Card-Based Cryptography Meets Formal Verification*

Alexander Koch, Michael Schrempp, and Michael Kirsten

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
alexander.koch@kit.edu, michi.schrempp@freenet.de, kirsten@kit.edu

Abstract. Card-based cryptography provides simple and practicable protocols for performing secure multi-party computation (MPC) with just a deck of cards. For the sake of simplicity, this is often done using cards with only two symbols, e.g., ♣ and ♡. Within this paper, we also target the setting where all cards carry distinct symbols, catering for use-cases with commonly available standard decks and a weaker indistinguishability assumption. As of yet, the literature provides for only three protocols and no proofs for non-trivial lower bounds on the number of cards. As such complex proofs (handling very large combinatorial state spaces) tend to be involved and error-prone, we propose using formal verification for finding protocols and proving lower bounds. In this paper, we employ the technique of software bounded model checking (SBMC), which reduces the problem to a bounded state space, which is automatically searched exhaustively using a SAT solver as a backend.

Our contribution is threefold: (a) We identify two protocols for converting between different bit encodings with overlapping bases, and then show them to be card-minimal. This completes the picture of tight lower bounds on the number of cards with respect to runtime behavior and shuffle properties of conversion protocols. For computing AND, we show that there is no protocol with finite runtime using four cards with distinguishable symbols and fixed output encoding, and give a four-card protocol with an expected finite runtime using only random cuts. (b) We provide a general translation of proofs for lower bounds to a bounded model checking framework for automatically finding card- and run-minimal (i.e., the protocol has a run of minimal length) protocols and to give additional confidence in lower bounds. We apply this to validate our method and, as an example, confirm our new AND protocol to have its shortest run for protocols using this number of cards. (c) We extend our method to also handle the case of decks on symbols ♣ and ♡, where we show run-minimality for two AND protocols from the literature.

* This is an authors' manuscript version of an article to appear in "New Generation Computing", and constitutes an extended version of a proceedings paper with the same title that appeared at ASIACRYPT 2019 with DOI 10.1007/978-3-030-34578-5_18 [KSK19]. We replaced the proof sketch of Theorems 1 and 2 with a full formal version. Moreover, we adapted our verification tool to handle more general decks, which allows us to additionally show run-minimality of two-color deck AND protocols from the literature. These new results are mainly described in Sections 8 and 9.

Keywords: secure multiparty computation · card-based cryptography · formal verification · bounded model checking · standard decks · two-color decks

1 Introduction

Card-based cryptographic protocols allow to perform secure multi-party computation (MPC), i.e., jointly computing a function while not revealing more information about each individual input than absolutely necessary, with just a (regular) deck of playing cards, as long as they have indistinguishable backs. Let us start with an example. Assume that Alice and Bob meet in a bar and spend the evening together. After quite some chat, they would like to find out whether to have a second date. They are faced with the following problem: In case only one of them likes to meet again, this would cause an uncomfortable embarrassment, if he or she is the first to come out.[1] Fortunately, Alice is a notable cryptographer and likes card games, so she has with her a standard deck of cards. She remembers the protocol by Niemi and Renvall [NR99] for computing the AND function of two bits, here for outputting "yes", if both players share this mutual interest, and "no" otherwise. Using this MPC protocol hides the input of the respective other player, unless it is obvious from their own input and output, hence hiding a "yes"-choice given of only one player, from the other.

In order to get a feeling for how such card-based protocols work, let us introduce the said protocol by Niemi and Renvall. The protocol uses five cards with distinguishable symbols, which we denote – for simplicity[2] – as $\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}$ and $\boxed{5}$. It is essential that the cards' backs are indistinguishable, such that when they are put face-down on the table, the only thing observable is $\boxed{}\,\boxed{}\,\boxed{}\,\boxed{}\,\boxed{}$. With these cards, the two players can encode a commitment to a bit (yes or no) by the order of two cards $\boxed{i}\,\boxed{j}$, $i, j \in \{1, \ldots, 5\}$ (with $i \neq j$) via the encoding

$$\boxed{i}\,\boxed{j} \mathrel{\hat{=}} \begin{cases} 0, & \text{if } i < j, \\ 1, & \text{if } i > j. \end{cases}$$

Alice inputs her bit by putting the cards $\boxed{1}\,\boxed{2}$ face-down and in the respective order on the table (she puts $\boxed{1}\,\boxed{2}$ for input 0, and $\boxed{2}\,\boxed{1}$ for input 1), while Bob does the same using his cards $\boxed{3}\,\boxed{4}$. We need an additional helper-card, here a $\boxed{5}$, which is put to the left of the two players' cards.

The protocol starts by swapping Alice's second card with Bob's first card in the card sequence (pile) on the table. The resulting card configuration has an interesting property, namely that the order of the cards $\boxed{1}$ and $\boxed{4}$ in this sequence already encodes the output of the protocol, i.e., it reads $\boxed{4}\,\boxed{1}$ if the output is 1, and $\boxed{1}\,\boxed{4}$ otherwise. Hence, by securely removing the cards $\boxed{2}$ and

[1] This is known as the "dating problem".

[2] Alice and Bob in the story might, e.g., use 7, 8, 9, 10, and a queen with any symbol.

[3] (which is explained below), one directly obtains the output. We see this by inspecting all possible cases:

Bits	Input sequence	After swap	Removing [2]+[3]
$(0,0)$	[5] [1] [2] [3] [4]	[5] [1] [3] [2] [4]	[5] [1] x x [4]
$(0,1)$	[5] [1] [2] [4] [3]	[5] [1] [4] [2] [3]	[5] [1] [4] x x
$(1,0)$	[5] [2] [1] [3] [4]	[5] [2] [3] [1] [4]	[5] x x [1] [4]
$(1,1)$	[5] [2] [1] [4] [3]	[5] [2] [4] [1] [3]	[5] x [4] [1] x

We can remove the cards [2] and [3], while keeping the relative order of all cards in the sequence intact, by cutting the cards, i.e., rotating the sequence by a random offset which is unknown to the players. We can then securely turn the first card and remove it in case it is a [2] or a [3]. Due to the cut, the turned card is random and hence does not reveal anything about the inputs. When both cards are removed, we reach a configuration where [5] is the first card by the same procedure where the two remaining cards encode the AND result. Here, the [5] played the crucial role of a separator that keeps the relative order of the remaining cards – starting from the separator – intact, when doing a random cut. (A formal version of this protocol is given in Protocol 2 and Figure 7.)

In this paper, we are interested in whether we can do away with the helping card [5], and whether there are simpler protocols. Moreover, in order to handle the increasing combinatorial state space (relative to protocols on decks of just ♣ and ♡), we introduce formal verification to the field of card-based cryptography.

1.1 Secure Multiparty Computation with Cards

In combining different protocols, one can do much more than just computing the AND function. For example, it is possible to compute arbitrary Boolean circuits by combining the well-known fact that any circuit can be expressed using only NOT and AND gates, with a method to duplicate the physically encoded bit in case of forking wires, which we make explicit by a COPY gate. In the encoding above, NOT simply inverts the order of the two cards, and a COPY protocol is given, e.g., by Mizuki [M16]. Using this setup, we can do general MPC for any function *without needing to trust a possibly corrupted computer*.

A particular advantage of protocols using physical assumptions is that they can provide a *bridge to reality*. Examples are given by Glaser, Barak, and Goldston as well as Fisch, Freund, and Naor, who give a protocol for proving in zero-knowledge that a nuclear warhead (to be disarmed due to an international treaty) conforms to a prescribed template, without giving away anything about its internal design [GBG14; FFN14]. In our setting of cryptography with cards, this bridge is used if the cryptographic protocol is embedded in a real card game, e.g., to prevent cheating[3]. Here, using computers is not only cumbersome, but there

[3] As an example, in a Duplicate Bridge tournament, one might prove that all sessions are handed the same cards, eliminating the need of a trusted dealer (no pun intended).

is no guarantee that the card sequence on a player's hand is the one he or she or he inputs into the software, hence we have no bridge to the physical world.

Another application of such protocols is to explain MPC in an interesting and motivating way to students in cryptography lectures. Card-based cryptography tries to find protocols for the above-mentioned AND and COPY functionalities which are card-minimal, simple and practicable. For simplicity, many protocols in card-based cryptography work with specially constructed decks, e.g., of only the two symbols ♣ and ♡. This is easy for explanation, and there are nice and easily describable protocols, such as the five-card trick by den Boer [dB89] and the six-card AND protocol by Mizuki and Sone [MS09].

However, the setting where all cards are distinguishable, as described above, has several advantages. Firstly, we assume little about the indistinguishability of cards, which leads to stronger security guarantees. (This is closer to the indistinguishable version of tamper-evident seals, e.g., scratch-off cards, by Moran and Naor [MN10].) We only need the backs (or envelopes wrapping the cards, if one wishes) to be indistinguishable. Secondly, these standard decks are more commonly available, in contrast to constructed decks. If one were to use standard decks for the protocols above, they would need multiple copies of the same card. Thirdly, this setting may lead to protocols using less cards than the optimal ones in the two-color deck setting. In fact, as our paper shows, one may use less cards than in the two-color deck setting. For example, our new four-card Las Vegas AND protocol presented in Section 5 uses only a very basic, practicable shuffling mechanism, namely random cuts, and uses one card less than the provably card-minimal Las Vegas AND protocol (restricted to certain types of practical shuffles) in the two-color deck setting. As of yet, there has only been little research in this direction, with [NR99; M16] being the only works that consider the setting where all cards have distinguishable symbols, called "standard deck" setting. Nothing is known about non-trivial lower bounds on the number of cards, which is likely due to the large state space, as there are many more distinguishable card re-orderings compared to the two-color case.

Within this paper, our interest is to find an automatic way of constructing compact card-based protocols which are secure and correct, based on only the two standard operations *turn* and *shuffle*, given the desired number of cards. We exploit the observation that, to the best of our knowledge, all findings in the literature employ only protocols with runs of comparatively small length using only a small number of cards. Based on the hypothesis that we may always find some number n which is greater than or equal to any run-minimal card-protocol, we apply the automatic off-the-shelf formal program-verification technique *software bounded model checking (SBMC)* [BCC+99]. This technique allows, given such a bound n, to encode a program verification task into a decidable set of logical equations, which can then be solved by a SAT or an SMT solver. In this work, we propose an automatic method based on SBMC that, given the desired numbers of cards and protocol length, either constructs such a protocol if and only if one exists, or proves the underlying SAT formula to be unsatisfiable, i.e., shows that no such protocol exists. Based thereon, we

propose that the cumbersome and error-prone task of finding such protocols or proving their non-existence by hand may be supported or complemented by such an automatic approach which is flexibly adaptable to a variety of card-based protocols and desired restrictions.

Prior to our work, it was not yet clear which role the input encoding plays when devising new protocols. This is the question on whether it can make a difference regarding the possibility of a protocol if we provide, e.g., $\boxed{1}\,\boxed{2}$ to Alice and $\boxed{3}\,\boxed{4}$ to Bob, or $\boxed{1}\,\boxed{3}$ to Alice and $\boxed{2}\,\boxed{4}$ to Bob. We provide an analysis of this question, showing that with certain restrictions, there is a relatively large freedom in choosing the input (and/or output) bases. This is a useful prerequisite in proving the impossibility of a protocol with a given number of cards.

1.2 Contribution

Our contribution consists in providing interesting new protocols and impossibility results, as well as a fully automatic method based on formal verification to support such findings. The specific advances therein are the following (cf. also Table 1 for a comparison to the literature):

(1) A four-card AND protocol in the standard deck setting, improving upon the work by Niemi and Renvall [NR99] by one card, and reaching the theoretical minimum on the number of cards. W.r.t. shuffling, this protocol only uses an expected number of 6 random cuts, compared to 7.5 random cuts in a (shortened) variant of Niemi and Renvall [NR99]. Additionally, the protocol has a natural interpretation and the fact that it uses only random cuts makes it particularly easy to implement in an actively secure way [KW20].

(2) We show that under certain conditions the cards for encoding input or output can be chosen freely. For one-bit output protocols and if five or more cards are available, we can freely choose both input and output bases by only extending the protocol by expected three shuffle and three turn steps. For this matter, we identify two protocols for converting a bit encoding if the new encoding shares one card with the old one.

(3) We show that there is no finite-runtime protocol for converting between bases with non-empty intersection using four cards. Moreover, there cannot be a finite-runtime AND protocol with four cards if we fix the basis in advance.

(4) We introduce formal verification to card-based cryptography by providing a technique which automatically finds new protocols using as few as possible operations and searches for lowest bounds on card-minimal protocols.

(5) We extend this technique to more general decks, and show two AND protocols in the *two-color* (or *two-symbol*) deck setting, i.e., where the deck constitutes a multiset of cards on symbols ♣ and ♡, to be run-minimal. Finally, we employ our formal verification method to give a formal guarantee that we may safely reduce the maximal permutation set size and thereby optimize the running time for our protocol-finding technique in Section 8. This is due to the fact that in the two-color setting the number of possible sequences in a protocol state may be significantly smaller than the number of possible permutations on the deck.

Table 1. Minimum number of cards required by AND and basis conversion protocols, subject to the running time and shuffle restrictions specified in the first two columns. Note that random cuts are a subclass of uniform closed shuffles.

Running Time	Shuffle Restr.	#Cards	Protocol	Lower Bound
AND Protocols:				
Las Vegas	random cuts	4	Theorem 3	– (trivial)
finite	–	$\geq 5^{a}, \leq 8$	[M16, Sect. 3.4]	Theorem 2
finite	uniform closed	$\geq 5^{a}, \leq 8$	[M16, Sect. 3.4]	Theorem 2
Disjoint Basis Convert Protocols:				
finite	uniform closed	4	[M16, Sect. 3.2]	– (trivial)
Overlapping Basis Convert Protocols:				
Las Vegas	random cuts	3	Theorem 4	– (trivial)
finite	–	5	Theorem 5	Theorem 1
finite	uniform closed	5	Theorem 5	Theorem 1

[a] Lower bound result only holds for fixed output basis, flexible case is still open.

1.3 Related Work

The feasibility of card-based cryptographic MPC is due to den Boer [dB89], Crépeau and Kilian [CK93], and Niemi and Renvall [NR98], with a formal model given by Mizuki and Shizuya [MS14]. The only two papers looking at standard deck solutions are by Niemi and Renvall [NR99] and Mizuki [M16]. Lower bounds on card-based cryptographic protocols are given by Koch, Walzer, and Härtel [KWH15], Kastner et al. [KKW+17], and Koch [K18] for the two-color deck setting. The card-minimal protocol for this setting, using only practicable (i.e., uniform closed) shuffles, is given by Abe et al. [AHM+18] and uses five cards. The state trees used for protocols in this paper are devised by Koch, Walzer, and Härtel [KWH15] and Kastner et al. [KKW+17].

To the best of our knowledge, this is the first work which applies formal methods to the field of card-based cryptography. However, a large range of research has been done using formal methods in the more general field of secure two-party and multiparty computations. This can be clustered into either analyzing security protocols given as high-level, abstract (and usually idealized) models, or program-based approaches targeting real(istic) protocol (software) implementations. Avalle, Pironti, and Sisto [APS14] further structure this into the two main approaches of automated model extraction and automated code generation. We refer the interested reader to overviews as given by Blanchet [B12] or Avalle, Pironti, and Sisto, and only go into a few selected works for which we identified closer links to our approach, e.g., using software bounded model checking (SBMC), SAT solvers on real(istic) protocol implementations, or relating in the analyzed security model. Standard cryptographic assumptions using lower-level computational models are – albeit more realistic – usually harder to formalize and automate. One notable line of research is CBMC-GC [FHK+14] which builds on top of

the tool CBMC [CKL04]. It uses SBMC in a compiler framework translating secure computations of ANSI C programs into an optimized Boolean circuit which can subsequently be implemented securely utilizing the garbled circuit approach. Another similar setting to ours is analyzed by Rastogi, Swamy, and Hicks [RSH19], who also assume an "honest-but-curious" attacker model. Therein, a domain-specific language is built on top of the F* language, a full-featured, verification-oriented, effectful programming language by Swamy et al. [SHK+16]. Swamy et al. then implement MPC programs with enabled formal verification provided by the semantics of the language.

1.4 Outline

We give the computational model of card-based protocols, security definitions, etc. and the necessary preliminaries as well as a basic setup for software bounded model checking in Section 2. Section 3 discusses which freedom one has when choosing the specific cards for encoding inputs and outputs to card-based protocols and introduces a formal relabeling operation. We give lower bounds on the number of cards for AND and basis-conversion protocols in Section 4. A four-card Las Vegas AND protocol and two basis-conversion protocols are presented in Section 5 and Section 6, respectively. Section 7 gives results from applying our formal verification setup based on SBMC to our new AND protocol. In Sections 8 and 9, we describe our new results for the two-color deck case.

2 Preliminaries

In this section, we first formally introduce card-based protocols with their computational model (including some basic required notions), a convenient formal protocol representation, a suitable security notion, and the formal requirements for proving lower bounds. Secondly, we introduce our applied formal technique called software bounded model checking, on which, thirdly, we establish our general technique for automatically finding card- and run-minimal protocols.

2.1 Card-Based Protocols

Formally, a *deck* $\mathcal{D}$ of cards is a multiset over a *(deck) alphabet* or symbol set Σ. We denote multisets by $[\![\cdot]\!]$, e.g., $[\![\heartsuit, \heartsuit, \clubsuit, \clubsuit]\!]$ is a deck over $\{\heartsuit, \clubsuit\}$. In this paper, except for Sections 8 and 9, we focus mainly on decks $\mathcal{D} = [\![1, \ldots, n]\!]$, $n \in \mathbb{N}$, where each symbol occurs exactly once. Following Mizuki [M16], we call these decks *standard decks*, because decks of common card games are a good representation of such formal decks.

A card that is lying on a table (as usual in card-based protocols) can have two orientations, namely *face-up* (showing the symbol of the card), or *face-down*. A special *back symbol* '?' that is not part of Σ represents what is visible about a card that is turned face-down. In this way, we can describe a *card lying on the table* by a fraction symbol $\frac{a}{b}$, where exactly one of a and b is '?', and the other is

a symbol from Σ. Here, a represents part that is visible from the card when it lies down, and hence $\frac{a}{b}$ is a face-down card if $a = ?$, and a face-up card if $a \in \Sigma$. As card-based protocols usually involve some turning-over of the cards, this status will likely change during a protocol, causing the numerator and denominator to be swapped.

Card-based protocols then proceed on sequences of such cards $(\alpha_1, \ldots, \alpha_{|\mathcal{D}|})$ where all cards from the deck $\mathcal{D}$ are lying on the table as just described and in the given order. The *visible sequence* of such a sequence then arises by just taking the "visible" numerator of all cards. For example, if the sequence is $(\frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{\clubsuit}{?}, \frac{?}{\clubsuit})$, the corresponding visible sequence of the cards is $(?, ?, \clubsuit, ?)$. The *sequence trace* of a finite protocol run, and analogously its *visible sequence trace*, is then the sequence of all card sequences and visible sequences, respectively, as they arise during the run. Let $\mathsf{Seq}^{\mathcal{D}}$ denote the set of sequences on deck $\mathcal{D}$.

In the following we will often just use the *symbol sequence* that contains only the card symbols ($(\heartsuit, \heartsuit, \clubsuit, \clubsuit)$ in the example above) as a shorthand for the corresponding face-down cards. This is due to Kastner et al. [KKW+17, Cor. 2 and Lem. 4], who showed that it does not increase the computational power of card-based protocols to leave cards face-up longer than necessary, and that one can safely assume that any face-down cards that are turned over during a step in the protocol are directly turned back after learning its symbol.

For encoding a bit, we additionally assume a linear order on the card symbols in Σ, which is the usual order on $\mathbb{N}$ for standard decks, and $\clubsuit < \heartsuit$ for simple two-element decks. Two face-down cards with distinct symbols $s_1, s_2 \in \Sigma$ then *encode a bit* via the following encoding rule introduced by Niemi and Renvall [NR99]:

$$s_1\, s_2 \mathrel{\hat{=}} \begin{cases} 0, & \text{if } s_1 < s_2, \\ 1, & \text{if } s_1 > s_2. \end{cases}$$

Card-based protocols proceed by mainly two actions on the sequence or pile of cards: We can introduce uncertainty (about which card is which) by shuffling them in arbitrary or in certain controlled ways, e.g., by cutting the cards in quick succession, so that players do not know which card ended up where in the card sequence (or pile). Slightly more formal, a (uniform) shuffle is specified by a permutation set, from which one element is drawn uniformly at random and applied to the cards, without the players learning which one it was. Secondly, we may turn over cards and publicly learn their symbol, and act on the basis of this information. Moreover, we may deterministically permute the cards.

A protocol computes a Boolean function $f\colon \{0,1\}^2 \to \{0,1\}$ if the possible start sequences, corresponding to the player inputs $b \in \{0,1\}^2$, do encode these inputs as described above, and that the cards that are declared to contain the output value upon termination of the protocol, do encode the output value $o = f(b)$ for each respective input $b \in \{0,1\}^2$ as described above. A more formal definition in terms of the tree representation introduced in Section 2.1 is given at the end of that section.

Permutations and Groups. Let S_n denote the *symmetric group* on $\{1,\dots,n\}$. For elements $x_1,\dots,x_k \in \{1,\dots,n\}$ the *cycle* $(x_1\ x_2\ \dots\ x_k)$ is the *cyclic* permutation π with $\pi(x_i) = x_{i+1}$ for $1 \le i < k$, $\pi(x_k) = x_1$ and $\pi(x) = x$ for all x not occurring in the cycle. Every permutation can be written as a composition of pairwise disjoint cycles. For example, $(1\ 3\ 2)(4\ 5)$ maps $1 \mapsto 3, 3 \mapsto 2, 2 \mapsto 1, 4 \mapsto 5$, and $5 \mapsto 4$. The identity permutation is denoted as id.

Given permutations $\pi_1,\dots,\pi_k \in S_n$, $\langle \pi_1,\dots,\pi_k\rangle$ denotes the group generated by $\pi_1,\dots,\pi_k$. A shuffle is a *random cut* if its permutation set is the group $\langle\pi\rangle = \{\pi^0,\dots,\pi^{l-1}\}$ generated by a single element π *which is a cycle* $(x_1\ x_2\ \dots\ x_l)$. A shuffle is called a *random bisection cut* if its permutation set is generated by a π which is the composition of pairwise disjoint cycles of length 2. Finally, an S_k*-shuffle* is a shuffle with permutation set S_k.

Computational Model and Protocol State Tree Representation. For our formal descriptions, we make heavy use of the KWH trees introduced by Koch, Walzer, and Härtel [KWH15] and shown to be equivalent to the computational model by Mizuki and Shizuya [MS14; MS17] in the work by Kastner et al. [KKW+17]. For this matter, let us first describe what a state during a run of a card-based protocols is. We start by an example, namely the state of a protocol in the very beginning, i.e., after the players have put their cards encoding their inputs on the table:

12 34	X_{00}
12 43	X_{01}
21 34	X_{10}
21 43	X_{11}

As mentioned above, we resort to only write symbol sequences instead of full card sequences. Each line in the state as depicted by the above boxed information rows describes a card sequence that is possible at this point in time in the protocol, together with a certain type of polynomial in the variables $X_{00}, X_{01}, X_{10}, X_{11}$. For example, the first line of the state can be read as "the sequence $(\frac{?}{1}, \frac{?}{2}, \frac{?}{3}, \frac{?}{4})$ lies at the table with the symbolic probability X_{00}", i.e., with the probability that $(0,0)$ is the input of the protocol (which is left as a variable, instead of a concrete value, as the input distribution can be arbitrary). Note that 12, and 34 encode 0 as required for input $(0,0)$ and that the order of the rows is of no significance in the above depiction. We capture the notion of a state more formally in the following definition:

Definition 1 (State). *Let $\mathcal{D}$ be a deck of a protocol $\mathcal{P}$ computing a Boolean function $f\colon \{0,1\}^2 \to \{0,1\}$. A* state *$\mu$ of $\mathcal{P}$ is a map $\mu\colon \mathsf{Seq}^{\mathcal{D}} \to \mathbb{X}_2$, where $\mathbb{X}_2$ denotes the polynomials over the variables X_b for $b \in \{0,1\}^2$ of the form $\sum_{b\in\{0,1\}^2} \beta_b X_b$, for $\beta_b \in [0,1] \subset \mathbb{R}$, and $\mu(s)$ for $s \in \mathsf{Seq}^{\mathcal{D}}$ is interpreted as the probability that s is the actual sequence on the table, in terms of the symbolic probabilities on the inputs.*

Defined this way, the boxes drawn throughout the paper are just depictions of such a (state) map, i.e., we just write down all sequences $s \in \mathsf{Seq}^{\mathcal{D}}$ that are not

assigned probability 0, and annotate it to their right with the polynomial $\mu(s)$. (An alternative characterization of a state is given by Koch [K19, Def. 7.1].)

Every standard-deck protocol starts by a state as above:

$$\boxed{\begin{array}{ll} 12\,34 & X_{00} \\ 12\,43 & X_{01} \\ 21\,34 & X_{10} \\ 21\,43 & X_{11} \end{array}}$$

but we eventually add further cards ($\boxed{5}$, $\boxed{6}$, ...) if the deck is larger to the right of the players bits. The state (or KWH) tree of a protocol is then a directed tree where the nodes are states as above, with annotations at the outgoing edges of each state, specifying the action that is performed next. Let μ be the state with the outgoing annotation, then the possible actions are defined as:

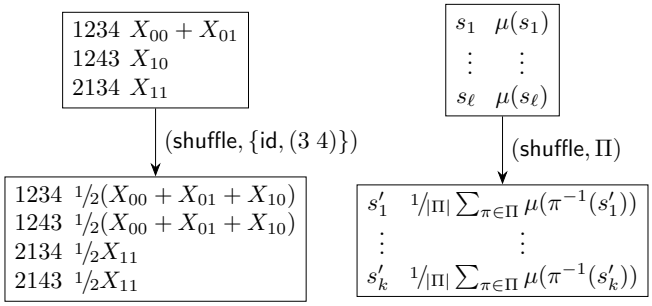


Fig. 1. A shuffle operation, given by example (left), and via the general rule (right).

1. $(\mathsf{shuffle}, \Pi)$ leads to a μ' as in Figure 1, where $\Pi \subseteq S_{|\mathcal{D}|}$ is a permutation set.
2. (turn, T) branches the tree into states μ_v for each observation v possible by revealing the cards at positions from the set $T \subseteq \{1, \ldots, |\mathcal{D}|\}$, as in Figure 2. μ_v contains the sequences from μ which are compatible with the observation v. For each sequence s compatible with v, we have $\mu_v(s) := \mu(s)/\Pr[v]$, where $\Pr[v] \in (0, 1]$ is the probability of observing v. Note that we omit the implicit operation to turn the card back face-down, as motivated above.
3. (perm, π) permutes the sequences of μ according to π.
4. $(\mathsf{result}, p_1, p_2)$ stops the computation and returns the cards at p_1, p_2 as output.

A protocol computes a Boolean function $f\colon \{0,1\}^2 \to \{0,1\}$ if the start state (tree root) encodes each $b \in \{0,1\}^2$ in the first four cards (the remaining cards being at fixed positions), and in the leaf nodes of the protocol's state tree, it holds for the positions given by the result operation that the cards at these positions encode a value $o \in \{0,1\}$ if all X_i occurring in $\mu(s)$ for sequence s satisfy $f(i) = o$ *(Correctness)*.

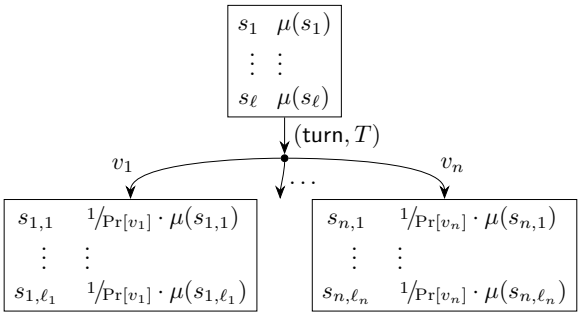


Fig. 2. A turn operation. Here, $v_1, \ldots, v_n$, are the possible observation by turning the cards at positions in T. For each $i \in \{1, \ldots, n\}$ the $s_{i,1}, \ldots, s_{i,\ell_i}$ are the sequences from $s_1, \ldots, s_\ell$ which are compatible with v_i. Note that in secure protocols, the probability of observing v_i, denoted as $\Pr[v_i]$, is constant.

We say that a protocol has *finite runtime* if its tree is finite. It is a *Las Vegas* protocol, if it is not finite runtime, but the expected length of any path in its tree, i.e., the expected value of the length of an arbitrary descending path in the tree starting from the root (as a random variable, where the randomness is in the choice of the path), is finite. Note that while we consider looping protocols, we do not consider the case where a complete restart is necessary. For self-similar infinite trees, we simplify by drawing edges to earlier states.

Security of Card-Based Protocols. We slightly adjust the security notion from the literature to standard decks. For more details, we refer to Koch [K19]. Since different encodings for the same bit are possible, we want the encoding basis of the output bit to not give away anything about the inputs. We say that a protocol is *secure* if at any turn operation the probability for each observation v is a constant $\rho \in [0, 1]$ (using $\sum_{i \in \{0,1\}^2} X_i = 1$), *and additionally* if at any result operation the probability of each output basis is constant in the same sense.

Similar to the work by Kastner et al. [KKW+17], for our impossibility proofs and formalizations with bounded model checkers, it is also useful to consider a weaker form of security, which is a necessary criterion for security as defined above: A protocol is *possibilistically output-secure*, if at any state of the protocol, every output can still be possible. This weakens the normal security guarantee, as the probability for a given input sequence could be higher in this state. One could even be able to exclude a specific input sequence, if the corresponding output can still be possible through another input sequence. Together with possibilistic input-security, this discussion leads to the following formal definition:

Definition 2 (cf. [KKW+17]). *A protocol $\mathcal{P} = (\mathcal{D}, U, Q, A)$ computing a function $f\colon \{0,1\}^2 \to \{0,1\}$ has* possibilistic input security (possibilistic output security) *if it is correct, i.e., the probability of the output being $O = f(I)$ is* 1*, and*

for uniformly[4] *random input* I *and any visible sequence trace* v *with* $\Pr[v] > 0$ *as well as any input* $i \in \{0,1\}^2$ *(any output* $o \in \{0,1\}$*) we have* $\Pr[v \,|\, I = i] > 0$ $(\Pr[v \,|\, f(I) = o] > 0)$.

Proving Lower Bounds. Let us begin by defining an equivalence relation on the states that helps to greatly reduce the complexity of impossibility proofs by identifying states that are only a permuted version of each other:

Definition 3 (Similarity). *We call two states, or analogously two reduced states as defined next,* μ *and* μ' similar*, if there is a permutation* π *such that applying* (perm, π) *to* μ *gives rise to* μ'*. For notation, let* $\langle \mu \rangle_\sim$ *be the equivalence class of* μ *up to similarity, i.e., the set of all states that are permuted versions of* μ *as defined by similarity.*

In other words, μ is similar to μ' if it is equal to μ' up to column permutation on the sequences part of the state depiction.

As in the work by Kastner et al. [KKW+17, Definition 3], we define *reduced states*, where states are not annotated by their symbolic probabilities, but by the result that is specified by their inputs – a formal definition follows below. This simplifies impossibility proofs by reducing information and the state space. Any such reduced tree captures only a weak form of security, possibilistic security, as discussed above where each output (reachable in principle) needs to be still possible. Showing that a protocol is impossible even in this weak setting implies its general impossibility.

To obtain a reduced state tree, we project all the symbolic probabilities of the sequences of all states in a state tree to a *type* (representing the possible future output associated with the sequence in a correct protocol, see below), which can be any $o \in \{0,1\}$. For this, let $\mathcal{P}$ be a protocol computing a function $f\colon \{0,1\}^2 \to \{0,1\}$ and μ be a state in the state tree. For any sequence s with $\mu(s)$ being a polynomial with positive coefficients for the variables $X_{b_1}, \ldots, X_{b_i}$ $(i \geq 1)$, set $\hat{\mu}(s) \coloneqq o \in \{0,1\}$ if $o = f(b_1) = f(b_2) = \ldots = f(b_i)$ in the resulting *reduced state* $\hat{\mu}$. We call sequences in $\hat{\mu}$ according to their type *o-sequences*. Moreover, we introduce the *additional type* $\perp$ for sequences s where $\mu(s)$ does have positive coefficients for variables representing input that would map to different output, as in $X_{00} + X_{11}$ when $f(0,0) \neq f(1,1)$[5].

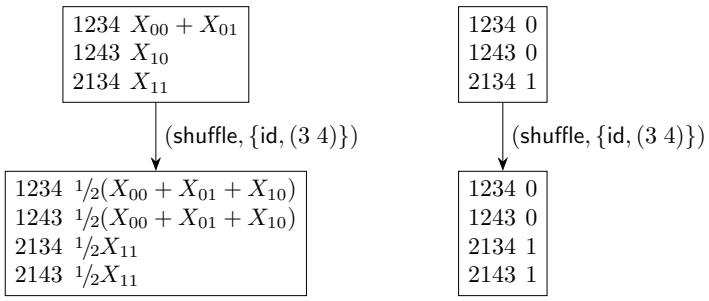
Definition 4 (Reduced State). *Let* $\mathcal{P}$ *be a protocol computing a Boolean function* $f\colon \{0,1\}^2 \to \{0,1\}$ *with deck* $\mathcal{D}$*. Then a* reduced state $\hat{\mu}$ *of* $\mathcal{P}$ *is a map* $\hat{\mu}\colon \mathsf{Seq}^{\mathcal{D}} \to \{0,1,\perp\}$ *which maps a sequence* $s \in \mathsf{Seq}^{\mathcal{D}}$ *to its type (as defined above).*

[4] Actually, the distribution does not matter, as long as $\Pr[I = i] > 0$ for all $i \in \{0,1\}^2$.

[5] It is clear that if a state with a $\perp$ sequence arises, then the protocol has to abort later, as if this sequence would actually lie on the table, it is no longer clear whether an input sequence encoding $(0,0)$, or an input sequence encoding $(1,1)$ was on the table at the start.

If μ is a (non-reduced) state of $\mathcal{P}$, we can map it to its reduced state as follows: The reduced state $\hat{\mu}$ *of* $\mathcal{P}$ arising from μ *is defined via* $\hat{\mu}(s) \coloneqq t_s$*, where* t_s *is the type of* $\mu(s)$*. Note that it is always possible to map a state to its reduced version.*

As an example, let us look at the tree excerpt on the left of Figure 1, and its reduced version (here, shown on the right), when assuming it is part of a protocol computing AND:



For example, the annotation of 1234 in the first state, $X_{00} + X_{01}$, is mapped to its type 0, as it only contains variables representing inputs (namely $(0, 0)$ and $(0, 1)$) that result in output 0. Note that by using reduced states, we bring the state space from the countably infinite to the finite, which is a necessary step for the impossibility proofs, albeit using it only allows us to show impossibility to the weaker notion of possibilistic security (which nevertheless is a necessary condition for full security, hence the even stronger impossibility claim).

A reduced state is *turnable at position* $i \in \{1, \ldots, |\mathcal{D}|\}$, if for each symbol $c \in \Sigma$, there is, among the sequences s with symbol c at position i, an r-sequence for each $r \in \{0, 1\}$ in the image of the function computed by the protocol, and/or a $\perp$-sequence. This essentially means that after the turn at i all outputs are still possible, capturing the notion of output-possibilistic security. The reduced state is *turnable* if it is turnable at a position $i \in \{1, \ldots, |\mathcal{D}|\}$.

For proving impossibility results, we make use of the backwards calculus as given by Koch [K18]. We highlight the main ideas here, but refer to it for details.

Definition 5 (Backwards Shuffle). *Let* $\mathcal{G}$ *be a non-empty set of reduced states of a protocol* $\mathcal{P}$*. Then* $\mathsf{shuf}^{-1}(\mathcal{G})$ *is the set of reduced states* μ' *of* $\mathcal{P}$ *such that there is a permutation set* Π *(containing* id*, and dependent on* μ'*) such that* $(\mathsf{shuffle}, \Pi)$ *applied to* μ' *results in a reduced state in* $\mathcal{G}$*. In other words,* $\mathsf{shuf}^{-1}(\mathcal{G})$ *is the set of states that are transformed into a state in* $\mathcal{G}$ *by a shuffle. Note that the trivial shuffle is allowed, i.e.,* $\mathcal{G} \subseteq \mathsf{shuf}^{-1}(\mathcal{G})$*.*

For example, if $\mathcal{G}$ would consist of just one state, μ, where $o_1, \ldots, o_4$ are distinct symbols[6]:

$$\boxed{\begin{array}{ll} o_1o_2\,o_3o_4 & 0 \\ o_1o_2\,o_4o_3 & 0 \\ o_2o_1\,o_3o_4 & 1 \\ o_2o_1\,o_4o_3 & 1 \end{array}},$$

then $\mathsf{shuf}^{-1}(\mathcal{G})$ would contain exactly the following eight states:

$$\boxed{\begin{array}{ll} o_1o_2\,o_3o_4 & 0 \\ o_1o_2\,o_4o_3 & 0 \\ o_2o_1\,o_3o_4 & 1 \\ o_2o_1\,o_4o_3 & 1 \end{array}}, \boxed{\begin{array}{ll} o_1o_2\,o_4o_3 & 0 \\ o_2o_1\,o_3o_4 & 1 \\ o_2o_1\,o_4o_3 & 1 \end{array}}, \boxed{\begin{array}{ll} o_1o_2\,o_3o_4 & 0 \\ o_2o_1\,o_3o_4 & 1 \\ o_2o_1\,o_4o_3 & 1 \end{array}}, \boxed{\begin{array}{ll} o_1o_2\,o_3o_4 & 0 \\ o_1o_2\,o_4o_3 & 0 \\ o_2o_1\,o_4o_3 & 1 \end{array}}, \boxed{\begin{array}{ll} o_1o_2\,o_3o_4 & 0 \\ o_1o_2\,o_4o_3 & 0 \\ o_2o_1\,o_3o_4 & 1 \end{array}},$$

$$\boxed{\begin{array}{ll} o_1o_2\,o_4o_3 & 0 \\ o_2o_1\,o_4o_3 & 1 \end{array}}, \boxed{\begin{array}{ll} o_1o_2\,o_4o_3 & 0 \\ o_2o_1\,o_3o_4 & 1 \end{array}}, \boxed{\begin{array}{ll} o_1o_2\,o_3o_4 & 0 \\ o_2o_1\,o_4o_3 & 1 \end{array}}.$$

To see this, observe that the first one is just μ, which is contained by definition, as the trivial shuffle $(\mathsf{shuffle}, \{\mathsf{id}\})$ will map it to itself. Moreover, all the other states in this list result in μ by the $(\mathsf{shuffle}, \{\mathsf{id}, (3\ 4)\})$. The above list is exhaustive as we cannot generate a 0-sequence or a 1-sequence via a shuffle if it was not already present in the state on which the shuffle was applied. (Note that in the generation of this list we make use of the assumption that id is always contained in a shuffle, which is the case for closed shuffles anyway, but w.l.o.g. otherwise also, as in the case that id would not be contained, we could replace the shuffle by a conjugated version that is pre-/postfixed by a corresponding deterministic perm operation, cf. Kastner et al. [KKW+17].)

Definition 6 (Backwards Turn). *Let $\mathcal{G}$ be a non-empty set of reduced states of a protocol $\mathcal{P}$. Then, $\mathsf{turn}_{\mathrm{f}}^{-1}(\mathcal{G})$ is the set of reduced states μ' of $\mathcal{P}$, such that $\mu' \in \mathcal{G}$, or that there is a position $i \in \{1, \ldots, |\mathcal{D}|\}$ such that $(\mathsf{turn}, \{i\})$ applied to μ' results in reduced states that are contained in $\mathcal{G}$. In other words, it is the set of states being in $\mathcal{G}$, or having a turnable position i such that all immediate successor states from a turn at i are in $\mathcal{G}$.*

For example, if $\mathcal{G}$ would consist of three reduced states $\mu_1, \ldots, \mu_3$, which each have a constant column at the fourth position:

$$\boxed{\begin{array}{ll} 1234 & 0 \\ 1324 & 0 \\ 2134 & 1 \end{array}}, \boxed{\begin{array}{ll} 1243 & 0 \\ 1423 & 1 \\ 2143 & 1 \end{array}}, \boxed{\begin{array}{ll} 1342 & 0 \\ 1432 & 1 \\ 3142 & 1 \end{array}}.$$

[6] While we chose for the example the same state as depicted on p. 21 in the impossibility proof where it is later used, note that *there*, $\mathcal{G}$ also already includes all the depicted eight states *including* any deterministic permutations (via the similarity relation) of all these, and hence is a much larger set to start with.

Then $\mathsf{turn}_{\mathrm{f}}^{-1}(\mathcal{G})$ would contain, in addition to the states in $\mathcal{G}$, exactly the following four reduced states:

$$\boxed{\begin{array}{|c|}\hline 1234\ 0\\ 1324\ 0\\ 2134\ 1\\ 1243\ 0\\ 1423\ 1\\ 2143\ 1\\ 1342\ 0\\ 1432\ 1\\ 3142\ 1\\ \hline\end{array}}, \boxed{\begin{array}{|c|}\hline 1234\ 0\\ 1324\ 0\\ 2134\ 1\\ 1243\ 0\\ 1423\ 1\\ 2143\ 1\\ \hline\end{array}}, \boxed{\begin{array}{|c|}\hline 1234\ 0\\ 1324\ 0\\ 2134\ 1\\ 1342\ 0\\ 1432\ 1\\ 3142\ 1\\ \hline\end{array}}, \boxed{\begin{array}{|c|}\hline 1243\ 0\\ 1423\ 1\\ 2143\ 1\\ 1342\ 0\\ 1432\ 1\\ 3142\ 1\\ \hline\end{array}}.$$

Here, first observe that the first state is just a combination of all three states, whereas the second, third and fourth is a combination of μ_1 and μ_2, of μ_1 and μ_3 and of μ_2 and μ_3, respectively. When forming the "backwards turn" set, we can just combine states with a constant column of distinct symbols into one, as a turn at the position where these individual states had a constant column branches/gives rise to exactly these individual states.

We call $\mathsf{turn}_{\mathrm{f}}^{-1}(\cdot)$ and $\mathsf{shuf}^{-1}(\cdot)$ *backwards turn* and *backwards shuffle*. Define by $\mathsf{cl}_{\mathrm{f}}(\mathcal{G})$ *the closure* of $\mathsf{turn}_{\mathrm{f}}^{-1}(\cdot)$ and $\mathsf{shuf}^{-1}(\cdot)$ operations on $\mathcal{G}$. Note here, that if a finite-runtime protocol exists for a given start state, then there exists a sequence of shuffle/turn operations which, applied to the start state, will result in a final state. Therefore if we assume $\mathcal{G}$ to be the set of all possible final states for a deck $\mathcal{D}$, then it holds that if the start state is not in $\mathsf{cl}_{\mathrm{f}}(\mathcal{G})$, then no finite-runtime protocol for $\mathcal{D}$ can exist.

2.2 Automatic Formal Verification Using SBMC

In the following, we introduce an automatic technique from formal program verification, namely software bounded model checking (SBMC), to the field of card-based cryptography. We first describe the general technique of using SBMC to check for software properties, before we explain how we apply it to search for cryptographically secure card-based protocols. In a nutshell, we translate the task to a reachability problem in software programs (which will later-on be a program encoding operations on an abstract state tree as described above), which the SBMC tool encodes into an instance of the SAT problem.

We assume we are given an imperatively defined function f under the form of an imperative program (for example, written in the C language), that uses some parameter values taken among a set of possible start values I. An entry $i \in I$ is a list of values, one value for each such parameter: it gives a value to everything that a run of f depends on, such as its input variables, or anything that is considered non-deterministic (i.e., of arbitrary, but fixed, value for any concrete evaluation of f) from the point of view of f. For this reason, those parameters are qualified as "non-deterministic", to distinguish them from normal parameters used in a programming language to pass information around. Moreover, some values can be "derived", thus, computed in f from the non-deterministic parameter values,

or declared as constants in f, and both values of non-deterministic parameters or derived values can then be used as normal parameters in the program. We are also given a software property to be checked about f, in the form $C^{\text{ant}} \Rightarrow C^{\text{cons}}$, where *ant* and *cons* stand for antecedent and consequence respectively. Both C^{ant} and C^{cons} are sets of Boolean statements. A Boolean statement is a statement of f that evaluates to a Boolean value, for example, a simple statement checking that some computed intermediate value is positive. An entry i is said to satisfy a set of Boolean statements if and only if all Boolean statements in the set evaluate to true during the execution of f using the non-deterministic parameter values i, and is said to fail the set of Boolean statements otherwise. The property $C^{\text{ant}} \Rightarrow C^{\text{cons}}$ requires that for all possible entries $i \in I$, *if* i satisfies C^{ant}, *then* i satisfies C^{cons}. As an example, assume f computes, given i, two intermediate integer values v_1 and v_2, and then returns a third value v_3. The property to be checked could, e.g., be: *if* v_1 is negative, *then* v_2 is positive and v_3 is odd. A solver that is asked to check a software property $C^{\text{ant}} \Rightarrow C^{\text{cons}}$ thus exhaustively searches for an entry i that satisfies C^{ant} but fails C^{cons}. The property is valid if and only if there does not exist any such entry i, i.e., it is impossible to find.

SBMC is a fully-automatic static program analysis technique used to verify whether such a software property is valid, given a function and a property to be checked. It covers all possible inputs within a specified bound. It is static in the sense that programs are analyzed without executing them on concrete values or considering any side channels. Instead, programs are symbolically executed and exhaustively checked for errors up to a certain bound, restricting the number of loop iterations to limit runs through the program to a bounded length. This is done by unrolling the control flow graph of the program and translating it into a formula in a decidable logic that is satisfiable if and only if a program run exists which satisfies C^{ant} and fails C^{cons}. The variables in the formula are the non-deterministic parameters of f, and their possible values are taken from I.

This reduces the problem to a decidable satisfiability problem. Modern SAT-solving technology can then be used to verify whether such a program run exists, in which case an erroneous input has been found, and the run is presented to the user. If the solver cannot find such a program run, it may be either because the property is valid, or because it is invalid only for some run which exceeds the bound. In some cases, SBMC is also able to infer statically which bound is sufficient to bring a definitive conclusion.

2.3 Automatic Formal Verification for Card-Based Protocols

Our approach employs a standardized program representation of the KWH trees introduced by Koch, Walzer, and Härtel [KWH15] (and described in the beginning of this section). This allows a general programmatic encoding of both shuffle and turn operations, as well as of the fixed input state (indicated by the input card sequences from the table in the very beginning of this paper), the non-deterministic reachable states, and the logical function to be computed securely.

The input state is trivially derived from the specified numbers of cards as the size and order of the players' commitments is fixed and the (without loss of

generality) consecutively ordered card sequence of (distinguishable) helper-cards is simply prepended to the input card sequence, annotated with their respective input probabilities. Any input state thus consists of exactly four distinguishable card sequences. Based on this input state, the program performs a loop, which successively performs turn or shuffle operations based on the input state and computes the resulting states from which it continues performing turn or shuffle operations. The loop ends when the specified bound (representing the length of the protocol to be found) is reached, checks whether the final state is indeed a valid computation of the secure function, and (if and only if the check is successful) the found protocol is then presented to the user.

However, this task involves multiple computational complexities, most notably both the number of (possibly) reachable states, and the choice of the next operation, i.e., either choosing the card(s) to be turned or which shuffle to perform. We partially overcome the first computational complexity by not considering Las Vegas protocols as this relieves us from checking every reachable sequence of states to be finite. In fact, we compute all reachable states after every protocol operation, but only check each of them to be valid, and then proceed our operations on only one of them, which is non-deterministically chosen among them. The second computational complexity consists in first non-deterministically choosing whether to shuffle or to turn, and then to perform the respective operation. The turn operation is less interesting as it is mostly the obvious implementation for updating the computed state and its probabilities using mostly standard imperative program operations, except that the turn observations are again non-deterministically chosen, hence making the SBMC tool consider any of them to be possible. The more interesting operation is the shuffle operation, as it must randomly draw a set of permutations on which the thereby reachable states are computed. We implement this by non-deterministically choosing a set of permutations from a precomputed set of all generally possible permutations. Both the amount and the choices of the respective permutations are chosen non-deterministically. Moreover, we have the ability to restrict our experiments to only closed shuffles, and can even bound the shuffle set size to keep the running time of the verification time acceptable, if needed (albeit possibly reducing the strength of the results, cf. Section 9). For example, in our analysis of the run-minimality of Protocol 1, we bounded the permitted size of the permutation sets by the (arguably quite reasonable) number 8, in order to keep the execution times still manageable for our experiments. Note that our technique from Section 9 shows that only a bound of 12 would be really safe to assume, leaving a small gap in the argumentation as we superficially exclude exactly the possible 12-element alternating groups A_4 as shuffles steps from the possible protocol candidates, when showing that no shorter protocol can exist. We leave it for future work to tweak the code such that the looser bound of 12 is within reach with our technique.

Finally, after iterating the afore-mentioned loop for the specified bound number with the described operations and restricting that final state indeed computes the secure function, we specify the software property C^{cons} to be checked simply

as the Boolean value `false`. This trivially unsatisfiable property implies that the verification task always fails once there exist input and non-deterministic parameters such that the respective program run reaches the statement in the program which checks this property. The SBMC tool exhaustively searches for a run of the specified length through the program which leads from the starting state to a correct and secure state which satisfies the given security notion, i.e., reaches the above-metioned statement. Hence, if there exists any protocol of the specified length which computes the secure function and for which the specified operations and valid intermediate states (representing KWH-trees) exist, such a protocol is presented by our method. If no such protocol can be found, we know there is no card-based protocol of the specified length satisfying all our restrictions on permitted turn and shuffle operations, as well as intermediate and final states. This means there exists no model for the SAT formula which encodes the set of all permitted program runs given our specified requirements.

Hence, assuming our translation of KWH trees and respective protocol operations into a simple imperative program are correct, this method can then be used in an iterative manner to strengthen the bounds from the literature. Note that this is largely based on the so-called "small-scope hypothesis", i.e., a large number of bugs are already exposed for small program runs. We apply this hypothesis to the setting of card-based security protocols as all protocols in the literature only use a small number of turn and shuffle operations and the length of any found protocol is below ten operations.

This approach can be generalized to search for card-based protocols using a pre-defined number of actions and adhering to a given formal security notion. We have written a general program[7] to search for such situations parameterized in the desired restrictions on actions and security notions. Note that, in order to cope with the still considerable state space size, we use the refined security notion of output-possibilistic security.

3 On the Choice of Cards for Input and Output

We essentially show that the choice of input basis (or output basis, but not necessarily both) is irrelevant for the functioning of the protocol. In rare cases, one has to append two operations to existing protocols to make them fully basis flexible. In the Niemi–Renvall protocol shown above, the protocol description specifies Alice's cards to be of symbols $1, 2$, and Bob's to be of symbols $3, 4$ and the helping card to be a 5. To simplify later proofs and to demonstrate an interesting symmetry in card-based protocols, we show that this choice is irrelevant for the functioning of the protocol.

For this, we define a *relabeling* from deck alphabet Σ to a deck alphabet Σ', i.e., a bijective function $\lambda\colon \Sigma \to \Sigma'$.[8] A relabeling of a sequence $s = (s_1, \ldots, s_n)$ is a relabeling of each of its symbols, i.e., $\lambda(s) := (\lambda(s_1), \ldots, \lambda(s_n))$. A relabeling of a

[7] The source code is available under https://github.com/mi-ki/cardCryptoVerification.

[8] In case of the decks being a subset of $\mathbb{N}$, we may use usual permutation notation. We require that if λ maps x to y, then the cardinalities of x and y are equal in the deck.

state is given by the relabeling of all its sequences, a relabeling of a protocol/state (sub)tree is the relabeling of all its states as described by Figures 3 and 5.

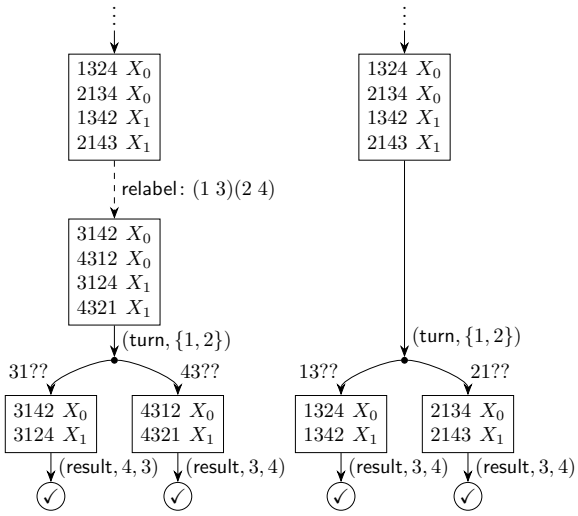


Fig. 3. Example of the relabel action, swapping the card symbols of 1 and 3, and of 2 and 4, respectively. This action is for abbreviated writing only, it does not actually relabel the physical cards, which seems impossible without learning their symbols. Hence, the tree on the left is virtually translated to the right. Note that the relabeling only affects the sequences, the observations at edges belonging to turn actions and may swap the order of the indices in result operations.

Lemma 1. *If $\mathcal{P}$ is a protocol with deterministic output basis, one can relabel the cards without affecting the functioning.*

Note that the deterministic output basis restriction is important, because if we have a randomized output encoding such as in Figure 4 on the left, a relabeling might affect the monotonicity of the encoding of only one of the possible output bases. In this case, we make use of the following lemma, as illustrated Figure 4.

Lemma 2. *Every protocol with one-bit output and a randomized output basis can be transformed into a protocol with deterministic output basis, by inserting a shuffle and a turn before any result operation with randomized output basis.*

4 Impossibility of Finite-Runtime Four-Card AND and Basis Conversion with Overlapping Bases

In this section we give our main impossibility results.

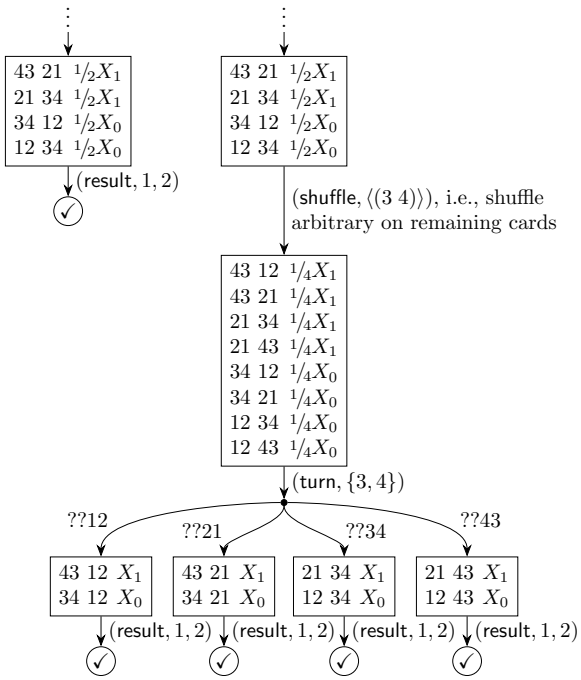


Fig. 4. Example of making the basis deterministic, cf. Lemma 2. On the left, you can see a tree part with one-bit output and randomized basis, i.e., the output basis may be $\{1,2\}$ or $\{3,4\}$, each with a probability of $1/2$. We can make it known to the players, i.e., deterministic, by splitting up the state via an S_k-shuffle (here: $k=2$) on the remaining cards (so that they no longer contain any information), turning these and then doing the result operation. By what is visible in the turn, one can derive the output basis.

Theorem 1. *There is no four-card finite-runtime basis conversion protocol for overlapping bases with deck* $\mathcal{D} = [\![1,2,3,4]\!]$.

Proof. We proceed by using the backwards calculus technique by Koch [K18], as described in Section 2.1. That is, we start with the set of final states $\mathcal{G}$ of basis conversion protocols. Then, we iteratively build a (possibly) larger set by adding states which reach the states of the current set by a shuffle or a turn, in order to obtain the closure $\mathsf{cl}_\mathsf{f}(\mathcal{G})$. As we consider only reduced states (cf. Section 2.1), the set of possible states is finite, hence, applying $\mathsf{turn}_\mathsf{f}^{-1}(\cdot)$ and $\mathsf{shuf}^{-1}(\cdot)$ operations to the (growing) set of states, starting from $\mathcal{G}$, will become stationary. Finally, it remains to be shown that the start state is not contained in the derived closure.

We assume w.l.o.g.[9] the input basis $\{1,2\}$ with helping cards 3 and 4, and the output basis $\{o_1 < o_2\} \subset \{1,2,3,4\}$. For the basis conversion impossibility, we will require $|\{1,2\} \cap \{o_1,o_2\}| = 1$ (which we call *basis intersection requirement*

[9] For the impossibility result, the symbols of the cards are irrelevant, as we could prepend a relabel operation to any protocol, to bring it into this form.

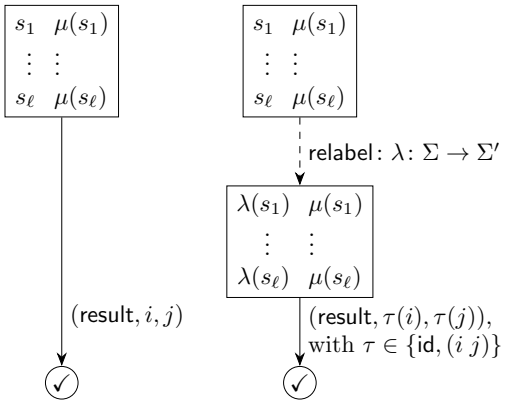


Fig. 5. The formal rule for relabeling leaf nodes of one-bit output protocols. Let $r_1 = s_k[i], r_2 = s_k[j] \in \mathcal{D}$ be the output symbols (before relabeling) of some arbitrary sequence s_k of μ. Then, $\tau = \mathsf{id}$, if $r_1 < r_2$ implies $\lambda(r_1) < \lambda(r_2)$ (λ is monotone on r_1, r_2) and $\tau = (i\ j)$ otherwise.

in the following). However, whether we use this requirement or not, the closure $\mathsf{cl}_\mathrm{f}(\mathcal{G})$ remains the same[10]. Hence, we will use this requirement only in the last step of the proof when we show that the start state is not in $\mathsf{cl}_\mathrm{f}(\mathcal{G})$, and reuse the closure for the AND protocol impossibility proof in Theorem 2.

After setting the stage, we start by describing the set $\mathcal{G}_0$ from which we will derive the closure $\mathsf{cl}_\mathrm{f}(\mathcal{G}_0)$ according to the backwards calculus technique described above. Let $o_3 < o_4$ be the remaining two symbols, i.e., $\{o_3, o_4\} = \{1,2,3,4\} \setminus \{o_1, o_2\}$. Thus, the final state is (up to similarity[11]) any choice of at least one 1-sequence and one 0-sequence of the states on the left set[12]:

$$\left\langle \begin{array}{ll} o_1o_2\,o_3o_4 & 0 \\ o_1o_2\,o_4o_3 & 0 \\ o_2o_1\,o_3o_4 & 1 \\ o_2o_1\,o_4o_3 & 1 \end{array} \right\rangle_\sim \qquad \left\langle \begin{array}{ll} o_3o_4\,o_1o_2 & 0 \\ o_3o_4\,o_2o_1 & 0 \\ o_4o_3\,o_1o_2 & 1 \\ o_4o_3\,o_2o_1 & 1 \end{array} \right\rangle_\sim$$

The state set on the right contains the template for final states with output basis $\{o_3 < o_4\}$, which we will include in the starting set $\mathcal{G}_0$, as they are reachable from final states with output basis $\{o_1, o_2\}$ by the backwards calculus anyway, due to the existence of the disjoint basis conversion protocol by Mizuki [M16] (again, with any choice of at least one 1- and one 0-sequence). As long as we can still show that the start state is not in $\mathsf{cl}_\mathrm{f}(\mathcal{G}_0)$, it is okay to enlarge $\mathcal{G}_0$, since our

[10] This is the (reduced-state) closure on the final states of arbitrary one-bit-output functions for the given deck.

[11] Refer to Definition 3. We do not want to assume anything on at which positions the output lies, hence we include all permutations of the states into the discussion.

[12] with the output being encoded in positions 1, 2, or at different positions, if looking at the permuted versions of the state.

claim is only made stronger (using the monotonicity property of the backwards operations $\mathsf{turn}_{\mathrm{f}}^{-1}(\cdot)$ and $\mathsf{shuf}^{-1}(\cdot)$).

We have $\mathsf{shuf}^{-1}(\mathcal{G}_0) = \mathcal{G}_0$, because any subset of a state from $\mathcal{G}_0$ which contains at least one 1-sequence and one 0-sequence (which is required as otherwise 1-/0-sequences cannot be generated out of thin air by a shuffle) is already in $\mathcal{G}_0$. Hence, we consider $\mathcal{G}_1 := \mathsf{turn}_{\mathrm{f}}^{-1}(\mathcal{G}_0)$, i.e., the states which are turnable at a position i, where all immediate child nodes after turning at i are in $\mathcal{G}_0$. W.l.o.g.[13] we fix the turn to be at position 4. Following Koch [K18, Lemma 3], we use that $\mathcal{G}_1 = \mathsf{turn}_{\mathrm{f}}^{-1}(\mathcal{G}_0) = \mathcal{G}_0 \cup \mathsf{turn}_{\mathrm{f}}^{-1}(\mathsf{cc}(\mathcal{G}_0))$ holds, where $\mathsf{cc}(\mathcal{G}_0)$ is the set of states in $\mathcal{G}_0$ that have a constant column, i.e., the union of these four equivalence classes up to similarity:

$$\left\langle \left[\begin{array}{cc} o_1o_2\,o_3o_4 & 0 \\ o_2o_1\,o_3o_4 & 1 \end{array}\right] \right\rangle_{\sim} \quad \left\langle \left[\begin{array}{cc} o_1o_2\,o_4o_3 & 0 \\ o_2o_1\,o_4o_3 & 1 \end{array}\right] \right\rangle_{\sim} \quad \left\langle \left[\begin{array}{cc} o_3o_4\,o_1o_2 & 0 \\ o_4o_3\,o_1o_2 & 1 \end{array}\right] \right\rangle_{\sim} \quad \left\langle \left[\begin{array}{cc} o_3o_4\,o_2o_1 & 0 \\ o_4o_3\,o_2o_1 & 1 \end{array}\right] \right\rangle_{\sim}$$

The states from $\mathcal{G}_1 \setminus \mathcal{G}_0$ look as follows:

$$\left[\begin{array}{cc} \ldots a & 0 \\ \ldots a & 1 \\ \hline \ldots b & 0 \\ \ldots b & 1 \\ \hline \ldots c & 0 \\ \ldots c & 1 \\ \hline \ldots d & 0 \\ \ldots d & 1 \end{array}\right], \qquad (*)$$

where at least two of the four (two-sequence) blocks are present, and $a, b, c, d \in \mathcal{D}$ are pairwise distinct. We show that a further backwards turn does not enlarge the set by showing $\mathsf{cc}(\mathcal{G}_1) = \mathsf{cc}(\mathcal{G}_0)$. For this, note that the states from $\mathsf{cc}(\mathcal{G}_0)$ (i.e., the blocks, considered in isolation) have exactly two constant columns, but with the specific pairing that if one of the constant columns consists of o_1, the other one consists of o_2 and vice versa, or if one consists of o_3, the other one consists of o_4 and vice versa.

Using this structure, we can deduce that states from $\mathcal{G}_1 \setminus \mathcal{G}_0$ *with a constant column*, say w.l.o.g.[14] at position 3, have the respective paired symbol (of the o_1-o_2 or o_3-o_4 constant-column symbol pairing) in the fourth column. Therefore[15], these states can have at most two sequences in total, i.e., they are already in $\mathcal{G}_0$. This shows $\mathsf{turn}_{\mathrm{f}}^{-1}(\mathcal{G}_1) = \mathcal{G}_1$.

[13] As we consider states up to similarity, we can just permute each of these states constituting the full turnable state in such a way that their constant column is at position 4

[14] Analogous to before, as all constituting states of the set are up to similarity, we have free choice in choosing a position at which the constant column should be.

[15] As both sequences in a block have identical symbols in column 4, and given the pairwise distinctiveness of these symbols between blocks, there are at most two such sequences within a state.

Now, for the main step of the proof, we define $\mathcal{G}_2 := \mathsf{shuf}^{-1}(\mathcal{G}_1)$ and $\mathcal{G}_3 := \mathsf{turn}_{\mathrm{f}}^{-1}(\mathcal{G}_2)$. Since the shuffling is unrestricted, applying another backwards shuffle to $\mathcal{G}_2$ cannot produce a larger set, as we can always replace two consecutive shuffles by an equivalent single shuffle. The remaining proof will show $\mathcal{G}_3 = \mathcal{G}_2$, in which case no further enlargement is possible. Finally, showing that the start state is not in $\mathcal{G}_2$ finishes the proof.

As $\mathcal{G}_2$'s states are subsets of $\mathcal{G}_1$'s states[16], $\mathsf{cc}(\mathcal{G}_2)$'s general form is as on the left, from which we can leave out further sequences, as long as we still have at least one 1-sequence and one 0-sequence:

$$\left[\begin{array}{ll} \ldots da & 0 \\ \ldots da & 1 \\ \hline \ldots db & t_1 \\ \hline \ldots dc & t_2 \end{array}\right] \qquad \left[\begin{array}{ll} \ldots da & 0 \\ \ldots da & 1 \\ \hline \ldots db & t_1 \\ (\ldots ab & \overline{t_1}) \\ \hline \ldots dc & t_2 \\ (\ldots ac & \overline{t_2}) \\ \hline (\ldots xd & t_3) \\ (\ldots yd & \overline{t_3}) \end{array}\right], \qquad (\star)$$

where $t_i \in \{0, 1\}$ $(i = 1, 2, 3)$ are the types of the sequences and $\overline{t_i} = 1 - t_i$ their inverses. To see this, observe that states of the form on the left are subsets of the form on the right, where x, y are either both set to a, or one is set to b and the other to c, and, where we leave out at least all sequences interfering with our wish of a constant column in this position (i.e., the sequences in parentheses in the form on the right). With the variables introduced above, we assume a (constant-column symbol) pairing between a and d, and between b and c[17]. This is the only way to obtain a maximal number of sequences with a d in column 3 for a state in $\mathcal{G}_1$. Hence, states in $\mathsf{cc}(\mathcal{G}_2)$ have at least 2 but at most 4 sequences.

Our aim is to show that the set of these states is $\mathsf{cc}(\mathcal{G}_0)$ again, i.e., that $\mathsf{cc}(\mathcal{G}_2) = \mathsf{cc}(\mathcal{G}_0)$. (In other words, we show that it is impossible to reach any state in $\mathcal{G}_1$ via a shuffle from a state of $\mathsf{cc}(\mathcal{G}_2) \setminus \mathsf{cc}(\mathcal{G}_0)$, which will be shown to be empty.) In the following, we do a case distinction on the number of sequences of states $\mu \in \mathsf{cc}(\mathcal{G}_2)$.

Let us prepend this case distinction with two general observations that will be used in the following. First, every shuffle set Π that is used to map $\mu \in \mathsf{cc}(\mathcal{G}_2)$ to a state $\mu' \in \mathcal{G}_1$ will contain a permutation π with $\pi(3) \neq 3$, i.e., one that moves the constant (third) column, as otherwise we cannot generate necessary additional sequences with a non-d symbol at position 3. As defined by Koch, Walzer, and Härtel [KWH15], we call a state *i/j-state* if it has i 0-sequences and j 1-sequences. Using this notation, we have that, if $\mu \in \mathsf{cc}(\mathcal{G}_2)$ is an i/j-state, then

[16] We assume w.l.o.g. that any shuffle contains the id permutation, hence, non-trivial shuffling generates new sequences. Consequently, backwards shuffling then only leaves out sequences, which we describe in set-theoretic terms, by abuse of notation.

[17] Note that this only refers to the 2-line subblocks of a state.

the reached $\mu' \in \mathcal{G}_1$ after the shuffle will be a i'/j'-state with $i' \geq 2i$ and $j' \geq 2j$, as the shuffle generates $i + j$ new sequences with d at a position $\pi(3) \neq 3$.

Now, let us first consider (a) the case that μ has *three* or *four sequences*. In this case, there cannot be a permutation $\pi \in \Pi$ with $\pi(3) = 4$, as there are only two possible sequences with a d in position 4 in states of $\mathcal{G}_1$, and this (i.e., having a permutation that maps the d-column to column 4) is the only way to obtain these two sequences ending with d, as no other column contains a d in μ. Hence, in this case, we have $i' + j' \leq 6$ due to the two unreachable sequences ending with d. Moreover, as $\mathcal{G}_1$ is built from blocks with one 0- and one 1-sequence, we know that $i' = j'$. But this allows us already to exclude the case of $i + j > 2$, because if, e.g., $i = 1$ and $j = 2$ (or vice versa), then $i' \geq 2$ and $j' \geq 4$, but $j' = i'$ yields $i' + j' = 8$, and if $i = j = 2$, then we also have $i' + j' = 8$, both contradicting $i' + j' \leq 6$. Hence, $\mathsf{cc}(\mathcal{G}_2)$ cannot contain any state with three or four sequences.

Now, let (b) μ contain *two sequences*. For this case, we consider choices of two sequences from a state in $\mathcal{G}_1 \setminus \mathcal{G}_0$ of $(*)$ with d in column 3. (We will show below that in the current case we can choose these more specifically from the state on the right of $(\star)$, without the parentheses.)[18] If we choose both sequences to end with da, the state would be in $\mathsf{cc}(\mathcal{G}_0)$, which is, however, inconsistent with the state being in $\mathcal{G}_1 \setminus \mathcal{G}_0$. Hence, there is at most one sequence of each of the following types: sequences ending with da, with db and with dc. If we choose to include a sequence ending with da, then it is inconsequential whether we choose one ending with db or with dc (only the d-a constant-column symbol pairing assigns a a special role). W.l.o.g. we choose a sequence ending with db in the following. This leaves us with two choices, either to include a sequence ending with da or to exclude it. In total, we can obtain three states that are not already in $\mathsf{cc}(\mathcal{G}_0)$:

$$\boxed{\begin{array}{ll} bcda & 0 \\ acdb & 1 \end{array}} \qquad \boxed{\begin{array}{ll} bcda & t \\ cadb & \bar{t} \end{array}} \qquad \boxed{\begin{array}{ll} acdb & t \\ badc & \bar{t} \end{array}},$$

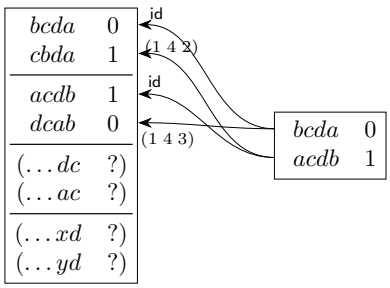
where $t \in \{0, 1\}$ is the type of the sequence. However, the third state is similar to the second one via the permutation (1 4), so we do not need to consider this case. Each of these states needs one 1- and one 0-sequence, which we can fix w.l.o.g. in the first state. This is because the first state is similar to the first state with swapped 0 and 1 types, also via the permutation (1 4).

We want to show that there is no way to shuffle these two states into a state of $\mathcal{G}_1 \setminus \mathcal{G}_0$ as given in $(*)$. As a first step, we show that, more specifically, it suffices to demonstrate the slightly stricter claim that there is no way to shuffle these two states into a state of $\mathcal{G}_1 \setminus \mathcal{G}_0$ as given on the right of $(\star)$ (including the sequences with parentheses). This is because of the following: As the two-sequence states considered here each have a sequence ending with da, our shuffle needs to reach the other sequence ending with da, in order to complete the block ending with

[18] To see that this is not already immediate, observe that the state on the right of $(\star)$ was chosen to maximize the number of d's in column 3, and is not as general as saying that the state is of $(*)$ with a d in column 3. However, this loss of generality does not restrict the general form of $\mathsf{cc}(\mathcal{G}_2)$ on the left of $(\star)$.

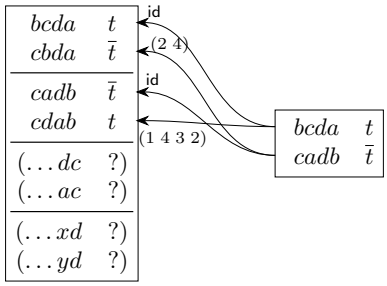
a in $\mathcal{G}_1 \setminus \mathcal{G}_0$. Because of the d-a pairing, this sequence also has a d in the third column. Hence, the state reached by the shuffle has at least three ds in the third column. However, as we start with two sequences with distinct types (and all symbols are distinct in the standard deck setting) any permutation $\pi \in \Pi \setminus \{\mathsf{id}\}$ that increases the number of ds in that column (by $\pi(3) = 3$) at least doubles the number of sequences. Hence, the resulting state in $\mathcal{G}_1 \setminus \mathcal{G}_0$ has at least four ds in column 3 and is therefore of the form in (⋆).

Consequently, for the first state, we have the following scenario:



Reaching the state on the left by a shuffle contains at least $\{\mathsf{id}, (1\ 4\ 3), (1\ 4\ 2)\}$. But applying (1 4 2) to the first sequence yields a sequence *cadb*, which is not possible in the scheme on the left side due to being the third sequence with a trailing b.

The case of the second state is as follows:



Reaching the state on the left by a shuffle contains at least $\{\mathsf{id}, (2\ 4), (1\ 4\ 3\ 2)\}$. But if we apply (2 4) to the first sequence, we obtain *badc*, and if we apply (1 4 3 2) to the second sequence, this gives the sequence *adbc*. The two additional sequences both end with a c, hence they would form a block in the scheme on the left, which is not possible, as the resulting block would miss a constant b-column. This shows that $\mathsf{cc}(\mathcal{G}_2) = \mathsf{cc}(\mathcal{G}_0)$.

The start state of base conversion protocols is (up to similarity)

$$\left\langle \begin{array}{|ll|} \hline 12\,34 & 0 \\ 21\,34 & 1 \\ \hline \end{array} \right\rangle_{\sim}$$

with the basis intersection requirement $|\{1,2\} \cap \{o_1, o_2\}| = 1$. Because of this, the state is not in $\mathcal{G}_0$. As it has a constant column, it would need to be in $\mathsf{cc}(\mathcal{G}_2)$ which is equal to $\mathsf{cc}(\mathcal{G}_0)$ by the argument above. Hence, the state is not in $\mathcal{G}_2$. □

Theorem 2. *There is no four-card finite-runtime AND protocol with deck* $\mathcal{D} = [\![1,2,3,4]\!]$ *with fixed-in-advance output basis.*

Proof. As the final states are (without the basis intersection requirement) the same as in the proof of Theorem 1, we use the closure $\mathsf{cl}_\mathrm{f}(\mathcal{G}_0)$ derived there, and show that the start state of an AND protocol is not contained in $\mathsf{cl}_\mathrm{f}(\mathcal{G}_0)$. For this, observe that the start state of an AND protocol is (up to similarity) as from the following set:

$$\left\langle \begin{array}{|ll|} \hline 12\,34 & 0 \\ 21\,34 & 0 \\ 12\,43 & 0 \\ 21\,43 & 1 \\ \hline \end{array} \right\rangle_{\sim}$$

In particular, it has three 0-sequences and one 1-sequence, which excludes it from being in $\mathcal{G}_0$ or $\mathcal{G}_1$ (derived in the proof of Theorem 1 above), as the numbers of 0- and 1-sequences differ. Moreover, observe that it has in each column exactly two distinct symbols, each exactly twice. For states in $\mathcal{G}_2$ (which are subsets of $\mathcal{G}_1$) it holds that each symbol occurs at most twice in the turn column 4, where each (two-sequence) block ending with one such symbol consists of one 1-sequence and/or one 0-sequence. If we try to leave out sequences from the $\mathcal{G}_1$ template (for the subsets of $\mathcal{G}_2$) to obtain a state of type 3/1, we lose the property of having each occurring symbol exactly twice. Hence, the start state cannot be in $\mathcal{G}_2$. □

5 Card-Minimal Protocols for AND

Theorem 3. *There is a four-card Las Vegas AND protocol with deck* $\mathcal{D} = [\![1,2,3,4]\!]$ *using only random cuts.*

Proof. See Figure 6 and Protocol 1.

Table 2. The different states of Protocol 1 after $\boxed{1}$ was revealed in the first turn. The permutation to be applied in this case is (3 4). The situation is similar in all other cases.

Bits	Sequence	After permutation	Removing $\boxed{3}$
(0, 0)	$\boxed{1}\boxed{2}\boxed{3}\boxed{4}$	$\boxed{1}\boxed{2}\boxed{4}\boxed{3}$	$\boxed{1}\boxed{2}\boxed{4}\boxed{\text{x}}$
(0, 1)	$\boxed{1}\boxed{2}\boxed{4}\boxed{3}$	$\boxed{1}\boxed{2}\boxed{3}\boxed{4}$	$\boxed{1}\boxed{2}\boxed{\text{x}}\boxed{4}$
(1, 0)	$\boxed{1}\boxed{3}\boxed{4}\boxed{2}$	$\boxed{1}\boxed{3}\boxed{2}\boxed{4}$	$\boxed{1}\boxed{\text{x}}\boxed{2}\boxed{4}$
(1, 1)	$\boxed{1}\boxed{4}\boxed{3}\boxed{2}$	$\boxed{1}\boxed{4}\boxed{2}\boxed{3}$	$\boxed{1}\boxed{4}\boxed{2}\boxed{\text{x}}$

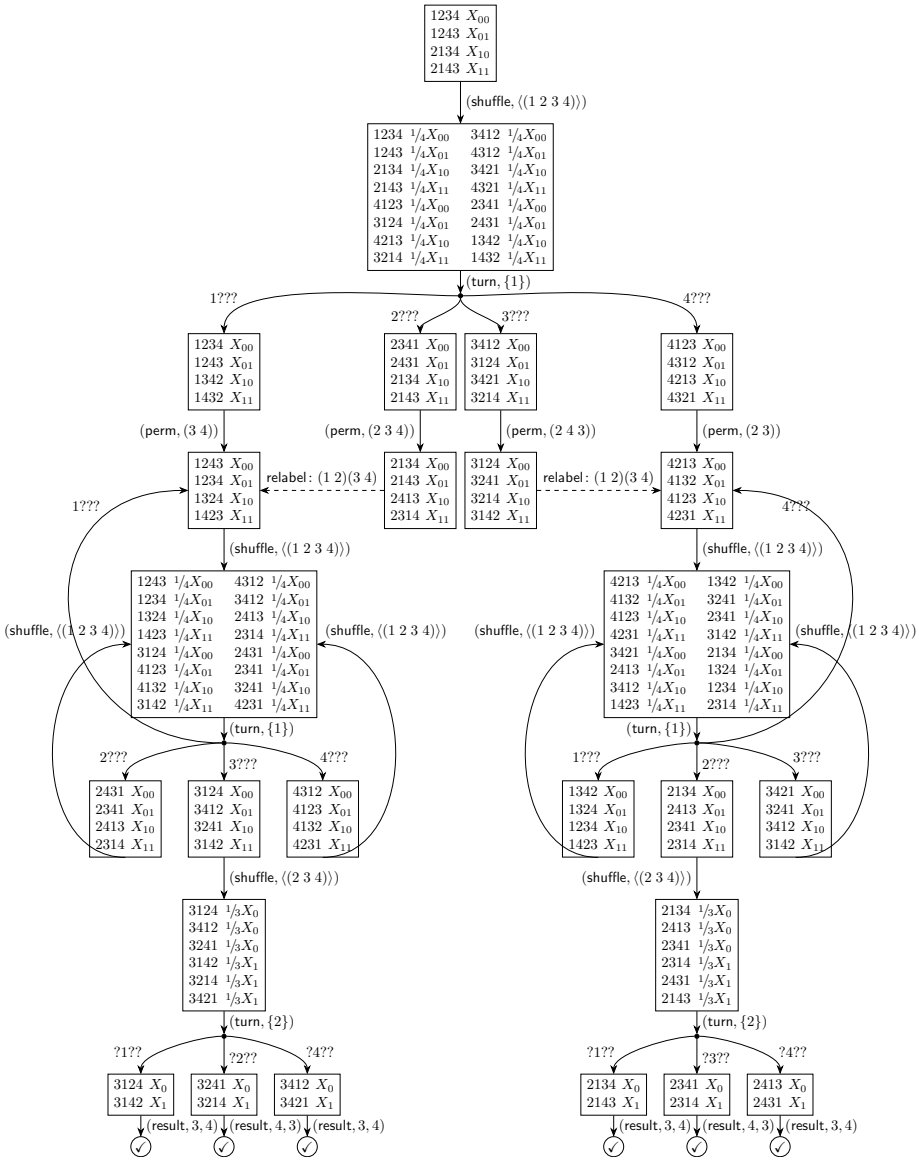


Fig. 6. Four-card Las Vegas AND protocol using random cuts, cf. Protocol 1. Here, $X_0 \coloneqq X_{00} + X_{01} + X_{10}$ and $X_1 \coloneqq X_{11}$. The relabel operations are not actual actions to be performed but help abbreviate the write-up of the protocol, see Section 3.

In order to get a better understanding of why the protocol works and how it is related to the protocol by Niemi and Renvall [NR99], let us consider exemplarily the case that the first card to be revealed is a 1, the other cases are analogous. In this situation, let us look at the different cases, given in Table 2. Using the method as before, we can remove $\boxed{3}$ by performing a random cut while leaving the relative order intact ($\boxed{1}$ here is assigned the role of the $\boxed{5}$ in Niemi and Renvall's protocol) and waiting until it appears when turning. Later we can remove the $\boxed{1}$ from the remaining cards, to get the output encoded using the cards $\boxed{2}$ and $\boxed{4}$. A closer analysis of the situation after removing $\boxed{3}$ shows that one can take a shortcut when one is not bound to the output being cards $\boxed{2}$ $\boxed{4}$ (which is not our goal, because in the other cases besides the first turn being 1 it is different anyway, and one would have to add conversion protocols to ensure this). The situation is as follows: The remaining three cards are either a cyclic rotation (cut) of the sequence $\boxed{1}$ $\boxed{2}$ $\boxed{4}$, if the output is 0, or a cyclic rotation of the sequence $\boxed{1}$ $\boxed{4}$ $\boxed{2}$, otherwise. A cut cannot rotate a sequence of the former type to become the other, or vice versa. After the cut we can safely turn any card and, from the resulting symbol, deduce in which order the other cards must be output to encode the protocol result.

Protocol 1 Our four-card AND protocol. The first bit is in basis $\{1,2\}$, the second in $\{3,4\}$, and the output in $\{1,2,3,4\}\setminus\{v_2,v_3\}$, where v_2, v_3 are the last two revealed symbols. See Figure 6 for a KWH tree representation.

$(\mathsf{shuffle}, \langle(1\ 2\ 3\ 4)\rangle)$
$v_1 := (\mathsf{turn}, \{1\})$
if $v_1 = 1$ **then** $(\mathsf{perm}, (3\ 4))$
else if $v_1 = 2$ **then** $(\mathsf{perm}, (2\ 3\ 4))$
else if $v_1 = 3$ **then** $(\mathsf{perm}, (2\ 4\ 3))$
else if $v_1 = 4$ **then** $(\mathsf{perm}, (2\ 3))$

Let $\pi := (1\ 3)(2\ 4)$
repeat
  $(\mathsf{shuffle}, \langle(1\ 2\ 3\ 4)\rangle)$
  $v_2 := (\mathsf{turn}, \{1\})$
until $v_2 = \pi(v_1)$

$(\mathsf{shuffle}, \langle(2\ 3\ 4)\rangle)$
$v_3 := (\mathsf{turn}, \{2\})$
Let $\sigma := (1\ 4)(2\ 3)$
if $v_3 = \sigma(v_2)$ **then** $(\mathsf{result}, 4, 3)$
else $(\mathsf{result}, 3, 4)$

For an analysis of the number of shuffle steps in the protocol, observe that we have performed two shuffles until we reach the loop condition, which holds

with probability $1/4$. After the loop, we have one additional shuffle step. Hence, the expected number of shuffles is $3 + \sum_{n=1}^{\infty} \left(1 - \frac{1}{4}\right)^n = 6$.

Comparison to Niemi and Renvall [NR99]. The previous protocol, using five cards, was described in the introduction. For a pseudo-code description, see Protocol 2.

Protocol 2 Five-card AND protocol by Niemi and Renvall [NR99]. The first bit is in basis $\{1, 2\}$, the second in basis $\{3, 4\}$. The output basis is $\{1, 4\}$. See also Figure 7 for a KWH tree representation.

```
(perm, (3 4))
repeat
  | (shuffle, ⟨(1 2 3 4 5)⟩)
  | v := (turn, {1})
until v = 2 or v = 3
repeat
  | (shuffle, ⟨(2 3 4 5)⟩)
  | v := (turn, {2})
until v = 2 or v = 3
repeat
  | (shuffle, ⟨(3 4 5)⟩)
  | v := (turn, {3})
until v = 5
(result, 4, 5)
```

As Niemi and Renvall state, their running time in the number of shuffle steps is calculated as follows: Their protocol starts with a shuffle and repeats this with probability $3/5$. The second loop contains a shuffle and has a repeating probability of $3/4$. The shuffle in the final loop is repeated with probability $2/3$. In total, the expected running time is $3 + \sum_{n=1}^{\infty} \left(\frac{3}{5}\right)^n + \sum_{n=1}^{\infty} \left(\frac{3}{4}\right)^n + \sum_{n=1}^{\infty} \left(\frac{2}{3}\right)^n = 3 + 1.5 + 3 + 2 = 9.5$. However, for a fair comparison to our protocol, we eliminate the last loop from their protocol, as its only function is to ensure that the output is in basis $\{1, 4\}$, which our protocol does not guarantee. In this case, the modified Niemi–Renvall protocol has an expected number of $3 + 1.5 + 3 = 7.5$ shuffle steps. Hence, our four-card AND protocol needs one card less and outperforms the Niemi–Renvall protocol by an expected number of 1.5 shuffle steps.

6 Card-Minimal Protocols for Basis Conversion with Overlapping Bases

In this section, we give two protocols for converting a basis encoding in the case where the old and the new encoding share a card. The first protocol has an expected (finite) running time of three shuffle and turn operations. While it has

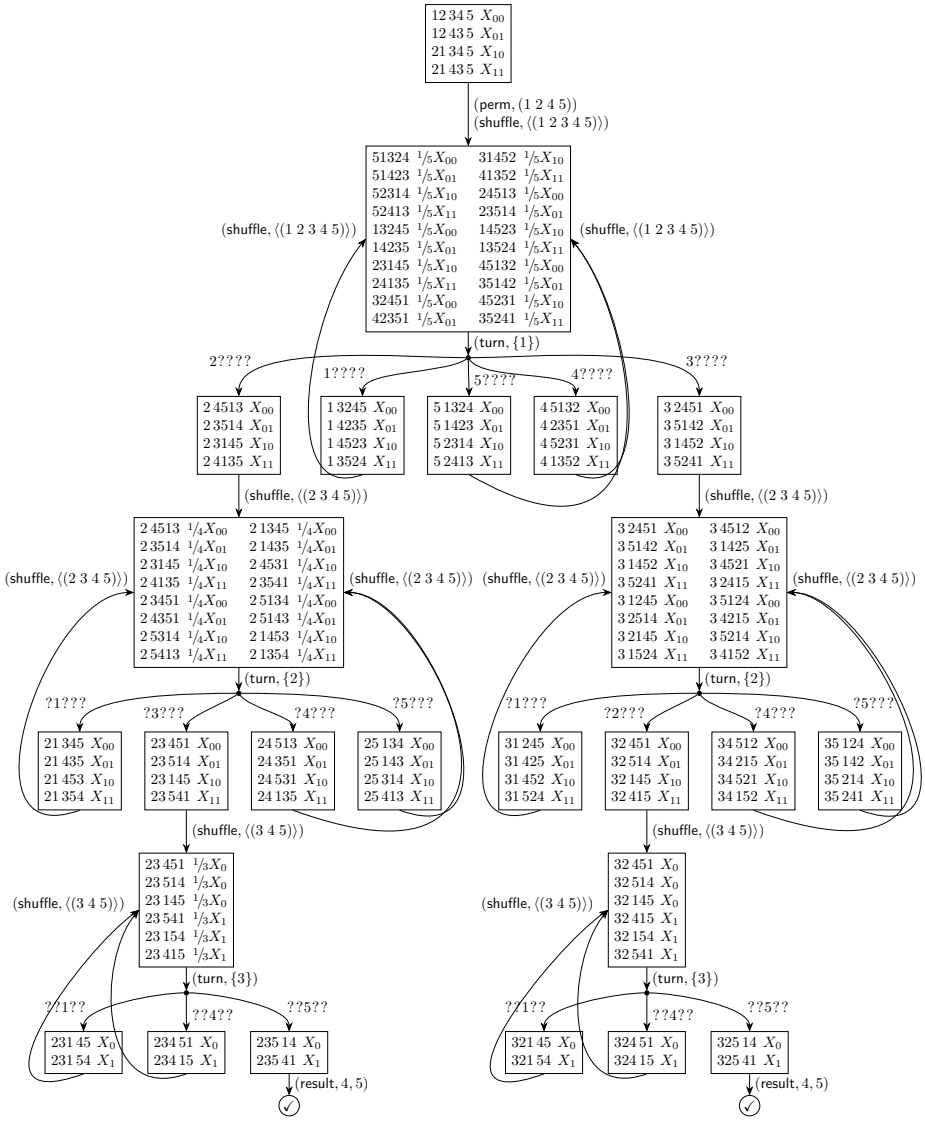


Fig. 7. KWH tree of the five-card AND protocol given by Niemi and Renvall [NR99] with $\mathcal{D} = [\![1, 2, 3, 4, 5]\!]$ using only random cuts, cf. Protocol 2. Note that $X_0 \coloneqq X_{00}+X_{01}+X_{10}$ and $X_1 \coloneqq X_{11}$. The output is in basis $\{1, 4\}$.

not been explicit in the literature, it is in a way implicit in the protocol by Niemi and Renvall [NR99], as the authors aimed to get a fixed-in-advance output basis.

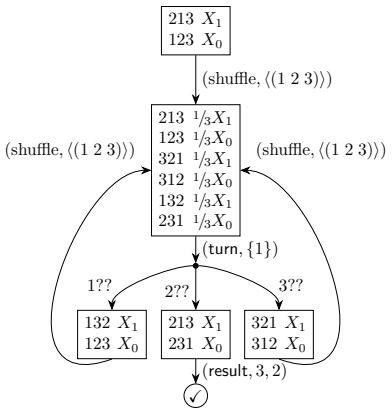


Fig. 8. Three-card Las Vegas basis conversion for $\mathcal{D} = [\![1, 2, 3]\!]$ with uniform closed shuffles.

Theorem 4. *There is a three-card Las Vegas basis-conversion protocol for overlapping bases with deck $\mathcal{D} = [\![1, 2, 3]\!]$ and uniform closed shuffles.*

Proof. See Figure 8 and Protocol 3.

Protocol 3 Three-card Las Vegas basis conversion protocol as given in Figure 8 with $\mathcal{D} = [\![1, 2, 3]\!]$, input basis $\{1, 2\}$ and output basis $\{1, 3\}$

repeat
  (shuffle, ⟨(1 2 3)⟩)
  $v :=$ (turn, {1})
until $v = 2$
(result, 3, 2)

Theorem 5. *There is a five-card finite-runtime basis conversion protocol for overlapping bases with deck $\mathcal{D} = [\![1, 2, 3, 4, 5]\!]$. It only uses two random bisection cuts as shuffle operations.*

Proof. This is just applying the basis conversion by Mizuki [M16] twice, cf. Protocol 4.

Protocol 4 Five-card finite-runtime conversion protocol with overlapping bases for $\mathcal{D} = [\![1, 2, 3, 4, 5]\!]$, input basis $\{1, 2\}$ and output basis $\{1, 3\}$

$(\mathsf{shuffle}, \langle(1\ 2)(4\ 5)\rangle)$
$v := (\mathsf{turn}, \{1\})$
if $v = 2$ **then** $(\mathsf{perm}, (1\ 2)(4\ 5))$

$(\mathsf{shuffle}, \langle(1\ 3)(4\ 5)\rangle)$
$v := (\mathsf{turn}, \{4\})$
if $v = 4$ **then** $(\mathsf{result}, 1, 3)$
else $(\mathsf{result}, 3, 1)$

```
1 struct sequence {
2   uint val[numberOfCards];
3   struct fractions probs;
4 };
```

Listing 1. C struct holding the state trees.

7 An Illustration of Our Verification Methodology

In the following, we exemplify our translation of card-based cryptographic AND protocols using standard decks to the bounded model checker CBMC, which takes programs in the C language. For our experiments, we used CBMC 5.11 with the built-in solver based on the SAT-solver MiniSat 2.2.0 [CKL04; ES03]. All experiments are performed on an AMD Opteron(tm) 6172 CPU at 2.10 GHz with 48 cores and 256 GB of RAM.

We translate KWH trees in the C language using a simple encoding into a bounded C program with only static structures and no pointers, e.g., we employ C structs (see Listing 1) holding an array of card sequences for the sequence s, attached with their respective values for each probability (for the probabilistic security notion) or dependency (for output-possibilistic security) X_i occurring in $\mu(s)$, which is simply encoded by another C struct `fractions`. The sequences are constructed using non-deterministic values restricted by respective software conditions to enforce a lexicographic ordering. Moreover, we assign the starting values in $\mu(s)$ with fixed (i.e., deterministic) values based on the constructed sequences. Subsequently, an array of (consecutively) reachable states is constructed non-deterministically using simple implementations of the turn and the shuffle operation as explained in Section 2. We then repeatedly (after each turn/shuffle) check whether all possible resulting (non-deterministic) states correctly and securely compute the specified function, e.g., here a secure AND.

An example shuffle operation is shown in Listing 2 for the case of output-possibilistic security. Therein, the keyword `__CPROVER_assume` is used by the bounded model checker to restrict all program runs passing this statement to satisfy the specified (Boolean) condition. By assigning values using the spe-

```
1 uint permSetSize = nondet_uint();
2 __CPROVER_assume (0 < permSetSize);
3 __CPROVER_assume (permSetSize <= NUM_POSS_SEQ);
4 uint permutationSet[permSetSize][numberOfCards];
5 uint takenPermutations[NUM_POSS_SEQ] = { 0 };
6
7 for (uint i = 0; i < permSetSize; i++) {
8   uint permIndex = nondet_uint();
9   __CPROVER_assume (permIndex < NUM_POSS_SEQ);
10  __CPROVER_assume (!takenPermutations[permIndex]);
11
12  takenPermutations[permIndex] = 1;
13  for (uint j = 0; j < numberOfCards; j++) {
14    permutationSet[i][j] =
15      startState.seq[permIndex][j] - 1;
16  }
17 }
18 struct state result =
19   doShuffle(startState, permutationSet, permSetSize);
20 __CPROVER_assume (isBottomFree(result));
```

Listing 2. Simplified shuffle operation for CBMC.

cial function `nondet_uint()`, we assign a non-deterministic non-negative integer number, which is restricted to values greater than zero and at most of value `NUM_POSS_SEQ` (which is a variable computed by the pre-processor and is the maximum number of sequences possible with the given deck) in the following program statement. In the shown example, the non-determinism is used to construct a set of permitted permutation sets (to be used by the shuffle operation), which makes the SBMC tool inspect the following program code for all possible assignments of this value. If necessary, this may result in a fully exhaustive search, however, the prover is often able to restrict the domain based on further program statements and dependencies seen in the rest of the program. A similar trick is used when computing the concrete permutations using the non-deterministic value of `permIndex` in order to check all possible permutations which possibly move the values, but preserve all existing numbers in the sequence itself. This is done using the `int`-array `takenPermutations`, which is first initialized to zero and, when choosing a concrete permutation, assumed to be zero at position `permIndex`, however set to the number one right afterwards (such that it is not permitted to be chosen again). In the subsequent inner loop, the permutations are assigned choosing the according cards from the sequences in the start state using the non-deterministic value `permIndex`. Finally, the shuffle is applied, resulting in the state variable `result`, which is then checked using a further method `isBottomFree` to not contain any sequences with impermissible values for X_i, which would result in incorrect computations of the AND function.

We applied our approach to the computation of a secure AND protocol using four cards in order to, firstly, substantiate our proof that no protocol of a length below six can be found, and, secondly, automatically find a permitted protocol using six operations. For the running times and formula size (i.e., numbers of variables and clauses) generated by our method, we refer to Table 3 on p. 37.

8 Verification of Run-Minimality in Two-Color Deck Protocols

For the two-color deck setting, a card-minimal Las Vegas AND protocol using only four cards was given by Koch, Walzer, and Härtel [KWH15]. While they use only closed shuffles, some of the shuffles are non-uniform and hence, the protocol is rather difficult to implement. However, we argue that it is insightful to analyze whether the protocol features a shortest run. For this, let us note that there are two possible versions of this protocol: by contracting two subsequent closed shuffles, we can generate a protocol with fewer but non-closed shuffles. Both protocols are given in Figure 9 and Protocol 5, where $\Pi_1, \mathcal{F}_1, \Pi_2, \mathcal{F}_2$ are permutation groups and probability distributions are as follows:

$$\begin{aligned} \Pi_1 &\coloneqq \langle (1\ 2)(3\ 4) \rangle, \quad \mathcal{F}_1 \colon \mathsf{id} \mapsto 1/3, (1\ 2)(3\ 4) \mapsto 2/3, \\ \Pi_2 &\coloneqq \langle (1\ 3)(2\ 4) \rangle, \quad \mathcal{F}_2 \colon \mathsf{id} \mapsto 1/3, (1\ 3)(2\ 4) \mapsto 2/3, \end{aligned} \tag{1}$$

and $\alpha_1, \alpha_2, \alpha_3$ are placeholders for one or two actions, which are for the full protocol as follows:

$$\begin{aligned} \alpha_1 &\coloneqq (\mathsf{shuffle}, \langle (1\ 3)(2\ 4) \rangle); (\mathsf{shuffle}, \langle (2\ 3) \rangle), \\ \alpha_2 &\coloneqq (\mathsf{shuffle}, \langle (1\ 3) \rangle); (\mathsf{shuffle}, \Pi_1, \mathcal{F}_1), \\ \alpha_3 &\coloneqq (\mathsf{shuffle}, \langle (3\ 4) \rangle); (\mathsf{shuffle}, \Pi_2, \mathcal{F}_2), \end{aligned} \tag{2}$$

and for the protocol using contracted shuffles as below:

$$\begin{aligned} \alpha_1 &\coloneqq (\mathsf{shuffle}, \{\mathsf{id}, (1\ 3)(2\ 4), (2\ 3), (1\ 2\ 4\ 3)\}), \\ \alpha_2 &\coloneqq (\mathsf{shuffle}, \{\mathsf{id}, (1\ 3), (1\ 3)(2\ 4), (1\ 4\ 3\ 2)\}, \mathcal{F}_3), \\ \mathcal{F}_3 &\colon \mathsf{id} \mapsto 1/6, (1\ 3) \mapsto 1/6, (1\ 3)(2\ 4) \mapsto 1/3, (1\ 4\ 3\ 2) \mapsto 1/3, \\ \alpha_3 &\coloneqq (\mathsf{shuffle}, \{\mathsf{id}, (3\ 4), (1\ 3)(2\ 4), (1\ 3\ 2\ 4)\}, \mathcal{F}_4), \\ \mathcal{F}_4 &\colon \mathsf{id} \mapsto 1/6, (3\ 4) \mapsto 1/6, (1\ 3)(2\ 4) \mapsto 1/3, (1\ 3\ 2\ 4) \mapsto 1/3. \end{aligned} \tag{3}$$

Run-Minimality Results. To summarize our run-minimality results derived from our adaption of the program to the two-color setting, we showed by formal verification that the closed AND protocol variant has a shortest run of 6 steps, relative to all closed four-card AND protocols. This is because our method excluded the possibility of an input-possibilistic[19] closed four-card AND protocol

[19] Because it found a possible output-possibilistic (but not input-possibilistic) protocol run, we had to strengthen the search criteria to protocols which are at least input-probabilistic.

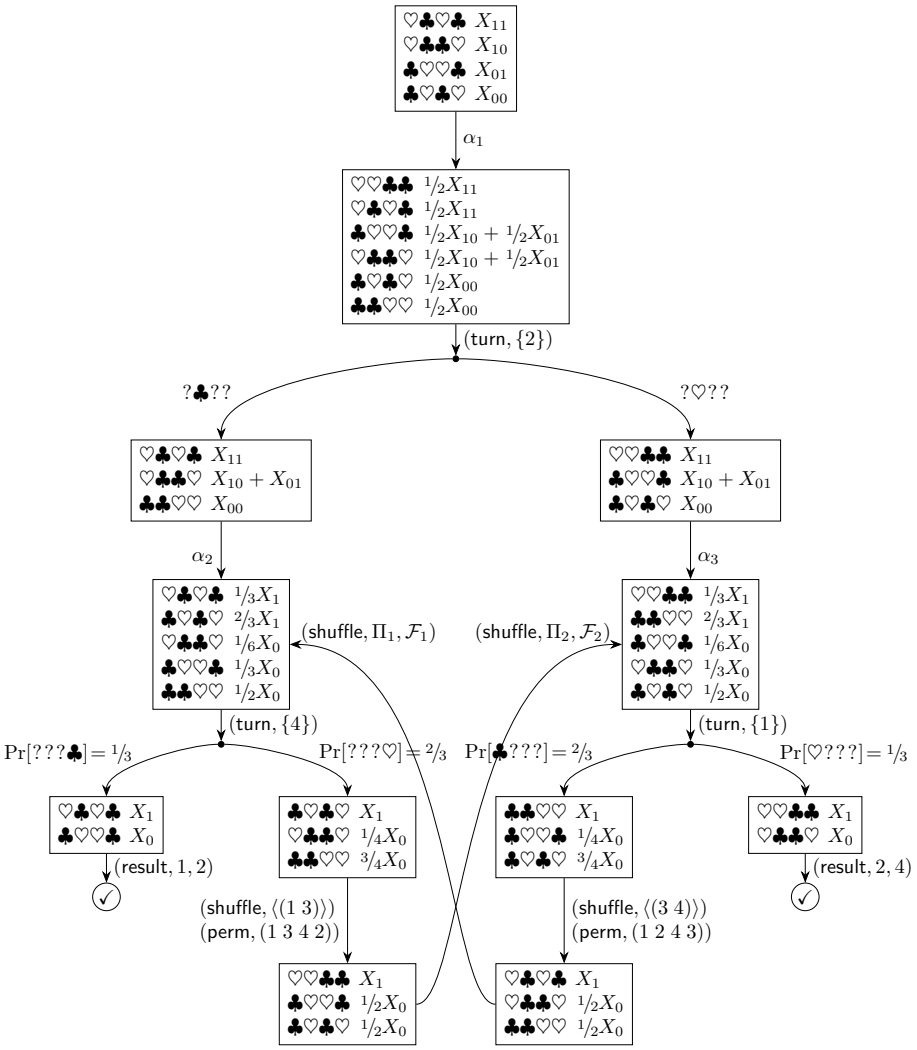


Fig. 9. The four-card protocol by Koch, Walzer, and Härtel [KWH15], with placeholders as specified in the text to define two similar variants of the same protocol. The contracted, non-closed variant has a shortest run of length 4, while the closed variant has a shortest run of length 6.

Protocol 5 Two protocols to compute AND using four cards, cf. also Figure 9. The placeholders Π_i, $\mathcal{F}_i$ are given in (1) and the α_i are defined in (2) and (3).

α_1
(turn, {2})
if $v = (?, \clubsuit, ?, ?)$ **then**
  (turn, {2}) // turn back
  α_2
1 (turn, {4})
  if $v = (?, ?, ?, \clubsuit)$ **then**
    (result, 1, 2)
  else if $v = (?, ?, ?, \heartsuit)$ **then**
    (turn, {4}) // turn back
    (shuffle, {id, (1 3)})
    (perm, (1 3 4 2))
    (shuffle, $\Pi_2, \mathcal{F}_2$)
    goto 2
else if $v = (?, \heartsuit, ?, ?)$ **then**
  (turn, {2}) // turn back
  α_3
2 (turn, {1})
  if $v = (\heartsuit, ?, ?, ?)$ **then**
    (result, 2, 4)
  else if $v = (\clubsuit, ?, ?, ?)$ **then**
    (turn, {1}) // turn back
    (shuffle, {id, (3 4)})
    (perm, (1 2 4 3))
    (shuffle, $\Pi_1, \mathcal{F}_1$)
    goto 1

with a run of length 5. Moreover, our contracted AND protocol is run-minimal in that no (output-possibilistic) four-card AND protocol with a run of length 3 exists. See also Table 3. In the following, we describe the changes for our verification method.

As the program by Koch, Schrempp, and Kirsten [KSK19] is already very general, the adaptions for covering the two-color settings required only little changes. The programs mainly differ in the assignment of the start state, in the following code snippets identified by `start`, for the protocol. In the following, the variable `NUM_SYM` specifies the number of distinct card symbols, which was not needed in the standard deck setting, as there it was identical to the total number of cards. In Listing 3, the variable `N` specifies this total number of cards.

In the standard deck setting, each player gets distinct symbols 1 and 2, or 3 and 4, respectively (as shown in the first two lines in Listing 3). For the two-color deck setting, it suffices to require that the individual cards for each player are pairwise distinct as shown in the first two lines in Listing 4. Moreover, we simply

Table 3. Running times for showing/disproving protocol existence for standard and two-color decks. While all rows having "✓" in the column "Protocol" indicate that a protocol run is output by our method with the CBMC running time as indicated in the table, these do not automatically feature probabilistic security. Hence, we add references to protocols with the given parameters, which should not (generally) be understood as having been discovered using our method.

#Cards	Shuffles	#Steps	Protocol	#Var.	#Clauses	Time
STANDARD DECKS						
4	closed	5	✗[a]	67.3 M	266.4 M	114.1 h
4	closed	6	✓, also Figure 6	68.2 M	269.7 M	45.3 h
TWO-COLOR DECKS						
4	–	3	✗	5.2 M	20.3 M	46 min
4	–	4	✓, also Figure 9 with (3)	6.9 M	27.0 M	50 min
4	closed	5	✗[b]	12.3 M	47.2 M	7.9 h
4	closed	6	✓, also Figure 9 with (2)	9.3 M	34.4 M	45 min
5	closed	4	✓, also Figure 14	22.3 M	87.2 M	45 min

[a] This holds only w.r.t. protocols with shuffle size of at most 8, excluding subgroups of size 12.
[b] For this, we had to strengthen the security to input-possibilistic security.

```
__CPROVER_assume ((i != 0 && i != 1) || start[i] == 1 || start[i] == 2);
__CPROVER_assume ((i != 2 && i != 3) || start[i] == 3 || start[i] == 4);
for (uint i = 4; i < N; i++) {
    start[i] = i + 1;
}
```

Listing 3. Simplified start sequence assignment in the standard deck for CBMC.

numbered the helper cards consecutively for the standard deck setting (see the loop in Listing 3), but allowed an arbitrary assignment of valid card symbols in the two-color deck setting (see the loop in Listing 4).

Besides the introduction of the variable `NUM_SYM`, these are the main changes that were needed in order to cover the two-color deck setting. Note that we moreover adapted the script that calls the SBMC tool together with our C program to compute the new number of possible sequences. For the standard deck setting, the number was simply the factorial of the total number of cards. In the two-color deck setting, this is the binomial coefficient of the two different amounts of cards with distinct symbols.

9 Verification of Shuffle Set Size Maximality

In the following, we exploit the fact that the number of possible sequences in a protocol state may be significantly smaller than the number of possible permutations on the deck for the two-color setting. We therefore extend our formal verification technique to additionally establish a formal guarantee that

```
__CPROVER_assume (start[1] != start[0]);
__CPROVER_assume (start[3] != start[2]);
for (uint i = 4; i < N; i++) {
    start[i] = nondet_uint();
    __CPROVER_assume (0 < start[i]);
    __CPROVER_assume (start[i] <= NUM_SYM);
}
```

Listing 4. Simplified start sequence assignment in the two-color deck for CBMC.

```
uint seqIdx1 = nondet_uint();
uint seqIdx2 = nondet_uint();
__CPROVER_assume (seqIdx1 < seqIdx2);
minState.sequence[seqIdx1].probs = {1, 0}; // set probability to X_0
minState.sequence[seqIdx2].probs = {0, 1}; // set probability to X_1

struct state nextState = performShuffle(minState);
uint foundValidState = isValid(nextState);
assert (foundValidState);
```

Listing 5. Simplified maximality verification for CBMC.

it suffices to search protocols with a smaller permutation set size (i.e., also the shuffle set size). Hence, the number of possible shuffles gets significantly smaller, which reduces the work for the SBMC tool and thus leads to significantly smaller running times.

We can write a simple program – via some simple adaptions from the program in Section 7 – that serves as an input for the SBMC tool to verify the maximality of a given shuffle set size. The shuffle operation from Listing 2 is adapted such that we can specify a lower bound for the non-deterministic variable `permSetSize`. We search for a single shuffle operation such that a valid output state is reached from a "minimal state", i.e., a state that has at most one 1-sequence and one 0-sequence (that should not be mixed together in the shuffle). In Listing 5 this is done by setting the probabilities of two arbitrary distinct sequences in that state to be the inverse of each other, i.e., (1 0) and (0 1). In the end, we check whether, after performing a shuffle operation on this state, we can still reach a valid state afterwards. Note that, since we are looking for worst-case maximality bounds, it suffices to employ the output-possibilistic setting (see Definition 2) which reduces the search complexity.

For the verification of a maximal shuffle set size, we can run the SBMC tool on this program for various lower bounds for `permSetSize` until we find the smallest value such that no valid state is reachable anymore. This gives us a guarantee that larger shuffle set sizes cannot produce smaller protocol runs and we can hence use this value for an upper bound on the shuffle set size in the approach from Section 8.

The described functionality in the C program is shown in Listing 5. Therein, `seqIdx1` and `seqIdx2` are the non-deterministically chosen indices for the zero- and one-sequence, which are assumed to be distinct. The minimal start state is given by the variable `minState` (which contains an array of sequences). We perform a non-deterministic shuffle operation on `minState` by calling the method `performShuffle`. Finally, we ask the SBMC tool to check whether the produced `nextState` is a valid state using the final `assert` statement.

Note that the results of this section in determining the maximal useful shuffle set size hold not only for AND but also for *all Boolean functions* that have at least two possible outputs. The results are summarized in Table 4.

Table 4. Running times for proving shuffle set size maximality. For some of the settings with closedness requirement we specify ranges, which should indicate that the larger range is already impossible due to the size restrictions of subgroups. See [N14a; N14b] for reference.

#Cards	Shuffles	Shuffle Size	Valid Shuffle	#Var.	#Clauses	Time
Standard Decks						
4	–	12	✓, cf. Figure 11	5.2 M	12.9 M	51.9 min
4	–	13	✗	13.8 M	55.9 M	2.4 h
4	closed	12	✓, cf. Figure 11	13.5 M	54.0 M	16.9 min
4	closed	13 – 24	✗[a]	–	–	–
Two-color Decks						
4	–	12	✓, cf. Figure 10	1.5 M	6.1 M	54 sec
4	–	13	✗	1.6 M	6.5 M	70 sec
4	closed	8	✓, cf. Figure 10	1.4 M	5.0 M	69 sec
4	closed	(9 –) 12	✗	2.2 M	8.2 M	3.2 min
5	–	48	✓	13.9 M	57.0 M	3.4 h
5	–	49	✗	14.2 M	58.1 M	11.4 h
5	closed	12	✓	4.9 M	18.9 M	26.1 min
5	closed	20	?[b]	9.1 M	35.4 M	–
5	closed	24	?[b]	11.8 M	46.4 M	–
5	closed	25 – 120	✗[c]	–	–	–

[a] As the largest proper subgroup is of size 12, there is nothing to show. (S_4 creates $\perp$-sequences).
[b] This run did not finish in time, or ran into the self-set timeout bound of 5 days.
[c] >48 permutations is impossible even non-closed, and 60 is the only *proper* subgroup size >24.

As an example, see Figure 10 (left) for the maximal shuffle set size (of 12) that is useful in four-card two-color protocols in general. Here, the shuffle starts from a minimal 2-sequence state that was chosen arbitrarily and non-deterministically by our SAT solver, but is likely to have maximal Hamming distance among their sequences. For protocols using only *closed* shuffles, our method showed that this bound is 8 permutations, as there is no larger closed permutation set that can result in a valid state, cf. Figure 10 (right). These bounds are fully tight.

In the *five-card* two-color setting, closed protocols can make use of shuffle groups of at most 24 permutations. It is an open question whether this is a tight bound, but we know that there is a 12 element shuffle that is valid. However, it still

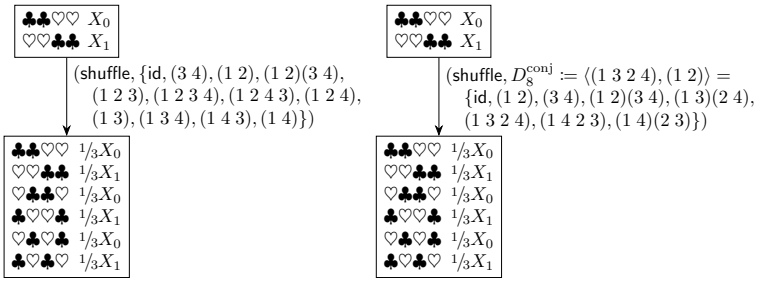


Fig. 10. Situation discovered by our formal method to find a minimal state and a maximal permutation set (of size 12 (left) and 8 (right), respectively), such that applying this shuffle to the minimal state does not generate an invalid state (with $\bot$-sequences). Our method showed that larger shuffle sets (left) or groups (right) cannot result in valid states, allowing us to reduce the shuffle set size in verification steps without loosing generality. Here, D_8^{conj} denotes a dihedral group of order 8.

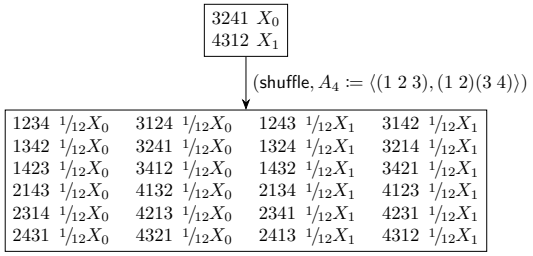


Fig. 11. Situation discovered by our formal method to find a minimal state and a maximal permutation set (of size 12, namely the alternating group A_4), such that applying this shuffle to the minimal state does not generate an invalid state (with $\bot$-sequences), in the standard deck setting.

allows us to restrict the maximal shuffle group size to 24 when searching protocols. For this five-card case and arbitrary non-closed shuffle sets, the maximal shuffle set size that does not introduce $\bot$-sequences on a minimal state is 48. This is a tight bound.

Additionally, we have adapted this method to the standard deck setting as well and have determined that the largest permutation set permissible in a protocol on four cards is 12. This also holds for the closed case, i.e., there is a group with 12 elements, namely the alternating group A_4, that, if performed on a minimal state, can result in a state that does not contain any $\bot$-sequences.

10 Conclusion

In this paper, we proposed a new method to search card-based protocols for any secure computation, by giving a general formal translation applicable to be used by the formal technique of software bounded model checking (SBMC). This

method allows us to find new protocols automatically, and prove lower bounds on required shuffle and turn operations for any protocol, and provide an example for the computation of a minimal AND protocol. We also found a new protocol that only uses the theoretical minimum of four distinguishable cards for an AND computation. Moreover, we supported this finding by our automatic method in showing the impossibility of any protocol using less shuffle and turn operations using only practicable shuffles (random cuts). The protocol is hence optimal w.r.t. the running time restriction "restart-free Las-Vegas". For the four-card standard deck setting, we showed that there is no finite runtime protocol, regardless of the shuffle operations used. This result completes the picture of tight lower bounds for the four-card setting. Additionally, we showed tight lower bounds on basis conversions for single bits and proposed the missing protocols, and establish the theorem that using a minimum of five cards, both input- and output-bases can be chosen freely, which fosters our impossibility result for the four-card setting.

Finally, we extended our verification method to the case of decks using only two colors, which is more common in the field of card-based cryptography. In this setting, we were able to show two variants of a card-minimal Las Vegas AND protocol to be also run-minimal, i.e., the protocol has a run of minimal length. Moreover, for the case of 4 cards, we derived tight upper bounds on the size of the maximal usable permutation set, of 12 and 8 for general and closed protocols, respectively. As this is not restricted to AND protocols, but applies more generally, we believe this to be of independent interest for researchers in the field of card-based cryptography.

Open Problems. Let us point out some open problems in the card-based security area that could be approached based on the findings in this paper: (1) For finite-runtime protocols, there exist no proven tight lower bounds on the required number of cards (five to eight cards). We recommend more research applying computer-aided formal methods at this point, as the state space for five or more cards is very large. (2) Our verification approach is fast for finding protocols and/or lower bounds on the operations needed in a protocol for given shuffle-restrictions. However, this is based on the assumption that protocols exist already for a given predefined length to find or confirm impossibility results. Investigating computer-aided formal methods for universal impossibility results might be worthwhile. (3) The two most common settings in card-based cryptography are the standard deck setting with only distinguishable cards and the two-color decks using ♣ and ♡. However, it may be possible that by mixing these settings (e.g., only distinguishable cards with one pair of identical cards), we might find more efficient protocols (especially in the finite runtime setting). For such a mixed setting, Shinagawa and Mizuki [SM19] provide nice results to use in further research.

Acknowledgments. The authors would like to thank the anonymous reviewers for their detailed and helpful comments and suggestions.

Appendix: Further Protocols

This appendix contains the 8-card AND protocol by Mizuki [M16] (Figure 12) and a second four-card protocol which uses a number of 4.5 shuffles in expectation, which are, however, non-closed and hence, more impractical to implement, cf. Figure 13. Moreover, we have added a variant of the protocol by Abe et al. [AHM+18] where we save one permutation step in the beginning, in Figure 14.

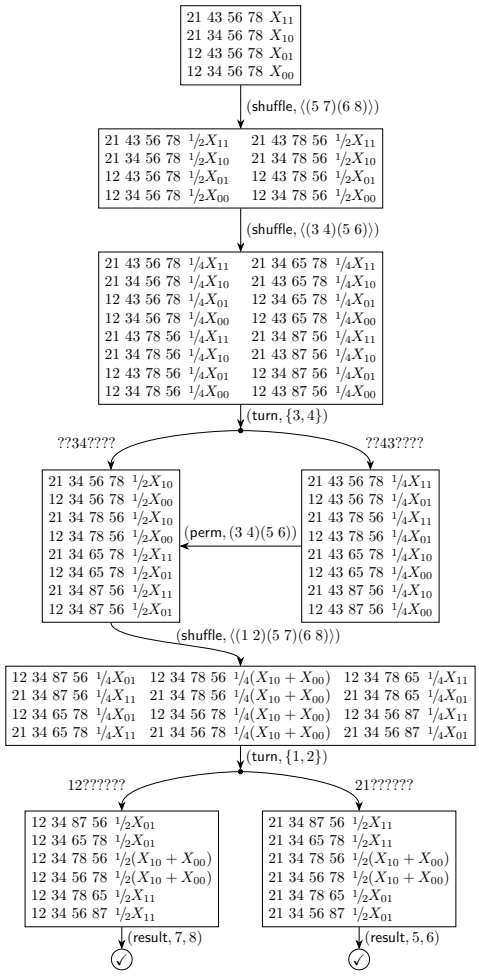


Fig. 12. The eight-card finite-runtime AND protocol by Mizuki [M16], with $\mathcal{D} = [\![1, \ldots, 8]\!]$ and uniform-closed shuffles. Output is in basis $\{5, 6\}$ or $\{7, 8\}$, each with probability $1/2$.

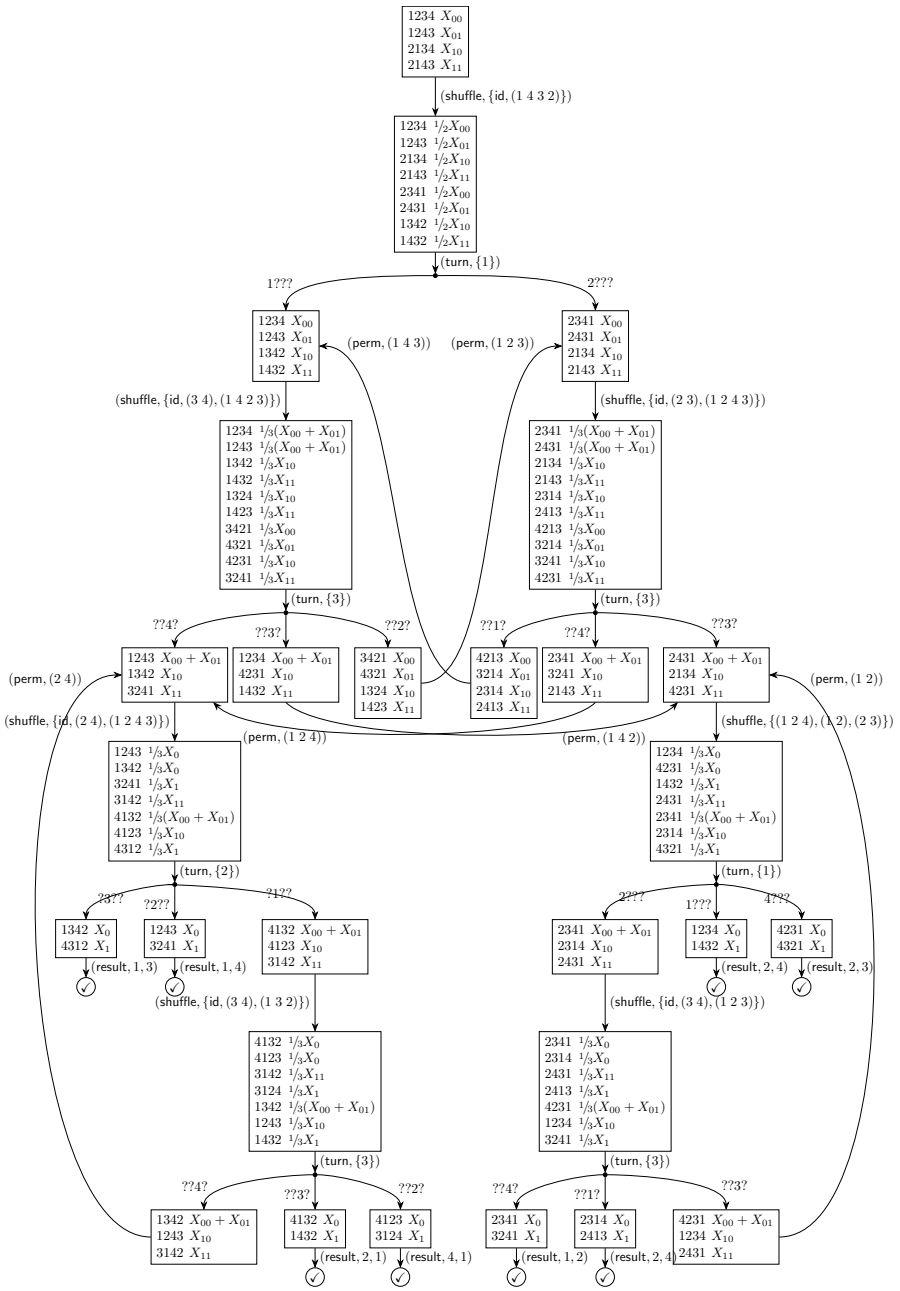


Fig. 13. A four-card Las Vegas AND protocol with deck $\mathcal{D} = [\![1, 2, 3, 4]\!]$ and uniform shuffles. Note that $X_0 \coloneqq X_{00} + X_{01} + X_{10}$ and $X_1 \coloneqq X_{11}$. The output is in one of the bases $\{1, 3\}, \{1, 4\}, \{2, 3\}, \{3, 4\}$, determined by the position of the final state in the tree, and can be converted as needed.

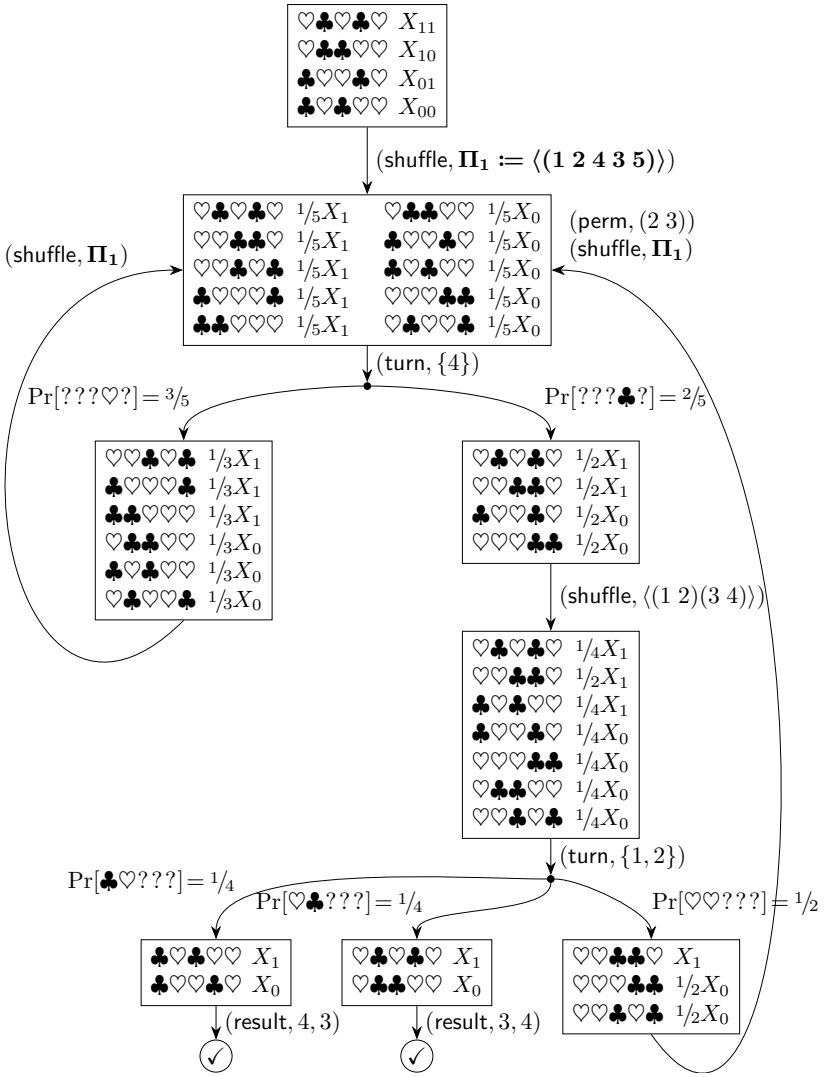


Fig. 14. A slightly shorter version of the five-card two-color Las Vegas AND protocol with uniform closed shuffles given by Abe et al. [AHM+18]. Here, we save one initial permutation step at the cost of using the slightly more complex shuffle Π_1 that is not as easy to perform as just cutting the cards (albeit still a "random cut", i.e., a cyclic group generated by a cycle). As our counting method for the number of steps assumes single-card turns, observe that the two-card turn step in the end can be split into two single-card turns, where turning the first card can already result in the final state on the left. Hence, its shortest run consists of only four steps. (This protocol version was found when trying to prove the run-minimality of [AHM+18] w. r. t. closed five-card two-color AND protocols – whether a protocol with these parameters and a run of only three steps exists, remains open.)

References

[AHM+18] Y. Abe, Y.-i. Hayashi, T. Mizuki, and H. Sone. "Five-Card AND Protocol in Committed Format Using Only Practical Shuffles". In: *APKC@AsiaCCS 2018*. Ed. by K. Emura et al. ACM, 2018, pp. 3–8. DOI: 10.1145/3197507.3197510.

[APS14] M. Avalle, A. Pironti, and R. Sisto. "Formal verification of security protocol implementations: a survey". In: *Formal Asp. Comput.* 26.1 (2014), pp. 99–123. DOI: 10.1007/s00165-012-0269-9.

[B12] B. Blanchet. "Security Protocol Verification: Symbolic and Computational Models". In: *POST 2012*. Ed. by P. Degano and J. D. Guttman. LNCS 7215. Springer, 2012, pp. 3–29. DOI: 10.1007/978-3-642-28641-4_2.

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. "Symbolic Model Checking without BDDs". In: *TACAS 1999*. Ed. by W. R. Cleaveland. LNCS 1579. Springer, 1999, pp. 193–207. DOI: 10.1007/3-540-49059-0_14.

[CK93] C. Crépeau and J. Kilian. "Discreet Solitary Games". In: *CRYPTO '93*. Ed. by D. R. Stinson. LNCS 773. Springer, 1993, pp. 319–330. DOI: 10.1007/3-540-48329-2_27.

[CKL04] E. M. Clarke, D. Kroening, and F. Lerda. "A Tool for Checking ANSI-C Programs". In: *TACAS 2004*. Ed. by K. Jensen and A. Podelski. LNCS 2988. Springer, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.

[dB89] B. den Boer. "More Efficient Match-Making and Satisfiability: The Five Card Trick". In: *EUROCRYPT '89*. Ed. by J. Quisquater and J. Vandewalle. LNCS 434. Springer, 1989, pp. 208–217. DOI: 10.1007/3-540-46885-4_23.

[ES03] N. Eén and N. Sörensson. "An Extensible SAT-solver". In: *SAT 2003*. Ed. by E. Giunchiglia and A. Tacchella. LNCS 2919. Springer, 2003, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37.

[FFN14] B. Fisch, D. Freund, and M. Naor. "Physical Zero-Knowledge Proofs of Physical Properties". In: *CRYPTO 2014*. Ed. by J. A. Garay and R. Gennaro. LNCS 8617. Springer, 2014, pp. 313–336. DOI: 10.1007/978-3-662-44381-1_18.

[FHK+14] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith. "CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations". In: *CC 2014*. Ed. by A. Cohen. LNCS 8409. Springer, 2014, pp. 244–249. DOI: 10.1007/978-3-642-54807-9_15.

[GBG14] A. Glaser, B. Barak, and R. J. Goldston. "A zero-knowledge protocol for nuclear warhead verification". In: *Nature* 510 (2014), pp. 497–502. DOI: 10.1038/nature13457.

[K18] A. Koch. *The Landscape of Optimal Card-based Protocols*. 2018. Cryptology ePrint Archive, Report 2018/951.

[K19] A. Koch. "Cryptographic Protocols from Physical Assumptions". PhD thesis. Karlsruhe: KIT, 2019. DOI: 10.5445/IR/1000097756.

[KKW+17] J. Kastner, A. Koch, S. Walzer, D. Miyahara, Y.-i. Hayashi, T. Mizuki, and H. Sone. "The Minimum Number of Cards in Practical Card-based Protocols". In: *ASIACRYPT 2017*. Ed. by T. Takagi and T. Peyrin. LNCS 10626. Springer, 2017, pp. 126–155. DOI: 10.1007/978-3-319-70700-6_5.

[KSK19] A. Koch, M. Schrempp, and M. Kirsten. "Card-Based Cryptography Meets Formal Verification". In: *ASIACRYPT 2019,* Proceedings, Part I. Ed. by S. D. Galbraith and S. Moriai. LNCS 11921. Springer, Nov. 25, 2019, pp. 488–517. DOI: 10.1007/978-3-030-34578-5_18.

[KW20] A. Koch and S. Walzer. "Foundations for Actively Secure Card-based Cryptography". In: *Fun with Algorithms, FUN 2021*. Ed. by M. Farach-Colton, G. Prencipe, and R. Uehara. Vol. 157. LIPIcs. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, 17:1–17:23. DOI: 10.4230/LIPIcs.FUN.2021.17.

[KWH15] A. Koch, S. Walzer, and K. Härtel. "Card-based Cryptographic Protocols Using a Minimal Number of Cards". In: *ASIACRYPT 2015*. Ed. by T. Iwata and J. H. Cheon. LNCS 9452. Springer, 2015, pp. 783–807. DOI: 10.1007/978-3-662-48797-6_32.

[M16] T. Mizuki. "Efficient and Secure Multiparty Computations Using a Standard Deck of Playing Cards". In: *CANS 2016*. Ed. by S. Foresti and G. Persiano. LNCS 10052. Springer, 2016, pp. 484–499. DOI: 10.1007/978-3-319-48965-0_29.

[MN10] T. Moran and M. Naor. "Basing cryptographic protocols on tamper-evident seals". In: *Theor. Comput. Sci.* 411.10 (2010), pp. 1283–1310. DOI: 10.1016/j.tcs.2009.10.023.

[MS09] T. Mizuki and H. Sone. "Six-Card Secure AND and Four-Card Secure XOR". In: *FAW 2009*. Ed. by X. Deng et al. LNCS 5598. Springer, 2009, pp. 358–369. DOI: 10.1007/978-3-642-02270-8_36.

[MS14] T. Mizuki and H. Shizuya. "A formalization of card-based cryptographic protocols via abstract machine". In: *Int. J. Inf. Sec.* 13.1 (2014), pp. 15–23. DOI: 10.1007/s10207-013-0219-4.

[MS17] T. Mizuki and H. Shizuya. "Computational Model of Card-Based Cryptographic Protocols and Its Applications". In: *IEICE Transactions* 100-A.1 (2017), pp. 3–11. DOI: 10.1587/transfun.E100.A.3.

[N14a] V. Naik. "Subgroup structure of symmetric group:S4". In: *Groupprops, The Group Properties Wiki*. 2014. URL: https://groupprops.subwiki.org/wiki/Subgroup_structure_of_symmetric_group:S4.

[N14b] V. Naik. "Subgroup structure of symmetric group:S5". In: *Groupprops, The Group Properties Wiki*. 2014. URL: https://groupprops.subwiki.org/wiki/Subgroup_structure_of_symmetric_group:S5.

[NR98] V. Niemi and A. Renvall. "Secure Multiparty Computations Without Computers". In: *Theor. Comput. Sci.* 191.1-2 (1998), pp. 173–183. DOI: 10.1016/S0304-3975(97)00107-2.

[NR99] V. Niemi and A. Renvall. "Solitaire Zero-knowledge". In: *Fundam. Inform.* 38.1-2 (1999), pp. 181–188. DOI: 10.3233/FI-1999-381214.

[RSH19] A. Rastogi, N. Swamy, and M. Hicks. "Wys*: A DSL for Verified Secure Multi-party Computations". In: *POST 2019*. Ed. by F. Nielson and D. Sands. LNCS 11426. Springer, 2019, pp. 99–122. DOI: 10.1007/978-3-030-17138-4_5.

[SHK+16] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin. "Dependent types and multi-monadic effects in F". In: *POPL 2016*. Ed. by R. Bodık and R. Majumdar. ACM, 2016, pp. 256–270. DOI: 10.1145/2837614.2837655.

[SM19] K. Shinagawa and T. Mizuki. "Secure Computation of Any Boolean Function Based on Any Deck of Cards". In: *FAW 2019*. Ed. by Y. Chen et al. LNCS 11458. Springer, 2019, pp. 63–75. DOI: 10.1007/978-3-030-18126-0_6.