Fig. 11d and also here, we observe that the two graphs are equal.

We conclude by showing that equations (6)-(8) suffice to prove Lemma 1. Let $\mathcal{A}$ be an adversary. We now first plug-in the definition of advantage, then use Equation 6 and Equation 7 and then add zeroes in telescopic sum-style, using Equation 8. We then use the triangle equality for absolute value, apply the definition of advantage again and obtain Lemma 1.

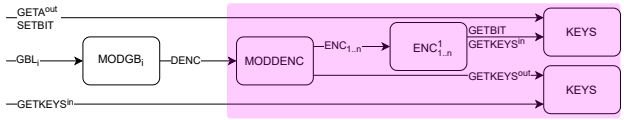
$$\begin{aligned}
& \mathsf{Adv}(\mathcal{A}; \mathtt{SEC}^0(\mathtt{GB}_{\mathrm{yao}}), \mathtt{SEC}^1(\mathtt{SIM}_{\mathrm{yao}})) \\
= \;& |\Pr\big[1 = \mathcal{A} \to \mathtt{SEC}^0(\mathtt{GB}_{\mathrm{yao}})\big] - \Pr\big[1 = \mathcal{A} \to \mathtt{SEC}^1(\mathtt{SIM}_{\mathrm{yao}})\big]| \\
\overset{(6),(7)}{=} \;& |\Pr\big[1 = \mathcal{A} \to \mathcal{R}^1_{\mathrm{circ}} \to \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},1})\big] - \Pr\big[1 = \mathcal{A} \to \mathcal{R}^d_{\mathrm{circ}} \to \mathtt{LSEC}^1(\mathtt{GB}_{\mathrm{yao},d})\big]| \\
\overset{(8)}{=} \;& |\Pr\big[1 = \mathcal{A} \to \mathcal{R}^1_{\mathrm{circ}} \to \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},1})\big] \\
& + \big(\sum_{i=1}^{d-1} - \Pr\big[1 = \mathcal{A} \to \mathcal{R}^i_{\mathrm{circ}} \to \mathtt{LSEC}^1(\mathtt{GB}_{\mathrm{yao},i})\big] + \Pr\big[1 = \mathcal{A} \to \mathcal{R}^{i+1}_{\mathrm{circ}} \to \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},i+1})\big]\big) \\
& - \Pr\big[1 = \mathcal{A} \to \mathcal{R}^d_{\mathrm{circ}} \to \mathtt{LSEC}^1(\mathtt{GB}_{\mathrm{yao},d})\big]| \\
= \;& |\sum_{i=1}^{d} \Pr\big[1 = \mathcal{A} \to \mathcal{R}^i_{\mathrm{circ}} \to \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},1})\big] - \Pr\big[1 = \mathcal{A} \to \mathcal{R}^i_{\mathrm{circ}} \to \mathtt{LSEC}^1(\mathtt{GB}_{\mathrm{yao},d})\big]| \\
\le \;& \sum_{i=1}^{d} |\Pr\big[1 = \mathcal{A} \to \mathcal{R}^i_{\mathrm{circ}} \to \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},1})\big] - \Pr\big[1 = \mathcal{A} \to \mathcal{R}^i_{\mathrm{circ}} \to \mathtt{LSEC}^1(\mathtt{GB}_{\mathrm{yao},d})\big]| \\
= \;& \sum_{i=1}^{d} \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^i_{\mathrm{circ}}; \mathtt{LSEC}^0(\mathtt{GB}_{\mathrm{yao},i}), \mathtt{LSEC}^1(\mathtt{SIM}_{\mathrm{yao},i})).
\end{aligned}$$



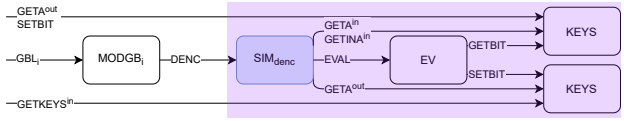
(a) Real layer security game $\texttt{LSEC}_i^0(\texttt{GB}_{\text{yao},i})$ with $\texttt{GB}_{\text{yao},i}$ highlighted in orange.



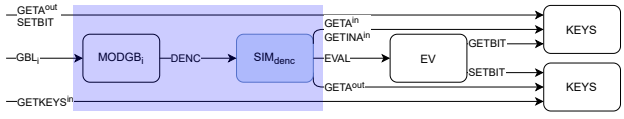
(b) Hybrid layer security game $\texttt{HYB}_i$. Reduction $\mathcal{R}_{\text{layer},i}$ is highlighted in red.



(c) Hybrid layer security game $\texttt{HYB}_i$. Subgame $\texttt{GDENC}$ is highlighted in pink.



(d) Ideal layer security game $\texttt{LSEC}_i^1(\texttt{GB}_{\text{yao},i}^1)$. Subgame $\texttt{GDENC}_{\text{sim}}$ is highlighted in purple.



(e) Ideal layer security game $\texttt{LSEC}_i^0(\texttt{GB}_{\text{yao},i}^1)$. Layer simulator $\texttt{GB}_{\text{yao},i}^1$ is marked in blue.

Oracle of $\texttt{MOD-DENC}$

$\mathsf{DENC}(\ell, r, op, j)$

$\tilde{g}_j \leftarrow \bot$

$Z_j^{\text{out}} \leftarrow \mathsf{GETKEYS}^{\text{out}}(j)$

for $(b_\ell, b_r) \in \{0,1\}^2$ **do**
  $b_j \leftarrow op(b_\ell, b_r)$
  $k_j^0 \leftarrow Z_j^{\text{out}}(b_j)$
  $c_{\mathsf{in}}^0 \leftarrow \mathsf{ENC}_i(b_\ell, k_j^0, 0^\lambda)$
  $c_{\mathsf{in}}^1 \leftarrow \mathsf{ENC}_i(b_\ell, 0^\lambda, 0^\lambda)$
  $c \leftarrow\!\!\$\ \mathsf{ENC}_r(b_r, c_{\mathsf{in}}^0, c_{\mathsf{in}}^1)$
  $\tilde{g}_j \leftarrow \tilde{g}_j \cup c$
return $\tilde{g}_j$

Oracles of $\texttt{MODGB}_i$

$\mathsf{GBL}_i(\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{op})$

assert $\tilde{C} = \bot$
assert $\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{op} \neq \bot$
assert $|\boldsymbol{\ell}|, |\boldsymbol{r}|, |\boldsymbol{op}| = n$
for $j = 1..n$ **do**
  $(\ell, r, op) \leftarrow (\boldsymbol{\ell}(j), \boldsymbol{r}(j), \boldsymbol{op}(j))$
  $\tilde{C}_j \leftarrow \mathsf{DENC}(\ell, r, op)$
$\tilde{C} \leftarrow \tilde{C}_{1..n}$
return $\tilde{C}$

Oracle of $\texttt{SIM}_{denc}$

$\mathsf{DENC}(\ell, r, op, j)$

$\tilde{g}_j \leftarrow \bot$
$\mathsf{EVAL}_j(\ell, r, op)$
$S_j^{\text{out}}(0) \leftarrow \mathsf{GETA}^{\text{out}}(j)$
$S_r^{\text{in}}(0) \leftarrow \mathsf{GETA}^{\text{in}}(r)$
$S_r^{\text{in}}(1) \leftarrow \mathsf{GETINA}^{\text{in}}(r)$
$S_\ell^{\text{in}}(0) \leftarrow \mathsf{GETA}^{\text{in}}(\ell)$
$S_\ell^{\text{in}}(1) \leftarrow \mathsf{GETINA}^{\text{in}}(\ell)$
for $(d_\ell, d_r) \in \{0,1\}^2$ **do**
  $k_\ell^{\text{in}} \leftarrow S_\ell^{\text{in}}(d_\ell)$
  $k_r^{\text{in}} \leftarrow S_r^{\text{in}}(d_r)$
  if $d_\ell = d_r = 0$:
    $k_j^{\text{out}} \leftarrow S_j^{\text{out}}(0)$
  else $k_j^{\text{out}} \leftarrow 0^\lambda$
  $c_{\mathsf{in}} \leftarrow\!\!\$\ enc_{k_r^{\text{in}}}(k_j^{\text{out}})$
  $c \leftarrow\!\!\$\ enc_{k_\ell^{\text{in}}}(c_{\mathsf{in}})$
  $\tilde{g}_j \leftarrow \tilde{g}_j \cup c$
return $\tilde{g}_j$

Oracles of $\texttt{ENC}_i^0$

$\mathsf{ENC}_i(b, m_0, m_1)$

$Z_i^{\text{in}} \leftarrow \mathsf{GETKEYS}^{\text{in}}$
$c \leftarrow\!\!\$\ enc(Z_i^{\text{in}}(b), m_0)$
return c

Oracles of $\texttt{ENC}_i^1$

$\mathsf{ENC}_i(b, m_0, m_1)$

$Z_i^{\text{in}} \leftarrow \mathsf{GETKEYS}^{\text{in}}(i)$
$z_i^{\text{in}} \leftarrow \mathsf{GETBIT}^{\text{in}}(i)$
if $z_i \neq b$ **then**
  $c \leftarrow\!\!\$\ enc(Z_i^{\text{in}}(b), m_1)$
if $z_i = b$ **then**
  $c \leftarrow\!\!\$\ enc(Z_i^{\text{in}}(b), m_0)$
return c

(f) Code of $\texttt{MODGB}_i$, $\texttt{MODDENC}$, $\texttt{SIM}_{\text{denc}}$, $\texttt{ENC}^0$, $\texttt{ENC}^1$.

Fig. 12: Proof of Lemma 2 (Layer Security).

7 Proof of Lemma 2

The proof of Lemma 2 consists of the following two claims:

Claim 3 (Real Code Equivalence) *$\forall 1 \leq i \leq d$, it holds that*

$$\texttt{LSEC}^0(\texttt{GB}_{yao,i}) \overset{code}{\equiv} \mathcal{R}^i_{layer} \to \texttt{IND-CPA}^0_{1..n}(se),$$

where $\mathcal{R}^i_{layer}$ is defined in Figure 12b.

Claim 3 follows by definition of $\texttt{GB}_{\text{yao},i}$.

Claim 4 (Ideal Code Equivalence) *$\forall 1 \leq i \leq d$, it holds that*

$$\texttt{LSEC}^1(\texttt{SIM}_{yao,i}) \overset{code}{\equiv} \mathcal{R}^i_{layer} \to \texttt{IND-CPA}^1_{1..n}(se),$$

where $\mathcal{R}^i_{layer}$ is defined in Figure 12b.

Claim 3 and Claim 4 together directly yield Lemma 2. We now turn to the proof of Claim 4.

Proof (Proof of Claim 4). Claim 4 is the technical heart of the proof. $\texttt{HYB}_i := \mathcal{R}^i_{\text{layer}} \to \texttt{IND-CPA}^1_{1..n}(se)$ is depicted in Figure 12b and Figure 12c. The only difference between these two figures is the highlighting. Namely, we highlight the game GDENC in pink. If we can show that

$$\texttt{GDENC} \overset{\text{code}}{\equiv} \texttt{GDENC}_{\text{sim}}, \tag{9}$$

we obtain Claim 4, since replacing GDENC in $\texttt{HYB}_i$ by $\texttt{GDENC}_{\text{sim}}$ (see Figure 12d for this replacement) directly yields the ideal layer security game $\texttt{LSEC}^1(\texttt{SIM}_{\text{yao},i})$, depicted in Figure 12e—the only difference between Figure 12d and Figure 12e is the highlighting. The proof of Equation 9 is an inlining argument, see Figure 13.

Proof of Equation 9. The proof proceeds via code comparison. The left-most column in Fig. 13. shows GDENC and the rightmost column shows the oracle of $\texttt{SIM}_{denc}$—we refer to Fig. 12 for the code of both.

From the leftmost column in Fig. 13 to the second column, we inline the code of calls to the respective ENC packages. We move the $\mathsf{GETKEYS}^{\text{in}}$ queries before the **for** loop. From the second to the third column, we also inline the KEYS packages (cf. Fig. 6 for its code). For disambiguation, we add superscript in to the state of the top KEYS package and superscript out to the state of the lower KEYS package. To obtain to the 4th column, we apply the bijective maps $b_\ell \mapsto b_\ell \oplus z^{\text{in}}_\ell$ and $b_r \mapsto b_r \oplus z^{\text{in}}_r$. Observe that $b_\ell = z^{\text{in}}_\ell \wedge b_r = z^{\text{in}}_r$ is true if and only if $b_\ell \oplus z^{\text{in}}_\ell = b_r \oplus z^{\text{in}}_r = 0$. We further replace $\texttt{bflag}^{\text{out}} \leftarrow 1$ by $\texttt{aflag}^{\text{out}} \leftarrow 1$: The read operation on this variable is performed by the $\mathsf{GETKEYS}$ oracle and there, the reading is performed on $\texttt{bflag} \vee \texttt{aflag}$, so that it does not matter which flag is set. Moreover, variable $\texttt{bflag}^{\text{in}}$ is never set to 1 and thus, we can replace **assert** $\texttt{bflag}^{\text{in}} \vee \texttt{aflag}^{\text{out}} = 1$ by **assert** $\texttt{aflag}^{\text{out}} = 1$.

From column 6 to column 5, we inline the EV and the KEYS package (we perform both steps in one). The main challenge consists of comparing column 4 and column 5, which follows by mapping $S_i(0)$ to $Z_i(z_i)$ and $S_i(1)$ to $Z_i(z_i \oplus 1)$. This concludes the inlining proof.

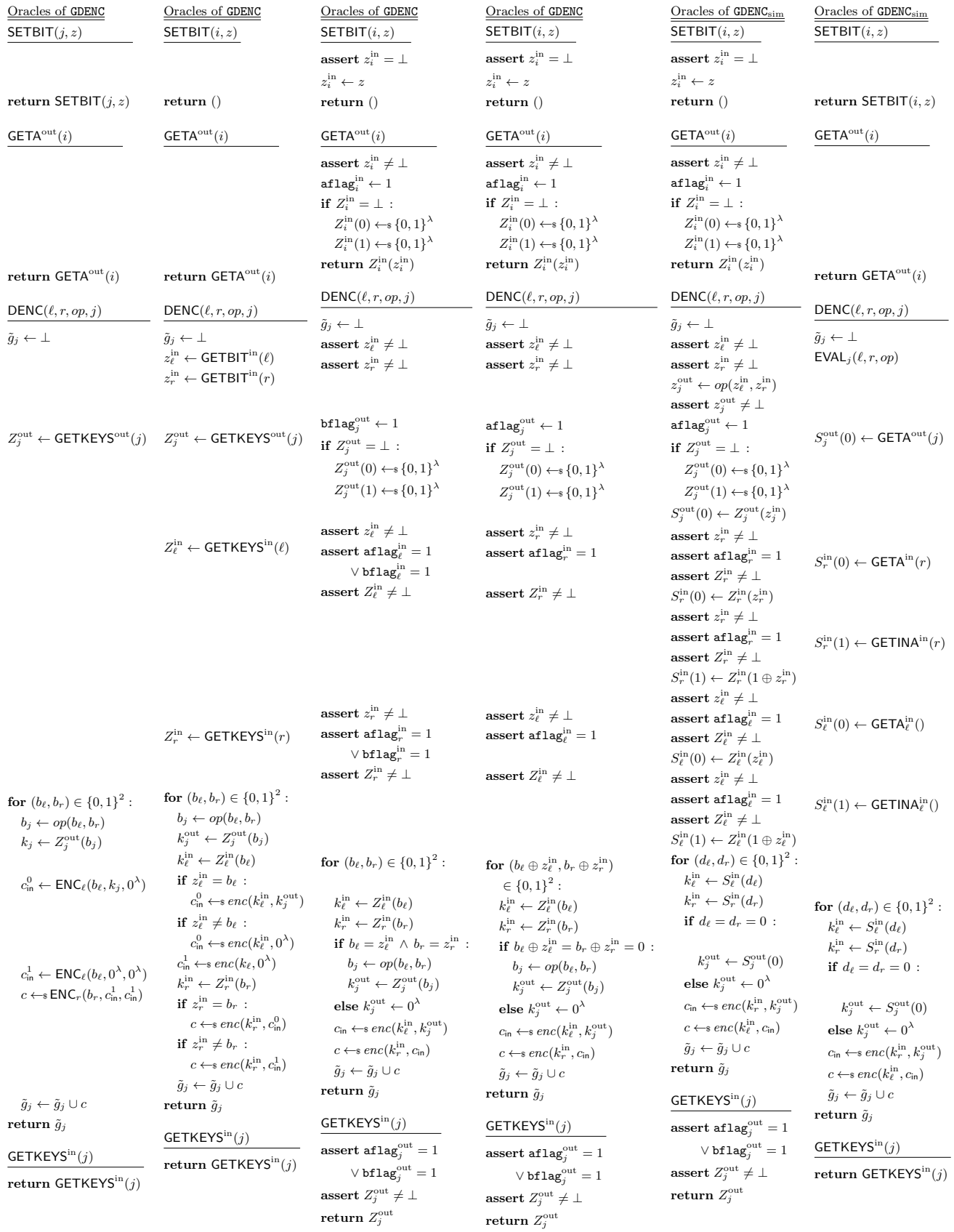


Fig. 13: Inlining for code-equivalence of GDENC and $\text{GDENC}_{\text{sim}}$.

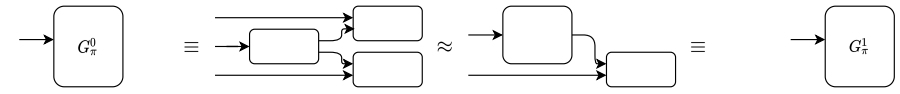
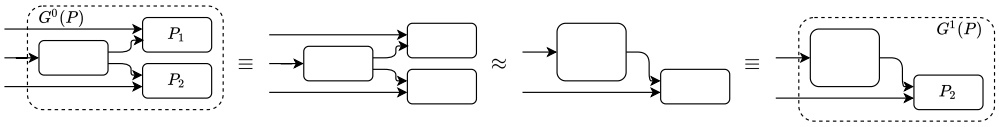
	Syntax notion	Example sequence of games
[BR06]	algorithm tuple, e.g. $\pi = (\pi_1, \pi_1)$	
[BDF+18]	algorithm tuple, e.g. $\pi = (\pi_1, \pi_1)$	
this work	package tuple, e.g. $\pi = (\mathsf{P}_1, \mathsf{P}_1)$	

Table 1: Overview over code-based game-hopping approaches by Bellare and Rogaway [BR06], state-separating proofs [BDF+18] and state-separating constructions in this work. The table compares the notions of cryptographic constructions and the schematic sequence of games in a typical proof.

8 Discussion

Selective vs. adaptive security. In Section 4, we introduced two versions of the ideal functionality for circuit garbling, one for correctness and one for security. We need two different security notions because of the difference between *selective* ($\mathtt{F}_d^{\mathrm{gbl}}$) and *adaptive* ($\mathtt{F'}_d^{\mathrm{gbl}}$) security, as introduced by BHR. The selective security notion asks the simulator to garble circuit and input together. In the adaptive security game, on the other hand, the simulator first has to present the adversary with the garbled circuit who only then chooses the input to be garbled. BHR use only one "ideal functionality" *ev* to express this difference since their *security game* controls when the simulator is invoked, implicitly enforcing that the input is set before the simulator is called, see Fig. 1. In our case though (and in the SSP style of security modeling in general), the simulator is invoked directly by the adversary and we therefore need a way for the simulator to check via a CHECK query whether inputs have already been set. Only when inputs have already been set, the simulator should garble inputs via $\mathsf{GETA}^{\mathrm{out}}$ and gates via GBL.

For the sake of defining correctness, both ideal functionalities provide the same functionality as the CHECK oracle does not yield any new information. CHECK is only relevant when the oracles of the ideal functionality are called by *different* packages (adversary and simulator, in the security game). We thus chose to define correctness with respect to $\mathtt{F'}_d^{\mathrm{gbl}}$, the simpler and in some sense more natural version of the two.

When it comes to security though, the output interface of the ideal functionality is split between adversary and simulator, hence the two ideal functionalities differ in the information they provide to the simulator. Yao's garbing scheme is known to be adaptively secure for circuits of small depth or width [JW16,KKP21] and other garbling schemes [BHR12a,BHK13,HJO+16,GS18,JO20] are adaptively security for general or restricted circuit classes. However, for general circuits, Yao's garbling scheme is only known to provide selective security and thus our security proof is with respect to $\mathtt{F}_d^{\mathrm{gbl}}$.

Local arguments and "irrelevant code". We mentioned in the introduction that SSPs allow us to describe local arguments, i.e. arguments that only reason about a subgame, but allow us to derive statements about the game as a whole. The connection between the subgame and the bigger game does not require reasoning about semantics, but instead is purely syntactical. This work shows several examples:

- Switch of key semantics (Claim 2)
- Reduction of circuit garbling security to layer garbling security (Lemma 1)
- Reduction of layer garbling security to encryption scheme security (Lemma 2)

In addition, our proof style differs in a subtle, but useful way from existing SSPs. Existing works using SSP-style proofs either (1) define security as a monolithic security game in the Bellare-Rogaway style of code-based game-playing and then perform a perfect equivalence step in which the game is split into a modular description [BDF+18], [CHK21], or (2) the security definition is tailored to the specific cryptographic construction in question

[BDE+21], [BCK22]. We propose a third approach: Defining the syntax of the primitive as a package tuple and the security notion itself as composition of packages. Thereby, the security notion becomes modular. Moreover the concrete implementation of the individual packages might be describable as package composition. As a result, when there is a natural modular description (as there is for Yao's garbling scheme), we can take advantage of SSP techniques for the entire security proof, without the need to modularize the games in the proof separately from the construction.

The different definition and proof styles are compared in Table 1. The first row shows a code-based game-playing proof [BR06], expressed schematically in the SSP style. The first and last games in the sequence of game hops are monolithic games parameterized by algorithms, and all intermediate games are of the same size. BDFKK [BDF+18] (second row) propose a more flexible approach to the intermediate game hops, but keeps the monolithic first and last games so that an additional proof is necessary to establish equivalence between monolithic and modular games. In our case, the first and last game are modular games parameterized by packages which aligns computation model and security notion.

Acknowledgments. We are greatful to Pihla Karanko, Markulf Kohlweiss, Kirthivaasan Puniamurthy, Luisa Zeppelin, and the participants of the *Advanced Topics in Cryptography* course 2021 at Aalto University for useful suggestions on the presentation. We thank François Dupressoir for insightful discussions about the EasyCrypt security proof of Yao's garbled circuits in [ABB+17].

Sabine Oechsner was supported by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 669255 (MPCPRO), the Concordium Blockhain Research Center, Aarhus University, Denmark, and the Danish Independent Research Council under Grant-ID DFF-8021-00366B (BETHE). This work was supported by the Blockchain Technology Laboratory at the University of Edinburgh and funded by Input Output Global and the Academy of Finland.

References

ABB+17. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. A fast and verified software stack for secure function evaluation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1989–2006. ACM Press, October / November 2017.

AHR+21. Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. Ssprove: A foundational framework for modular cryptographic proofs in coq. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–15. IEEE, 2021.

BCK22. Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Key-schedule security for the TLS 1.3 standard. Cryptology ePrint Archive, Report 2021/137, 2022. to appear at S&P 2022, `https://eprint.iacr.org/2021/467`.

BDE+21. Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. Key-schedule security for the TLS 1.3 standard. Cryptology ePrint Archive, Report 2021/467, 2021. `https://eprint.iacr.org/2021/467`.

BDF+18. Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 222–249. Springer, Heidelberg, December 2018.

BDJ+21. David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. An interactive prover for protocol verification in the computational model. *IEEE Secur. Priv.*, 19, 2021.

BGHZ11. Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 71–90. Springer, Heidelberg, August 2011.

BHK13. Mihir Bellare, Viet Tung Hoang, and Sriram Keelveedhi. Instantiating random oracles via UCEs. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 398–415. Springer, Heidelberg, August 2013.

BHR12a. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 134–153. Springer, Heidelberg, December 2012.

BHR12b. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

Bla06. Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *2006 IEEE Symposium on Security and Privacy*, pages 140–154. IEEE Computer Society Press, May 2006.

BPW04. Michael Backes, Birgit Pfitzmann, and Michael Waidner. A general composition theorem for secure reactive systems. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 336–354. Springer, Heidelberg, February 2004.

BR06. Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

CHK21. Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1847–1864. USENIX Association, 2021.

CKKR19. Jan Camenisch, Stephan Krenn, Ralf Küsters, and Daniel Rausch. iUC: Flexible universal composability made simple. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part III*, volume 11923 of *LNCS*, pages 191–221. Springer, Heidelberg, December 2019.

DKO21. François Dupressoir, Konrad Kohbrok, and Sabine Oechsner. Bringing state-separating proofs to easycrypt - A security proof for cryptobox. *IACR Cryptol. ePrint Arch.*, page 326, 2021.

GGP10. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 465–482. Springer, Heidelberg, August 2010.

GKR08. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 39–56. Springer, Heidelberg, August 2008.

GLNP15. Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast garbling of circuits under standard assumptions. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 567–578. ACM Press, October 2015.

GS18. Sanjam Garg and Akshayaram Srinivasan. Adaptively secure garbling with near optimal online complexity. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 535–565. Springer, Heidelberg, April / May 2018.

HJO+16. Brett Hemenway, Zahra Jafargholi, Rafail Ostrovsky, Alessandra Scafuro, and Daniel Wichs. Adaptively secure garbled circuits from one-way functions. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 149–178. Springer, Heidelberg, August 2016.

HS15. Dennis Hofheinz and Victor Shoup. GNUC: A new universal composability framework. *Journal of Cryptology*, 28(3):423–508, July 2015.

HUM09. Dennis Hofheinz, Dominique Unruh, and Jörn Müller-Quade. Polynomial runtime and composability. Cryptology ePrint Archive, Report 2009/023, 2009. `https://eprint.iacr.org/2009/023`.

JO20. Zahra Jafargholi and Sabine Oechsner. Adaptive security of practical garbling schemes. In Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran, editors, *INDOCRYPT 2020*, volume 12578 of *LNCS*, pages 741–762. Springer, Heidelberg, December 2020.

JW16. Zahra Jafargholi and Daniel Wichs. Adaptive security of Yao's garbled circuits. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 433–458. Springer, Heidelberg, October / November 2016.

KKP21. Chethan Kamath, Karen Klein, and Krzysztof Pietrzak. On treewidth, separators and yao's garbling. *IACR Cryptol. ePrint Arch.*, page 926, 2021.

KS08. Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

KTR20. Ralf Küsters, Max Tuengerthal, and Daniel Rausch. The IITM model: A simple and expressive model for universal composability. *J. Cryptol.*, 33(4):1461–1584, 2020.

LP09. Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009.

Mau10. Ueli Maurer. Constructive cryptography - a primer (invited paper). In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, page 1. Springer, Heidelberg, January 2010.

Mau12. Ueli Maurer. Constructive cryptography - A new paradigm for security definitions and proofs. In Sebastian Mödersheim and Catuscia Palamidessi, editors, *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*, volume 6993 of *Lecture Notes in Computer Science*, pages 33–56. Springer, 2012.

MT13. Daniele Micciancio and Stefano Tessaro. An equational approach to secure multi-party computation. In Robert D. Kleinberg, editor, *ITCS 2013*, pages 355–372. ACM, January 2013.

NPS99. Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the First ACM Conference on Electronic Commerce (EC-99)*, pages 129–139, 1999.

PSSW09. Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, Heidelberg, December 2009.

RR21. Mike Rosulek and Lawrence Roy. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. LNCS, pages 94–124. Springer, Heidelberg, 2021.

Weg87. Ingo Wegener. *The complexity of Boolean functions*. BG Teubner, 1987.

Yao86. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

ZRE15. Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.