Symbolic security of garbled circuits *

Baiyu Li[†] Daniele Micciancio[‡]

April 30, 2018

Abstract

We present the first computationally sound symbolic analysis of Yao's garbled circuit construction for secure two party computation. Our results include an extension of the symbolic language for cryptographic expressions from previous work on computationally sound symbolic analysis, and a soundness theorem for this extended language. We then demonstrate how the extended language can be used to formally specify not only the garbled circuit construction, but also the formal (symbolic) simulator required by the definition of security. The correctness of the simulation is proved in a purely syntactical way, within the symbolic model of cryptography, and then translated into a concrete computational indistinguishability statement via our general computational soundness theorem. We also implement our symbolic security framework and the garbling scheme in Haskell, and our experiment shows that the symbolic analysis performs well and can be done within several seconds even for large circuits that are useful for real world applications.

1 Introduction

Secure computation protocols [Yao82, Yao86, GMW87, BGW88], showing that any function can be evaluated by two or more distrustful parties in a secure way, are a cornerstone of cryptography, and one of the most complex security problems ever envisioned and solved by cryptographers. The complexity of designing and analyzing (general) secure computation protocols stems in good part from the fact that they require the construction of not just a single security application, but of an entire class of applications, each described by a function specified in a (low level, but still general purpose) computational model, e.g., that of arbitrary Boolean circuits. So, in a sense, protocols for secure computation problems are not individual security applications, but *compilers* to translate specifications (e.g., circuits to be computed) to secure solutions, often to be validated with respect to a strong simulation-based definition of security. In fact, much work on the implementation of secure computation (e.g., see [BNP08, MNPS04, BLW08, Mal11]) takes the form of compilers and execution engines. In this paper, we focus on the two party secure computation problem and Yao's garbled circuits [Yao82, Yao86], the first, and still most popular (in its many variants) solution to this problem. Even disregarding implementation issues, it is indicative of the complexity of this problem, that the first proof of security for Yao's garbled circuit construction [LP09] appeared approximately 30 years after the protocol was originally proposed [Yao82, Yao86].

*Research supported in part by the National Science Foundation (NSF) under grant CNS-1528068. Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

[†]University of California, San Diego, USA. E-mail: baiyu@cs.ucsd.edu

[‡]University of California, San Diego, USA. E-mail: daniele@cs.ucsd.edu

Following a line of research initiated by Abadi and Rogaway [AR07], we consider the possibility of simplifying and formalizing the design and analysis of secure (two-party) computation protocols using a hybrid approach, consisting of the following steps:

- Setting up a symbolic execution model, which provides a simple language to describe (and analyze) cryptographic computations without all the details and complications of concrete (complexity based) computational models.
- Proving a general *computational soundness* result, showing that what can be proved symbolically in this abstract model of computation, also holds true when the symbolic language is instantiated with computational cryptographic functions satisfying standard (computational) notions of security.
- Prove that the protocol is secure in a purely symbolic/syntactical way, i.e., within the abstract model.
- Conclude, via the computational soundness theorem, that the standard implementation of the protocol (using a concrete, computational instantiation of the cryptographic primitives) satisfies the computational indistinguishability security properties expected by cryptographers, and demanded by actual applications.

The usefulness and viability of this computationally sound symbolic approach to security analysis has been investigated and demonstrated in a number of papers. Previous work includes foundational results [AR07, MW04, Mic10], and applications to a number of different settings, like key distribution protocols [MP06, Pan07, MP08], access control in XML databases [AW08], password guessing attacks [BWA10], and more.

The goal of this paper is to demonstrate the applicability of this attractive methodology to the analysis of secure computation protocols, and, specifically, Yao's protocol for secure two party computation. Perhaps surprisingly, we are able to show that a very simple extension of symbolic cryptography languages already considered in the past are sufficient to both model and analyze this type of protocols. While we focus on Yao's protocol in one of its simplest variants, we believe that there is a general lesson to be learned: computationally sound symbolic analysis can be a powerful tool to manage the complexity of high level cryptographic applications.

We believe that the use of these methods is not limited to the mechanic validation of protocols that are seemingly too complex to be checked by hand, but it can actually help to carry out the security analysis at a sufficiently high (still precise and computationally meaningful) level of abstraction, so that formal proofs can be validated (and, most importantly, understood) by humans. Further extensions of the language and techniques described in this paper may also offer a basis to study optimizations and extensions of Yao's basic protocol, and, perhaps, even the construction of verified optimizing compilers for secure computation that translate between different variants of cryptographic constructions, while at the same time checking that the transformations preserve both functionality and security.

Contributions and Technical Overview As outlined above, our goal is to describe Yao's garbled circuits by simple "symbolic" cryptographic expressions, e.g., expressions of the form $\{\!\{(\mathsf{K}_1, \{\!\{\mathsf{K}_2\}\!\}_{\mathsf{K}_1})\}\!\}_{\mathsf{K}_3}$, representing the encryption under key K_3 of a pair, consisting of a key K_1 and a random message K_2 encrypted under K_1. Here we are using the compact notation $\{\!\{m\}\!\}_k$, quite common in symbolic cryptography, to represent the encryption of m under k. (In this introduction we appeal on the reader's intuition to interpret the meaning of symbolic expressions, and refer to Section 2 for formal definitions.) It is important to note that expressions like $E = \{\!\{(\mathsf{K}_1, \{\!\{\mathsf{K}_2\}\!\}_{\mathsf{K}_1})\}\!\}_{\mathsf{K}_3}$ do not represent the result of running a set of encryption algorithms, but they are purely syntactical objects, and can be manipulated as such. Of course, expressions like these can also be mapped to probability distributions over bit-strings, once an appropriate encryption scheme has been chosen to implement $\{\!\!|\cdot|\!\!\}_k$, and random values are chosen for all the K_i symbols used in the expression. The resulting distribution is what an real adversary would see when the protocol is implemented and executed in practice.

A simple language of this type was suggested in the pioneering work of Abadi and Rogaway [AR07], which also showed how to map these expressions to symbolic patterns that capture the adversary's view or knowledge of the computation. E.g., the expression E described above could be mapped to the pattern $\{\!\!|\square|\!\!\}$, representing the fact that the adversary can tell this is a cipher-text, but nothing else because it does not know the encrypting key K_3. More realistically, this expression could be mapped to the pattern $\{\!\!|(\!|\mathbb{K}, \{\!\!|\mathbb{K}|\!\!\}|\!)|\!\!\}_{\mathsf{K}_3}$ to capture the fact that the standard notion of encryption does not hide the size of the message being encrypted (and, thereby, it may reveal information on the "structure" or "shape" $(\!|\mathbb{K}, \{\!\!|\mathbb{K}|\!\!\}|\!)$ of the payload,) and may also reveal partial information about the encryption key K_3. (Protecting the identity of the recipient key K_3 is an extra security feature, typically called "anonymous encryption".) Abadi and Rogaway [AR07] also proved a computational soundness result, showing that the symbolic notion of equivalence induced by these patterns (i.e., two expressions are equivalent if they map to the same symbolic pattern, possibly up to variable renaming), matches precisely the notion of computational indistinguishability (i.e., the probability distributions generated by the two expressions cannot be told apart by any efficient adversary), provided a certain technical condition of encryption cycles is met.

In this work, we follow the approach of [Mic10], which allows to bypass the key-cycles technicality by using a co-inductive definition of symbolic adversarial knowledge. The language of [AR07, Mic10] allows to use only (arbitrarily nested) encryption, but it has been extended in [Mic09] to provide a computationally sound treatment of pseudorandom generators. As a first contribution of this work, we further extend the language (and computational soundness results) of [AR07, Mic10, Mic09] allows to include also randomly chosen bits, and a *controlled-swap* operation $\pi[b](e_0, e_1)$ that randomly permutes $\{e_0, e_1\}$ depending on the value of the (randomly chosen) bit b. (This is described in Section 2.)

Next, we show how this simple extended language is enough to express Yao's garbling procedure in a purely symbolic way. This requires to describe a method to map arbitrary circuits to symbolic expressions, rather than simply providing a single expression or sequence of expressions (as used, for example, in a multi-step protocol.) In turn, this requires a good way to handle arbitrary circuits within symbolic computations. The way circuits are typically formalized (as an unstructured list of gates and wires, similar to representing a graph by unstructured sets of nodes and edges) is not very convenient. As a second contribution of this work, we propose an inductive method and syntax to describe circuits, where larger circuits are built in a modular way from smaller ones, starting from the basic case of single gates. (For simplicity, we consider only two types of gates: a NAND gate mapping two Boolean inputs to one output, and a "duplicate" gate mapping a single input to two identical outputs.) This modular description of circuits supports both the formal definition of circuit mapping functions, and associated proofs of security, by *structural induction*. We remark that this circuit description language is by no means new, and it is strongly inspired by similar ideas used in modern high level programming languages, like Hughes' arrows [Hug00, Hug04].

As a disclaimer, we should note that the arrow syntax used in this paper is a good match for the mathematical definition of circuits, and it is a convenient formalism to specify and analyze circuit-manipulating programs (like compilers for secure computation), but it is not necessarily intended as a user friendly method to specify computations. But alternative syntax to describe circuit/arrow computations in a programmer friendly way exist [Pat01, LWY10], it can be automatically translated into the mathematical (inductive) arrows notation, and it is readily found implemented in mainstream programming languages like Haskell to structure complex software libraries, like graphical user interfaces, robotics applications, hardware description languages, and more. So, we will not be concerned on the usability of the arrow notation to directly specify application circuits, and refer the interested reader to the programming language literature for more information.

What is more relevant, in the context of this paper, is that we are able to use our extended language for symbolic cryptography, and the structural arrow-like formalization of circuits, to give a formal, yet conceptually simple description of

- Yao's circuit garbling procedure,
- a symbolic simulator, used to prove the security of Yao's construction, and
- a detailed, formal proof showing that the output of Yao's garbling procedure and the output of the simulator, are symbolically equivalent, i.e., they map to equivalent symbolic patterns.

We remark that all these definitions and proofs are purely symbolic, and they work by induction on the structure of the circuits, reducing the security analysis to the verification of a small number of base cases and inductive steps. It follows from our computational soundness theorem that, when implemented using standard cryptographic primitives, the resulting construction achieves the standard security notion of computational indistinguishability used in cryptography.

It is important to note that the connection between symbolic security, and computational security is not established at the level of garbled circuits, but it is proved in the context of a general soundness theorem for a generic, simple language of cryptographic expressions. The language is designed to be powerful enough to express garbled circuits and the associated simulation procedure, but it is otherwise independent of the specific circuit garbling problem. We believe that this greatly simplifies and elucidates both the computational soundness result (which is proved for a simple, application independent language,) and the application to garbled circuits (which is described and analyzed in a purely symbolic manner.)

Other related work Since the detailed security proof of garbled circuits in [LP09], there have been many studies on various security properties of garbled circuits. For a recent summary see for example [BHR12b]. The security notion used in [LP09] is sometimes called *selective security*, in which an adversary must choose an input before the circuit is provided to the simulator. A more useful notion in practice is *adaptive security*, in which a simulator must be able to return a simulated garbled circuit back to the adversary given only the circuit, and the adversary can adaptively choose an input value after seeing the garbled circuit. There is a number of works that explore adaptive security of garbled circuits, for example [BHR12a,BGG+14,HJO+16]. Jafargholi and Wichs [JW16] showed that Yao's original construction of garbled circuits is already adaptively secure with a security loss of $2^{O(d)}$, where d is the circuit depth, and this result has been further generalized in [JKK+17]. As a first step toward the symbolic modeling of garbling schemes, in this paper, we focus on selective security.

Adaptive security in general can be solved by using the "erasure" approach [BH92] or by assuming non-standard primitives such as non-committing encryption [CFGN96]. In the symbolic setting, adaptive security with standard assumptions was considered in the past in the context of symmetric-key encryption protocols [Pan07]. That approach can be adapted to our symbolic model to deal with adaptive security of garbled circuits. But such extension may require a non-trivial amount of work and is beyond the scope of the current paper, so we leave it for future study.

Machine-checked proofs have been developed for cryptographic systems through several computer-aided verification tools such as CryptoVerif [Bla06], CertiCrypt [BGZB09], EasyCrypt [BGHB11], and so on. These tools apply formal methods in conjunction with cryptography-specific constructions, and they impose rigorous proof styles. In a recent work [ABB+17], Almeida et. al. formalized Yao's secure function evaluation protocol in which the circuit garbling scheme is a central component, and, among many things, it then devised a machine-checked selective security proof of the garbling scheme using EasyCrypt (with customized extensions to allow using hybrid arguments and simulation-based proofs). Comparing to our work, the construction and the security goal of the garbling scheme in their work is similar, but their mechanized proofs argue computational security directly in the logic system of EasyCrypt, which are different from the symbolic style proofs in our work.

Paper organization The rest of the paper is organized as follows. In Section 2 we provide formal definitions for symbolic cryptography, background on computational soundness, and our extended symbolic language (and computational soundness theorem) to describe garbled circuits. Our inductive method to define circuits is presented in Section 3. In Section 4, we use our language of symbolic cryptography and the structural definition of circuits, to give a formal description of Yao's circuit garbling procedure. Section 5 contains the main results of the paper, with the description of a symbolic simulator, and a formal proof that it is (symbolically) equivalent to real garbled circuit computations. Computational security of garbled circuits, as described in this paper, automatically follows from the general soundness results given in Section 2. In Section 6, we report our implementation of the symbolic garbling procedure and the simulator, and we provide some experimental results on automated testings performed against our implementation. We conclude our paper in Section 7.

2 Preliminaries

In this section we introduce basic notation used by symbolic and computational cryptography. For a positive integer n, we write $[n] = \{1, \dots, n\}$. We use the bit 0 for the Boolean value *false*, and 1 for *true*. For $n \geq 1$, $\{0,1\}^n$ is the set of all Boolean vectors of length n. We can concatenate two Boolean vectors $x \in \{0,1\}^n$ and $y \in \{0,1\}^m$ to obtain $xy \in \{0,1\}^{n+m}$. For any $x \in \{0,1\}^n$, we can think x as a concatenation of n bits, written as $x = x_1 \cdots x_n$, where $x_1, \dots, x_n \in \{0,1\}$. For any $x, y \in \{0,1\}$, the NAND function $x \uparrow y = \neg(x \wedge y)$ maps x and y to 0 if and only if both x and y are 1.

2.1 Symbolic cryptography

Our symbolic cryptographic expressions extend those defined in [Mic09] with random bits and a swap operation, which we need to model garbled circuits. Informally, symbolic expressions are built from random keys and (possibly random) bits, using a symmetric encryption scheme, a (length doubling) pseudorandom generator, a pairing (concatenation) operation, and the (random) permutation of pairs. Just as in computational cryptography it is convenient to group bit-strings according to their length, in symbolic cryptography it is customary to classify expressions according to their *shape*, which captures the expression size in a representation independent way. The set of possible shapes for a symbolic expression is defined by the grammar:

$$\mathbf{Shape} \to \mathbb{B} \mid \mathbb{K} \mid (\!|\mathbf{Shape}, \mathbf{Shape}|\!) \mid \{\!|\mathbf{Shape}|\!\}$$

representing the shapes of bits, keys, pairs (of two sub-expressions of arbitrary shape), and encryptions (of messages of arbitrary shape), respectively. For example $(\!|\mathbb{K}, \{\!|\mathbb{B}|\!\}|\!)$ is the shape of a pair consisting of a key and the encryption of a single bit message. Let $\mathbf{B} = \{\mathsf{B}_i \mid i = 1, 2, \dots\}$ be a set of atomic bit symbols, and $\mathbf{K} = \{\mathsf{K}_i \mid i = 1, 2, \dots\}$ a set of atomic key symbols, representing independent uniformly random bits and independent uniformly random keys, respectively. For any shape $s \in \mathbf{Shape}$, we define a corresponding set of expressions of shape s (denoted $\mathbf{Exp}(s)$) according to the grammar rules:

$$
\begin{aligned}
\mathbf{Exp}(\mathbb{B}) &\to 0 \mid 1 \mid \mathsf{B}_i \mid \neg\mathbf{Exp}(\mathbb{B}) \\
\mathbf{Exp}(\mathbb{K}) &\to \mathsf{K}_i \mid \mathsf{G}_0(\mathbf{Exp}(\mathbb{K})) \mid \mathsf{G}_1(\mathbf{Exp}(\mathbb{K})) \\
\mathbf{Exp}(\{\!|s|\!\}) &\to \{\!|\mathbf{Exp}(s)|\!\}_{\mathbf{Exp}(\mathbb{K})} \\
\mathbf{Exp}((\!|s,t|\!)) &\to (\mathbf{Exp}(s), \mathbf{Exp}(t)) \\
\mathbf{Exp}((\!|s,s|\!)) &\to \pi[\mathbf{Exp}(\mathbb{B})](\mathbf{Exp}(s), \mathbf{Exp}(s)).
\end{aligned}
$$

where s, t range over $\mathbf{Shape}$, B_i ranges over $\mathbf{B}$, and K_i ranges over $\mathbf{K}$. Most symbols are self explanatory: $\neg b$ represents the logical negation of bit b, $(\mathsf{G}_0(k), \mathsf{G}_1(k))$ represents the output of a length doubling pseudorandom generator on seed k (with $\mathsf{G}_0(k)$ the first half of the output, and $\mathsf{G}_1(k)$ the second half,) $\{\!|e|\!\}_k$ is the encryption of e under key k, (e_0, e_1) is the ordered pair with sub-expressions e_0 and e_1, and for any bit b and expressions e_0, e_1 of the same shape, $\pi[b](e_0, e_1)$ represents the pair (e_0, e_1) with the two components swapped if $b = 1$. For example, $\{\!|\mathsf{G}_0(\mathsf{K}_1)|\!\}_{\mathsf{G}_1(\mathsf{K}_1)}$ represents the encryption of the first half $\mathsf{G}_0(\mathsf{K}_1)$ of a pseudorandom string (obtained by applying the pseudorandom generator on seed K_1,) encrypted under the second half of the pseudorandom string, while $\pi[\mathsf{B}_1](\mathsf{G}_0(\mathsf{K}_1), \mathsf{G}_1(\mathsf{K}_1))$ represents a pseudorandom string (output by the pseudorandom generator on seed K_1), with the first and second half of the string permuted (swapped) at random depending on the value of the (random) bit B_1.

Note that we can iteratively apply the pseudorandom generator on a key expression k to obtain expressions such as $\mathsf{G}_{b_1}(\mathsf{G}_{b_2}(\cdots(\mathsf{G}_{b_n}(k))))$ for $n \geq 0$ and $b_1, b_2, \ldots, b_n \in \{0, 1\}$. Such expressions are abbreviated as $\mathsf{G}_{b_1 b_2 \ldots b_n}(k)$. Let ε denote the empty bit-string, and let $\{0, 1\}^*$ denote the set of all bit-strings. For any set $S \subseteq \mathbf{Exp}(\mathbb{K})$, we define the sets

$$
\begin{aligned}
\mathsf{G}^*(S) &= \{\mathsf{G}_w(k) \mid k \in S, w \in \{0,1\}^*\} \\
\mathsf{G}^+(S) &= \{\mathsf{G}_w(k) \mid k \in S, w \in \{0,1\}^*, w \neq \varepsilon\}
\end{aligned}
$$

obtained by applying the (first or second half of the) pseudorandom generator zero (resp. one) or more times to a key in S. So, for example, $\mathsf{G}^*(\mathbf{K}) = \mathbf{Exp}(\mathbb{K})$ is the set of all (random or pseudorandom) keys. For convenience, we write $\mathbf{K}^*$ for $\mathsf{G}^*(\mathbf{K})$ and $\mathbf{K}^+$ for $\mathsf{G}^+(\mathbf{K})$. If $S = \{k\}$ is a singleton set, we usually write $\mathsf{G}^+(k)$ and $\mathsf{G}^*(k)$ instead of $\mathsf{G}^+(\{k\})$ and $\mathsf{G}^*(\{k\})$.

Patterns are extensions of expressions that include the construct $\{\!|s|\!\}_{\mathbf{Exp}(\mathbb{K})}$ to represent the encryption of an unknown expression of shape s. The pattern $\{\!|s|\!\}_{\mathbf{Exp}(\mathbb{K})}$ has shape $\{\!|s|\!\}$. Formally, patterns are defined by a grammar with variables $\mathbf{Pat}(s)$ indexed by $s \in \mathbf{Shape}$, and the same set of rules as those given for $\mathbf{Exp}(s)$, with the addition of one more rule

$$\mathbf{Pat}(\{\!|s|\!\}) \to \{\!|s|\!\}_{\mathbf{Exp}(\mathbb{K})}.$$

$\mathbf{Pat}(s)$ is the set of all patterns of shape s, and $\mathbf{Pat}$ is the set of all patterns (of any shape). Notice that $\mathbf{Pat}(\mathbb{B}) = \mathbf{Exp}(\mathbb{B})$ and $\mathbf{Pat}(\mathbb{K}) = \mathbf{Exp}(\mathbb{K})$ because only encryption gives raise to nontrivial patterns.

Computational evaluation Throughout this paper we let κ be the security parameter for cryptographic primitives in the computational setting. For simplicity, all keys are assumed to have length κ. We use $\mathsf{negl}(\kappa)$ to denote an arbitrary negligible function of κ, i.e., $\mathsf{negl}(\kappa) < 1/\kappa^c$ for any constant $c > 0$ and sufficiently large κ. To instantiate our symbolic framework, we assume the existence of a length-doubling pseudorandom generator $\mathcal{G}$ and an IND-CPA secure symmetric encryption scheme $(\mathcal{E}, \mathcal{D})$ with keys of length κ.

Definition 1 (Pseudorandom generator). *A deterministic function $\mathcal{G} : \{0,1\}^\kappa \to \{0,1\}^{2\kappa}$ is a secure length-doubling pseudorandom generator if it can be computed in polynomial time and, for any PPT distinguisher $\mathcal{A}$ we have*

$$\left| \Pr_{s \leftarrow \{0,1\}^{2\kappa}} \{\mathcal{A}(s) = 1\} - \Pr_{r \leftarrow \{0,1\}^{\kappa}} \{\mathcal{A}(\mathsf{G}(r)) = 1\} \right| \le \mathsf{negl}(\kappa).$$

For any symmetric encryption scheme $(\mathcal{E}, \mathcal{D})$ and $b \in \{0,1\}$, the *left-right encryption oracle* $\mathcal{O}_{\mathcal{E},b}$ first samples a uniformly random key $k \leftarrow \{0,1\}^\kappa$, and then it answers any encryption query of the form (m_0, m_1) with a ciphertext $\mathcal{E}(k, m_b)$, where m_0 and m_1 are of the same length.

Definition 2 (IND-CPA secure symmetric encryption scheme). *A pair of PPT algorithms $(\mathcal{E}, \mathcal{D})$ is an* IND-CPA secure *symmetric encryption scheme with key length κ if the followings hold:*

- **Correctness:** *For any $k \in \{0,1\}^\kappa$ and $m \in \{0,1\}^*$,* $\Pr\{\mathcal{D}(k, \mathcal{E}(k,m)) = m\} = 1$;
- **Security:** *For any PPT distinguisher $\mathcal{A}$,*

  $$\left| \Pr\{\mathcal{A}^{\mathcal{O}_{\mathcal{E},0}}(1^\kappa) = 1\} - \Pr\{\mathcal{A}^{\mathcal{O}_{\mathcal{E},1}}(1^\kappa) = 1\} \right| \le \mathsf{negl}(\kappa),$$

  where the probability is over the random choices of $\mathcal{A}$.

We assume that the size of a cipher-text $\mathcal{E}(k,m)$ is a function of the size of the input m, i.e., if two messages have the same length, then their encryption also have the same length. We do not make any special assumption on the encoding of pairs (e_0, e_1), except that e_0 and e_1 can be recovered from (e_0, e_1), and that the size of (e_0, e_1) depends only on the size of e_0 and the size of e_1. For any $x \in \{0,1\}^\kappa$, let $\mathcal{G}_0(x)$ and $\mathcal{G}_1(x)$ be the first and second halves of the bit-string $\mathcal{G}(x)$, so that $\mathcal{G}(x) = \mathcal{G}_0(x)\mathcal{G}_1(x)$. Let σ be a function mapping **B** to $\{0,1\}$, and **K** to $\{0,1\}^\kappa$. We can extend σ to map any symbolic expression to a distribution on bit-strings as follows:

$$\begin{aligned}
&\sigma(0) = 0, && \sigma(1) = 1, \\
&\sigma(\mathsf{G}_0(k)) = \mathcal{G}_0(\sigma(k)), && \sigma(\neg b) = 1 - (\sigma(b)), \\
&\sigma(\mathsf{G}_1(k)) = \mathcal{G}_1(\sigma(k)), && \sigma(\{\!|e|\!\}_k) = \mathcal{E}(\sigma(k), \sigma(e)), \\
&\sigma((e_0, e_1)) = (\sigma(e_0), \sigma(e_1)), && \sigma(\pi[b](e_0, e_1)) = \begin{cases} (\sigma(e_0), \sigma(e_1)) & \text{if } \sigma(b) = 0 \\ (\sigma(e_1), \sigma(e_0)) & \text{if } \sigma(b) = 1 \end{cases}
\end{aligned}$$

where $k \in \mathbf{Exp}(\mathbb{K})$, and $b \in \mathbf{Exp}(\mathbb{B})$. The computational evaluation $[\![e]\!]$ of an expression e is defined as the probability distribution obtained by first choosing a uniformly random key and bit assignment σ, and then picking a sample from $\sigma(e)$.[1] It is easy to check (by induction) that any two expressions of the same shape evaluate to bit-strings of the same length.

Lemma 1. *For any shape s, all strings in $[\![\mathbf{Exp}(s)]\!]$ have the same bit-length.*

Using this property, we can associate a bit-length to any shape s as the bit-length $|s|$ of any string in the set $[\![\mathbf{Exp}(s)]\!]$, and extend the evaluation of expressions to evaluation of patterns by defining

$$\sigma(\{\!|s|\!\}_k) = \mathcal{E}(\sigma(k), 0^{|s|}).$$

[1]Notice that, even for fixed σ and e, the image $\sigma(e)$ is a probability distribution because it involves the use of a probabilistic encryption scheme $\mathcal{E}$.

Independence of pseudorandom keys The following definitions are given in [Mic09] to provide a (computationally sound) treatment of symbolic pseudorandom generators. For any two keys $k_1, k_2 \in \mathbf{K}^*$, if $k_2 \in \mathsf{G}^*(k_1)$ then we say that k_1 *yields* k_2, and denote this as $k_1 \preceq k_2$, meaning that k_2 can be obtained from k_1 by repeated application of the pseudorandom generator. By $k_1 \prec k_2$ we mean that $k_1 \preceq k_2$ and $k_1 \neq k_2$. We say that k_1 and k_2 are *independent* if neither $k_1 \preceq k_2$ nor $k_2 \preceq k_1$. The keys $\{k_1, \dots, k_n\}$ form an independent set if k_i and k_j are independent for all $i \neq j$. The root of any set of keys S is $\mathbf{Roots}(S) = S \setminus \mathsf{G}^+(S)$. Thus S is independent if and only if $S = \mathbf{Roots}(S)$. We recall the following theorem from [Mic09] which shows that independent symbolic keys correspond to (computational) pseudorandom bit-strings.

Theorem 1 ([Mic09, Theorem 1])**.** *Let $k_1, \dots, k_n \in \mathbf{K}^*$ be a sequence of symbolic keys. Then for any secure length-doubling pseudorandom generator $\mathcal{G}$, the following two conditions are equivalent:*

1. *The keys $k_1, \dots, k_n$ are symbolically independent (i.e., $k_i \preceq k_j$ if and only if $i = j$).*
2. *The probability distribution $[\![k_1, \dots, k_n]\!]$ is computationally indistinguishable from $[\![r_1, \dots, r_n]\!]$ where $r_1, \dots, r_n \in \mathbf{K}$ are distinct atomic key symbols.*

Equivalence and Renaming of patterns We consider patterns up to simple operations that do not change the probability distributions associated to them. First, let $\equiv$ be the smallest congruence relation on **Pat** such that

$$\begin{aligned} \neg 0 &\equiv 1 & \pi[0](e_0, e_1) &\equiv (e_0, e_1) \\ \neg 1 &\equiv 0 & \pi[1](e_0, e_1) &\equiv (e_1, e_0) \\ \neg(\neg b) &\equiv b & \pi[\neg b](e_0, e_1) &\equiv \pi[b](e_1, e_0) \end{aligned}$$

for all $e_0, e_1 \in \mathbf{Pat}(s)$, and $b \in \mathbf{Pat}(\mathbb{B})$. It should be clear from the computational interpretation of $\pi[b]$ and $\neg b$ that for any two equivalent patterns $e_0 \equiv e_1$ and any assignment σ, the probability distributions $\sigma(e_0)$ and $\sigma(e_1)$ are identical. Similarly, we define a *random bit renaming* as a function $\alpha_B : \mathbf{B} \to \{b, \neg b \mid b \in \mathbf{B}\}$ such that its projection $\alpha'_B : \mathbf{B} \to \mathbf{B}$ (defined by the condition $\alpha_B(b) \in \{\alpha'_B(b), \neg\alpha'_B(b)\}$) is a bijection on $\mathbf{B}$. Random bit renamings are extended to patterns $\alpha_B : \mathbf{Pat}(s) \to \mathbf{Pat}(s)$ in the obvious way, and it is easy to check that for any pattern $e \in \mathbf{Pat}(s)$ and assignment σ, the distributions $\sigma(e)$ and $\sigma(\alpha_B(e))$ are identical.

For keys, we consider a form of renaming that may change the distribution associated to an expression or pattern, but in a computationally indistinguishable way. Following [Mic09], we define a *pseudorandom key renaming* as a mapping $\alpha_K : S \to \mathbf{K}^*$ on $S \subseteq \mathbf{K}^*$ that preserves G, i.e.,

$$\mathsf{G}_w(k_1) = k_2 \qquad \Longleftrightarrow \qquad \mathsf{G}_w(\alpha_K(k_1)) = \alpha_K(k_2)$$

for all $w \in \{0,1\}^*$ and $k_1, k_2 \in S$. We restate some useful properties of key renamings proved in [Mic09]:

1. [Mic09, Lemma 1] Any pseudorandom key renaming $\alpha_K : S \to \mathbf{K}^*$ is a bijection from S to $\alpha_K(S)$. Moreover, S is independent if and only if $\alpha_K(S)$ is independent.
2. [Mic09, Lemma 2] Any pseudorandom key renaming α_K with domain S can be uniquely extended to a pseudorandom key renaming $\bar{\alpha}_K$ with domain $\mathsf{G}^*(S)$. In particular, any pseudorandom key renaming can be uniquely specified as an extension $\bar{\alpha}_K$ of a bijection $\alpha_K : A \to B$ between independent sets $A = \mathbf{Roots}(S)$ and $B = \alpha_K(A)$.
3. [Mic09, Lemma 5] For any pseudorandom key renaming $\alpha_K : S \to \mathbf{K}^*$ and set of keys $A \subseteq S$, $\alpha_K(\mathbf{Roots}(A)) = \mathbf{Roots}(\alpha_K(A))$.

$$\mathbf{p}(b, S) = b \qquad \mathbf{p}((e_0, e_1), S) = (\mathbf{p}(e_0, S), \mathbf{p}(e_1, S))$$

$$\mathbf{p}(k, S) = k \qquad \mathbf{p}(\pi[b](e, e_0), S) = \pi[b](\mathbf{p}(e, S), \mathbf{p}(e_0, S))$$

$$\mathbf{p}(\{\!|s|\!\}_k, S) = \{\!|s|\!\}_k \qquad \mathbf{p}(\{\!|e|\!\}_k, S) = \begin{cases} \{\!|\mathbf{p}(e, S)|\!\}_k & \text{if } k \in S \\ \{\!|s|\!\}_k & \text{if } k \notin S \end{cases}$$

Figure 1: The pattern function $\mathbf{p} : \mathbf{Pat} \times \wp(\mathbf{Pat}(\mathbb{K})) \to \mathbf{Pat}$, defined for all $b \in \mathbf{Exp}(\mathbb{B})$, $k \in \mathbf{Exp}(\mathbb{K})$, $e, e_0 \in \mathbf{Exp}(s)$, $e_1 \in \mathbf{Exp}(t)$

Pseudorandom key renamings α_K can also be extended to patterns $\alpha_K : \mathbf{Pat}(s) \to \mathbf{Pat}(s)$ in the obvious way, and while the distributions $\sigma(e)$ and $\sigma(\alpha_K(e))$ may, in general be different, they are always computationally indistinguishable.

The following lemma is an easy consequence of Theorem 1, and, despite the fact that we use a larger class of expressions, the proof is virtually identical to that of [Mic09, Corollary 1]. For completeness, a formal proof can be found in the appendix.

Lemma 2. *For any pattern e and pseudorandom key renaming α_K, the distributions $[\![e]\!]$ and $[\![\alpha_K(e)]\!]$ are computationally indistinguishable.*

We refer to a pair of mappings $\alpha = (\alpha_B, \alpha_K)$ (consisting of a random bit renaming α_B and a pseudorandom key renaming α_K) as a *pseudorandom renaming*, or simply a *renaming*. For any pattern $e \in \mathbf{Pat}(s)$, we write $\alpha(e) = \alpha_K(\alpha_B(e)) = \alpha_B(\alpha_K(e))$ for the result of applying the renamings to the pattern e.[2] Two patterns e_0 and e_1 are *equivalent up to renaming*, denoted as $e_0 \approx e_1$, if there exists a renaming $\alpha = (\alpha_B, \alpha_K)$ such that $e_0 \equiv \alpha(e_1)$. When we want to emphasize the renaming α, we write $e_0 \approx_\alpha e_1$. It follows from the previous statements that patterns that are equivalent up to renaming evaluate to probability distributions that are computationally indistinguishable.

Pattern computation Following [Mic10], the mapping from expressions to patterns is defined by two functions:

- A function $\mathbf{p}(e, S)$ mapping an expression (or pattern) e and set of keys $S \subseteq \mathbf{K}^*$ to the pattern representing the view of e to an adversary that can decrypt under (all and only) the keys in S.
- A function $\mathbf{r}(p)$ mapping a pattern p to a corresponding set of keys, which may be recoverable by an adversary that sees all the parts of p.

The definition of these functions is virtually identical to the one given in [Mic09] for expressions with pseudorandom keys, extended with an additional case for our "controlled swap" expressions. Informally, $\mathbf{p}(e, S)$ replaces all subexpressions of e of the form $\{\!|e'|\!\}_k$ for some $k \notin S$ and $e' \in \mathbf{Pat}(s)$, with the pattern $\{\!|s|\!\}_k$. The formal definition is given in Fig. 1.

The formal definition of $\mathbf{r}$ is more technical, and uses the auxiliary functions **Keys** and **Parts** describing the keys and parts of an expression given in Fig. 2. As a matter of notation, for any two expressions e' and e, we say that e' is a *sub-expression* of e, denoted as $e' \Subset e$, if $e' \in \mathbf{Parts}(e)$. Notice that encryption keys k are not considered sub-expressions of $\{\!|e|\!\}_k$, as, even an adversary with unlimited decryption capabilities

[2]Notice that the mappings α_B and α_K commute, so they can be applied in any order.

$$
\begin{aligned}
\mathbf{Keys}(b) &= \emptyset & \mathbf{Parts}(b) &= \{b\} \\
\mathbf{Keys}(k) &= \{k\} & \mathbf{Parts}(k) &= \{k\} \\
\mathbf{Keys}(\{\!|e|\!\}_k) &= \{k\} \cup \mathbf{Keys}(e) & \mathbf{Parts}(\{\!|e|\!\}_k) &= \{\{\!|e|\!\}_k\} \cup \mathbf{Parts}(e) \\
\mathbf{Keys}(\{\!|s|\!\}_k) &= \{k\} & \mathbf{Parts}(\{\!|s|\!\}_k) &= \{\{\!|s|\!\}_k\} \\
\mathbf{Keys}((e_0, e_1)) &= \mathbf{Keys}(e_0) \cup \mathbf{Keys}(e_1) & \mathbf{Parts}((e_0, e_1)) &= \{(e_0, e_1)\} \cup \mathbf{Parts}(e_0) \cup \mathbf{Parts}(e_1) \\
\mathbf{Keys}(\pi[b](e_0, e_1)) &= \mathbf{Keys}(e_0) \cup \mathbf{Keys}(e_1) & \mathbf{Parts}(\pi[b](e_0, e_1)) &= \{\pi[b](e_0, e_1)\} \cup \mathbf{Parts}((e_0, e_1))
\end{aligned}
$$

Figure 2: The definition of the keys and parts of a sub-expression. As usual $b \in \mathbf{Exp}(\mathbb{B})$, $k \in \mathbf{Exp}(\mathbb{K})$.

cannot, in general, recover k from $\{\!|e|\!\}_k$. Informally, $\mathbf{r}(e)$ is defined as the set of all keys that can be potentially recovered from $\mathbf{Parts}(e)$. In [Mic09], this is defined using a general framework to model partial information in symbolic security analysis. For simplicity, here we only give the definition specialized to our class of expressions.

Definition 3. *For any $e \in \mathbf{Pat}$, we define the key recovery function $\mathbf{r} : \mathbf{Pat} \to \wp(\mathbf{Pat}(\mathbb{K}))$ as follows:*

$$
\mathbf{r}(e) = \mathsf{G}^*\left(\{k \in \mathbf{Keys}(e) \mid (k \Subset e) \vee (\exists k' \in \mathbf{Keys}(e).k \prec k')\}\right)
$$

Informally, $\mathbf{r}(e)$ contains all keys k from $\mathbf{Keys}(e)$ (and pseudorandom keys that can be derived from k) such that either k appears in e as a sub-expression, or k is related to some other key in $\mathbf{Keys}(e)$. The intuition behind this definition is that the adversary can learn a key k either by reading it directly from the parts of e, or by combining different pieces of partial information about k. We refer the reader to [Mic09] for further discussion and justification of this definition.

One can check by induction that the following commutative properties hold for $\mathbf{p}$ and $\mathbf{r}$: For any pattern $e \in \mathbf{Pat}$, set of keys $S \subseteq \mathbf{K}^*$, and pseudorandom renaming α, we have $\alpha(\mathbf{p}(e, S)) = \mathbf{p}(\alpha(e), \alpha(S))$, and $\alpha(\mathbf{r}(e)) = \mathbf{r}(\alpha(e))$.

Computational soundness We can now return to the framework of [Mic10] to associate computationally sound symbolic patterns to cryptographic expressions. The functions $\mathbf{p}$ and $\mathbf{r}$ are used to define, for any $e \in \mathbf{Pat}$, a key recovery operator

$$
\mathcal{F}_e(S) = \mathbf{r}(\mathbf{p}(e, S))
$$

mapping any set of keys $S \subseteq \mathsf{G}^*(\mathbf{K})$, to the set of keys potentially recoverable by an adversary that is capable of decrypting under the keys in S. This operator is used in [Mic10] to prove the following general computational soundness result.

Theorem 2 ([Mic10, Theorem 1]). *Assume the functions $\mathbf{p}$, $\mathbf{r}$ satisfy the following properties:*

1. $\mathbf{p}(e, \mathbf{K}^*) = e$
2. $\mathbf{p}(\mathbf{p}(e, S), T) = \mathbf{p}(e, S \cap T)$ *for all* $S, T \subseteq \mathbf{K}^*$
3. $\mathbf{r}(\mathbf{p}(e, T)) \subseteq \mathbf{r}(e)$ *for all* $T \subseteq \mathbf{K}^*$
4. *The distributions* $[\![e]\!]$ *and* $[\![\mathbf{p}(e, \mathbf{r}(e))]\!]$ *are computationally indistinguishable.*

Then, the key recovery operator $\mathcal{F}_e$ has a (unique) greatest fixed point $\mathrm{Fix}(\mathcal{F}_e) = \cap_{i>0}\mathcal{F}_e^{(i)}(\mathbf{K}^*)$*, and the pattern*

$$\mathbf{Pattern}(e) = \mathbf{p}(e, \mathrm{Fix}(\mathcal{F}_e))$$

is computationally sound, in the sense that $[\![\mathbf{Pattern}(e)]\!]$ *and* $[\![e]\!]$ *are computationally indistinguishable distributions.*

One can check that the functions $\mathbf{p}$ and $\mathbf{r}$ satisfy all the conditions 1 to 3 in Theorem 2. For the last condition, the following lemma shows that $[\![e]\!]$ and $[\![\mathbf{p}(e, \mathbf{r}(e))]\!]$ are indistinguishable for all patterns e. The proof is omitted due to space constraint. Using the soundness theorem of the general symbolic framework of [Mic10] we can then conclude that our symbolic semantics is computationally sound.

Lemma 3. *For any $e \in$ **Pat**, the probability distributions $[\![e]\!]$ and $[\![\mathbf{p}(e, \mathbf{r}(e))]\!]$ are computationally indistinguishable.*

Recall that renamings commute with the pattern function $\mathbf{p}$, i.e., for any expression e and for any set of keys $S \subseteq \mathbf{K}^*$, $\mathbf{p}(\alpha(e), \alpha(S)) = \alpha(\mathbf{p}(e, S))$. It follows that $\mathbf{Pattern}(\alpha(e)) = \alpha(\mathbf{Pattern}(e))$, and therefore we can extend the computational soundness theorem to pattern equivalence up to renaming. That is, for any two expressions e_1 and e_2, symbolic equivalence (up to pseudorandom renaming) of their patterns $\mathbf{Pattern}(e_1)$ and $\mathbf{Pattern}(e_2)$ implies that the two probability distributions $[\![e_1]\!]$ and $[\![e_2]\!]$ are computationally indistinguishable.

Theorem 3. *For any two symbolic expressions e_0, e_1, if $\mathbf{Pattern}(e_0) \approx \mathbf{Pattern}(e_1)$, then $[\![e_0]\!]$ and $[\![e_1]\!]$ are computationally indistinguishable.*

3 Inductive Circuits

Traditionally, boolean circuits are described by two sets of gates $\{g_i\}_{i=1}^q$ and wires $\{w_i\}_{i=1}^p$ and a description of how they are connected together. Each wire carries a boolean value, that is either given as part of the input to the circuit, or is computed by a gate. Each gate is associated to a number of input and output wires, and sets the value of the output wires to some fixed function of the values of the input wires. For simplicity, we consider circuits using just two types of gates:

- a NAND gate that on input two boolean values x_0, x_1, computes the output $y = x_0 \uparrow x_1$, and
- a DUP gate, which duplicates the value on its single input wire x to its two output wires $y_0 = y_1 = x$.

The NAND function itself is complete for the set of all boolean functions, and the DUP gate can be used to implement arbitrary fan-out. So any boolean circuit can be converted to this notation. A circuit with n input wires and m output wires computes a boolean function $f : \{0,1\}^n \to \{0,1\}^m$.

This traditional formalization of circuits is completely unstructured, making it inconvenient to use in symbolic constructions and proofs of security. Below we present an alternative way to describe boolean circuits, which is inductive (larger circuits are built from smaller ones), and supports definitions and proofs by structural induction.

We begin by putting some structure on the set of input and output wires of a circuit, by defining the notion of a *wire bundle*. Informally, the shape of a wire bundle is defined by a well parenthesized expression like $(\circ, (\circ, \circ))$. Formally, we can define bundle to be either a single wire (represented by the symbol $\circ$), or an ordered pair (u, v) where u and v are wire bundles. The size of a bundle is simply the number of wires in it, i.e., the number of $\circ$ subexpressions. Each wire $\circ$ carries a bit $b \in \{0, 1\}$, and a bundle of n wires naturally carries a bit vector in $\{0, 1\}^n$, but the additional bundle structure will give us easier access to individual bits, without having to index them. We remark that the grouping of wires is not associative, i.e., $((u, v), w)$ is different from $(u, (v, w))$.

We define circuits inductively, specifying a number of basic circuits, and some general operations to combine them together. Each circuit takes as input a bundle of wires, and produces as output another bundle. The set of circuits with input shape s and output shape t is denoted by $\mathrm{Circuit}(s, t)$. Circuits, their inputs and outputs, and the functions they compute, are formally specified in the following definition, with the base and inductive cases illustrated in Fig. 3 and 4.

Definition 4. *A* circuit *is either a basic circuit from the set* $\{\mathbf{Swap}, \mathbf{Assoc}, \mathbf{Unassoc}, \mathbf{Dup}, \mathbf{NAnd}\}$*, or it is a composite circuit built using operations* $\ggg$ *and* $\mathbf{First}$*. The semantics of basic circuits are:*

- $\mathbf{Swap}$ *consumes wires* (u, v) *and produces wires* (v, u)*.*
- $\mathbf{Assoc}$ *consumes wires* $(u, (v, w))$ *and produces wires* $((u, v), w)$*.*
- $\mathbf{Unassoc}$ *consumes wires* $((u, v), w)$ *and produces wires* $(u, (v, w))$*.*
- $\mathbf{Dup}$ *consumes a single wire* w *and produces wires* (w, w)*.*
- $\mathbf{NAnd}$ *consumes wires* (u, v)*, where* u *and* v *are single wires carrying bits* x *and* y*, and its output is a single wire that carries the bit* $x \uparrow y$*.*

For composite circuits, assume C_0 *is a circuit that takes* u *as input wires and produces output wires* w*, and* C_1 *a circuit that takes* w *as input wires and produces output wires* v*. Then*

- $C_0 \ggg C_1$ *is a circuit that takes input* u *and produces output* v*, obtained by first applying* C_0 *on* u *to get an intermediate result* w*, and then applying* C_1 *on* w *to get* v*.*
- $\mathbf{First}(C_0)$ *is a circuit that takes input wires* (u, u') *and produces output wires* (w, u') *for any wires* u'*, where* w *is the output of* C_0 *on input* u*, and* u' *is left unchanged by the circuit.*

To evaluate a circuit, we define the function $\mathbf{Ev}(C, w)$ that takes a circuit $C \in \mathrm{Circuit}(s, t)$ and a wire bundle w of shape s, and return a bundle of shape t according to the above semantics. For simplicity, we usually just write $C(x)$ for the boolean value carried on the wires $u = \mathbf{Ev}(C, w)$ where x is the value carried on w.

We remark that the circuit concatenation operation $\ggg$ is associative, i.e., $(C_0 \ggg C_1) \ggg C_2$ and $C_0 \ggg (C_1 \ggg C_2)$ produce the same circuit. So, we may omit the parentheses when writing a sequence of concatenations $C_0 \ggg C_1 \ggg C_2$.

For a circuit C, we say that C' is a *sub-circuit* of C if one of the following holds:

- $C' = C$, or
- $C = C_0 \ggg C_1$ and C' is a sub-circuit of C_0 or C_1, or
- $C = \mathbf{First}(C_0)$ and C' is a sub-circuit of C_0.

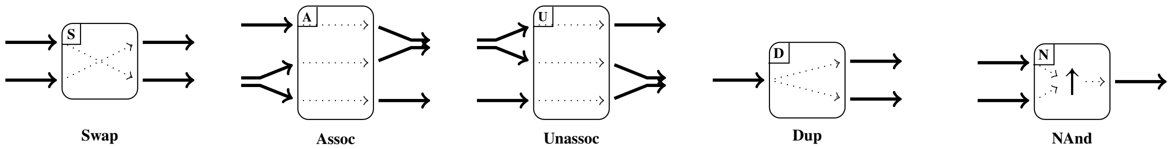

Figure 3: The atomic circuits **Swap**, **Assoc**, **Unassoc**, **Dup**, and **NAnd**. The dotted lines indicate how values are transferred from input wires to output wires. For **Swap**, **Assoc**, and **Unassoc**, an arrow may represent a bundle of more than one wires.

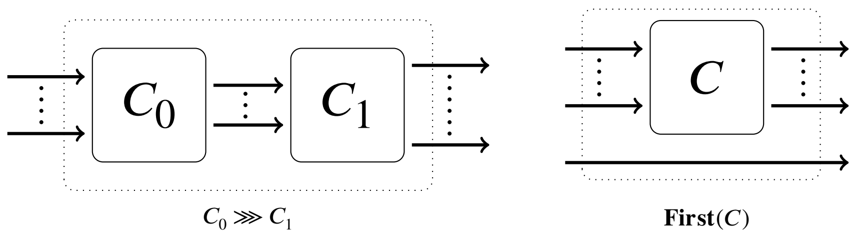

Figure 4: Composite circuits $C_0 \ggg C_1$ and **First**(C) using operations $\ggg$ and **First** on circuits C_0, C_1, C. Dotted lines draw the boundaries of composite circuits.

Example 1. To illustrate our circuit notation, consider the function $f((x, y), z) = (x \wedge y, y \rightarrow z)$, where $y \rightarrow z \equiv \neg y \vee z$ is the logical implication operation. First we define an operation **Second** on circuits such that **Second**(C) is a circuit that takes as input a wire bundle (u, v) and produces as output a bundle (u, w), where v is the input of C and w is the output of C:

$$\textbf{Second}(C) = \textbf{Swap} \ggg \textbf{First}(C) \ggg \textbf{Swap}$$

Since $x \uparrow x = \neg x$, the circuit $\textbf{Not} = \textbf{Dup} \ggg \textbf{NAnd}$ computes the negation of an input bit, and the circuit $\textbf{And} = \textbf{NAnd} \ggg \textbf{Not} = \textbf{NAnd} \ggg \textbf{Dup} \ggg \textbf{NAnd}$ computes the function $(x, y) \mapsto (x \wedge y)$. Since $y \rightarrow z = (\neg y) \vee z = y \uparrow (\neg z)$, the circuit $\textbf{Imp} = \textbf{Second}(\textbf{Not}) \ggg \textbf{NAnd}$ computes the function $(y, z) \mapsto (y \rightarrow z)$. Putting them together, we obtain a circuit

$$\begin{aligned} C = \ & \textbf{First}(\textbf{Second}(\textbf{Dup}) \ggg \textbf{Assoc}) \ggg \textbf{Unassoc} \\ & \ggg \textbf{First}(\textbf{And}) \ggg \textbf{Second}(\textbf{Imp}) \end{aligned}$$

for the function $f((x, y), z) = (x \wedge y, y \rightarrow z)$, illustrated graphically in Fig. 5. Notice how the first part of the computation consisting of the **Dup**, **Assoc** and **Unassoc** gates is used to route the input wires to the appropriate subcircuit.

Remark 1. With our circuit notation, a circuit with q gates and p wires can be represented using a string of size $O(qd \log q)$, where d is the depth of the circuit. We can convert the traditional DAG-like circuit notation to our inductive circuit representation by organizing gates into layers according to their depth. For a layer with q_i gates, the computation of these gates can be described using $q_i \log q_i$ many **First** and **Second** operations together with q_i basic circuits. To rearrange wires after a layer of q_i gates, we can add $O(q_i \log q_i)$ many **Swap**, **Assoc**, and **Unassoc** gates. The entire circuit can be concatenated from layers using $\ggg$ operations. So the size of such representation is $O(qd \log q)$.

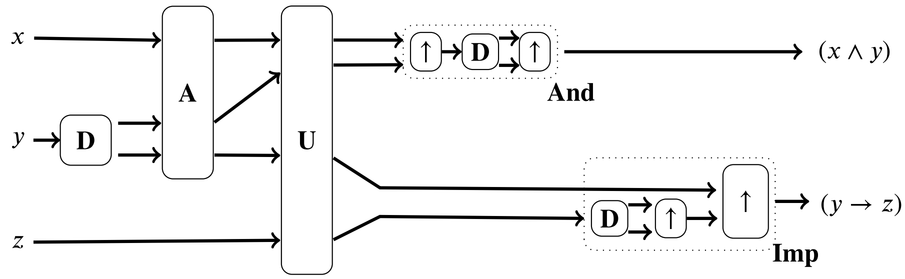

Figure 5: The circuit that computes the function $f((x, y), z) = (x \wedge y, y \to z)$.

4 Symbolic Garbling

Let us first recall the definition of circuit garbling schemes in the computational setting [LP09, BHR12a].

Definition 5 (Syntax)**.** *A* garbling scheme *is defined by a pair of PPT algorithms* ($\mathtt{Garble}, \mathtt{GEval}$)[3] *where*

- $\mathtt{Garble}(C, x) = (\widetilde{C}, \tilde{x})$*: The circuit garbling algorithm takes a circuit C and a boolean vector x as input, and it produces a garbled circuit $\widetilde{C}$ and a garbled input $\tilde{x}$.*

- $\mathtt{GEval}(\widetilde{C}, \tilde{x}) = y$*: The garbled circuit evaluation algorithm takes a garbled circuit $\widetilde{C}$ and a garbled input $\tilde{x}$ as input, and it produces a boolean vector y as output.*

Definition 6 (Correctness and security)**.** *For a garbling scheme* ($\mathtt{Garble}, \mathtt{GEval}$)*, we say that*

- *it is* correct *if* $\mathtt{GEval}(\mathtt{Garble}(C, x)) = C(x)$ *for all circuits C and boolean vectors x;*

- *it is* (selectively) secure *if there exists a PPT simulator* $\mathtt{Simulate}(\cdot, \cdot)$ *such that for any circuit C and input x, the distributions* $\mathtt{Simulate}(C, C(x))$ *and* $\mathtt{Garble}(C, x)$ *are computationally indistinguishable.*

Strictly speaking, a simulator should not gain access to a circuit, and instead, it should take the topology of a circuit as input. To simplify discussion, we use the actual circuit as its topology representation rather than introducing new notations. This can be justified by the facts that 1) there is only one primitive gate in our circuit notation, namely the NAND gate, and 2) our simulator (defined later) does not exploit the function computed by the NAND gate.

Symbolic garbled circuit We consider garbling schemes where the output of all algorithms `Garble`, `GEval`, and `Simulate` are expressions in our symbolic language **Exp**. This will allow us to analyze both the correctness and security properties of the scheme in a purely symbolic manner, without resorting to the power (and complications) of the full computational model of cryptography. The circuit garbling construction described here is essentially the one with the point-and-permute technique as described in [BMR90]. In this section we present `Garble` and `GEval`, and we will define `Simulate` and prove security in the next section.

[3] Usually a garbling scheme consists of three algorithms ($\mathtt{GCircuit}, \mathtt{GInput}, \mathtt{GEval}$) such that $\mathtt{GCircuit}(C) = (\widetilde{C}, L)$ produces a garbled circuit $\tilde{C}$ and labels L for the input wires, and $\mathtt{GInput}(L, x) = \tilde{x}$ produces garbled input $\tilde{x}$ using the labels. Such a syntax is useful to define adaptive security. However, we choose a simplified syntax of two algorithms that is sufficient to define selective security and convenient for our analysis.

Let ϵ denote a special symbolic expression whose computational evaluation is the empty string. We slightly change the notation of atomic key symbols by using both subscripts and superscripts to index them: an atomic key is a symbol K_i^j where $i \in \{1, 2, \ldots\}$ and $j \in \{0, 1\}$. With this notation, the set of atomic keys is now $\mathbf{K} = \{\mathsf{K}_1^0, \mathsf{K}_1^1, \mathsf{K}_2^0, \mathsf{K}_2^1, \ldots\}$. To hide the input of a circuit, the garbling algorithm encodes values carried on wires using *labels* of shape $(\!| \mathbb{B}, (\!| \mathbb{K}, \mathbb{K} |\!) |\!)$, one for each wire. We call a bundle of labels a *label expression*.

Formally, we first define a function `Label` that on input a bundle shape s, outputs a collection of wire labels:

$$\begin{aligned}
&\texttt{Label}(\circ) = (\mathsf{B}_h, (\mathsf{K}_h^0, \mathsf{K}_h^1)) \text{ where} \\
&\quad h \leftarrow \textbf{new} \\
&\texttt{Label}((s, t)) = (\texttt{Label}(s), \texttt{Label}(t))
\end{aligned}$$

The instruction $h \leftarrow \textbf{new}$ picks a fresh index h (e.g., using a counter), used to define a new symbolic label $(\mathsf{B}_h, (\mathsf{K}_h^0, \mathsf{K}_h^1))$.

A garbled input has two parts: an encoded input expression that is a bundle of shape $(\!| \mathbb{B}, \mathbb{K} |\!)$, and an output mask expression that is a bundle of bits. The function `GEnc` encodes a boolean vector using bits and keys in a label expression:

$$\begin{aligned}
&\texttt{GEnc}((\mathsf{B}, (\mathsf{K}^0, \mathsf{K}^1)), 0) = (\mathsf{B}, \mathsf{K}^0) \\
&\texttt{GEnc}((\mathsf{B}, (\mathsf{K}^0, \mathsf{K}^1)), 1) = (\neg \mathsf{B}, \mathsf{K}^1) \\
&\texttt{GEnc}((L_0, L_1), (x_0, x_1)) = (\texttt{GEnc}(L_0, x_0), \texttt{GEnc}(L_1, x_1))
\end{aligned}$$

The output masks are used to decode an encoded expression. It is formed by the bits in a label expression:

$$\begin{aligned}
&\texttt{GMask}((\mathsf{B}, (\mathsf{K}^0, \mathsf{K}^1))) = \mathsf{B} \\
&\texttt{GMask}((L_0, L_1)) = (\texttt{GMask}(L_0), \texttt{GMask}(L_1))
\end{aligned}$$

The core of the garbling algorithm is a recursive function `Gb`, which takes as input a circuit and a label expression for the input wires, and outputs a symbolic expression of the garbled circuit and a label expression for the output wires.

$$\begin{aligned}
&\texttt{Gb} :: \mathrm{Circuit}(s, t) \times \mathbf{Exp} \to \mathbf{Exp} \times \mathbf{Exp} \\
&\texttt{Gb}(\mathbf{Swap}, (u, v)) = \epsilon, (v, u) \\
&\texttt{Gb}(\mathbf{Assoc}, (u, (v, w))) = \epsilon, ((u, v), w) \\
&\texttt{Gb}(\mathbf{Unassoc}, ((u, v), w)) = \epsilon, (u, (v, w)) \\
&\texttt{Gb}(C_0 \ggg C_1, u) = (\widetilde{C}_0, \widetilde{C}_1), v \text{ where} \\
&\quad \widetilde{C}_0, w = \texttt{Gb}(C_0, u) \\
&\quad \widetilde{C}_1, v = \texttt{Gb}(C_1, w) \\
&\texttt{Gb}(\mathbf{First}(C), (u, w)) = \widetilde{C}, (v, w) \text{ where} \\
&\quad \widetilde{C}, v = \texttt{Gb}(C, u) \\
&\texttt{Gb}(\mathbf{Dup}, (b, (k^0, k^1))) = \epsilon, w \text{ where} \\
&\quad w = ((b, \mathsf{G}_0(k^0), \mathsf{G}_0(k^1)), (b, \mathsf{G}_1(k^0), \mathsf{G}_1(k^1))) \\
&\texttt{Gb}(\mathbf{NAnd}, ((b_i, (k_i^0, k_i^1)), (b_j, (k_j^0, k_j^1)))) = \widetilde{C}, w \text{ where} \\
&\quad h \leftarrow \textbf{new} \\
&\quad \widetilde{C} = \pi[b_i](\pi[b_j](\{\!|\{\!|(\neg \mathsf{B}_h, \mathsf{K}_h^1)|\!\}_{k_j^0}|\!\}_{k_i^0}, \{\!|\{\!|(\neg \mathsf{B}_h, \mathsf{K}_h^1)|\!\}_{k_j^1}|\!\}_{k_i^0}), \\
&\qquad\qquad \pi[b_j](\{\!|\{\!|(\neg \mathsf{B}_h, \mathsf{K}_h^1)|\!\}_{k_j^0}|\!\}_{k_i^1}, \{\!|\{\!|(\mathsf{B}_h, \mathsf{K}_h^0)|\!\}_{k_j^1}|\!\}_{k_i^1})) \\
&\quad w = (\mathsf{B}_h, (\mathsf{K}_h^0, \mathsf{K}_h^1))
\end{aligned}$$

The full garbling procedure can be obtained by composing the above functions. On input a circuit C and a boolean vector x, it picks random labels for the input wires using $\mathtt{Label}$, calls $\mathtt{Gb}$ to generate a garbled circuit $\widetilde{C}$ and output labels, and then calls $\mathtt{GEnc}$ and $\mathtt{GMask}$ to produce a garbled input $\tilde{x}$. Note that the second parameter of $\mathtt{GEnc}$ is a bundle of bits rather than a boolean vector. In the definition of $\mathtt{Garble}$ below we slightly abuse notation and use x to denote a bundle of bits $x_1, \ldots, x_n$ of a suitable shape, which can be efficiently constructed from x and s.

$$
\begin{aligned}
&\mathtt{Garble} :: \mathrm{Circuit}(s,t) \times \{0,1\}^n \to \mathbf{Exp} \\
&\mathtt{Garble}(C, x) = (\widetilde{C}, \tilde{x}) \text{ where} \\
&\quad u \leftarrow \mathtt{Label}(s) \\
&\quad \widetilde{C}, v = \mathtt{Gb}(C, u) \\
&\quad \tilde{x} = (\mathtt{GEnc}(u, x), \mathtt{GMask}(v))
\end{aligned}
$$

Next, we consider the garbled circuit evaluation algorithm $\mathtt{GEval}$. The core part of $\mathtt{GEval}$ is a recursive function $\mathtt{GEv}$ that takes a garbled circuit and an encoded input expression, producing an encoded output expression. Any encoded output is also an encoded input for evaluating subsequent garbled circuits. We include a circuit as another input of $\mathtt{GEv}$, which is used to determine the shapes of output wires. Ideally we can use the circuit's topology instead, but for simplicity we just use the circuit itself and we do not exploit the function computed by a circuit.

$$
\begin{aligned}
&\mathtt{GEv} :: \mathrm{Circuit}(s,t) \times \mathbf{Exp} \times \mathbf{Exp} \to \mathbf{Exp} \\
&\mathtt{GEv}(\mathbf{Swap}, \epsilon, (u, v)) = (v, u) \\
&\mathtt{GEv}(\mathbf{Assoc}, \epsilon, (u, (v, w)) = ((u, v), w) \\
&\mathtt{GEv}(\mathbf{Unassoc}, \epsilon, (u, (v, w)) = ((u, v), w) \\
&\mathtt{GEv}(\mathbf{Dup}, \epsilon, (b, k) = ((b, \mathsf{G}_0(k)), (b, \mathsf{G}_1(k))) \\
&\mathtt{GEv}(\mathbf{NAnd}, \widetilde{C}, ((b'_0, k_0), (b'_1, k_1))) = (b, k) \text{ where} \\
&\quad \pi[b_0](r_0, r_1) \quad = \widetilde{C} \\
&\quad \pi[b_1](e_0, e_1) \quad = \text{if } b'_0 \equiv b_0 \text{ then } r_0 \text{ else } r_1 \\
&\quad \{\!\{ \{\!\{ (b, k) \}\!\}_{k_1} \}\!\}_{k_0} = \text{if } b'_1 \equiv b_1 \text{ then } e_0 \text{ else } e_1 \\
&\mathtt{GEv}(C_0 \ggg C_1, (\widetilde{C}_0, \widetilde{C}_1), u) = \mathtt{GEv}(C_1, \widetilde{C}_1, w) \text{ where} \\
&\quad w = \mathtt{GEv}(C_0, \widetilde{C}_0, u) \\
&\mathtt{GEv}(\mathbf{First}(C), \widetilde{C}, (u, w)) = (v, w) \text{ where} \\
&\quad v = \mathtt{GEv}(C, \widetilde{C}, u)
\end{aligned}
$$

We briefly explain how $\mathtt{GEv}$ works. For the basic circuits **Swap**, **Assoc**, **Unassoc**, and **Dup** whose corresponding garbled circuits are ϵ, it simply rearranges the bits and keys in the encoded input to form an encoded output, except for **Dup** where it generates and then splits a pseudo-random key in the encoded output. For **NAnd**, it parses the corresponding garbled circuit as permutations controlled by atomic bits b_0, b_1, and it selects the entry corresponding to the bits b'_0, b'_1. In the above definition, we use pattern matching syntax that is usually found in functional programming languages to parse $\widetilde{C}$ and select the subexpression $\{\!\{ \{\!\{ (b, k) \}\!\}_{k_1} \}\!\}_{k_0}$. One can verify that, if (b'_i, k_i) is in the encoded input to **NAnd** for $i \in \{0, 1\}$, then $b'_i \in \{b_i, \neg b_i\}$ and the entry selected using bits b'_0, b'_1 are doubly encrypted under keys k_0,k_1. So the expression (b, k) extracted by $\mathtt{GEv}$ is well-defined. For the composite circuits $C_0 \ggg C_1$ and $\mathbf{First}(C)$, $\mathtt{GEv}$ produces an encoded output expression recursively in a way similar to how **Ev** evaluates these circuits.

Notice that the output of $\mathtt{GEv}$ are bit symbols rather than boolean values. The function $\mathtt{Decode}$ uses the output masks to decode a garbled output into a boolean vector:

$$
\begin{aligned}
&\mathtt{Decode}((b,k),b') = \text{if } b \equiv b' \text{ then } 0 \text{ else } 1\\
&\mathtt{Decode}((u_0,u_1),(d_0,d_1)) = (\mathtt{Decode}(u_0,d_0),\mathtt{Decode}(u_1,d_1))
\end{aligned}
$$

Finally, the full evaluation algorithm $\mathtt{GEval}$ is defined as[4]:

$$
\begin{aligned}
&\mathtt{GEval} :: \text{Circuit}(s,t) \times \textbf{Exp} \times \textbf{Exp} \to \{0,1\}^n\\
&\mathtt{GEval}(C,\widetilde{C},\tilde{x}) = \mathtt{Decode}(\mathtt{GEv}(C,\widetilde{C},u),d) \text{ where}\\
&\quad (u,d) = \tilde{x}
\end{aligned}
$$

The following theorem shows that our garbling scheme is correct. Briefly speaking, the encoded input expressions contain the sufficient bits and keys to obtain the encoded output from the garbled circuit expression, and the output masks provide information for decoding the encoded output.

Theorem 4. *For any circuit $C \in \text{Circuit}(s,t)$ and any boolean vector x of shape s,* $\mathtt{GEval}(C,\mathtt{Garble}(C,x)) = C(x)$.

5 Symbolic simulation and proof of security

In this section we define a simulator $\mathtt{Simulate}(\cdot,\cdot)$, and we then present our proof that, for any circuit C and any boolean vector x, the expressions $\mathtt{Garble}(C,x)$ and $\mathtt{Simulate}(C,C(x))$ are equivalent up to renaming. Together with the computational soundness theorem of our symbolic framework, such proof implies that the garbled circuit scheme of the previous section is computationally secure.

Symbolic simulator Recall that a simulator must output a symbolic expression that represents a garbled circuit and a garbled input, and a garbled input consists of an encoded input and output masks. The simulator has no access to the circuit input values, so it picks the random bit and the first random key from each label to form the encoded input:

$$
\begin{aligned}
&\mathtt{SEnc}((\mathsf{B},(\mathsf{K}^0,\mathsf{K}^1))) = (\mathsf{B},\mathsf{K}^0)\\
&\mathtt{SEnc}((L_0,L_1)) = (\mathtt{SEnc}(L_0),\mathtt{SEnc}(L_1))
\end{aligned}
$$

In order to correctly evaluate the simulated garbled circuit on the simulated garbled input, we adjust the output masks according to the circuit output value. Given a label expression and a boolean vector representing the circuit output value, the function $\mathtt{SMask}$ computes the output masks:

$$
\begin{aligned}
&\mathtt{SMask}((\mathsf{B},(\mathsf{K}^0,\mathsf{K}^1)),0) = \mathsf{B}\\
&\mathtt{SMask}((\mathsf{B},(\mathsf{K}^0,\mathsf{K}^1)),1) = \neg\mathsf{B}\\
&\mathtt{SMask}((L_0,L_1),(y_0,y_1)) = (\mathtt{SMask}(L_0,y_0),\mathtt{SMask}(L_1,y_1))
\end{aligned}
$$

The core of our simulator is a recursive function $\mathtt{Sim}$ that consumes a circuit and a label expression for input wires, and produces a symbolic expression of the simulated garbled circuit and a label expression for output wires:

[4] Notice that $\mathtt{Decode}$ outputs a bundle of bits. Here we slightly abuse notation and assume a boolean vector can be extracted from a bundle of bits.

$$
\begin{aligned}
&\mathtt{Sim} :: \text{Circuit}(s,t) \times \mathbf{Exp} \to \mathbf{Exp} \times \mathbf{Exp}\\
&\mathtt{Sim}(\mathbf{Swap}, (u,v)) = \epsilon, (v,u)\\
&\mathtt{Sim}(\mathbf{Assoc}, (u,(v,w))) = \epsilon, ((u,v),w)\\
&\mathtt{Sim}(\mathbf{Unassoc}, ((u,v),w)) = \epsilon, (u,(v,w))\\
&\mathtt{Sim}(C_0 \ggg C_1, u) = (\widehat{C}_0, \widehat{C}_1), v \text{ where}\\
&\quad \widehat{C}_0, w = \mathtt{Sim}(C_0, u)\\
&\quad \widehat{C}_1, v = \mathtt{Sim}(C_1, w)\\
&\mathtt{Sim}(\mathbf{First}(C), (u,w)) = \widehat{C}, (v,w) \text{ where } \widehat{C}, v = \mathtt{Sim}(C,u)\\
&\mathtt{Sim}(\mathbf{Dup}, (b,(k^0,k^1))) = \epsilon, w \text{ where}\\
&\quad w = ((b, (\mathsf{G}_0(k^0), \mathsf{G}_0(k^1))), (b, (\mathsf{G}_1(k^0), \mathsf{G}_1(k^1))))\\
&\mathtt{Sim}(\mathbf{NAnd}, ((b_i,(k_i^0,k_i^1)),(b_j,(k_j^0,k_j^1)))) = \widehat{C}, w \text{ where}\\
&\quad h \leftarrow \mathbf{new}\\
&\quad \widehat{C} = \pi[\mathsf{B}_i](\pi[\mathsf{B}_j](\{\!\!\{ \{\!\!\{ (\mathsf{B}_h, \mathsf{K}_h^0) \}\!\!\}_{k_j^0} \}\!\!\}_{k_i^0}, \{\!\!\{ \{\!\!\{ (\mathsf{B}_h, \mathsf{K}_h^0) \}\!\!\}_{k_j^1} \}\!\!\}_{k_i^0}),\\
&\qquad\qquad \pi[\mathsf{B}_j](\{\!\!\{ \{\!\!\{ (\mathsf{B}_h, \mathsf{K}_h^0) \}\!\!\}_{k_j^0} \}\!\!\}_{k_i^1}, \{\!\!\{ \{\!\!\{ (\mathsf{B}_h, \mathsf{K}_h^0) \}\!\!\}_{k_j^1} \}\!\!\}_{k_i^1}))\\
&\quad w = (\mathsf{B}_h, (\mathsf{K}_h^0, \mathsf{K}_h^1))
\end{aligned}
$$

Notice that, for any circuit C and any label expression u, if $\widetilde{C}, v = \mathtt{Gb}(C,u)$ and $\widehat{C}, w = \mathtt{Sim}(C,u)$, then the subscript h of any atomic key symbol K_h^i that appears in $(\widetilde{C}, v)$ and $(\widehat{C}, w)$ follows the same ordering.

Our simulator is composed of the above functions. It takes a circuit C and a boolean vector y as input, and it generates a simulated garbled circuit using Sim and a simulated garbled input using SEnc and SMask:

$$
\begin{aligned}
&\mathtt{Simulate} :: \text{Circuit}(s,t) \times \{0,1\}^m \to \mathbf{Exp}\\
&\mathtt{Simulate}(C, y) = (\widehat{C}, \hat{x}) \text{ where}\\
&\quad u \leftarrow \mathtt{Label}(s)\\
&\quad \widehat{C}, v = \mathtt{Sim}(C, u)\\
&\quad \hat{x} = (\mathtt{SEnc}(u), \mathtt{SMask}(v, y))
\end{aligned}
$$

Symbolic proof of security For this paper we present a pen-and-paper symbolic security proof, which can also be adapted to a machine-checked proof using verification tools. For any bit expression $b \in \mathbf{Pat}(\mathbb{B})$ and any $x \in \{0,1\}$, we introduce the notation $b^{\oplus x}$ to shorten our proofs:

$$
b^{\oplus x} = \begin{cases} b & \text{if } x = 0 \\ \neg b & \text{if } x = 1 \end{cases}
$$

We say that a label expression w is *strongly independent* if $\mathbf{Keys}(w)$ is a set of independent keys and, if $w = (b,(k^0,k^1))$ is a single label then $k^0 \neq k^1$, and if $w = (u,v)$ where u and v are label expressions, then u and v are both strongly independent and $\mathbf{Keys}(u) \cap \mathbf{Keys}(v) = \emptyset$.

Let us start with some technical lemmas that are helpful to derive our main result. The first lemma can be easily verified by induction on the definition of Gb.

Lemma 4. *For any circuit C and label expression u, if $\widetilde{C}, v = \mathtt{Gb}(C,u)$ and $k \in \mathbf{Keys}(\widetilde{C}) \cap \mathbf{Parts}(\widetilde{C})$, then $k \in \mathbf{K}$ is an atomic key symbol.*

Our next lemma shows that Gb produces strongly independent output labels from strongly independent input labels. Furthermore, any key in the output label expression is yielded from either a new atomic key introduced in the garbled circuit or a key in the input labels, and it does not yield any other key in the garbled circuit. The formal proof is done using structural induction on circuits, and it is omitted due to space constraint.

Lemma 5. *For any circuit C and any strongly independent label expression u such that $\widetilde{C}, v = \mathtt{Gb}(C, u)$, v is strongly independent, and the following hold for all $k \in \mathbf{Keys}(v)$:*

1. $\mathsf{G}^+(k) \cap \mathbf{Keys}((\widetilde{C}, u)) = \emptyset$*;*
2. $\exists k' \in \mathbf{Keys}((\widetilde{C}, u)) \cap \mathbf{Parts}((\widetilde{C}, u)).k' \preceq k$*.*

A quick observation on Gb is that, for any circuit C, if $k \in \mathbf{Pat}(\mathbb{K})$ appears in $\widetilde{C}$, then either k is in a plaintext message and so $k \in \mathbf{Parts}(\widetilde{C})$, or k is used as an encryption key. The former case has been considered in Lemma 4. The following lemma characterizes the latter case, and it can be proved using structural induction on circuits.

Lemma 6. *For any circuit C and any label expression u such that u is strongly independent and $\widetilde{C}, v = \mathtt{Gb}(C, u)$, if $\{\!\{e\}\!\}_k \in \mathbf{Parts}(\widetilde{C})$ for some expression e and some key $k \in \mathbf{Pat}(\mathbb{K})$, then the following hold:*

1. $\mathsf{G}^+(k) \cap \mathbf{Keys}(\widetilde{C}) = \emptyset$*;*
2. $\mathsf{G}^*(k) \cap \mathbf{Keys}(v) = \emptyset$*;*
3. $\exists k' \in \mathbf{Keys}((\widetilde{C}, u)) \cap \mathbf{Parts}((\widetilde{C}, u)).k' \preceq k$*.*

For the rest of paper, let us fix a circuit $C \in \mathrm{Circuit}(s, t)$ and a boolean vector $x \in \{0, 1\}^n$, where s is a shape of n wires and t is a shape of m wires. Let $e = (\widetilde{C}, \tilde{x}) = \mathtt{Garble}(C, x)$ be the symbolic expression of the garbled circuit and the garbled input of C on input x. Since $\mathcal{F}_e$ is monotone, the greatest fixed point of $\mathcal{F}_e$ exists and it can be computed in polynomially many steps. Let $S = \mathrm{Fix}(\mathcal{F}_e)$ and $e' = \mathbf{p}(e, S)$. Then $\mathbf{Roots}(S) \subseteq \mathbf{Keys}(e)$ and $S = \mathcal{F}_e(S) = \mathbf{r}(\mathbf{p}(e, S)) = \mathbf{r}(e')$. For any label $(b, (k^0, k^1))$, we say that it satisfies the *label invariant* if

$$b \in \mathbf{B}, \exists z \in \{0, 1\} \text{ such that } k^z \in S, k^{1-z} \notin S, \tag{1}$$

and we call z the *actual value* of the label $(b, (k^0, k^1))$.

Lemma 7. *For any sub-circuit C' of C, and for any label expression u, if $\widetilde{C'}, v = \mathtt{Gb}(C', u)$ and all labels $(b, (k^0, k^1)) \Subset u$ satisfy the label invariant, then all labels $(\bar{b}, (\bar{k}^0, \bar{k}^1)) \Subset v$ satisfy the label invariant.*

Proof. We use induction on the structure of circuit C'. For the base case, C' is an atomic circuit:

- $C' = \mathbf{Swap}$, $\mathbf{Assoc}$, or $\mathbf{Unassoc}$: Any label $(\bar{b}, (\bar{k}^0, \bar{k}^1)) \Subset v$ is also a sub-expression of u. So the lemma holds.

- $C' = \mathbf{Dup}$: Suppose $u = (b, (k^0, k^1))$ satisfies the label invariant with an actual value z. If $(\bar{b}, (\bar{k}^0, \bar{k}^1)) \Subset v$, then $\bar{b} = b$, $\bar{k}^0 = \mathsf{G}_h(k^0)$, and $\bar{k}^1 = \mathsf{G}_h(k^1)$ for some $h \in \{0, 1\}$. So $\bar{b} \in \mathbf{B}$. Let $\bar{z} = z$. Then $\bar{k}^{\bar{z}} = \mathsf{G}_h(k^z) \in S$. Assume towards a contradiction that $\bar{k}^{1-\bar{z}} \in S$. Then $\mathsf{G}_h(k^{1-z}) = \bar{k}^{1-\bar{z}} \in \mathsf{G}^*(k')$ for some $k' \in \mathbf{Keys}(e')$ where $k' \in \mathbf{Parts}(e')$ or $\exists k'' \in \mathbf{Keys}(e')$ such that $k' \prec k''$. Notice that $e' = \mathbf{p}((\widetilde{C}, \tilde{x}), S) = (\mathbf{p}(\widetilde{C}, S), \mathbf{p}(\tilde{x}, S))$, and $\tilde{x}$ contains only atomic keys. So $k'' \in \mathbf{Keys}(\widetilde{C'}) \subseteq \mathbf{Keys}(\widetilde{C})$. We have two cases:
  - $\mathsf{G}_h(k^{1-z}) \neq k'$: $k^{1-z} \in \mathsf{G}^*(k') \subseteq S$, a contradiction.
  - $\mathsf{G}_h(k^{1-z}) = k'$: Now $k' \notin \mathbf{Parts}(e')$, and thus $\{\!\{g'\}\!\}_{k'} \in \mathbf{Parts}(e')$ for some pattern g'. So $\{\!\{g'\}\!\}_{k'} \in \mathbf{Parts}(\mathbf{p}(\widetilde{C}, S))$ and $\{\!\{g\}\!\}_{k'} \in \mathbf{Parts}(\widetilde{C})$ for some expression g such that $g' = \mathbf{p}(g, S)$. By Lemma 6, $\mathsf{G}^+(k') \cap \mathbf{Keys}(\widetilde{C}) = \emptyset$ and hence $k'' \notin \mathbf{Keys}(\widetilde{C})$, a contradiction.

  Therefore $(\bar{b}, (\bar{k}^0, \bar{k}^1))$ satisfies the label invariant.

- $C' = \textbf{NAnd}$: The only label in v is $(\mathsf{B}_h, (\mathsf{K}_h^0, \mathsf{K}_h^1))$. Notice that the expressions in $\textbf{Parts}(e)$ that contain $\mathsf{K}_h^0, \mathsf{K}_h^1$ are the following and their sub-expressions:

$$\{\!\!\{ \{\!\!\{ (\neg \mathsf{B}_h, \mathsf{K}_h^1) \}\!\!\}_{k_j^0} \}\!\!\}_{k_i^0}, \{\!\!\{ \{\!\!\{ (\neg \mathsf{B}_h, \mathsf{K}_h^1) \}\!\!\}_{k_j^0} \}\!\!\}_{k_i^1},$$
$$\{\!\!\{ \{\!\!\{ (\neg \mathsf{B}_h, \mathsf{K}_h^1) \}\!\!\}_{k_j^1} \}\!\!\}_{k_i^0}, \{\!\!\{ \{\!\!\{ (\mathsf{B}_h, \mathsf{K}_h^0) \}\!\!\}_{k_j^1} \}\!\!\}_{k_i^1},$$

where $((b_i, (k_i^0, k_i^1)), (b_j, (k_j^0, k_j^1))) = u$. Observe that these four expressions can be generated as

$$\{\!\!\{ \{\!\!\{ (\mathsf{B}_h^{\oplus(x_i \uparrow x_j)}, \mathsf{K}_h^{x_i \uparrow x_j}) \}\!\!\}_{k_j^{x_j}} \}\!\!\}_{k_i^{x_i}} \text{ for } x_i, x_j \in \{0, 1\}.$$

Let $\bar{z} = z_i \uparrow z_j$. By assumption, we have $k_i^{z_i}, k_j^{z_j} \in S$ and $k_i^{1-z_i}, k_j^{1-z_j} \notin S$, so $k^{\bar{z}} = \mathsf{K}_h^{\bar{z}} \in S$ and $k^{1-\bar{z}} = \mathsf{K}_h^{1-\bar{z}} \notin S$, and Condition 1 holds for $(\mathsf{B}_h, (\mathsf{K}_h^0, \mathsf{K}_h^1))$.

Next, consider composite circuits. Assume the lemma holds for all sub-circuits of C'. Then we have these cases:

- $C' = C_0' \ggg C_1'$: Suppose $\widetilde{C} = (\widetilde{C}_0', \widetilde{C}_1')$ where $\widetilde{C}_0', w = \mathtt{Gb}(C_0', u)$ and $\widetilde{C}_1', v = \mathtt{Gb}(C_1', w)$. Since C_0' and C_1' are both sub-circuits of C', by assumption we see that Condition 1 holds for all labels in u and consequently, for all labels in w, and so it holds for all labels in v.
- $C' = \textbf{First}(C'')$: Suppose $u = (u'', w)$ and $v = (v'', w)$ such that $\widetilde{C}, v'' = \mathtt{Gb}(C'', u'')$. For any label $(\bar{b}, (\bar{k}^0, \bar{k}^1)) \Subset v$, it is either a sub-expression of v'' or it is a sub-expression of w. For the former case, since C'' is a sub-circuit of C', Condition 1 holds for $(\bar{b}, (\bar{k}^0, \bar{k}^1))$ by induction hypothesis. For the latter case, since $w \Subset u$, Condition 1 holds for this label by assumption.

Therefore the lemma holds for any circuit C. □

Let $f = (\widehat{C}, \hat{x}) = \mathtt{Simulate}(C, C(x))$ be the symbolic expression of simulated garbled circuit of C on output $C(x)$. Let $T = \mathrm{Fix}(\mathcal{F}_f)$, which satisfies $\mathcal{F}_f(T) = \mathbf{r}(\mathbf{p}(f, T)) = T$. The following lemma shows that, for each key pair k^0, k^1 in f, exactly one of k^0 and k^1 is in T.

Lemma 8. *For any sub-circuit C' of C and any label expression u such that $\widehat{C}', v = \mathtt{Sim}(C', u)$, if all labels $(b, (k^0, k^1)) \Subset u$ satisfy the label invariant with actual value 0, then all labels $(\bar{b}, (\bar{k}^0, \bar{k}^1)) \Subset v$ satisfy the label invariant with actual value 0.*

Proof. We can directly apply the proof of Lemma 7 except for the base case when $C' = \textbf{NAnd}$:

- $C' = \textbf{NAnd}$: The label in v is $(\mathsf{B}_h, (\mathsf{K}_h^0, \mathsf{K}_h^1))$. The expressions in $\textbf{Parts}(f)$ that contain $\mathsf{K}_h^0, \mathsf{K}_h^1$ are the following and their sub-expressions:

$$\{\!\!\{ \{\!\!\{ (\mathsf{B}_h, \mathsf{K}_h^0) \}\!\!\}_{k_j^0} \}\!\!\}_{k_i^0}, \{\!\!\{ \{\!\!\{ (\mathsf{B}_h, \mathsf{K}_h^0) \}\!\!\}_{k_j^0} \}\!\!\}_{k_i^1},$$
$$\{\!\!\{ \{\!\!\{ (\mathsf{B}_h, \mathsf{K}_h^0) \}\!\!\}_{k_j^1} \}\!\!\}_{k_i^0}, \{\!\!\{ \{\!\!\{ (\mathsf{B}_h, \mathsf{K}_h^0) \}\!\!\}_{k_j^1} \}\!\!\}_{k_i^1},$$

where $((b_i, (k_i^0, k_i^1)), (b_j, (k_j^0, k_j^1))) = u$. Let $\bar{z} = 0$. By assumption, $k_i^0, k_j^0 \in T$ and $k_i^1, k_j^1 \notin T$. So $k^{\bar{z}} = \mathsf{K}_h^0 \in T$ and $k^{1-\bar{z}} = \mathsf{K}_h^1 \notin T$, and Condition 1 holds for $(\mathsf{B}_h, (\mathsf{K}_h^0, \mathsf{K}_h^1))$ with actual value 0.

For the rest of the cases, the proof of Lemma 7 applies with actual value 0. □

Now we are ready to prove our main result that the patterns of the real garbled circuit and the simulated garbled circuit are equivalent up to renaming.

Theorem 5. *For any circuit $C \in \mathrm{Circuit}(s,t)$ and any boolean vector $x \in \{0,1\}^n$, where s is a shape of n wires,* $\mathbf{Pattern}(\mathtt{Garble}(C,x)) \approx \mathbf{Pattern}(\mathtt{Simulate}(C, C(x)))$.

Proof. Let $u = ((\mathsf{B}_1, (\mathsf{K}_1^0, \mathsf{K}_1^1)), \dots, (\mathsf{B}_n, (\mathsf{K}_n^0, \mathsf{K}_n^1)))$ be the label expression in $\mathtt{Garble}$. Let $\widetilde{C}, v = \mathtt{Gb}(C,u)$. One can check that, for any sub-circuit C' of C, if $\widetilde{C}', v' = \mathtt{Gb}(C',u')$ and $\widehat{C}', w' = \mathtt{Sim}(C',u')$ for any label expression u' of an appropriate shape, then $v' = w'$. Since $\mathtt{Sim}$ is also applied on C and u in $\mathtt{Simulate}$, we can write $\widehat{C}, v = \mathtt{Sim}(C,u)$.

Let $e = (\widetilde{C}, \tilde{x}) = \mathtt{Garble}(C,x)$, $f = (\widehat{C}, \hat{x}) = \mathtt{Simulate}(C, C(x))$, $S = \mathrm{Fix}(\mathcal{F}_e)$, and $T = \mathrm{Fix}(\mathcal{F}_f)$. We can write $\widetilde{C} = (\widetilde{C}_1, \dots, \widetilde{C}_q)$ and $\widehat{C} = (\widehat{C}_1, \dots, \widehat{C}_q)$, where $\widetilde{C}_i, v_i = \mathtt{Gb}(C_i, u_i)$ and $\widehat{C}_i, v_i = \mathtt{Sim}(C_i, u_i)$ for some atomic sub-circuit C_i of C and some label expression u_i. To show $\mathbf{Pattern}(e) = \mathbf{p}(e,S) \approx \mathbf{p}(f,T) = \mathbf{Pattern}(f)$, we first show $(\mathbf{p}(\widetilde{C}_1, S), \dots, \mathbf{p}(\widetilde{C}_q, S)) \approx (\mathbf{p}(\widehat{C}_1, T), \dots, \mathbf{p}(\widehat{C}_q, T))$ with respect to a pseudorandom renaming $\alpha = (\alpha_B, \alpha_K)$, and then we show $\mathbf{p}(\tilde{x}, S) \approx_\alpha \mathbf{p}(\hat{x}, T)$.

For the first part, let α_B be the random bit renaming $\alpha_B(\mathsf{B}_i) = \mathsf{B}_i^{\oplus z_i}$ for all $\mathsf{B}_i \in \mathbf{B}$, where z_i is the actual value of the label that contains B_i. Let α_K be the bijection on $\mathbf{K}$ such that $\alpha_K(\mathsf{K}_i^{z_i}) = \mathsf{K}_i^0$ and $\alpha_K(\mathsf{K}_i^{1-z_i}) = \mathsf{K}_i^1$ for each $\mathsf{K}_i^0, \mathsf{K}_i^1$. We claim that, for any sub-circuit C' of C and for any label expression u', if $\widetilde{C}', v' = \mathtt{Gb}(C',u')$ and $\widehat{C}', v' = \mathtt{Sim}(C',u')$, then Condition 1 holds for all labels in v' and $\mathbf{p}(\widetilde{C}', S) \approx_\alpha \mathbf{p}(\widehat{C}', T)$.

Proof of claim: Notice that all labels in u satisfy Condition 1. By Lemma 7, all labels in v' also satisfy Condition 1. We use induction on the structure of C' to show $\mathbf{p}(\widetilde{C}', S) \approx_\alpha \mathbf{p}(\widehat{C}', T)$. For the base case, C' is an atomic circuit:

- $C' = $ **Swap**, **Assoc**, **Unassoc**, or **Dup**: Both $\widetilde{C}'$ and $\widehat{C}'$ are the empty garbled circuit ϵ, so $\mathbf{p}(\widetilde{C}', S) = \mathbf{p}(\widehat{C}', T)$.
- $C' = $ **NAnd**: Suppose $u' = ((b_i, (k_i^0, k_i^1)), (b_j, (k_j^0, k_j^1)))$ and $v' = (\mathsf{B}_h, (\mathsf{K}_h^0, \mathsf{K}_h^1))$. Let z_i, z_j and z_h be the actual values of the labels $(b_i, (k_i^0, k_i^1))$, $(b_j, (k_j^0, k_j^1))$ and $(\mathsf{B}_h, (\mathsf{K}_h^0, \mathsf{K}_h^1))$, respectively. We know from the proof of Lemma 7 that $z_h = z_i \uparrow z_j$. So we can apply α_K and get

$$
\begin{aligned}
\widetilde{C}' = {} & \pi[B_i](\pi[B_j](\{\!\!\{\{\!\!\{(\mathsf{B}_h^{\oplus(0\uparrow 0)}, \mathsf{K}_h^{0\uparrow 0})\}\!\!\}_{k_j^0}\}\!\!\}_{k_i^0}, \\
& \qquad\qquad \{\!\!\{\{\!\!\{(\mathsf{B}_h^{\oplus(0\uparrow 1)}, \mathsf{K}_h^{0\uparrow 1})\}\!\!\}_{k_j^1}\}\!\!\}_{k_i^0}), \\
& \quad \pi[B_j](\{\!\!\{\{\!\!\{(\mathsf{B}_h^{\oplus(1\uparrow 0)}, \mathsf{K}_h^{1\uparrow 0})\}\!\!\}_{k_j^0}\}\!\!\}_{k_i^1}, \\
& \qquad\qquad \{\!\!\{\{\!\!\{(\mathsf{B}_h^{\oplus(1\uparrow 1)}, \mathsf{K}_h^{1\uparrow 1})\}\!\!\}_{k_j^1}\}\!\!\}_{k_i^1})) \\
\approx_\alpha {} & \pi[B_i^{\oplus z_i}](\pi[B_j^{\oplus z_j}](\{\!\!\{\{\!\!\{(\mathsf{B}_h^{\oplus(0\uparrow 0)\oplus z_h}, \mathsf{K}_h^{(0\uparrow 0)\oplus z_h})\}\!\!\}_{k_j^0}\}\!\!\}_{k_i^0}, \\
& \qquad\qquad \{\!\!\{\{\!\!\{(\mathsf{B}_h^{\oplus(0\uparrow 1)\oplus z_h}, \mathsf{K}_h^{(0\uparrow 1)\oplus z_h})\}\!\!\}_{k_j^1}\}\!\!\}_{k_i^0}), \\
& \quad \pi[B_j^{\oplus z_j}](\{\!\!\{\{\!\!\{(\mathsf{B}_h^{\oplus(1\uparrow 0)\oplus z_h}, \mathsf{K}_h^{(1\uparrow 0)\oplus z_h})\}\!\!\}_{k_j^0}\}\!\!\}_{k_i^1}, \\
& \qquad\qquad \{\!\!\{\{\!\!\{(\mathsf{B}_h^{\oplus(1\uparrow 1)\oplus z_h}, \mathsf{K}_h^{(1\uparrow 1)\oplus z_h})\}\!\!\}_{k_j^1}\}\!\!\}_{k_i^1})) \\
\equiv {} & \pi[B_i](\pi[B_j](\{\!\!\{\{\!\!\{(\mathsf{B}_h^{\oplus\mu(z_i,z_j)}, \mathsf{K}_h^{\oplus\mu(z_i,z_j)})\}\!\!\}_{k_j^0}\}\!\!\}_{k_i^0}, \\
& \qquad\qquad \{\!\!\{\{\!\!\{(\mathsf{B}_h^{\oplus\mu(z_i,1-z_j)}, \mathsf{K}_h^{\oplus\mu(z_i,1-z_j)})\}\!\!\}_{k_j^1}\}\!\!\}_{k_i^0}), \\
& \quad \pi[B_j](\{\!\!\{\{\!\!\{(\mathsf{B}_h^{\oplus\mu(1-z_i,z_j)}, \mathsf{K}_h^{\oplus\mu(1-z_i,z_j)})\}\!\!\}_{k_j^0}\}\!\!\}_{k_i^1}, \\
& \qquad\qquad \{\!\!\{\{\!\!\{(\mathsf{B}_h^{\oplus\mu(1-z_i,1-z_j)}, \mathsf{K}_h^{\oplus\mu(1-z_i,1-z_j)})\}\!\!\}_{k_j^1}\}\!\!\}_{k_i^1})),
\end{aligned}
$$

where $\mu(d_i, d_j) = (d_i \uparrow d_j) \oplus z_h$ for $d_i, d_j \in \{0, 1\}$. In particular, $\mu(z_i, z_j) = 0$. By Condition 1, $k_i^{z_i}, k_j^{z_j}, \mathsf{K}_h^{z_h} \in S$, $k_i^{1-z_i}, k_j^{1-z_j}, \mathsf{K}_h^{1-z_h} \notin S$, and $b_i^{\oplus z_i}, b_j^{\oplus z_j} \in \mathbf{Parts}(\mathbf{p}(e, S))$. So $k_i^0, k_j^0 \in \alpha(S)$, $k_i^1, k_j^1 \notin \alpha(S)$, and the pattern $\alpha(\mathbf{p}(\widetilde{C}', S)) = \mathbf{p}(\alpha(\widetilde{C}'), \alpha(S))$ is equivalent to

$$\pi[B_i](\pi[B_j](\{\!\!\{\{\!\!\{(\mathsf{B}_h, \mathsf{K}_h^0)\}\!\!\}_{k_j^0}\}\!\!\}_{k_i^0}, \{\!\!\{\{\!\!\{(\!|\mathbb{B}, \mathbb{K}|\!)\}\!\!\}_{k_j^1}\}\!\!\}_{k_i^0}),$$
$$\pi[B_j](\{\!\!\{\{\!\!\{(\!|\mathbb{B}, \mathbb{K}|\!)\}\!\!\}\}\!\!\}_{k_i^1}, \{\!\!\{\{\!\!\{(\!|\mathbb{B}, \mathbb{K}|\!)\}\!\!\}\}\!\!\}_{k_i^1}))$$

On the other hand, by Lemma 8, $k_i^0, k_j^0, \mathsf{K}_h^0 \in T$ and $k_i^1, k_j^1, \mathsf{K}_h^1 \notin T$. So the pattern $\mathbf{p}(\widehat{C}', S)$ of $\widehat{C}'$ is

$$\pi[B_i](\pi[B_j](\{\!\!\{\{\!\!\{(\mathsf{B}_h, \mathsf{K}_h^0)\}\!\!\}_{k_j^0}\}\!\!\}_{k_i^0}, \{\!\!\{\{\!\!\{(\!|\mathbb{B}, \mathbb{K}|\!)\}\!\!\}_{k_j^1}\}\!\!\}_{k_i^0},$$
$$\pi[B_j](\{\!\!\{\{\!\!\{(\!|\mathbb{B}, \mathbb{K}|\!)\}\!\!\}\}\!\!\}_{k_j^1}), \{\!\!\{\{\!\!\{(\!|\mathbb{B}, \mathbb{K}|\!)\}\!\!\}\}\!\!\}_{k_j^1}))$$

Thus $\mathbf{p}(\widetilde{C}', S) \approx_\alpha \mathbf{p}(\widehat{C}', T)$.

For the induction step, assuming the claim holds for sub-circuits C_0' and C_1' of C, it is easy to check that the claim also holds for the cases $C' = \mathbf{First}(C_0')$ and $C' = C_0' \ggg C_1'$. Therefore our claim follows.

For the second part, let $y = C(x)$. Then for any $i \in [m]$, y_i is the actual value of the corresponding output wire. Since $\tilde{x} = ((\mathsf{K}_1^{x_1}, \mathsf{B}_1^{\oplus x_1}), \ldots, (\mathsf{K}_n^{x_n}, \mathsf{B}_n^{\oplus x_n}), (b_1, \ldots, b_m))$, we can calculate

$$\alpha(\tilde{x}) = ((\mathsf{K}_1^0, \mathsf{B}_1), \ldots, (\mathsf{K}_n^0, \mathsf{B}_n), (b_1^{\oplus y_i}, \ldots, b_m^{\oplus y_m})) = \hat{x}.$$

So $\tilde{x} \approx_\alpha \hat{x}$, and thus $\mathbf{p}(\tilde{x}, S) \approx_\alpha \mathbf{p}(\hat{x}, T)$.

Therefore the theorem holds. □

As a corollary of Theorem 3 and 5, we can now conclude that our garbled circuit scheme is computationally secure.

Corollary 1. *For any circuit $C \in \mathrm{Circuit}(s, t)$ and any $x \in \{0, 1\}^n$ where s is a shape of n wires, the probability distributions $[\![\texttt{Garble}(C, x)]\!]$ and $[\![\texttt{Simulate}(C, C(x))]\!]$ are computationally indistinguishable.*

6 Implementation and automated tests

As a proof of concept, we have implemented our symbolic framework as well as the garbling scheme and the simulator in Haskell. The source code can be found at `https://github.com/b5li/SymGC`. Our symbolic framework implementation closely follows the definitions in Section 2.1. In addition, we added a normalization operation `norm` on patterns, for example:

`norm (Not (Bit False)) = Bit True`	`norm (Perm (Bit False)` p q`) = Pair (norm` q`) (norm` p`)`
`norm (Not (Bit True)) = Bit False`	`norm (Perm (Bit True)` p q`) = Pair (norm` p`) (norm` q`)`
`norm (Not (Not` e`)) = norm` e	`norm (Perm (Not` b`)` p q`) = norm (Perm` b q p`)`

The equivalence relation $\equiv$ on patterns, defined in Section 2.1, is checked using syntactic equality on normalized patterns. Random bit renaming and pseudo-random key renaming are implemented using maps on normalized bit and key patterns. Thus we can check equivalence up to renaming by first applying renaming maps to normalized patterns and then checking for equivalence.

To build symbolic expressions of the real and the simulated garbled circuits, the pseudo-code definitions of the garbling scheme and the simulator in Sections 4 and 5 were directly translated into Haskell code. The bit and key renamings α_B and α_K were constructed recursively as in the proof of Lemma 5.

So far, given a circuit and a boolean vector of an appropriate shape, our programs are able to produce symbolic expressions of the real and the simulated garbled circuits, compute their patterns, and check if these patterns are equivalent up to renaming. The whole implementation consists of about 500 lines of Haskell code, and its performance is fairly good: For example, with a randomly generated circuit that contains about 10000 NAND subcircuits and a 112-dimension boolean vector, the entire process of generating the real and the simulated garbled circuits, computing their patterns, and checking for symbolic equivalence runs in about 1.3 second on a Linux desktop with an Intel I7-4790 CPU running at 3.60GHz. Notice that the number of NAND subcircuits and the dimension of the input vector together determine the number of atomic keys in the garbled circuit expression, which affects how fast the greatest fixed point of the recoverable key set can be reached. Further optimization is possible, for example, we could expand our circuit notation by adding AND and XOR as basic circuits. As a reference, an AES encryption circuit usually consists of about 5k AND and 20k XOR gates, which can be implemented using about 90k **NAnd** inductively.

We conducted automated tests using the QuickCheck test framework to perform symbolic security analysis on randomly generated circuits and boolean vectors, and the performance results are shown in Fig. 6.

We remark that our automated tests run on a circuit-by-circuit basis, that is, given a circuit and a boolean vector, the test ensures that the resulting garbled circuit is computationally secure. In fact, our program can check that, for any cryptographic system that is built using primitives in our symbolic framework, an instance for a given input is computationally secure. It is also interesting to translate our proofs into a machine-checked flavor using verification tools, but such work is out of the scope of the current paper, and we would like to explore it in the future.

7 Conclusion

We presented a lightweight and sound symbolic security framework and an inductive notation for formal specification of boolean circuits. Our symbolic language extends on previous work by including random bits and controlled swap operation; such an extension is not trivial as we need the additional rules for equivalence on patterns and the pseudorandom bit renaming to build a sound language. We proved that Yao's garbled circuit scheme is symbolically secure, which can be translated into selective computational security thanks to the computational soundness theorem. By abstracting the probabilistic behaviors in the computational soundness theorem, one may find concise and clean descriptions of security properties, e.g., the label invariant in our proof of garbled circuit schemes. As a result, the complete security proof can be presented in a way that is not only easy to be mechanized by computer-aided verification tools but also manageable for human readers to understand and verify. We remark that helping cryptographers comprehend formal security proofs could be of great benefit to them to find optimization opportunities that might be hidden in complex computational proofs.

Our symbolic language does not limit us to just circuit garbling schemes. With a small extension to include the xor operation on key expressions, we can formally specify Yao's computational secret sharing scheme [Bei11, VNS+03] and prove its security. We briefly present such analysis in the Appendix. As another example, we can formalize the protocol of OT length extension via a pseudorandom generator [LM18] in this extended framework, and we can prove it is secure against a static semi-honest adversary. A natural but challenging upgrade is to consider active security of cryptographic protocols; we leave it to future work.

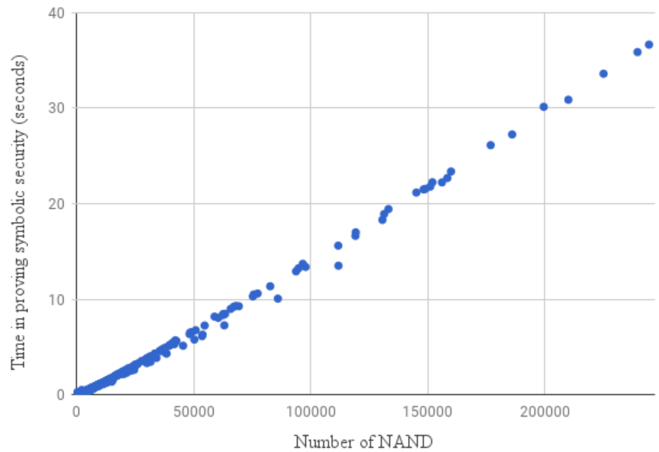

Figure 6: Running times of proving symbolic security of the garbling scheme using our implementation. Experiments were run on a Linux desktop with an Intel I7-4790 CPU running at 3.60GHz. Each point corresponds to a randomly generated test case, where the circuit may contain up to 250k **NAnd** subcircuits and the input vector may have up to 128 components. For each test case we measure the total time spent on generating the real and the simulated garbled circuit expressions, computing their patterns, and then checking for symbolic equivalence on patterns. The horizontal axis measures the number of **NAnd** subcircuits in a circuit, and the vertical axis measures the time in seconds.

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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*, pages 1989–2006, 2017.

[AR07] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptology*, 20(3):395, 2007.

[AW08] Martín Abadi and Bogdan Warinschi. Security analysis of cryptographically controlled access to XML documents. *J. ACM*, 55(2):6:1–6:29, 2008.

[Bei11] Amos Beimel. Secret-sharing schemes: A survey. In *Coding and Cryptology - Third International Workshop, IWCC 2011, Qingdao, China, May 30-June 3, 2011. Proceedings*, pages 11–46, 2011.

[BGG+14] Dan Boneh, Craig Gentry, Sergey Gorbunov, Shai Halevi, Valeria Nikolaenko, Gil Segev, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurthy. Fully key-homomorphic encryption, arithmetic circuit ABE and compact garbled circuits. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pages 533–556, 2014.

[BGHB11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 71–90, 2011.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[BGZB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 90–101, New York, NY, USA, 2009. ACM.

[BH92] Donald Beaver and Stuart Haber. Cryptographic protocols provably secure against dynamic adversaries. In *Advances in Cryptology - EUROCRYPT '92, Workshop on the Theory and Application of of Cryptographic Techniques, Balatonfüred, Hungary, May 24-28, 1992, Proceedings*, pages 307–323, 1992.

[BHR12a] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, pages 134–153, 2012.

[BHR12b] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 784–796, 2012.

[Bla06] Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154, Oakland, California, May 2006.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, pages 192–206, 2008.

[BMR90] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. ACM.

[BNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multi-party computation. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 257–266, 2008.

[BWA10] Mathieu Baudet, Bogdan Warinschi, and Martín Abadi. Guessing attacks and the computational soundness of static equivalence. *Journal of Computer Security*, 18(5):909–968, 2010.

[CFGN96] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 639–648, 1996.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.

[HJO+16] Brett Hemenway, Zahra Jafargholi, Rafail Ostrovsky, Alessandra Scafuro, and Daniel Wichs. Adaptively secure garbled circuits from one-way functions. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, pages 149–178, 2016.

[Hug00] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[Hug04] John Hughes. Programming with arrows. In *Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14-21, 2004, Revised Lectures*, pages 73–129, 2004.

[JKK+17] Zahra Jafargholi, Chethan Kamath, Karen Klein, Ilan Komargodski, Krzysztof Pietrzak, and Daniel Wichs. Be adaptive, avoid overcommitting. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 133–163, 2017.

[JW16] Zahra Jafargholi and Daniel Wichs. Adaptive security of yao's garbled circuits. In *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I*, pages 433–458, 2016.

[LM18] Baiyu Li and Daniele Micciancio. Equational security proofs of oblivious transfer protocols. In *Public-Key Cryptography - PKC 2018 - 21st IACR International Conference on Practice and Theory of Public-Key Cryptography, Rio de Janeiro, Brazil, March 25-29, 2018, Proceedings, Part I*, pages 527–553, 2018.

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

[LWY10] Sam Lindley, Philip Wadler, and Jeremy Yallop. The arrow calculus. *J. Funct. Program.*, 20(1):51–69, 2010.

[Mal11] Lior Malka. Vmcrypt: modular software architecture for scalable secure computation. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 715–724, 2011.

[Mic09] Daniele Micciancio. Symbolic encryption with pseudorandom keys. Cryptology ePrint Archive, Report 2009/249, 2009. `https://eprint.iacr.org/2009/249`.

[Mic10] Daniele Micciancio. Computational soundness, co-induction, and encryption cycles. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, pages 362–380, 2010.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 287–302, 2004.

[MP06] Daniele Micciancio and Saurabh Panjwani. Corrupting one vs. corrupting many: The case of broadcast and multicast encryption. In *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, pages 70–82, 2006.

[MP08] Daniele Micciancio and Saurabh Panjwani. Optimal communication complexity of generic multicast key distribution. *IEEE/ACM Trans. Netw.*, 16(4):803–813, 2008.

[MW04] Daniele Micciancio and Bogdan Warinschi. Completeness theorems for the abadi-rogaway language of encrypted expressions. *Journal of Computer Security*, 12(1):99–130, 2004.

[Pan07] Saurabh Panjwani. Tackling adaptive corruptions in multicast encryption protocols. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, pages 21–40, 2007.

[Pat01] Ross Paterson. A new notation for arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.*, pages 229–240, 2001.

[VNS+03] V. Vinod, Arvind Narayanan, K. Srinathan, C. Pandu Rangan, and Kwangjo Kim. On the power of computational secret sharing. In *Progress in Cryptology - INDOCRYPT 2003, 4th International Conference on Cryptology in India, New Delhi, India, December 8-10, 2003, Proceedings*, pages 162–176, 2003.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167, 1986.

A Proofs of Lemmas

The proofs of some lemmas and theorems were omitted from the main paper due to space limit. We provide them in this section for completeness.

Lemma 2. *For any pattern e and pseudorandom key renaming α_K, the distributions $[\![e]\!]$ and $[\![\alpha_K(e)]\!]$ are computationally indistinguishable.*

Proof. The pseudo-random renaming α_K can be treated as an extension of a bijection $\mu : A \to B$, where $A = \textbf{Roots}(\textbf{Keys}(e))$ and $B = \alpha_K(\textbf{Roots}(\textbf{Keys}(e)))$ are independent sets of keys. Suppose $A = \{k_1, \dots, k_n\}$. For every distinguisher $\mathcal{D}$ that can distinguish $[\![e]\!]$ and $[\![\alpha_K(e)]\!]$ with advantage η, we can build a distinguisher $\mathcal{A}$ for $[\![A]\!]$ and $[\![B]\!]$ with the same advantage. $\mathcal{A}$ takes as input a sample δ of either $[\![A]\!]$ or $[\![B]\!]$, where $\delta = \{x_1, \dots, x_n\}$ for bitstrings x_i of length κ. Given this sample, $\mathcal{A}$ computes an evaluation of e or $\alpha_K(e)$ as follows: For every atomic bit symbol $\mathsf{B} \in \mathbf{B}$, evaluate it to 0, for every $k_i \in \mathbf{Roots}(\mathbf{Keys}(e))$, $i = 1, \ldots, n$, evaluate it to x_i, and evaluate e as a natural extension according the computational evaluation rules of our expressions. Finally $\mathcal{A}$ runs $\mathcal{D}$ on the resulting evaluation, and returns whatever $\mathcal{D}$ returns.

Notice that the input to $\mathcal{D}$ is $[\![e]\!]$ if and only if the input to $\mathcal{A}$ is $[\![A]\!]$. Since A (resp. B) is an independent set of keys, $[\![A]\!]$ (resp. $[\![B]\!]$) is indistinguishable from a set of n truly random keys, and $[\![A]\!]$ and $[\![B]\!]$ are indistinguishable. Therefore η is negligible, and $[\![E]\!]$ and $[\![\alpha_K(e)]\!]$ are industinguishable. □

Theorem 4. *For any circuit $C \in \mathrm{Circuit}(s,t)$ and any boolean vector x of shape s,* $\texttt{GEval}(C, \texttt{Garble}(C, x)) = C(x)$.

Proof. Formally, the correctness of our garbling scheme can be seen as follows. Let $\tau : \mathbf{Exp}(\mathbb{B}) \to \{0,1\}$ be a function such that

$$\tau(b) = \begin{cases} 0 & \text{if } b \equiv \mathsf{B}_i \text{ for some } \mathsf{B}_i \in \mathbf{B} \\ 1 & \text{if } b \equiv \neg\mathsf{B}_i \text{ for some } \mathsf{B}_i \in \mathbf{B} \end{cases}$$

Suppose $\widetilde{C}$ and $\tilde{x}$ are the garbled circuit and the garbled input expressions for a circuit C and a boolean vector x. We claim that, if $(b, (k^0, k^1))$ is a label and (b', k') is an encoded input/output for a wire, and if z is the boolean value on that wire when evaluating C on x, then $\tau(b) = z$ and $k' = k^z$. One can easily check that our claim holds for input wires of a circuit. We show the claim holds for output wires by structural induction:

- $C \in \{\mathbf{Swap}, \mathbf{Assoc}, \mathbf{Unassoc}, \mathbf{Dup}\}$: The bits in the encoded output are exactly the bits in encoded input, so the claim holds.

- $C = \mathbf{NAnd}$: Suppose $(b_0, (k_0^0, k_0^1))$ and $(b_1, (k_1^0, k_1^1))$ are the input labels used by $\texttt{Gb}$ to generate $\widetilde{C}$. Notice that $\widetilde{C}$ is a permutation (as two swaps controlled by bits b_0, b_1) of four ciphertexts $\{\!\{ \{\!\{(b, \mathsf{K}_h^z)\}\!\}_{k_0^i} \}\!\}_{k_1^j}$ for $i, j \in \{0,1\}$ and $z = i \uparrow j$, such that $b \equiv \mathsf{B}_h$ for some $\mathsf{B}_h \in \mathbf{B}$ if and only if $i \uparrow j = 0$. Let (b_0', k_0') and (b_1', k_1') be the encoded input, and let $z_0 = \tau(b_0')$ and $z_1 = \tau(b_1')$. Then by assumption we see that the encoded output (b, K_h^z) is doubly encrypted under keys $k_0^{z_0}$,$k_1^{z_1}$ in the ciphertext corresponding to b_0', b_1'. So $\tau(b) = z = z_0 \uparrow z_1$ and our claim holds for $\mathbf{NAnd}$.

- $C = C_0 \ggg C_1$ or $C = \mathbf{First}(C')$: These two cases can be easily verified by induction.

We can extend τ to a bundle of bits in the obvious way. Our claim implies $\tau(v) = y$ where v is the encoded output of evaluating $\widetilde{C}$ on $\tilde{x}$ and $y = C(x)$. Notice that $\tau(v)$ can be computed by $\texttt{Decode}$ on v and the output mask d, so $\texttt{GEval}(\widetilde{C}, \tilde{x}) = C(x)$ and our garbling scheme is correct. □

Lemma 3. *For any $e \in \mathbf{Pat}$, the probability distributions $[\![e]\!]$ and $[\![\mathbf{p}(e, \mathbf{r}(e))]\!]$ are computationally indistinguishable.*

Proof. We first prove the case where $\mathbf{Roots}(\mathbf{Keys}(e)) \subseteq \mathbf{K}$. Let $e' = \mathbf{p}(e, \mathbf{r}(e))$. By the definition of $\mathbf{p}$, we see that e' is obtained from e by replacing all subexpressions of the form $\{\!\{e''\}\!\}_k$, where $k \in \mathbf{Keys}(e) \setminus \mathbf{r}(e)$ and $e'' \in \mathbf{Pat}(s)$, by the pattern $\{\!\{s\}\!\}_k$. Let $R = \mathbf{Keys}(e) \setminus \mathbf{r}(e)$. For any $k \in R$, the following three properties must be satisfied:

- $k \in \mathbf{K}$: If $k \notin \mathbf{K}$, then by assumption that $\mathbf{Roots}(\mathbf{Keys}(e)) \subseteq \mathbf{K}$, we have $k \in \mathbf{Keys}(e) \setminus \mathbf{Roots}(\mathbf{Keys}(e))$, and thus $k' \preceq k$ for some $k' \in \mathbf{Keys}(e)$. This implies that $k \in \mathbf{r}(e)$, a contradiction.

- $\mathsf{G}^*(k) \cap \mathbf{Parts}(e) = \emptyset$: If $k \in \mathbf{Parts}(e)$, then $k \in \mathbf{r}(e)$. If $\mathsf{G}_w(k) \in \mathbf{Parts}(e)$ for some non-empty bitstring w, then $\mathsf{G}_w(k) \in \mathbf{Keys}(e)$ and $\mathsf{G}_w(k) \prec k$, which implies $k \in \mathbf{r}(e)$.

- $\mathsf{G}^+(k) \cap \mathbf{Keys}(e) = \emptyset$: If $\mathsf{G}_w(k) \in \mathbf{Keys}(e)$ for some non-empty bitstring w, then as in the previous property, $\mathsf{G}_w(k) \prec k$ and thus $k \in \mathbf{r}(e)$.

These properties show that we can obtain $[\![e]\!]$ as follows: First we independently sample two random assignments σ and δ. For each sub-expression of e, if it has the form $\{\!|e'|\!\}_k$ for some $k \in R$, then we evaluate it using δ; otherwise we evaluate it using σ. Moreover, if we let δ be the assignment such that $\delta(\{\!|e'|\!\}_k) = \mathcal{E}(k, 0^{|s|})$ for all $k \in R$, where s is the shape of e', then the resulting distribution is just $[\![\mathbf{p}(e, \mathbf{r}(e))]\!]$. This means that we can turn a distinguisher for $[\![e]\!]$ and $[\![\mathbf{p}(e, \mathbf{r}(e))]\!]$ into an adversary attacking the IND-CPA security of the encryption scheme $(\mathcal{E}, \mathcal{D})$. Therefore $[\![e]\!]$ and $\mathbf{p}(e, \mathbf{r}(e))$ are computationally indistinguishable.

For the general case, suppose $\mathbf{Roots}(\mathbf{Keys}(e)) = \{k_1, \dots, k_n\}$ and let $\bar{\alpha}_K$ be the extension of the pseudo-random key renaming $\alpha_K : \mathbf{Roots}(\mathbf{Keys}(e)) \to \{\mathsf{K}_1, \dots, \mathsf{K}_n\}$ such that $\alpha_k(k_i) = \mathsf{K}_i$ for $i = 1, \dots, n$. Notice that $\alpha_K(\mathbf{Roots}(\mathbf{Keys}(e))) = \mathbf{Roots}(\mathbf{Keys}(\alpha_K(e)))$, so it follows from the above analysis that $[\![\alpha_K(e)]\!]$ and $[\![\mathbf{p}(\alpha_K(e)), \mathbf{r}(\alpha_K(e))]\!]$ are indistinguishable. Since $\mathbf{p}(\alpha_K(e), \mathbf{r}(\alpha_K(e))) = \mathbf{p}(\alpha_K(e), \alpha_K(\mathbf{r}(e))) = \alpha_K(\mathbf{p}(e, \mathbf{r}(e)))$, the distributions $[\![\alpha_K(e)]\!]$ and $[\![\alpha_K(\mathbf{p}(e, \mathbf{r}(e)))]\!]$ are also indistinguishable. Hence $[\![e]\!]$ and $[\![\mathbf{p}(e, \mathbf{r}(e))]\!]$ are indistinguishable. $\square$

Lemma 5. *For any circuit C and any strongly independent label expression u such that $\widetilde{C}, v = \mathtt{Gb}(C, u)$, v is strongly independent, and the following hold for all $k \in \mathbf{Keys}(v)$:*

1. $\mathsf{G}^+(k) \cap \mathbf{Keys}((\widetilde{C}, u)) = \emptyset$;
2. $\exists k' \in \mathbf{Keys}((\widetilde{C}, u)) \cap \mathbf{Parts}((\widetilde{C}, u)).k' \preceq k$.

Proof. Assume u is strongly independent. We use induction on the structure of C. As the base cases, let us first consider atomic circuits:

- $C = \mathbf{Swap}$, $\mathbf{Assoc}$, $\mathbf{Unassoc}$, or $\mathbf{Dup}$: Since $\widetilde{C} = \epsilon$, $\mathbf{Keys}((\widetilde{C}, u)) = \mathbf{Keys}(u)$ and $\mathbf{Parts}((\widetilde{C}, u)) = \mathbf{Parts}(u)$. Using directly the definition of $\mathtt{Gb}$, it is easy to check that the lemma is satisfied for these circuits.

- $C = \mathbf{NAnd}$: v contains two distinct atomic keys K_h^0 and K_h^1, so v is strongly independent. Clearly $\mathbf{Parts}(\widetilde{C}) = \{\mathsf{K}_h^0, \mathsf{K}_h^1\}$ and $\mathsf{G}^+(\mathsf{K}_h^i) \cap \mathbf{Keys}((\widetilde{C}, u)) = \emptyset$ for all $i \in \{0, 1\}$, so the lemma is satisfied.

Next, consider the composite circuits $\mathbf{First}(C')$ and $C_0 \ggg C_1$. Assume the lemma holds for all proper sub-circuits of C. Then:

- $C = \mathbf{First}(C')$: Suppose $u = (u', w)$, $v = (v', w)$, and $\widetilde{C} = \widetilde{C'}$ such that $\widetilde{C'}, v' = \mathtt{Gb}(C', u')$. Since u is strongly independent, u' is strongly independent, and by induction we have that v' is strongly independent. Fix any $k \in \mathbf{Keys}(w)$, and assume towards a contradiction that $\exists k' \in \mathsf{G}^*(k) \cap \mathbf{Keys}(v')$. Then by induction on $\widetilde{C'}$ and k',there is $k'' \in \mathbf{Keys}((\widetilde{C'}, u')) \cap \mathbf{Parts}((\widetilde{C'}, u'))$ such that $k'' \preceq k'$, and so $k'' \preceq k$ or $k \prec k''$. If $k'' \preceq k$, since u is independent, we have $k'' \notin \mathbf{Keys}(u')$, and thus $k'' \in \mathbf{Parts}(\widetilde{C'})$, which contradicts $k \in \mathbf{Keys}(w)$. If $k \prec k''$, then k'' is not atomic and $k'' \notin \mathbf{Parts}(\widetilde{C'})$, and so $k'' \in \mathbf{Keys}(u')$, which implies u is not independent, a contradiction too. Therefore $\mathsf{G}^*(k) \cap \mathbf{Keys}(v') = \emptyset$ and v is strongly independent.

  To check (1) and (2), fix any $k \in \mathbf{Keys}(v)$, and there are two cases:

  - $k \in \mathbf{Keys}(v')$: By induction on C' and u', there exists $k' \in \mathbf{Keys}((\widetilde{C'}, u')) \cap \mathbf{Parts}((\widetilde{C'}, u'))$ such that $k' \preceq k$, and (2) holds. Since v is strongly independent and $\mathsf{G}^+(k) \cap \mathbf{Keys}((\widetilde{C'}, u')) = \emptyset$, we have $\mathsf{G}^+(k) \cap \mathbf{Keys}(w) = \emptyset$. So (1) holds.

- $k \in \mathbf{Keys}(w)$: (2) holds immediately. Since u is strongly independent, $\mathsf{G}^+(k) \cap \mathbf{Keys}(u) = \emptyset$. Since v is strongly independent, $\mathsf{G}^*(k) \cap \mathbf{Keys}(v') = \emptyset$ and thus $\mathsf{G}^+(k) \cap \mathbf{Keys}(\widetilde{C'}) = \emptyset$. So (1) also holds.

- $C = C_0 \ggg C_1$: Suppose $\widetilde{C} = (\widetilde{C}_0, \widetilde{C}_1)$ where $\widetilde{C}_0, w = \mathtt{Gb}(C_0, u)$ and $\widetilde{C}_1, v = \mathtt{Gb}(C_1, w)$. By induction, w is independent and so v is independent.

  Now fix any $k \in \mathbf{Keys}(v)$. By induction, we have $\mathsf{G}^+(k) \cap \mathbf{Keys}((\widetilde{C}_1, w)) = \emptyset$, and there exists $k' \in \mathbf{Keys}((\widetilde{C}_1, w)) \cap \mathbf{Parts}((\widetilde{C}_1, w))$ such that $k' \preceq k$. There are two cases:

  - $k' \in \mathbf{Parts}(\widetilde{C}_1)$: Clearly (1) holds. We also have that k' is atomic, and so $k' = k$ and $\mathsf{G}^+(k') \cap \mathbf{Keys}((\widetilde{C}_0, u)) = \emptyset$; hence (2) holds.
  - $k' \in \mathbf{Parts}(w)$: By induction, $\mathsf{G}^+(k') \cap \mathbf{Keys}((\widetilde{C}_0, u)) = \emptyset$, and there exists $k'' \in \mathbf{Keys}((\widetilde{C}_0, u)) \cap \mathbf{Parts}((\widetilde{C}_0, u))$ such that $k'' \preceq k' \preceq k$. So both (1) and (2) hold.

Therefore the lemma follows. □

Lemma 6. *For any circuit C and any label expression u such that u is strongly independent and $\widetilde{C}, v = \mathtt{Gb}(C, u)$, if $\{\!|e|\!\}_k \in \mathbf{Parts}(\widetilde{C})$ for some expression e and some key $k \in \mathbf{Pat}(\mathbb{K})$, then the following hold:*

1. $\mathsf{G}^+(k) \cap \mathbf{Keys}(\widetilde{C}) = \emptyset$;
2. $\mathsf{G}^*(k) \cap \mathbf{Keys}(v) = \emptyset$;
3. $\exists k' \in \mathbf{Keys}((\widetilde{C}, u)) \cap \mathbf{Parts}((\widetilde{C}, u)).k' \preceq k$.

Proof. We use induction on the structure of C. For the base case, consider the atomic circuits: If $C = \mathbf{Swap}$, **Assoc**, **Unassoc**, or **Dup**, then $\widetilde{C} = \epsilon$ and the lemma holds trivially. If $C = \mathbf{NAnd}$, then $k \in \mathbf{Keys}(u) \cap \mathbf{Parts}(u)$, and one can easily verify that all three statements hold.

For a composite circuit C, assume the lemma holds for all proper sub-circuits of C. Then:

- $C = \mathbf{First}(C')$: Suppose $u = (u', w)$ and $v = (v', w)$ such that $\widetilde{C'}, v' = \mathtt{Gb}(C', u')$ and $\widetilde{C} = \widetilde{C'}$. If $\{\!|e|\!\}_k \in \mathbf{Parts}(\widetilde{C}) = \mathbf{Parts}(\widetilde{C'})$, by induction hypothesis we have $\mathsf{G}^+(k) \cap \mathbf{Keys}(\widetilde{C'}) = \emptyset$ and so (1) holds, and there exists $k' \in \mathbf{Keys}((\widetilde{C'}, u')) \cap \mathbf{Parts}((\widetilde{C'}, u'))$ such that $k' \preceq k$ and so (3) holds. Furthermore, if $k' \in \mathbf{Parts}(\widetilde{C'})$, then k' is a new atomic key introduced in $\widetilde{C'}$ and $\mathsf{G}^*(k) \cap \mathbf{Keys}(w) = \emptyset$; if $k' \in \mathbf{Parts}(u')$, then by strongly independence of u, we also have $\mathsf{G}^*(k) \cap \mathbf{Keys}(w) = \emptyset$. So $\mathsf{G}^*(k) \cap \mathbf{Keys}(v) = \emptyset$ and (2) holds.

- $C = C_0 \ggg C_1$: Suppose $\widetilde{C} = (\widetilde{C}_0, \widetilde{C}_1)$ where $\widetilde{C}_0, w = \mathtt{Gb}(C_0, u)$ and $\widetilde{C}_1, v = \mathtt{Gb}(C_1, w)$. Fix $\{\!|e|\!\}_k \in \mathbf{Parts}(\widetilde{C})$, and we have two cases:

  - $\{\!|e|\!\}_k \in \mathbf{Parts}(\widetilde{C}_0)$: By induction hypothesis we have $\mathsf{G}^+(k) \cap \mathbf{Keys}(\widetilde{C}_0) = \emptyset$, $\mathsf{G}^*(k) \cap \mathbf{Keys}(w) = \emptyset$, and $\exists k' \in \mathbf{Keys}((\widetilde{C}_0, u)) \cap \mathbf{Parts}((\widetilde{C}_0, u))$ such that $k' \preceq k$. So (3) holds immediately.

    Assume towards a contradiction that there exists $k'' \in \mathsf{G}^+(k) \cap \mathbf{Keys}(\widetilde{C}_1)$. Then k'' is not atomic and $\{\!|e'|\!\}_{k''} \in \mathbf{Parts}(\widetilde{C}_1)$ for some expression e'. So by induction on $\widetilde{C}_1$, there exists $r \in \mathbf{Keys}((\widetilde{C}_1, w)) \cap \mathbf{Parts}((\widetilde{C}_1, w))$ such that $r \preceq k''$. Then either $k \preceq r$ or $r \prec k$. There are two subcases:

    * If $r \in \mathbf{Parts}(\widetilde{C}_1)$, then r is a new atomic key introduced in $\widetilde{C}_1$ and $r \preceq k$. So $k \notin \mathbf{Keys}(\widetilde{C}_0)$, a contradiction.

* If $r \in \mathbf{Parts}(w)$, then $r \in \mathbf{Keys}(w)$. By Lemma 5 we have $\mathsf{G}^+(r) \cap \mathbf{Keys}(\widetilde{C}_0) = \emptyset$. Since $\mathsf{G}^*(k) \cap \mathbf{Keys}(w) = \emptyset$, we cannot have $k \preceq r$. So $r \prec k$, which contradicts the fact that $k \in \mathbf{Keys}(\widetilde{C}_0)$.

Both cases lead to a contradiction, so $\mathsf{G}^+(k) \cap \mathbf{Keys}(\widetilde{C}) = \emptyset$ and (1) holds.

To check (2), assume there exists $k'' \in \mathsf{G}^*(k) \cap \mathbf{Keys}(v)$. By Lemma 5, there exists $r \in \mathbf{Keys}((\widetilde{C}_1, w)) \cap \mathbf{Parts}((\widetilde{C}_1, w))$ such that $r \preceq k''$. Using the same argument as above, we see that $\mathsf{G}^*(k) \cap \mathbf{Keys}(v) = \emptyset$ and hence (2) holds.

- $\{\!|e|\!\}_k \in \mathbf{Parts}(\widetilde{C}_1)$: By induction hypothesis we have $\mathsf{G}^+(k) \cap \mathbf{Keys}(\widetilde{C}_1) = \emptyset$, $\mathsf{G}^*(k) \cap \mathbf{Keys}(v) = \emptyset$ and so (2) holds, and there exists $k' \in \mathbf{Keys}((\widetilde{C}_1, w)) \cap \mathbf{Parts}((\widetilde{C}_1, w))$ such that $k' \preceq k$. We consider two cases of k':
  * $k' \in \mathbf{Parts}(\widetilde{C}_1)$: (3) holds immediately. By Lemma 4, k' is a new atomic key in $\widetilde{C}_1$, and hence $\mathsf{G}^*(k') \cap \mathbf{Keys}(\widetilde{C}_0) = \emptyset$. So $\mathsf{G}^+(k) \cap \mathbf{Keys}(\widetilde{C}_0) = \emptyset$ and (1) holds.
  * $k' \in \mathbf{Parts}(w)$: By Lemma 5, $\mathsf{G}^+(k') \cap \mathbf{Keys}((\widetilde{C}_0, u)) = \emptyset$, and there exists $k'' \in \mathbf{Keys}((\widetilde{C}_0, u)) \cap \mathbf{Parts}((\widetilde{C}_0, u))$ such that $k'' \preceq k'$. It follows that $\mathsf{G}^+(k) \cap \mathbf{Keys}(\widetilde{C}_0) = \emptyset$ and $k'' \preceq k$. So (1) and (3) hold.

Therefore the lemma holds. □

B Yao's secret sharing scheme

Here we show how to extend our symbolic framework with the xor expressions to give a sound symbolic security proof of Yao's secret sharing scheme [Bei11, VNS+03].

Formally, we extend the syntax of $\mathbf{Exp}(\mathbb{K})$ as:

$$\begin{aligned} \mathbf{Exp}(\mathbb{K}) &\rightarrow \mathbf{Key} \mid \mathbf{Exp}(\mathbb{K}) \oplus \mathbf{Exp}(\mathbb{K}) \mid \mathsf{C}_0 \mid \mathsf{C}_1 \mid \mathsf{C}_2 \mid \cdots \\ \mathbf{Key} &\rightarrow \mathsf{K}_i \mid \mathsf{G}_0(\mathbf{Key}) \mid \mathsf{G}_1(\mathbf{Key}) \end{aligned}$$

where K_i ranges over $\mathbf{K}$, and $\mathsf{C}_0, \mathsf{C}_1, \mathsf{C}_2, \ldots$ are new constant symbols representing keys in $\{0,1\}^\kappa$. At the same time, we modify the grammar rule of $\mathbf{Exp}(\{\!|s|\!\})$ as:

$$\mathbf{Exp}(\{\!|s|\!\}) \rightarrow \{\!|\mathbf{Exp}(s)|\!\}_{\mathbf{Key}}$$

Note that the encryption expressions remain the same as in Section 2, and $\mathbf{Key} = \mathsf{G}^*(\mathbf{K}) = \mathbf{K}^*$ is the set of possible encryption keys. So symbolic properties on pseudorandom keys do not change. For all $0 \leq i < 2^\kappa$, let $\bar{i} \in \{0,1\}^\kappa$ be the binary representation of i. Any computational evaluation function σ can be extended in the obvious way:

$$\sigma(k_0 \oplus k_1) = \sigma(k_0) \veebar \sigma(k_1), \qquad \sigma(\mathsf{C}_i) = \bar{i},$$

where $\veebar$ is the bitwise xor operation on bitstrings.

We extend the congruence relation $\equiv$ with the following rules: For all $k, k', k'' \in \mathbf{Pat}(\mathbb{K})$ and for all $0 \leq i, j < 2^\kappa$, let

$$\begin{aligned} k \oplus k' &\equiv k' \oplus k, & (k \oplus k') \oplus k'' &\equiv k \oplus (k' \oplus k''), \\ k \oplus \mathsf{C}_0 &\equiv k, & k \oplus k &\equiv \mathsf{C}_0, \end{aligned}$$
$$\mathsf{C}_i \oplus \mathsf{C}_j \equiv \mathsf{C}_h \text{ for some } 0 \leq h < 2^\kappa \text{ such that } \bar{i} \veebar \bar{j} = \bar{h}.$$

Pseudorandom bit renamings and pseudorandom key renamings remain the same. We consider an additional mapping $\alpha_\oplus : \mathbf{Pat}(\mathbb{K}) \to \mathbf{Pat}(\mathbb{K})$ such that it is compatible with $\equiv$, i.e., $\alpha_\oplus(k \oplus k') \equiv \alpha_\oplus(k) \oplus \alpha_\oplus(k')$ for all $k, k' \in \mathbf{Pat}(\mathbb{K})$. Moreover, we require that, for all $k \in \mathbf{Pat}(\mathbb{K})$:

- if $k = \mathsf{C}_i$ for some C_i, then $\alpha_\oplus(k) = k$;
- if $k \in \mathbf{Key}$, then $\alpha_\oplus(k) = k \oplus \mathsf{C}_j$ for some C_j;
- if $k = k' \oplus k''$, then $\alpha_\oplus(k) = \alpha_\oplus(k') \oplus \alpha_\oplus(k'')$.

Then for all $k, k' \in \mathbf{Pat}(\mathbb{K})$, $k \equiv k'$ if and only if $\alpha_\oplus(k) \equiv \alpha_\oplus(k')$. We extend $\alpha_\oplus$ to all patterns in the obvious way. It is easy to check that for any pattern e the distributions $[\![e]\!]$ and $[\![\alpha_\oplus(e)]\!]$ are the same. Now, a pseudorandom renaming is a triple $\alpha = (\alpha_B, \alpha_K, \alpha_\oplus)$, and we write $\alpha(e) = \alpha_\oplus(\alpha_K(\alpha_B(e)))$

To compute the pattern of an expression, we keep the definitions of $\mathbf{p}$, **Keys**, and **Parts** unchanged. For any set S of keys, let $S^\oplus$ be the closure of S under $\oplus$. Then we modify the definition of $\mathbf{r}$ to include keys that can be derived using the xor operation:

$$\mathbf{r}(e) = \mathsf{G}^* \left(\left\{ k \in \mathbf{Keys}(e) \mid (k \Subset e \vee \exists k' \in \mathbf{Keys}(e). k \prec k') \right\} \right)^\oplus .$$

For any $e \in \mathbf{Pat}$, the key recovery operator $\mathcal{F}_e : \wp(\mathbf{Pat}(\mathbb{K})) \to \wp(\mathbf{Pat}(\mathbb{K}))$ has the same definition as in Section 2: for any $S \subseteq \mathbf{Pat}(\mathbb{K})$, $\mathcal{F}_e(S) = \mathbf{r}(\mathbf{p}(e, S))$. One can check that the conditions in Theorem 2 (with $\mathbf{K}$ replaced by $\mathbf{Pat}(\mathbb{K})$) still hold with these changes, and thus the extended symbolic framework is sound.

A secret sharing scheme Π for n parties $p_1, \ldots, p_n$ consists of a pair of algorithms ($\mathtt{share}$, $\mathtt{recon}$) and an access structure defined by a boolean circuit $C : \{0,1\}^n \to \{0,1\}$. Any set P of parties can be encoded using a boolean vector $x^P \in \{0,1\}^n$ such that $x^P_i = 1$ if and only of $p_i \in P$. The probabilistic algorithm $\mathtt{share}$ takes a circuit C and a secret $y \in \{0,1\}^n$, and it produces n secret shares $\{\tilde{y}_i\}_{i=1}^n$, one for each party; the algorithm $\mathtt{recon}$ takes a set of secret shares, and it outputs $y' \in \{0,1\}^n$. The scheme Π is *correct* if for any set P of parties with $C(x^P) = 1$ and any $y \in \{0,1\}^n$, if $\{\tilde{y}_i\}_{i=1}^n \leftarrow \mathtt{share}(C, y)$, then $\mathtt{recon}(\{\tilde{y}_j\}_{p_j \in P}) = y$. It is *computationally secure* if for any $y_0, y_1 \in \{0,1\}^n$, if $\{\tilde{y}_{h,i}\}_{i=1}^n \leftarrow \mathtt{share}(C, y_h)$ for $h \in \{0,1\}$, then for any set P of parties such that $C(x^P) = 0$, the distributions $\{\tilde{y}_{0,j}\}_{p_j \in P}$ and $\{\tilde{y}_{1,j}\}_{p_j \in P}$ are computationally indistinguishable.

Yao's secret sharing scheme Π is a computational secret sharing scheme for monotone boolean circuits, i.e., circuits that consist of AND and OR gates and have a single bit output. To describe such circuits in our inductive circuit notation, we remove **NAnd** and add **And** and **Or** circuits: both of **And** and **Or** have two input wires and one output wire, and they compute the boolean and and or functions, respectively. We refer the readers to [VNS+03] for a detailed formulation and analysis in the computational setting. In the symbolic setting, we can describe $\mathtt{share}$ and $\mathtt{recon}$ as the following functions:

- $\mathtt{share}$ takes a circuit C with a single bit output and a boolean vector $y \in \{0,1\}^n$, and it outputs an expression consisting of n shares $(\mathtt{ct}, k_i)$ for $i = 1, \ldots, n$, where $\mathtt{ct}$ represents the ciphertexts that are known to every party, and k_i is the key private to p_i.
- $\mathtt{recon}$ takes a circuit C and an expression representing n shares, and it outputs an expression representing the secret that could be reconstructed from the shares. Note that, when reconstructing from a set P of $m < n$ parties, if a party $p_i \notin P$ then we set the key k_i in the input expression to K_l for some K_l that would not appear in $\mathtt{share}(C, y)$.

$\mathtt{share} :: \mathrm{Circuit}(s, \circ) \times \{0,1\}^n \to \mathbf{Exp}$
$\mathtt{share}(C, y) = ((\mathtt{ct}, k_1), \ldots, (\mathtt{ct}, k_n))$ where
  $(\mathtt{ct}, v) = \mathtt{sh}(C, \mathsf{C}_y)$
  $(k_1, \ldots, k_n) = v$

$\mathtt{sh} :: \mathrm{Circuit}(s, t) \times \mathbf{Exp} \to \mathbf{Exp}$
$\mathtt{sh}(\mathbf{And}, k) = (\epsilon, (\mathsf{K}_h, k \oplus \mathsf{K}_h))$ where
  $h \leftarrow \mathbf{new}$
$\mathtt{sh}(\mathbf{Or}, k) = (\epsilon, (k, k))$
$\mathtt{sh}(\mathbf{Dup}, (k_i, k_j)) = ((\{\!|k_i|\!\}_{\mathsf{K}_h}, \{\!|k_j|\!\}_{\mathsf{K}_h}), \mathsf{K}_h)$ where
  $h \leftarrow \mathbf{new}$
$\mathtt{sh}(\mathbf{Swap}, (u, v)) = (\epsilon, (v, u))$
$\mathtt{sh}(\mathbf{Assoc}, (u, (v, w))) = (\epsilon, ((u, v), w))$
$\mathtt{sh}(\mathbf{Unassoc}, ((u, v), w)) = (\epsilon, (u, (v, w)))$
$\mathtt{sh}(C_0 \ggg C_1, w) = ((\mathtt{ct}_0, \mathtt{ct}_1), u)$ where
  $(\mathtt{ct}_1, v) = \mathtt{sh}(C_1, w)$
  $(\mathtt{ct}_0, u) = \mathtt{sh}(C_0, v)$
$\mathtt{sh}(\mathbf{First}(C), (v, w)) = (\mathtt{ct}, (u, w))$ where
  $(\mathtt{ct}, u) = \mathtt{sh}(C, v)$

$\mathtt{recon} :: \mathrm{Circuit}(s, \circ) \times \mathbf{Exp} \to \mathbf{Exp}$
$\mathtt{recon}(C, ((\mathtt{ct}, k_1), \ldots, (\mathtt{ct}, k_n))) = u$ where
  $u = \mathtt{re}(C, \mathtt{ct}, (k_1, \ldots, k_n))$

$\mathtt{re} :: \mathrm{Circuit}(s, t) \times \mathbf{Exp} \times \mathbf{Exp} \to \mathbf{Exp}$
$\mathtt{re}(\mathbf{And}, \mathtt{ct}, (k_0, k_1)) = k_0 \oplus k_1$
$\mathtt{re}(\mathbf{Or}, \mathtt{ct}, (k, k)) = k$
$\mathtt{re}(\mathbf{Dup}, (\mathtt{ct}_0, \mathtt{ct}_1), k) = (k_0, k_1)$ where
  $\{\!|k_0|\!\}_k = \mathtt{ct}_0$
  $\{\!|k_1|\!\}_k = \mathtt{ct}_1$
$\mathtt{re}(\mathbf{Swap}, \mathtt{ct}, (v, u)) = (u, v)$
$\mathtt{re}(\mathbf{Assoc}, \mathtt{ct}, ((u, v), w)) = (u, (v, w))$
$\mathtt{re}(\mathbf{Unassoc}, \mathtt{ct}, (u, (v, w))) = ((u, v), w)$
$\mathtt{re}(C_0 \ggg C_1, (\mathtt{ct}_0, \mathtt{ct}_1), u) = w$ where
  $v = \mathtt{re}(C_0, \mathtt{ct}_0, u)$
  $w = \mathtt{re}(C_1, \mathtt{ct}_1, v)$
$\mathtt{re}(\mathbf{First}(C), (u, w)) = (\mathtt{ct}, (v, w))$ where
  $v = \mathtt{re}(C, \mathtt{ct}, u)$

To show that this scheme is secure, let us fix any monotone boolean circuit C with n input wires and a set $P = \{p_{i_1}, \ldots, p_{i_m}\}$ of m parties such that $C(x^P) = 0$. One can show that the following lemma holds:

Lemma 9. *For any $y \in \{0,1\}^n$, let $((\mathtt{ct}, k_1), \ldots, (\mathtt{ct}, k_n)) = \mathtt{share}(C, y)$, and let $e = (\mathtt{ct}, (k_{i_1}, \ldots, k_{i_m}))$. If $\mathsf{C}_y \oplus k \in \mathrm{Fix}(\mathcal{F}_e)$ for some $k \in \mathbf{Pat}(\mathbb{K})$, then $k \notin \mathrm{Fix}(\mathcal{F}_e)$.*

Fix any $0 \le y_0, y_1 < 2^n$. For $h \in \{0,1\}$, let $((\mathtt{ct}^h, k_1^h), \ldots, (\mathtt{ct}^h, k_n^h)) = \mathtt{share}(C, y_h)$, and let $e_h = (\mathtt{ct}^h, (k_{i_1}^h, \ldots, k_{i_m}^h))$. If $\mathsf{C}_{y_0} \oplus k \Subset e_0$ then let $\alpha_\oplus^0$ be such that $\alpha_\oplus^0(k) \equiv k \oplus \mathsf{C}_{y_0}$; otherwise let $\alpha_\oplus^0$ be the identity map on k. Similarly we can define $\alpha_\oplus^1$ for e_1. Let α_B and α_K be identity maps. One can check that $\alpha_\oplus^0(\mathbf{Pattern}(e_0)) \equiv \alpha_\oplus^1(\mathbf{Pattern}(e_1))$, and thus $\mathbf{Pattern}(e_0)$ and $\mathbf{Pattern}(e_1)$ are equivalent up to the pseudorandom renaming α. Therefore Π is computationally secure.