Karlsruhe Institute of Technology

An Automated Approach to Generating Card-Based Cryptographic Protocols

Bachelor's Thesis by

Anne Elisabeth Hoff

at the KIT Department of Informatics
Institute of Information Security and Dependability (KASTEL)

Reviewer:	Prof. Dr. rer. nat. Bernhard Beckert
Advisor:	Dr. rer. nat. Michael Kirsten

24 October 2022 – 24 February 2023

Karlsruher Institut für Technologie
KIT-Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that the work presented in this thesis is entirely my own. I confirm that I specified all employed auxiliary resources and clearly acknowledged anything taken verbatim or with changes from other sources. I further declare that I prepared this thesis in accordance with the rules for safeguarding good scientific practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, 24 February 2023

Anne Hoff

(Anne Elisabeth Hoff)

Contents

1 Introduction

1.1 Motivation

Card-based protocols provide a method of performing multi-party computation without computers. Such protocols can be used in a classroom or lecture setting to teach students about secure multi-party computation in an accessible and easy to understand way. All that is needed, is a deck of playing cards whose backs are indistinguishable from each other and whose fronts are showing two different symbols, ♣ and ♡. As an example let us assume that we have two players *a* and *b* that want to securely compute the logical *AND* function of two bits. They can do so, by performing the *AND* protocol by Mizuki and Sone (2009). For this the two players input their bits as two face-down cards. They put $\boxed{\clubsuit}\,\boxed{\heartsuit}$ to if their input bit is 0 and $\boxed{\heartsuit}\,\boxed{\clubsuit}$ if their input bit is 1. These cards are placed in a line with two additional cards $\boxed{\clubsuit}\,\boxed{\heartsuit}$ behind. Thus the cards that are on the table in the beginning are ▨ ▨ ▨ ▨ $\boxed{\clubsuit}\,\boxed{\heartsuit}$ where the first two cards are the input of player *a* and the third and fourth card are the input of the player *b*. We then perform the protocol as follows:

1. We turn over the last two cards so the cards on the table look indistinguishable like this:
   ▨ ▨ ▨ ▨ ▨ ▨

2. Then we rearrange the cards like this:
   $\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}\,\boxed{5}\,\boxed{6} \longrightarrow \boxed{1}\,\boxed{3}\,\boxed{4}\,\boxed{2}\,\boxed{5}\,\boxed{6}$

3. Afterwards we split the cards into two:
   [▨ ▨ ▨] [▨ ▨ ▨]
   and randomly swap them with probability exactly 1/2.
   The result is either the original card sequence or the sequence in which the cards of the left side are now on the right side.

4. We rearrange our cards again:
   $\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}\,\boxed{5}\,\boxed{6} \longrightarrow \boxed{1}\,\boxed{4}\,\boxed{2}\,\boxed{3}\,\boxed{5}\,\boxed{6}$

5. Now we turn over the first two cards:

a) If the cards that were turned over are:
   ♣ ♡ ▨ ▨ ▨ ▨
   the result is encoded by the fifth and sixth card

b) If the cards that were turned over are:
   ♡ ♣ ▨ ▨ ▨ ▨
   the result is encoded by the third and fourth card

Depending on the cards that were turned over in the last steps, the players *a* and *b* can now turn over the two cards containing the result. The result is interpreted in the same way as the input. If the output is ♣ ♡ then the result of the calculation is 0 and if the output is ♡ ♣ the result is 1. The focus of the research on card-based cryptography is on finding new protocols, that are increasingly effective. That means, that they require fewer cards or there are fewer steps needed to perform the protocol. To automate the process of finding new card-based protocols, Koch et al. (2021) employ the technique of bounded model checking. They implement the basic actions and states of a protocol in a symbolic program and then use a bounded model checker to find valid runs through the symbolic program. If a valid run is found, the protocol is returned by the bounded model checker. If no valid run can be found, then there exists no protocol for the given parameters. Koch et al. (2021) provide a standardized program that can be used to find card-based protocols for the *AND* function. But computation using playing cards is not just focused on simple functions like an *AND* function. Niemi and Renvall (1998) and Nishida et al. (2015) have shown that we can construct a card-based protocol for any boolean function. And there are several examples of protocols for practical functions like a *COPY* function by Mizuki and Sone (2009) or even a protocol for a half adder by Mizuki et al. (2013) that use more cards and steps than simple binary boolean operators.

In this thesis we therefore want to take the technique introduced by Koch et al. (2021), and further develop it. We want to determine whether it is possible to generalize the technique and make it applicable to find protocols for any function like the *COPY* or half adder function. We also want to ascertain if we can increase the efficiency of the technique to make it applicable to larger problems.

1.2 Contribution

Firstly, we generalize the method of Koch et al. (2021) that uses bounded model checking to find card protocols, so that we can find protocols for any boolean function. We use this generalization to apply it to a range of new functions. For the *OR* as well as the *COPY* function, we give new protocols. We also show that there are no protocols for certain preconditions for the *OR* and *XOR* functions as well as the *COPY* and half adder functions. Additionally we share our findings about the effectiveness of various SAT solvers for the use within the bounded model checker.

Secondly we further universalize the method of Koch et al. (2021) by inserting a nested structure into the symbolic program. By doing that a protocol that has for example been found previously with bounded model checking or is a protocol from literature, can be used as an operation within another protocol. We provide a definition for this new operation. To be able to graphically represent logs that use this structure, we give an extension to the definition of KWH trees. We apply this method with a nested structure to the *COPY* and half adder function. For the *COPY* function we give a six card protocol which we found by this application. We show, that we can construct a protocol for any boolean function using only two protocols by making use of this six card *COPY* protocol. We also show that there are no protocols for certain prerequisites for the *COPY* and half adder function.

Thirdly we explore an alternative data structure within the implementation of the representation of sequences. We exchange the arrays of the original implementations with a Char datatype on which we can perform bitwise operations. We investigate if there is an improvement in the performance of the bounded model checker when using the new data structure.

1.3 Outline

We provide definitions drawn from the literature for card-based protocols in sections 2.1 and 2.2 and give an introduction to how bounded model checking can be used to find card-based protocols in sections 2.3 and 2.4. We introduce a definition of a new protocol action and extend the definition of KWH-trees in section 4.1.1 to include the additional action. This is the only adaption of our definitions given in chapter 2. In section 3.1 we generalize the method from section 2.4 and show how we can use it to find protocols for any function. We provide a description and the results of our tests that use an implementation of the generalizations from section 3.1 in section 3.2. We also experiment with the use of different SAT solvers in section 3.2.2. We further universalize the method of using bounded model checking to find card-based protocols in section 4.1 by introducing a nested structure within the symbolic program. After introducing an implementation in section 4.2 we present the results of our tests employing the nested structure in section 4.3. In chapter 5 we present an approach how to replace the data structure to represent sequence. In section 5.1 we provide an implementation of our symbolic program with the new data structure and how we can use bitwise operations to perform our actions within a protocol. We present our test setup and give the results and how the new data structure performed compared to the old one during bounded model checking in section 5.2. Finally, we provide a summary of the results in section 6.1. A few starting points for future research are listed in section 6.2.

2 Foundations

We specify the definitions of permutations in section 2.1 and give a description of card-based protocols in section 2.2. Section 2.2.3 explains the construction of KWH trees which will be used as a way of representing card-based protocols. Sections 2.3 and 2.4 will give a brief description of C bounded model checking and how it can be used to find card protocols.

2.1 Permutations

Permutations are central to card-based protocols. As we have seen in section 1.1 shuffles (which consist of multiple shuffles) and permutations are used as steps in the protocol to calculate the result Therefore, in the following we define the most important terms such as permutations, cycles, explain their properties and introduce notations. The subsequent explanations and definitions are gathered from Koch (2019).

Definition 1 (Permutation). *A permutation of a set X is the bijective function $\pi : X \longrightarrow X$.*

The set S_n ($n \in \mathbb{N}$) is called the *symmetric group* for a number n and contains all possible permutations of $X = \{1, ..., n\}$. Its neutral element is the identity (the permutation of length n, that maps every permutation in S_n onto itself) and it uses composition ($\circ$) as its group operation. We denote a *set of permutations* with $\Pi \subseteq S_n$.

A *cycle* is a tuple $(x_1 x_2 ... x_n)$ where $x_1, x_2, ... x_n \in X$ are all distinct elements. The *cyclic permutation* π is a permutation with $\pi(x_n) = x_1$ and $\pi(x_i) = x_{i+1}$ $(1 \leq i \leq n)$ and $\pi(x) = x$ for all x that are not occurring in the cycle but are in X. A composition of multiple cycles with disjoint sets can be written next to each other. Every permutation can be written in such a fashion, which we call *cycle decomposition.* From now on, we write permutations as cycle decompositions. For example, (123)(45) is a permutation with the mappings $\{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 1, 4 \mapsto 5, 5 \mapsto 4\}$.

2.2 Card-Based Protocols

The following definition of card-based protocols are gathered from Koch et al. (2021), Saito et al. (2020), Koch (2019), and Koch et al. (2015).

Each card-protocol is executed on a deck of cards. These are the cards we use to perform the protocol.

Definition 2 (Deck). *A deck $\mathcal{D}$ is a multiset over a symbol set or deck alphabet Σ.*

A deck is represented by $[[\cdot]]$. In our example in section 1.1 we use the deck $[[\clubsuit, \clubsuit, \clubsuit, \heartsuit, \heartsuit, \heartsuit]]$. In this thesis, we focus exclusively on the deck that consists of symbols from $\Sigma = \{\clubsuit\heartsuit\}$. We call this the *two-color deck*. A possible deck with the deck alphabet $\Sigma = \{\clubsuit\heartsuit\}$ could therefore be $[[\clubsuit, \clubsuit, \heartsuit, \heartsuit, \heartsuit]]$. All cards within a deck have indistinguishable backs. Cards with the same symbol from the deck alphabet also have a front that is indistinguishable from each other.

A card within a deck can either be turned face-up (the symbol of the card is showing) or face-down (the symbol of the card is hidden). To represent a card that is turned face down, we define the special symbol '?', which is not part of Σ. We can therefore depict a card lying on the table by a fraction $\frac{a}{b}$ where a and b are elements of $\Sigma \cup \{?\}$. The card that is face-up is represented by a, while the card that is face-down is represented by b. Thus, a face up card could be $\frac{\clubsuit}{?}$ while the same card facing down would be written as $\frac{?}{\clubsuit}$. To encode values with cards we need to have two cards. We call two cards that encode a value (usually 0 or 1) a commitment.

Definition 3 (Commitment). *A commitment to x is a pair of two face-down cards that encode the value x.*

Generally Koch et al. (2021) define the encoding for two bits c_1 and c_2 as follows:

$$c_1 c_2 = \begin{cases} 0 & \text{if } c_1 < c_2 \\ 1 & \text{if } c_1 > c_2 \end{cases} \tag{2.1}$$

For the two-color deck $\{\clubsuit\heartsuit\}$ we set the order as $\clubsuit < \heartsuit$. Therefore two cards from the two-color deck together form an encoding of a bit as follows: $\clubsuit\heartsuit$ encodes a 0 and $\heartsuit\clubsuit$ encodes a 1. Thus the commitment to 0 is the cards $\clubsuit\heartsuit$ and the commitment to 1 is the cards $\heartsuit\clubsuit$.

Card-based protocols operate on sequences of cards. A sequence consists of all cards in a deck. These cards have a specific order and orientation. Thus a sequence for the deck $[[\clubsuit, \clubsuit, \clubsuit, \heartsuit, \heartsuit, \heartsuit]]$ could be $(\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\heartsuit},)$.

Definition 4 (sequence). *A sequence* $\Gamma = (\alpha_1, \alpha_2, ..., \alpha_{|\mathcal{D}|})$ *contains all cards from a deck $\mathcal{D}$ in a given order and orientation.*

All possible sequences on the deck $\mathcal{D}$ are $Seq^{\mathcal{D}}$. Koch (2019, Chapter 6) defined the several different actions on sequences $\Gamma = (\alpha_1, \alpha_2, ..., \alpha_{|\mathcal{D}|})$ that a protocol can perform as follows:

Definition 5 (permutation (perm, π))**.** *A permutation (perm, π) with $\pi \in S_n$ permutes Γ according to π. The result of this operation is the sequence. $\pi(\Gamma)$.*

Take the sequence $\Gamma = (\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit})$ and the permutation π =(perm, (12)(34)). If we apply the given permutation π to the given sequence Γ, we receive the resulting sequence $\pi(\Gamma) = (\frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\clubsuit}, \frac{?}{\heartsuit})$.

Definition 6 (shuffle (shuffle, Π, $\mathcal{F}$))**.** *A shuffle (shuffle, Π, $\mathcal{F}$) has a permutation set $\Pi \subseteq S_n$ and a probability distribution $\mathcal{F}$ on Π. The shuffle draws a random permutation $\pi \in \Pi$ according to the probability distribution $\mathcal{F}$. Then it applies the chosen permutation to the sequence Γ.*

The definition of security will rely on the assumption that nothing about Γ is learned during the execution of this action which cannot be learned from Π and/or the visible part of the sequence. Shuffles can further have different properties such as closedness and uniformness.

Definition 7 (Closedness)**.** *A shuffle (shuffle, Π, $\mathcal{F}$) is closed if it is a subgroup of the symmetric group (Saito et al., 2020).*

That means that the set of possible permutations in a closed shuffle is invariant under repetition (Koch, 2019). We call a protocol *closed* if it does not contain shuffles, that are not closed.

Definition 8 (Uniformness)**.** *A shuffle (shuffle, Π, $\mathcal{F}$) is uniform if the distribution probability distribution $\mathcal{F}$ is uniform, meaning any $\pi \in \Pi$ is drawn with equal probability. (Saito et al., 2020).*

That means, that within a uniform shuffle action, every possible permutation has the same probability (Koch, 2019). A uniform shuffle can be written only as (shuffle, Π). Take as an example the sequence $\Gamma = (\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit})$ and the shuffle (shuffle, id, (123))) with an uniform probability distribution $\mathcal{F}$. The possible resulting sequences are either $(\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit})$, if the randomly chosen permutation π was id, or $(\frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit})$ if the randomly chosen permutation π was (123). The given shuffle (shuffle, id, (123))) is not closed. We can prove this, by showing that it is not invariant under repetition. If we apply the permutation π = (123) to the sequence $\Gamma' = (\frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit})$, we receive the new sequence $\Gamma'' = (\frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\clubsuit})$. This sequence Γ'' is not an element of the set of possible sequences that result from applying (shuffle, id, (123))) to the sequence $\Gamma = (\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit})$. Therefore the set of possible permutations in the shuffle is not invariant under repetition and thus the shuffle is not uniform.

Definition 9 (turn (turn, $\mathcal{T}$)). *A turn (turn, $\mathcal{T}$) with the turn set $\mathcal{T} \subseteq 1, ..., n$, flips the cards at the positions specified by $\mathcal{T}$.*

With $\frac{a}{b}$ being the card where a is facing up and b is facing down, after flipping the card we have $\frac{b}{a}$. As an example, executing a turn operation (turn, 1) on the sequence $\Gamma = (\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit})$ would result in the sequence $\Gamma = (\frac{\clubsuit}{?}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit})$. In our protocols after having turned the cards in the turn sets, and thus learned their symbol, we immediately turn them back around. This will not be explicitly written as its own turn operation.

Definition 10 (result (result, $p_1, p_2, ..., p_r$)). *The result (result, $p_1, p_2, ..., p_r$) is the output of a protocol. The positions $p_1, p_2, ..., p_r \in \{1, ..., l\}$ terminates the protocol and specifies that the output is $O = (\Gamma[p_1], \Gamma[p_2], ..., \Gamma[p_r])$.*

The cards at the positions p_1 to p_r have to be facing down when this action is performed. For our purpose the positions $p_1, p_2, ..., p_r \in \{1, ..., l\}$ also have to be one or more valid commitments. Thus for the sequence $\Gamma = (\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit})$, (result, 1,2) would be a valid result $O = (\Gamma[1], \Gamma[2]) = (\clubsuit, \heartsuit)$, while (result, 2,3) would be an invalid result $O' = (\Gamma[2], \Gamma[3]) = (\heartsuit, \heartsuit)$. For our protocols we adapt the definition by Koch et al. (2021) for the case that we have more than one output commitment. In this separate the output commitments with brackets to emphasize which cards form a commitment. The result (result, (1,2)(4,3)) would be a valid result $O'' = (\clubsuit, \heartsuit, \clubsuit, \heartsuit)$, where the first two cards are the cards that belong to commitment one and the other rwo cards encode commitment two.

2.2.1 Runtime of Card-Based Protocols

The runtime of card-based protocols refers to the amount of steps that have to be performed until a solution is reached. A *step* can be either a permutation, a turn, or a shuffle. We do not count the result action as a step. The protocol by Mizuki and Sone (2009) given in section 1.1, that executes first two perms and a shuffle and then a turn therefore has four steps. With card-based protocols we generally distinguish two types of runtime. We either have a fixed number of steps that it takes for the protocol to compute the result, or we do not have a fixed number but only an expected number of steps until the protocol produces a result. The former is called a finite runtime protocol and the latter a Las Vegas protocol.

Definition 11 (Finite Runtime). *Finite runtime protocols have a fixed bound on the number of steps (Koch et al., 2015).*

Therefore the runtime is bounded and can be precisely predicted (Koch, 2019). A KWH-Tree (section 2.2.3) of a finite runtime protocol is finite (Koch et al., 2021). The *AND* protocol from section 1.1 is a finite runtime protocol. It produces a result after four steps for every possible input.

Definition 12 (Las Vegas). *For Las Vegas protocols the number of steps until termination is only expectedly finite (Koch, 2019).*

Within a KWH-Tree a Las Vegas protocol makes use of restart actions, and/or its states form a cyclic diagram. (Koch et al., 2015). We distinguish *Restart-Free Las Vegas protocols* and *restarting Las Vegas protocols* (Koch et al., 2015). For Restart-Free Las Vegas there has to be a constant probability to exit for each cycle. We also cannot end up in a state where we have to restart the whole protocol from the beginning. A restart would mean, that both players have to provide their commitment again (Koch, 2019).

2.2.2 Correctness and Security of Card-Based Protocols

The definitions of correctness and security of card-based protocols are the ones Koch et al. (2021) use. A protocol is correct if it calculates the correct result from the input commitments for all possible correct inputs. More formally:

Definition 13 (Correctness). *A protocol P with input I computing a function f is correct, if the probability of the output O being $O = f(I)$ is 1.*

Generally a protocol is considered secure, if during the protocol execution, the visible cards do not reveal anything about the input and if the output of the protocol reveals nothing about the input apart from what can be derived from the result. Koch et al. (2021) provide a definition of security which we will call probabilsitic security, to distinguish it from other security definitions, that we will give below.

Definition 14 (Probabilistic Security). *A given protocol has probabilistic security, if for every turn operation the probability X_v for each observation v is a constant p between 0 and 1. For all observations of a turn operation $\sum_{v\in[0,1]^2} X_v = 1$ holds. Additionally, every probability X_w of each output basis w has to be a constant between 0 and 1 as well. For all observations of a output basis $\sum_{w\in[0,1]^2} X_w = 1$ holds.*

For the formalization with bounded model checkers Koch et al. (2021) defined two other types of security: input-possibilistic security and output-possibilistic security.

Definition 15 (Input-Possibilistic Security). *A protocol P is input-pssibilistically secure, if it is correct and for random input I (where $Pr[I = i] > 0$ for all $i \in \{0,1\}^2$) and any visible sequence trace v with $Pr[v] > 0$ as well as any input $i \in \{0,1\}^2$ we have $Pr[v|I = i] > 0$.*

Definition 16 (Output-Possibilistic Security). *A protocol P is input-pssibilistically secure, if it is correct and for random input I (where $Pr[I = i] > 0$ for all $i \in \{0,1\}^2$) and any visible sequence trace v with $Pr[v] > 0$ as well as any output $o \in \{0,1\}$ we have $Pr[v|f(I) = o] > 0$.*

Corrolary 1. *Every protocol that is input possibilistically secure protocol is also output possibilistically security.*

Proof. Assume that we have a input-possibilistic protocol. Then it is by definition correct. Also, every output is the result of at least one of the possible inputs. For a protocol with input possibilistic security, the probability for every input in every visible sequence trace is greater than 0. Therefore the probability for every output in every visible sequence trace is also greater than 0. □

Corrolary 2. *Not every protocol with output possibilistic security also possesses input-possibilistic security.*

Proof. A protocol calculating the boolean *XOR*, has two possible inputs that result in the output 0 and two possible inputs that result in the output 1. Therefore if we have a visible sequence trace, where the probability for each output is greater than 0, the probability for the one of the two inputs that have the same output could be 0 without violating the requirement that the possibility for all outputs has to be grater 0. □

This means the stronger security guarantee is input-possibilistic security and the weaker security guarantee is output-possibilistic security.

Remark 1. *If we want to prove the nonexistence of a protocol, proving that no protocol exists with output-possibilistic security is the stronger statement. For the existence of a protocol proving that a protocol has input-possibilistic security is the stronger statement.*

2.2.3 KWH-Trees

The descriptions of KWH-trees are taken from Koch et al. (2015) where KWH-trees are first introduced, as well as from Koch et al. (2021). KWH-Trees describe all possible executions of a protocol in the form of a tree. The nodes represent the sequences that are possible at a specific step in the protocol. The vertices have he action that the protocol prescribes in that situation associated to it. The tree branches when the visible sequence differs i.e. when a turn operation is performed.

To start, let us look at a single node which we will call a state. Our example is the start state of the six card *AND* protocol by Mizuki and Sone (2009). A state contains all sequences that are possible at a specific point in the protocol. Instead of depicting a sequence as $\Gamma = (\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit})$ we only depict the values of the cards as $\clubsuit\heartsuit\clubsuit\heartsuit\clubsuit\heartsuit$.

♣♡♣♡♣♡	X_{00}
♣♡♡♣♣♡	X_{01}
♡♣♣♡♣♡	X_{10}
♡♣♡♣♣♡	X_{11}

Apart from the sequence itself, we also track its symbolic probability. For input-possibilistic security, the symbolic probability describes that the sequence has the probability of X_{ij} that (i, j) is the input that produced this sequence. For output-possibilistic security, the symbolic probability describes that the sequence has the probability of X_i that (i) is the output that will be reached when executing the remaining protocol on the sequence. In our example the symbolic probabilities are X_{00}, X_{01}, X_{10} and X_{11}. Therefore the first line of our example (♣♡♣♡♣♡ X_{00}) can be interpreted as: the sequence $\Gamma = (\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit})$ has the symbolic probability X_{00}. We have taken a more formal definition of states from Koch et al., 2021.

Definition 17 (State). *For a protocol $\mathcal{P}$ that has a deck $\mathcal{D}$ and computes a boolean function f, a state μ is a map $\mu : Seq^{\mathcal{D}} \rightarrow \mathbb{X}_2$ where $\mathbb{X}_2$ describes 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 $\beta_b \in [0, 1] \subsetneq \mathbb{R}$. $\mu(s)$ for $s \in Seq^{\mathcal{D}}$ is interpreted as the probability that s is the actual sequence on the table, in terms of the symbolic probabilities on the inputs.*

We can now interpret the nodes as states, that show all sequences with a probability greater than 0. We can also describe the action defined in section 2.2 in the context of KWH-Trees. As an illustration we use the six card *AND* protocol by Mizuki and Sone, 2009 shown in fig. 2.1.

For a permutation (perm, π) (definition 5) a sequence s with probability $Pr(s)$ the resulting sequence of the permutation with $p\pi$ $\pi(s)$ will be assigned the probability $Pr(s)$. For our example in fig. 2.1 we have a permutation as the first operation. If we pick out the first sequence ♣♡♣♡♣♡ with probability X_{00}, we can see that after applying the permutation π =(243) the resulting sequence ♣♣♡♡♣♡ then has the probability X_{00}.

A shuffle operation (shuffle, Π, $\mathcal{F}$) (definition 6) contains multiple permutations $\pi \in \Pi$. So a sequence s will be permuted by a $\pi \in \Pi$ with the probability designated by $\mathcal{F}$. For our example we have a shuffle with uniform probability (definition 8) and thus every permutation $\pi \in \Pi$ has the same probability. If we continue with the sequence ♣♣♡♡♣♡, either the permutation id or the permutation (14)(25)(36) was applied to it, each with probability $\frac{1}{2}$.

After a turn (turn, $\mathcal{T}$) (definition 9), the visible sequence differs and we thus have a branching of the tree. In our example the turn at $\mathcal{T} = \{1, 2\}$ can produce two possible visible sequences, either ♣♡???? or ♡♣????. The probability for observing either of the

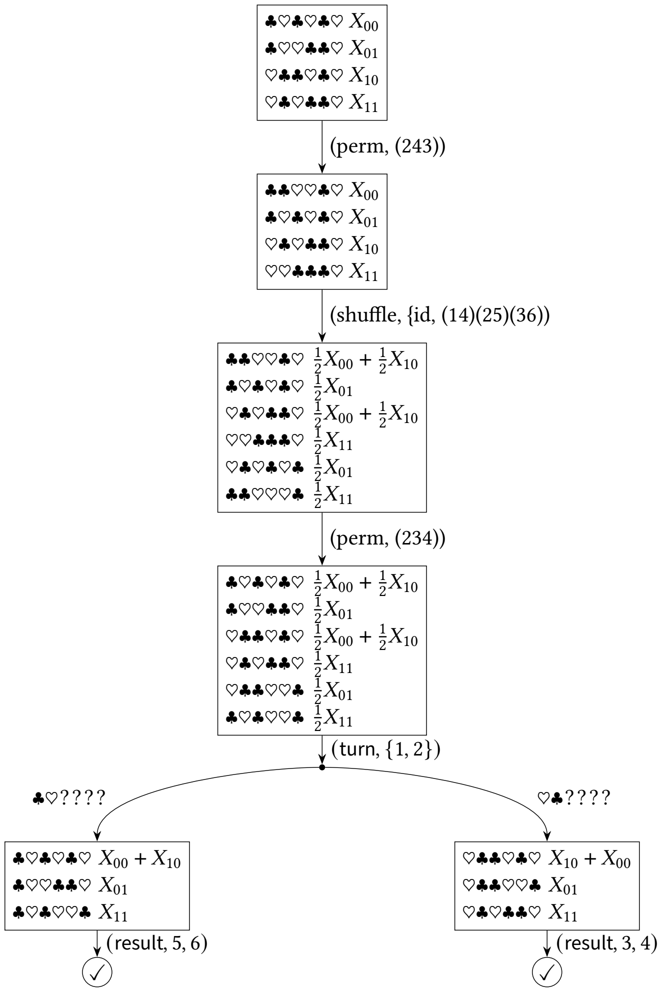

♣♡♣♡♣♡ X_{00}
♣♡♡♣♣♡ X_{01}
♡♣♣♡♣♡ X_{10}
♡♣♡♣♣♡ X_{11}
(perm, (243))
♣♣♡♡♣♡ X_{00}
♣♡♣♡♣♡ X_{01}
♡♣♡♣♣♡ X_{10}
♡♡♣♣♣♡ X_{11}
(shuffle, {id, (14)(25)(36))
♣♣♡♡♣♡ $\frac{1}{2}X_{00} + \frac{1}{2}X_{10}$
♣♡♣♡♣♡ $\frac{1}{2}X_{01}$
♡♣♡♣♣♡ $\frac{1}{2}X_{00} + \frac{1}{2}X_{10}$
♡♡♣♣♣♡ $\frac{1}{2}X_{11}$
♡♣♡♣♡♣ $\frac{1}{2}X_{01}$
♣♣♡♡♡♣ $\frac{1}{2}X_{11}$
(perm, (234))
♣♡♣♡♣♡ $\frac{1}{2}X_{00} + \frac{1}{2}X_{10}$
♣♡♡♣♣♡ $\frac{1}{2}X_{01}$
♡♣♣♡♣♡ $\frac{1}{2}X_{00} + \frac{1}{2}X_{10}$
♡♣♡♣♣♡ $\frac{1}{2}X_{11}$
♡♣♣♡♡♣ $\frac{1}{2}X_{01}$
♣♡♣♡♡♣ $\frac{1}{2}X_{11}$
(turn, {1, 2})
♣♡????
♡♣????
♣♡♣♡♣♡ $X_{00} + X_{10}$
♣♡♡♣♣♡ X_{01}
♣♡♣♡♡♣ X_{11}
♡♣♣♡♣♡ $X_{10} + X_{00}$
♡♣♣♡♡♣ X_{01}
♡♣♡♣♣♡ X_{11}
(result, 5, 6)
(result, 3, 4)
✓
✓

Fig. 2.1: Six card *AND* protocol by Mizuki and Sone, 2009.

sequences is the same for each possible input, so no information about the rest of the sequence is leaked.

The output of the protocol is given as (result, $p_1, p_2, ...p_n$ (definition 10). Within a KWH-Tree we can easily observe if the given $p_1, p_2, ...p_n$ encode a correct result, because the state with all its possible sequences is drawn. Thus we can observe in our example that for (result, 5,6) the sequences ♣♡♣♡♣♡ and ♣♡♡♣♣♡ correctly encode a 0 and the sequence ♣♡♣♡♡♣ correctly encodes a 1.

2.2.4 Card-Based Protocols for Multi-Party Computation

Card-based cryptographic protocols can perform secure multi-party computation. in general multi-party computation allows multiple parties, that each hold their private data, to calculate a function without revealing their private data to the others.

Definition 18 (Multi-Party Computation (MPC)). *Stiglic (2001) defines multi-party computation as follows: We assume a group of n players $C_1, C_2, \dots, C_n$ with inputs $c_1, c_2, \dots, c_n$, where the input c_i is the private input of player C_i. These players want to correctly and securely compute $f(c_1, c_2, \dots, c_n)$, where f is a public function. Securely means, that each player C_i learns nothing about the input of the other players, than what he can deduce from his own input c_i and the result of the function $f(c_1, c_2, \dots, c_n)$.*

Our definition of multi-party computation relies on the assumption, that the players are "honest but curious" (Rastogi et al., 2019) That means, that they do follow the protocol and do not try to manipulate it (e.g. by turning cards over that are not supposed to be turned over), but that they want to gain as much knowledge as they can about the other players input by observing the protocol.

For our application of card-based protocols, we focus on Boolean functions. A boolean function is a function whose arguments and results are one or more values out of the set 0,1. We adapt the definition by Koch et al. (2021) to include boolean functions of arbitrary input and output length.

Definition 19 (Compute a Boolean Function). *A protocol $\mathcal{P}$ computes a boolean function $f : [0,1]^i \longrightarrow [0,1]^j$ where $i, j \in \mathbb{N}$ if the following holds:*

- *The possible start sequences corresponding to the players inputs $b \in [0,1]^i$ do encode the inputs as the correct commitments.*
- *The cards that contain the output commitment(s) after the termination of the protocols, encode the output value $o(b)$ for every possible input $b \in [0,1]^j$.*

There are several explanations in the existing literature on how to construct a protocol for any boolean function from given boolean operators. Niemi and Renvall (1998) utilize a *AND*, *OR* and *NOT* operator as well as a *COPY* protocol. Nishida et al. (2015) also present an approach that uses *XOR*, *AND* and *COPY* protocols.

Let us take a closer look at the approach by Nishida et al. (2015). They utilize the four card *XOR* protocol shown in fig. A.2 as well as the *COPY* protocol described in this section. They also employ a variant of the six card *AND* protocol shown in fig. 2.1, they produces a commitment to $c_1 \wedge c_2$ and a commitment to c_2 (Nishida et al., 2015). Using those protocols, they give a protocol for the computation of any boolean function that uses $2n + 6$ cards to compute a function with n input commitments. Their protocol requires the function to be an *AND-XOR* expression. In general this can be achieved by expressing the desired function as a Shannon expansion (Sasao, 1999, Chapter 3):

$$\begin{aligned} f(x_1, x_2, \dots, x_n) &= \overline{x_1 x_2} \dots \overline{x_n} f(0,0,\dots,0) \oplus x_1 \overline{x_2} \dots \overline{x_n} f(1,0,\dots,0) \\ &\oplus \overline{x_1} x_2 \dots \overline{x_n} f(0,1,\dots,0) \oplus \dots \oplus x_1 x_2 \dots x_n f(1,1,\dots,1) \\ &= T_1 \oplus T_2 \oplus \dots \oplus T_n \end{aligned} \tag{2.2}$$

Once we have a function that is an exclusive disjunction of conjunctions, we can use the *AND*, *COPY* protocols to calculate the result of a $T_i = v_1 \wedge v_2 \wedge \dots$. For that we copy v_1 using the *COPY* protocol and then apply the *AND* protocol to v_1 and v_2. We receive $v_1 \wedge v_2$ as well as v_2. We now repeat these steps, copying $v_1 \wedge \cdots \wedge v_{i-1}$ and calculating $(v_1 \wedge \cdots \wedge v_{i-1}) \wedge v_i$ with the *AND* protocol. Once we have calculated the result of T_i, we can calculate the result of $(T_1 \oplus T_2 \oplus \cdots \oplus T_{i-1}) \oplus T_i$ with our *XOR* protocol.

Throughout this work we make use of several protocols from the existing literature.

Protocol	#Cards	#Steps	Shuffle Type	Runtime	
AND by Mizuki and Sone (2009)	6	4	uniform closed	Finite Runtime	fig. 2.1
AND by Koch et al. (2021)	5	4	uniform closed	Las Vegas	fig. A.1
XOR by Mizuki and Sone (2009)	4	4	uniform closed	Finite Runtime	fig. A.2
COPY by Mizuki and Sone (2009)	$2k + 4$ (k copies)	4	uniform closed	Finite Runtime	listing 2.1

Table 2.1: Protocols from literature, that are used in this thesis. All protocols are probabilistically secure (definition 14).

For the Finite Runtime *COPY* protocol of Mizuki and Sone (2009) we consider the case, that produces 1 copy. Because we need $2k + 4$ cards to produce k copies, the protocol producing 1 copy needs $2 * 1 + 4 = 6$ cards. Listing 2.1 describes the actions of the *COPY* protocol for one copy and six cards. The input for this protocol is the one commitment that should be copied. The other four cards are arranged as ♣♡♣♡.

Listing 2.1 The protocol to compute *COPY* by Mizuki and Sone (2009).

```
(perm, (2453))
(shuffle, {id, (14)(25)(36)})
(perm, (2354))
v = (turn, {1,2})
if v == (♣,♡,?,?,?,?) then
    (result, {(3,4),(5,6)})
else if v == (♡,♣,?,?,?,?) then
    (result, {(4,3),(6,5)})
```

2.3 Software Bounded Model Checking

The explanations and descriptions of bounded model checking are taken from (Koch et al., 2021).

Software Bounded Model Checking (SBMC) is a fully-automatic static technique from formal program verification. It analyzes programs given in programming language such a C, C++ or Java. We will focus on C Bounded Model Checking (CBMC) (Kroening and Tautschnig, 2014), which is a bounded model checker for C programs.

The main purpose of SBMC is to find violations of assertions in programs, or to prove that the assertions hold for all inputs within a given bond. The given programs are not executed by SBMC, but statically analyzed without executing them on specific values.

This is done by transforming the program into a control flow graph. Then the control flow graph is unwound and a formula is built from it. The resulting formula can then solved by a SAT or SMT solver. If the SAT formula can be satisfied, then a program run, which violates an assertion, is found. In this case the bounded model checker returns the faulty run. In the case that the formula is unsatisfiable, meaning that no faulty run can be found, there are two possibilities. Either the assertions hold and the property is valid, or the property is invalid for runs that were not considered because they are out of bounds.

As an input, SBMC is given an imperatively defined function in the form of an imperative program. This imperative program has a set of possible start values $\mathcal{I}$. An entry $i \in \mathcal{I}$ contains a list of values, one for each parameter that a run of the imperative program can depent on. A parameter can be for example a input variable, or nondeterministic values. Nondetermintistic parameters have arbitrary but fixed value for a concrete evaluation of the imperative program.

Further we need a software property to be checked. The property has a form of $C^{ant} \Rightarrow C^{cons}$, with C^{ant} and C^{cons} being boolean statements. This means that for all possible entries $i \in \mathcal{I}$ that if i satisfies C^{ant} then i also has to satisfy C^{cons}. The property is only valid if there is no entry i that satisfies C^{ant} but does not satisfy C^{cons}. An example for a property could be $C^{ant} = v_1$ is odd and $C^{cons} = v_1$ is a prime number. Then $i = \{v_1 = 3\}$ satisfies C^{ant} and c^{cons} but $i = \{v_1 = 9\}$ does not.

2.4 Using Software Bounded Model Checking to Find Card Protocols

Koch et al. (2021) apply the method of software bounded model checking described in section 2.3 to the task of finding card-based protocols. They employ CBMC a bounded model checker for programs written in C. Their program and the description here focus on finding protocols for the *AND* gate.

Koch et al. (2021) implement a representation of the components of KWH trees (see Sec. 2.2.3) within a standardized C program. The specific properties, such as the number of cards used or the level of security can be specified by setting constants within the C program. Operations within the protocol such as what operation will be performed or which cards encode the output are implemented as nondeterministic variables. As described in section 2.3, CBMC symbolically executes programs to find violations of the provided assertions, or to prove that the assertions hold for all inputs within a given bond. The assertion given by Koch et al. (2021) is a simple assert(0) at the end of the program. Thus if the assertion is reached and therefore violated, there exists a input and an assignment of the non-deterministic variables so that there is a run through the standardized program and with that a protocol. The error trace that CBMC returns is the run through the standardized program that produced the violation. This is the protocol we are looking for. If the assertion is not violated, that means that it was not reached and thus there is no run through the standardized program. This means that there exists no protocol.

The standardized C program contains implementations of the neccesary actions that a protocol can perform as well as fuctions that ensure the security and correctness of the program. The program can be structured in three main parts. First all possible input states as well as the start sequence are created. Afterwards, the shuffles and turns of the protocols are performed. Here there are checks, that ensure that the chosen shuffle and turns are allowed and that the security of the protocol is not violated. Finally there are functions that check if the previously executed shuffles and turns resulted in a valid output.

Certain properties of the program can be adjusted by setting constants within the standardized program before executing CBMC. The most important constants are the number of cards used as well as the amount of steps the protocol should have. Furthermore, the program by Koch et al. (2021) supports two security types: input-possibilistic security (definition 15) and output possibilistic security (definition 16). The bounded model checker comes to a result faster if it searches for a protocol with output-possibilistic security and not input possibilistic security (see table 3.2). As described in section 2.2.2, this is the stronger condition when it comes to the non-existence of protocols. If we find a protocol however, we would want it to be input possibilistic as well, because output-possibilistically secure protocols can in certain cases leak information about the input (see fig. 3.2). By default, the bounded model checker considers permutation sets of any size. The number of possible permutations of a sequence of n cards is $n!$. To calculate the number of possible permutation sets of n, we determine the size of the power set of the set of possible permutations. The power set is the set of of all subsets of a given set S. As the power set also includes the empty set, which we do not want to pick as a permutation set, we calculate the number of possible permutation sets of a sequence of n cards as $2^{n!} - 1$. As the number of possible permutation sets grows exponentially with increasing card numbers, we might want to reduce the complexity and thus keep the running time of the program lower by reducing the maximum size of the permutation set. Therefore a value for the permutation set size can be provided, and the bounded model checker will only search for protocols with a permutation set that is as big or smaller than the limit provided. This reduction can however, reduce the strength of the results, if there is no protocol found, because there could be a protocol for a shuffle set that is bigger than the given maximum shuffle set size.

The user can also determine whether he only wants to search for protocols with closed shuffles (definition 7) or not.

In the case that a program for the specified input exists, CBMC will find a run through the program that violates C^{cons} = assert(0). That means that the input and the non-deterministic parameters are chosen in a way that they form a valid protocol. The trace of the run therefore contains all the necessary information to recreate the protocol. All we have to do now is obtain the instantiations of the non-deterministic parameters from the trace. These are for example whether a specific step was a shuffle or turn operation, which shuffles were chosen for the shuffle set and which cards encode the output commitment. To transfer the protocol into a step-by step explanation in the form of a KWH-tree for example, we will also have to complete the missing branches of the protocol. This is because we only look at one possible post state after every possible turn. Therefore we will only receive one run from start state to result state through the protocol. We can allow ourselves to only look at one possible post state, because if we have one valid run from start state to result state, that is chosen with a probability greater than zero, we can always restart our protocol for the post states that were not looked at further. With this approach, we can complete every protocol found by the bounded model checker and receive a protocol that is a restarting Las Vegas protocol. However, for most of the protocols found by the bounded model checker, we can complete the protocol manually by exploiting similarities between the actions and states found by the bounded model checker and the ones we have to complete manually. With this approach we can construct protocols, that are Restart-Free Las Vegas or even Finite Runtime protocols.

Implementation of the representation of KWH trees

In order for us to implement a representation of the components of KWH trees (see Sec. 2.2.3) within a standardized C program, we need to have a data structure for the representation of the states. We implement states as a single array containing all possible sequences (see listing 2.2. Unlike in the states in KWH-trees (see section 2.2.3) where only the sequences with a probability greater than zero are depicted, the states in this program representation hold all possible sequences regardless of probability.

Listing 2.2 Representation of a state by Koch et al. (2021)

```
    /**
 * All sequences are remembered here, as seen from left to right, sorted alphabetically.
 * The states in this program are equal to the states in KWH trees.
 */
struct state {
    struct sequence seq[NUMBER_POSSIBLE_SEQUENCES];
};
```

A sequence itself contains an array of unsigned integers which holds the cards. For the case of using only two colours, we represent ♣ as the integer value 1 and the ♡ as the integer value 2. Thus the card sequence ♣♡♡♣ would be represented by the array `[1,2,2,1]`. The sequence also contains a struct that holds the probabilities for the sequence.

Listing 2.3 Representation of a sequence by Koch et al. (2021)

```
struct sequence {
    unsigned int val[N];
    struct fractions probs;
};
```

The array that holds the different probabilities has a different number of entries depending on the chosen probability. For input-possibilistic security (definition 15) it is the number of possible inputs i.e. 4, and for output-possibilistic security (definition 16) it is the number of possible outputs i.e. 2.

Listing 2.4 Representation of a fraction by Koch et al. (2021)

```
struct fraction {
    unsigned int num; // The numerator.
    unsigned int den; // The denominator.
};

struct fractions {
    struct fraction frac[NUMBER_PROBABILITIES];
};
```

After the input is generated, the program performs a loop with the amount of iterations specified at the beginning as the amount of steps. Within each loop, the program nondeterministically chooses whether to perform a turn or a shuffle operation. Starting from the input states it performs the chosen turns and shuffles and computes the resulting intermediate states until the loop ends.

If we choose a turn, we then nondeterministically choose an index (`turnPosition`) at which we turn the card. After that, the calculation of the resulting states consists of imperative operations that compute the new resulting states and update the probabilities. The new resulting states are computed as follows. We create a new state for each possible card colour. For every sequence that is still possible, we determine the card colour at the index we chose. This is done by reading the value of the array at the specified `turnPosition` (see listing 2.5).

Listing 2.5 Turning one card at index "turnPosition" in a sequence during the execution of a turn operation (Koch et al., 2021)

```
unsigned int turnedCardNumber = seq.val[turnPosition];
```

We then copy the probabilities of the sequence into the new state for the determined card colour. After we have done so for all sequences, we check whether our security definitions (see Sec. 2.2.2) still hold. For input- and output-possibilistic security, we need to find one sequence for every possible in-/output. After every turn operation, there are two possible states that can be reached from there. The state that occurs, if a ♣ has been turned and the state that occurs if ♡ has been turned. We calculate both of the possible post states and check if they are valid, but we only look at one of them for our following operations. The post state we look at is chosen nondeterministically.

For a shuffle operations we first nondeterministically determine the size of our shuffle set. If the size of the shuffle set is one i.e. we only choose one permutation from the set of possible permutations, we have a permutation (perm, π) instead of a shuffle (shuffle, Π, $\mathcal{F}$) (see Sec. 2.2). We also choose the permutations nondeterministically. Afterwards we can check for optional shuffle properties e.g. if the chosen permutations make a closed shuffle. Then we apply the shuffle we generated to our current state. We do this by iterating over every permutation `j` from the shuffle set and every sequence `i` in the state and applying the permutatuion j `j` to the sequence `i` Both the permutation `j` and the sequence `i` are arrays. For each card `k` in the sequence we read the value of the sequence at that array index `k`. Then we determine the index to write this value to, by determining the value of the permutation at that array index `k`- The predetermined value of the sequence, is then written to a new array `resultingSeq` at the index taken from the permutation array.

Listing 2.6 Applying permutation j to sequence i during the execution of a shuffle operation (Koch et al., 2021)

```
for (unsigned int k = 0; k < N; k++) {
    resultingSeq.arr[permutationSet[j][k]] = s.seq[i].val[k];
}
```

The result of the shuffle is the state after the shuffle operation. We now check for each sequence in the state, whether it is a bottom sequence.

Definition 20 (Bottom Sequence). *A sequence is a bottom sequence if it belongs to more than one possible output.*

If a shuffle results in one sequence being a bottom sequence, the protocol can no longer produce the correct result. That is because this sequence belongs to more than one possible output. Thus however the result will be defined, it will be wrong for one of this multiple possible outputs.

3 A Standard Program Representation for Finding Card-based Protocols for Any Boolean Function

In section 2.4 we have seen a general description of the implementation of a standardized program representation by Koch et al. (2021). Their representation was designed to find protocols for *AND*. Our aim is, to use bounded model checking (section 2.3) and the implementation by Koch et al. (2021) (section 2.4) to find protocols for any type of function. In section 3.1 we give a description of how to implement a standardized program representation, so that it is able to find a protocol for an arbitrary function. Using these principles, we implement a standardized program representation for a select number of functions in section 3.2 and perform a test on the efficiency of different SAT solvers in section 3.2.2. We present the protocols found using this standardized program representation in section 3.2.3 and discuss the size constraints for using the bounded model checker with our implementations in section 3.2.4. .

3.1 Adjustments to the Standard Program Representation for Boolean Functions

In general, a mathematical function assigns to each element from a set X (domain), an element from a set Y (co-domain). In these sections we will now take a closer look at how changes in the domain, co-domain and the function behaviour are represented within our program to ensure that we obtain a protocol for the desired function. First we will look at the function behaviour (definition 19). This means, which input maps to which output. Then we will consider the amount of input commitments for our functions i.e. the domain. Finally we will take a look at the amount of output commitments for our functions i.e. the co-domain.

3.1.1 Function Behaviour

The function behaviour determines the output value for each input value. As an example for this section, let us look at binary boolean operators. They take the same amount of input commitments (namely two) and have the same amount of output commitments (namely one). The program by Koch et al. (2021) is designed to find protocols for *AND*. However, there are many other binary boolean operators like *XOR* and *OR*. They differentiate themselves from *AND* not in their possible inputs and outputs, but in the way inputs are mapped to outputs. As seen in table 3.1, The input Com1 = 1 and Com2 = 0 would result in the output 1 for *OR* but 0 for *AND*.

Com1	Com2	*AND*	*OR*	*XOR*
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 3.1: Truth assignments for *AND*, *OR* and *XOR*

Our task now is to determine how the function behaviour is represented and encoded in the program. The different function behaviours express themselves within the link between input and output. Therefore, the function behaviour influences how the possibilities for output possibilistic security (definition 16) are calculated. For input possibilistic security (definition 15) and probabilistic security (definition 14), the examination if a sequence is a bottom sequence and the calculation of the output is impacted, because they both depend on the output.

Let us start with the calculation of the possibilities. For output possibilistic security, we determine in the beginning for all possible input sequences which sequence will be mapped onto the output 0 and which will be mapped onto the output 1. The mapping is recorded in the possibilities: X_0 if the output is 0 and X_1 if the output is 1. Within the code, the possibilities are held within the struct `fractions` where `frac[0]` contains the fraction for X_0 and `frac[1]`contains the fraction for X_1 (see listing 3.1). For *AND* the input sequence 11 (♡♣♡♣) has the output 1 and therefore has a possibility of $X_0 = 0$ and $X_1 = 1$. The other input sequences 00, 01 and 10 (♣♡♣♡, ♣♡♡♣ and ♡♣♣♡) have the output 0 and therefore has a possibility of $X_0 = 1$ and $X_1 = 0$. For *OR* on the other hand, for the first input sequence where both commitments are 0 ((♣♡♣♡), X_0 is set to 1 and X_1 is set to 0. For all other possible input sequences, it is set the other way round. For input possibilistic security and probabilistic security we instead calculate the possibility from the input not the output. Thus, it is not different for functions that have the same amount of input commitments.

Listing 3.1 The structs for sequences fractions.

```
struct fraction {
    unsigned int num; // The numerator.
    unsigned int den; // The denominator.
};

struct fractions {
    // NUMBER_PROBABILITIES = 2 for output possibilistic security,
    // and  = 4 for the other types of security.
    struct fraction frac[NUMBER_PROBABILITIES];
};

struct sequence {
    unsigned int val[N]; // N is the number of cards in a sequence
    struct fractions probs;
};
```

Now let us take a look at the adaptations for input possibilistic and probabilistic security, starting with the check for bottom sequences. As described in section 2.4 we check for bottom sequences (definition 20) every time we perform a shuffle. We also check it while testing if a state is a final state. We do that, because if we have a bottom sequence, our program is faulty. A sequence is a bottom sequence if it belongs to more than one possible input and we cannot give back a correct result for the sequence if it belongs to more than one output. To illustrate, a sequence would be a bottom sequence for *AND* if the sequence had input possibilities of $X_{11} \neq 0$ and $X_{01} \neq 0$. The same sequence would not be a bottom sequence for *OR* if the sequence had input possibilities of $X_{11} \neq 0$ and $X_{01} \neq 0$. This is because these input commitments both evaluate to 1. For output possibilistic security, the check for bottom sequences remains the same throughout all different function behaviours (see listing 3.2). For input possibilistic and probabilistic security on the other hand, we have to exclude that probabilities that represent inputs which map onto different outputs are present in the same sequence (see listing 3.3).

Listing 3.2 Extract from isBottom() for WEAK_SECURITY == 2 (output possibilistic security)

```
bottom = probs.frac[0].num && probs.frac[1].num;
```

Listing 3.3 Extract from isBottom() for WEAK_SECURITY != 2 (input possibilistic and probabilistic security) for *AND*(top), *OR*(middle) and *XOR*(bottom)

```
bottom = (probs.frac[0].num || probs.frac[1].num || probs.frac[2].
    num)
        && probs.frac[3].num;

bottom = (probs.frac[1].num || probs.frac[2].num || probs.frac[3].
    num)
        && probs.frac[0].num;

bottom = (probs.frac[0].num || probs.frac[3].num)
        && (probs.frac[1].num || probs.frac[2].num);
```

Lastly, we have to take a look at the function `isFinalStates` which checks whether a state is a final state. The function nondeterministically picks two parameters that are the column numbers, and then checks for each sequence that the values encode a 1 if the sequence is evaluating to 1 and a 0 if the sequence is evaluating to 0. The parameter `deciding` holds the output. For output possibilistic security, we can infer the value of `deciding` from the possibilities directly. Therefore, a sequence is deciding if $X_1 > 0$. That means, that value of `deciding` can be determined as seen for *XOR* in listing 3.4. For input possibilistic and probabilistic security the X_{ij} have to be specified that evaluate to one. In the case of *XOR* in listing 3.4 it is X_{01} and X_{10}.

Listing 3.4 Deciding for *XOR*

```
if (WEAK_SECURITY == 2) {
    deciding = (s.seq[i].probs.frac[1].num); //for output possibilistic security X_1 is deciding, X_0 is not
}
else {
    deciding = (s.seq[i].probs.frac[1].num)  //for the other security types  X_01 and X_10 are deciding for XOR
    || (s.seq[i].probs.frac[2].num);
}
```

3.1.2 Domain

For our example above, we have considered only binary boolean operators with two input commitments and one output commitment. But there are functions, that take a different amount of input commitments. One prominent example for this is the *COPY* function, that takes one commitment (definition 3) as an input, and duplicates it (listing 2.1). Another would be a three input majority function as described by Nishida et al., 2013, which takes three input commitments. Let us take a look at how a different amount of input commitments influences our protocol. In general, we have to make sure that we generate the start sequences correctly and have the right amount of probabilities for input possibilistic and probabilistic security.

To generate the start sequences, we make use of constants (aka. preprocessor variables). Two of them depend on the amount of input commitments a function has. The number of cards used for a commitment `COMMIT` has to match the number of cards that will be needed to encode all input commitments. In general, we need two cards for every input commitment. So the value of `COMMIT` is two times the number of commits. For the binary boolean operators we use 4 cards for their two input commitments. If we wanted to instead find protocols *COPY*, we would have to set `COMMIT` to 2, because we have only one input commitment. The number of start sequences `NUMBER_START_SEQS` usually depends on the desired protocol as well. For k players where both can either commit a 0 or a 1, we have 2^k start sequences. If we have other scenarios like three different commit options (e.g. -1, 0, 1) we can have a different number of start sequences and a different number of cards for commitments. The minimum size of the input sequence `N` depends on the amount of input and output cards. `N` has to be big enough so that it can hold the input commitments and later also the output commitments. For a three input majority function, we would need 3 commitments and thus at least 6 cards. So for the three input majority function `N` would have to be greater or equal 6. For *COPY* we would only need one commitment and two cards for our input. We do however need two commitments and four cards to encode the output. Therefore, we need `N` ≥ 4 for *COPY*.

The start sequences themselves are generated at the beginning of the program. The function `getStartSequence()` nondeterministically constructs a start sequence for the given commitment length COMMIT. To obtain a valid sequence we need to make sure that each player uses fully distinguishable cards. We therefore have to check for each input commitment separately, if the two cards are the same or not. Afterwards, we check, that all the possible inputs are represented within the start sequences. For a boolean operator this would be the input commitments 00, 01, 10 and 11, and thus we have to check for those four different sequences. For the *COPY* protocols we only have the input commitments 0 and 1. We check for a 0 by testing if the first card is smaller than the second, and we check for a 1 by testing if the first card is bigger than the second. If we have more than one input commitment, the first two cards are the first commitment, the third and fourth card the second commitment and so on. And we check for all the possible combinations of values for these commitments.

We also assign the input probabilities and possibilities to the start state, by checking if the given sequence is an input sequence in the start state. This works just as the check for all the possible inputs.

3.1.3 Codomain

Not only the amount of input commitments, but also the amount of output commitments can vary from function to function. Some problems can have more than a single bit as an output. Take for example the half adder (definition 21). It receives two bits as an input - just like the binary boolean operators. But instead of one output commitment it produces two. The sum of the two bits and the carry signal, which represents the overflow into the next bit. The amount of output commitments influences the amount of output possibilities as well as how we determine a final state.

Let us look at the output possibilities first. If all values are possible for the k output commitments, we have 2^k different output possibilities. For example, we have the 2 different output possibilities X_0 and X_1 for a binary boolean operator. For the half adder we would thus have 4 different output possibilities X_{00}, X_{01}, X_{10} and X_{11}. However it is not possible for any sum of two input commitments to be greater than 2. Thus the output possibility X_{11}, where sum and carry are both one is impossible. We are therefore left with only three output possibilities for the half adder. Another example, where not all combinations of values are possible for the output commitment is the *COPY* function. Here we have two output commitments. But because the function requires, that they are both the same, we have only two output possibilities X_0 and X_1.

We set the number of possibilities `NUMBER_PROBABILITIES` as a constant and then have to set the possibilities for output possibilistic security when we generate the start sequences. Here we have to match each input sequence to the possibility that belongs to its output sequence. For the half adder, the input commitment 00 has possibility X_{00}, the input commitments 01 and 10 have possibility X_{01} and the input 11 has possibility X_{10}.

Just as in section 3.1.1 with input possibilistic and probabilistic security, we now have to consider output possibilistic security, when checking for bottom sequences (definition 20). A bottom sequence, is a sequence that belongs to more than one possible output. Thus we have to check, that for each sequence there is only one output possibility greater than 0.

The last step when finding a protocol is checking that the result is valid. The function `isFinalState()` determines if a state contains columns containing the result bits. Therefore, we need to find two columns as the result for a binary boolean operator and 4 columns for the *COPY* protocol and the half adder protocol. We chose every index of our columns nondeterministically. Afterwards we also need to match the possibilities to the correct output. As explained in section 3.1.1, the parameter `deciding` encodes for a binary boolean operator whether the sequence evaluates to a 0 or 1. A general approach is to have one such deciding variable for each output commitment.

Listing 3.5 Extract from isFinalState for the half adder.

```
unsigned int decidingSum = 0;
if (WEAK_SECURITY == 2) {
    decidingSum = (s.seq[i].probs.frac[1].num);
} else {
    decidingSum = (s.seq[i].probs.frac[1].num) || (s.seq[i].probs.
        frac[2].num);
}
    // ....
unsigned int decidingCarry = s.seq[i].probs.frac[
    NUMBER_PROBABILITIES - 1].num;
```

In listing 3.5 we see the assignment of the deciding parameter for the two output commitments of the protocol. For the first parameter, the assignment of `decidingSum` is the same as for a logical *XOR*. For the second parameter, the assignment of `decidingCarry` is the same as for a logical *AND*. Because of the function behaviours of *AND* and *XOR*, we can not have them be both 1 for the same input. Thus the output 11 is impossible, just as stated above. After we have determined whether the sequence is deciding, we can check whether the cards at the previously chosen index encode the correct result for each deciding parameter and output commitment.

3.2 Implementing a Concrete Standardized Program Representation for a Select Number of Functions

This section takes the principles presented in section 3.1 and applies them to specific functions. We discuss the experimental setup in section 3.2.1. In section 3.2.2 we describe our attempt of improving the performance of our bounded model checker by using different SAT solvers. Afterwards we present the protocols we were able to find using the previously described setup in section 3.2.3. We also discuss the size constraints for using the bounded model checker with our implementations in section 3.2.4.

3.2.1 Structure and Execution of the Standardized Program Representation

To apply the principles presented in section 3.1, we implemented a standardized program representation of a range of different functions. The first set of functions we implemented, were binary boolean operators. As Koch et al. (2021) had already done extensive runs of their standard program representation for the *AND* protocol, we did not repeat those tests. Instead we implemented a standardized program representation for the *XOR* and *OR* operators. We also implemented a *COPY* function. It receives only one commitment as an input, and returns a copy of that commitment. Together with a functionally complete set of boolean operators, for example *AND* and *XOR*, we can realize any multivariable function (Nishida et al., 2015). To add a more practical function, we implemented a standardized program representation for a half adder as well. Excerpts from the implementation for all the symbolic programs can be found in appendix A.5. The complete code can be found in the repository at appendix A.4.

All the experiments were performed on an AMD Opteron(tm) 6172 CPU at 2.10 GHz with 48 cores and 256 GB of RAM. We used CBMC 5.68 with the built-in solver based on the SAT-solver MiniSat 2.2.1.

3.2.2 Exploring different SAT solvers

Using a standardized program representation has two distinct disadvantages. Firstly, the process of finding a protocol can take a long time, especially for a protocol that has a lot of cards and/or a lot of steps. As can be observed in table 3.2, for the built-in SAT solver MiniSat, finding a *XOR* protocol with four cards, two steps and input-possibilistic security took around 10 minutes. Proving that there is no *OR* protocol with four cards, two steps and input-possibilistic security it took 20 minutes. But for even bigger protocols the time until a protocol is found is likely to be even bigger. As an example, finding the protocol from fig. 3.5 took more than 23 hours. Another problem is the occurrence of an "out-of-memory" error for certain protocols with more cards and steps.

We attempted to improve the time it takes to find a protocol as well as the avoidance of "out of memory"-problems through the use of different SAT solvers within the bounded model checker. We chose Glucose [1] and CaDiCal [2] to test against the built-in SAT Solver MiniSat.

All the experiments were performed on an AMD Opteron(tm) 6172 CPU at 2.10 GHz with 48 cores and 256 GB of RAM. We used CBMC 5.57.0 for all the SAT Solvers. Every test was performed five times for each of the SAT solvers. The number of variables and clauses stayed the same throughout all five executions of the tests. Therefore the values given in table 3.2 are the absolute values in each test. For the time spans we measured, we each excluded the highest and the lowest result. From the remaining three values we calculated the arithmetic mean. This is also shown in table 3.2 accordingly. We determined the time span by measuring the time from the start of the execution of the test until the execution terminated. For the four card *XOR* protocol the program terminated after a valid protocol was found. Thus the times given in table 5.1 were measured from the start of the execution until a valid protocol was found. For the four card *OR* Protocol, the program terminated after it had found, that no valid protocol exists for the given inputs. For this protocol the times given in table 5.1 are the times from the start of the execution until it was proven that there is no protocol. The five card *OR* protocol terminated after an out of memory error. Therefore it did not return the amount of clauses and variables. The times given in table 5.1 are measured from the start of the execution of the protocol until the out of memory error occurred and the program terminated.

The test results show clearly, that using CaDiCal or Glucose did not improve the runtime of our protocols, nor did it result in less "out-of-memory" errors. Out of all SAT solvers MiniSat performed the best for both the four card *XOR* as well as the four card *OR* tests that were performed. It had a shorter runtime for both input-possibilistic as well as output-possibilistic security. Glucose was second fastest for both the four card *XOR* as well as the four card *OR* tests. The smallest difference in runtime between MiniSat and Glucose can be observed for the four card *OR* protocol with input-possibilistic security. Here the test for Glucose took only about two minutes longer than the test for MiniSat. The biggest difference could be observed for the four card *XOR* protocol with input-possibilistic security. Glucose was around 2.5 times slower than MiniSat. For our use case the performance of CaDiCal was even worse. The smallest difference between MiniSat and CaDiCal could be observed for the four card *OR* protocol with input-possibilistic security. Here CaDiCal was still around 1.8 times slower than MiniSat.

The test results also show, that the use of CaDiCal or Glucose did not result in less out of memory errors, at least for the program that was tested. For the five card *OR*, the execution

[1] https://www.labri.fr/perso/lsimon/research/glucose/

[2] http://fmv.jku.at/cadical/

	#Cards	#Steps	Security		Mini Sat	Glucose	CaDiCal
XOR	4	2	output-possibilistic	Runtime (s)	367,3	569	725,7
				Variables	10 829 386	10 829 386	10 829 386
				Clauses	39 863 588	39 863 588	39 905 499
XOR	4	2	input-possibilistic	Runtime (s)	618	1 560,3	1 823,7
				Variables	13 191 171	13 191 171	13 191 171
				Clauses	47 693 620	47 693 620	47 736 299
OR	4	2	output-possibilistic	Runtime (s)	481	1 062	3 470,7
				Variables	10 819 303	10 819 303	10 819 303
				Clauses	39 832 637	39 832 637	39 874 504
OR	4	2	input-possibilistic	Runtime (s)	1 218,3	1 335	2 158,7
				Variables	13 185 883	13 185 883	13 185 883
				Clauses	47 673 732	47 673 732	47 716 387
OR	5	2	output-possibilistic	error after (s)	3 830,7	1 637,7	2 761,3

Table 3.2: Experiment results for the SAT-Solvers MiniSat, CaDiCal and Glucose.

terminated with an out of memory error for all three tested SAT solvers. Here the out of memory error occurred more quickly with Glucose and CaDiCal than with MiniSat.

Based on the results of the experiment, we decide to use only the built-in solver MiniSat when using CBMC.

3.2.3 Protocols Discovered using the Adapted Standardized Program Representation

In this section, we present the protocols, that we discovered while symbolically executing our adapted standardized program representations. First, we will present our results for the functions *XOR* (section 3.2.3.1) and *OR* (section 3.2.3.2). Then we will present the results for *COPY* (section 3.2.3.3) and finally the results for the half adder function (section 3.2.3.4).

3.2.3.1 Protocols for *XOR*

The protocol in fig. 3.1 found by the bounded model checker is the protocol that is card and step minimal if we consider probabilistic (definition 14) and input-possibilistic security (definition 15). It takes only the four cards necessary for the two commitments (definition 3) as input. The protocol has a finite runtime, and performs one shuffle (definition 6) and one turn (definition 9). This *XOR* protocol is similar to the four card protocol by Mizuki and Sone (2009) that is shown in fig. A.2. While Mizuki and Sone (2009) used one random

#Cards	#Steps	Closed?	Security	Protocol	Runtime
4	1	yes	output-possibilistic	✓	Finite Runtime
4	1	yes	input-possibilistic	×	Finite Runtime
4	2	yes	output-possibilistic	✓(see fig. 3.2)	Finite Runtime
4	2	yes	probabilistic (*)	✓(see fig. 3.1)	Finite Runtime

Table 3.3: Protocols that were found through bounded model checking for the *XOR* boolean operator.
(*) the protocol that was found by the bounded model checker had output-possibilistic security, we completed it with the specific fractions of the probabilities, proving that it also satisfies the stronger prerequisites for probabilistic security

bisection cut and two permutation operations (definition 5), the protocol in fig. 3.1 replaces these operations with a single shuffle. Mizuki and Sone (2009) also turn two cards instead of our protocol which turns only one card. They are however both equivalent in regard to their results.

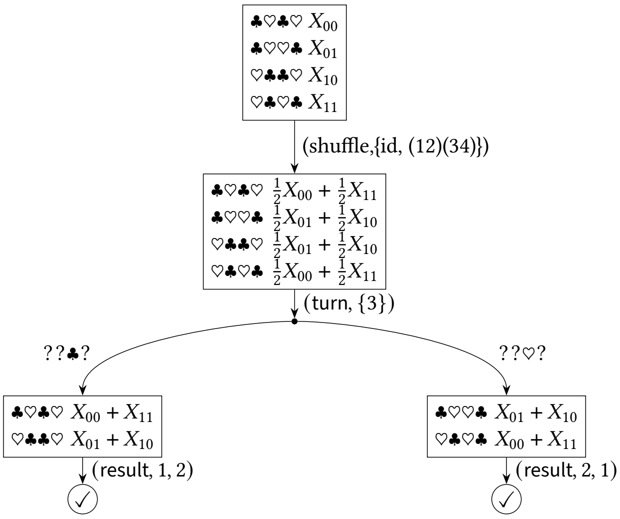


Fig. 3.1: A finite runtime protocol for *XOR* with probabilistic security and closed shuffles. It uses 4 cards and has 2 steps.

Search for an output possibilistic protocol (definition 16) for *XOR* using 4 cards, two steps and only closed shuffles, resulted in the protocol in fig. 3.2. Remarkable about this protocol is, that the first step performed is a turn operation. This reveals the fourth card and thus the value of the second commitment. This is however not a violation of output-possibilistic security. By definition, a protocol is possibilistically output-secure, if at every state in the protocol, every output is still possible (definition 16). Because we have two possible starting commitments that will result in 1 and two commitments that result in 0, turning one card and splitting the protocol in two, will not violate output-possibilistic security. As seen in fig. 3.2, after the turn, in each possible state, both outputs are still possible. This highlights the need for checking protocols for stronger security guarantees for protocols found through bounded model checking.

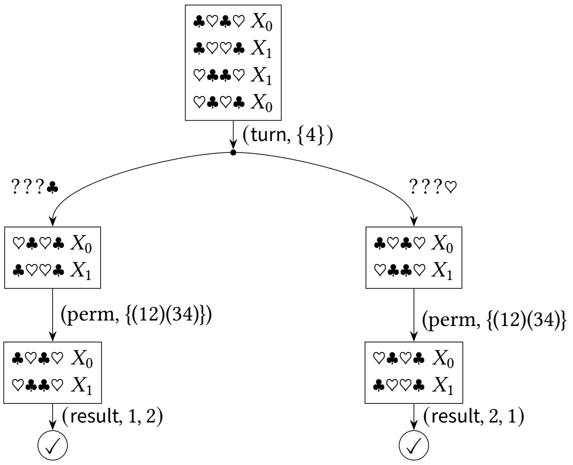


Fig. 3.2: A finite runtime protocol for *XOR* with output possibilistic security and closed shuffles. It uses 4 cards and has 2 steps.

We could also convert this protocol into a protocol with only one step. The permutation applied to both branches after the turn is unnecessary. Instead of performing the permutation, the result could be extracted after the turn. For the branch that resulted by turning ♣ the result could then be for example (result, 2, 1) and for the branch that resulted by turning ♡ a possible result would be (result, 1, 2). This protocol still has output-possibilistic security and is closed (definition 7), because it does not have any (non-closed) shuffles.

3.2.3.2 Protocols for *OR*

#Cards	#Steps	Closed?	Security	Protocol	Runtime
4	2	no	output-possibilistic	×	
4	3	no	output-possibilistic	×	
4	4	no	probabilistic (*)	✓(see fig. 3.3)	Las Vegas
4	4	yes	output-possibilistic	×	
4	4	yes	input-possibilistic	×	
4	5	no	probabilistic (*)	✓(see fig. A.4)	Las Vegas
4	5	yes	input-possibilistic	×	
4	6	yes	probabilistic (*)	✓(see fig. 3.4)	Las Vegas
5	2	no	probabilistic (*)	✓(see fig. A.3)	Las Vegas

Table 3.4: Protocols that were found through bounded model checking for the *OR* boolean operator.
(*) the protocol that was found by the bounded model checker had output-possibilistic security, we completed it with the specific fractions of the probabilities, proving that it also satisfies the stronger prerequisites for probabilistic security

There is no explicit protocol for *OR* in the existing literature. Stiglic (2001) describes the construction of an *OR* protocol from *AND* and *NOT* gates ($x_1 \vee x_2 = \neg(\neg x_1 \wedge \neg x_2)$).

We can perform a *NOT* protocol on an arbitrary number of commitments simultaneously, by executing one permutation operation. This permutation operation swaps the two cards of each commitment. As we can execute $\neg x_1$ and $\neg x_2$ simultaneously, we need two operations to execute the *NOT* operations in $\neg(\neg x_1 \wedge \neg x_2)$.

Additionally we need an existing *AND* protocol. The shortest finite runtime *AND* protocol in the literature is the protocol of Mizuki and Sone (2009) which uses six cards, two permutation operations and a random bisection cut. The two permutation operations and the random bisection cut can be combined into one shuffle operation (definition 6). Thus the protocol can be modified to have six cards and two steps. Using this *AND* protocol, we can construct a protocol for *OR*, that has six cards and four steps. This *OR* protocol has uniform closed shuffles and finite runtime. Using the *AND* protocol by Koch et al. (2015), which is a Las Vegas protocol with closed but non-uniform shuffles, we can construct an *OR* protocol that uses four cards and six steps.

With the use of bounded model checking, we were able to find protocols for *OR*, that used four cards. The protocol in fig. 3.3 has non-closed shuffles and probabilistic security. It uses four cards and has a best case of four steps. As can be seen in table 3.4 there exists no protocol with the same properties but closed shuffles. We could also show, that there are no protocols for four cards and less than four steps for any type of security and type of shuffles. The card minimal protocol for *OR*, that has closed shuffles is the protocol from fig. 3.4. It uses six steps. This protocol is very similar in structure to the four-card Las Vegas *AND* protocol by Koch et al. (2015). Every action up to and including the first turn action is the same. Afterwards the protocols differ slightly. The left branch that results from turning a ♣ in our *OR* protocol is identical operation wise to the right branch that results from turning a ♡ in the *AND* protocol by Koch et al. (2015). Our right branch performs the same action types as the left branch of the protocol by Koch et al., 2015. The concrete shuffles and turns differ however.

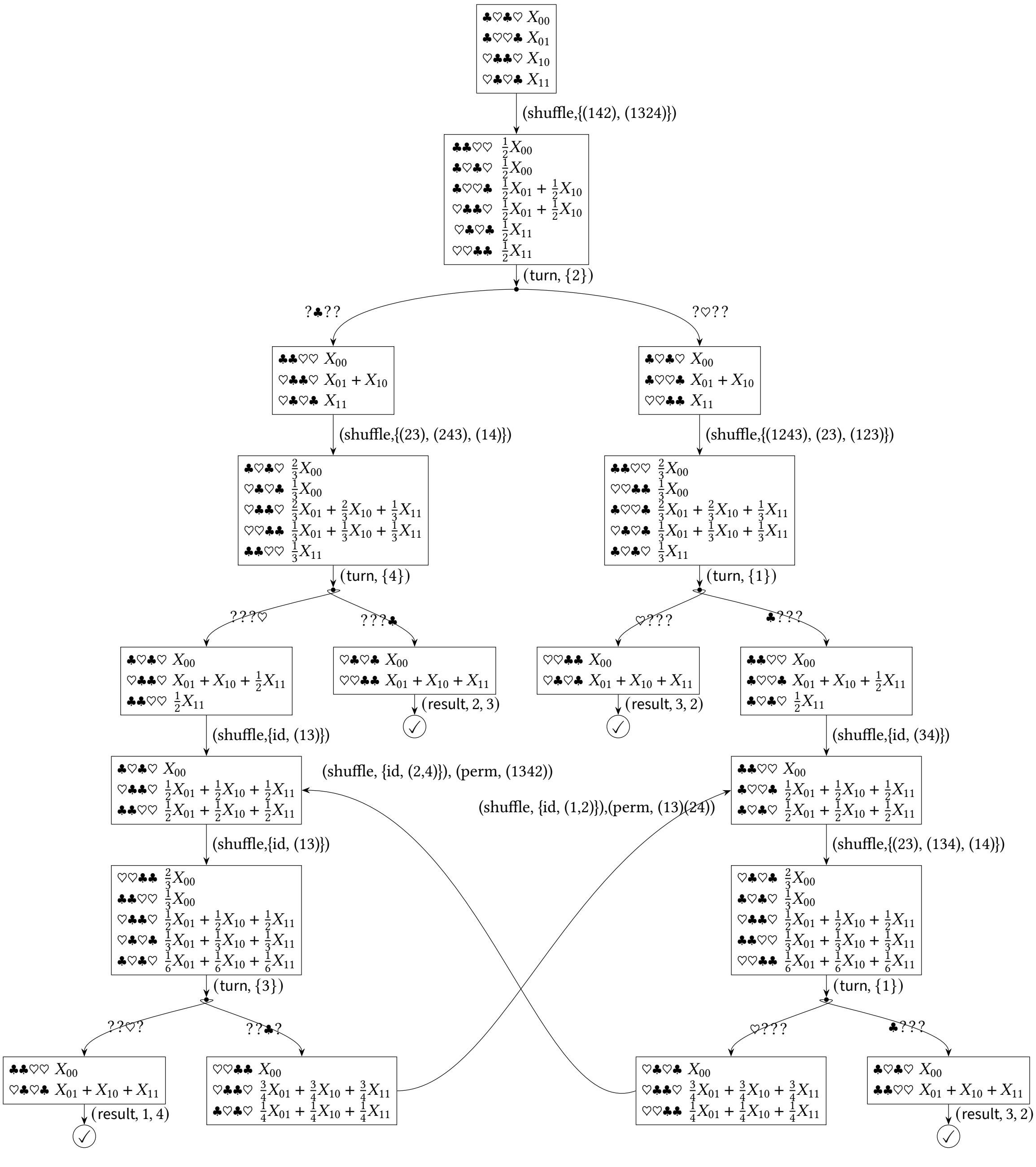

♣♡♣♡ X_{00}
♣♡♡♣ X_{01}
♡♣♣♡ X_{10}
♡♣♡♣ X_{11}
(shuffle,{(142), (1324)})
♣♣♡♡ $\frac{1}{2}X_{00}$
♣♡♣♡ $\frac{1}{2}X_{00}$
♣♡♡♣ $\frac{1}{2}X_{01} + \frac{1}{2}X_{10}$
♡♣♣♡ $\frac{1}{2}X_{01} + \frac{1}{2}X_{10}$
♡♣♡♣ $\frac{1}{2}X_{11}$
♡♡♣♣ $\frac{1}{2}X_{11}$
(turn, {2})
?♣??
?♡??
♣♣♡♡ X_{00}
♡♣♣♡ $X_{01} + X_{10}$
♡♣♡♣ X_{11}
♣♡♣♡ X_{00}
♣♡♡♣ $X_{01} + X_{10}$
♡♡♣♣ X_{11}
(shuffle,{(23), (243), (14)})
(shuffle,{(1243), (23), (123)})
♣♡♣♡ $\frac{2}{3}X_{00}$
♡♣♡♣ $\frac{1}{3}X_{00}$
♡♣♣♡ $\frac{2}{3}X_{01} + \frac{2}{3}X_{10} + \frac{1}{3}X_{11}$
♡♡♣♣ $\frac{1}{3}X_{01} + \frac{1}{3}X_{10} + \frac{1}{3}X_{11}$
♣♣♡♡ $\frac{1}{3}X_{11}$
♣♣♡♡ $\frac{2}{3}X_{00}$
♡♡♣♣ $\frac{1}{3}X_{00}$
♣♡♡♣ $\frac{2}{3}X_{01} + \frac{2}{3}X_{10} + \frac{1}{3}X_{11}$
♡♣♡♣ $\frac{1}{3}X_{01} + \frac{1}{3}X_{10} + \frac{1}{3}X_{11}$
♣♡♣♡ $\frac{1}{3}X_{11}$
(turn, {4})
(turn, {1})
???♡
???♣
♡???
♣???
♣♡♣♡ X_{00}
♡♣♣♡ $X_{01} + X_{10} + \frac{1}{2}X_{11}$
♣♣♡♡ $\frac{1}{2}X_{11}$
♡♣♡♣ X_{00}
♡♡♣♣ $X_{01} + X_{10} + X_{11}$
(result, 2, 3)
✓
♡♡♣♣ X_{00}
♡♣♡♣ $X_{01} + X_{10} + X_{11}$
(result, 3, 2)
✓
♣♣♡♡ X_{00}
♣♡♡♣ $X_{01} + X_{10} + \frac{1}{2}X_{11}$
♣♡♣♡ $\frac{1}{2}X_{11}$
(shuffle,{id, (13)})
(shuffle,{id, (34)})
♣♡♣♡ X_{00}
♡♣♣♡ $\frac{1}{2}X_{01} + \frac{1}{2}X_{10} + \frac{1}{2}X_{11}$
♣♣♡♡ $\frac{1}{2}X_{01} + \frac{1}{2}X_{10} + \frac{1}{2}X_{11}$
(shuffle, {id, (2,4)}), (perm, (1342))
(shuffle, {id, (1,2)}),(perm, (13)(24))
♣♣♡♡ X_{00}
♣♡♡♣ $\frac{1}{2}X_{01} + \frac{1}{2}X_{10} + \frac{1}{2}X_{11}$
♣♡♣♡ $\frac{1}{2}X_{01} + \frac{1}{2}X_{10} + \frac{1}{2}X_{11}$
(shuffle,{id, (13)})
(shuffle,{(23), (134), (14)})
♡♡♣♣ $\frac{2}{3}X_{00}$
♣♣♡♡ $\frac{1}{3}X_{00}$
♡♣♣♡ $\frac{1}{2}X_{01} + \frac{1}{2}X_{10} + \frac{1}{2}X_{11}$
♡♣♡♣ $\frac{1}{3}X_{01} + \frac{1}{3}X_{10} + \frac{1}{3}X_{11}$
♣♡♣♡ $\frac{1}{6}X_{01} + \frac{1}{6}X_{10} + \frac{1}{6}X_{11}$
♡♣♡♣ $\frac{2}{3}X_{00}$
♣♡♣♡ $\frac{1}{3}X_{00}$
♡♣♣♡ $\frac{1}{2}X_{01} + \frac{1}{2}X_{10} + \frac{1}{2}X_{11}$
♣♣♡♡ $\frac{1}{3}X_{01} + \frac{1}{3}X_{10} + \frac{1}{3}X_{11}$
♡♡♣♣ $\frac{1}{6}X_{01} + \frac{1}{6}X_{10} + \frac{1}{6}X_{11}$
(turn, {3})
(turn, {1})
??♡?
??♣?
♡???
♣???
♣♣♡♡ X_{00}
♡♣♡♣ $X_{01} + X_{10} + X_{11}$
(result, 1, 4)
✓
♡♡♣♣ X_{00}
♡♣♣♡ $\frac{3}{4}X_{01} + \frac{3}{4}X_{10} + \frac{3}{4}X_{11}$
♣♡♣♡ $\frac{1}{4}X_{01} + \frac{1}{4}X_{10} + \frac{1}{4}X_{11}$
♡♣♡♣ X_{00}
♡♣♣♡ $\frac{3}{4}X_{01} + \frac{3}{4}X_{10} + \frac{3}{4}X_{11}$
♡♡♣♣ $\frac{1}{4}X_{01} + \frac{1}{4}X_{10} + \frac{1}{4}X_{11}$
♣♡♣♡ X_{00}
♣♣♡♡ $X_{01} + X_{10} + X_{11}$
(result, 3, 2)
✓

Fig. 3.3: A Las Vegas protocol for *OR* with probabilistic security. The shuffles are not closed. It uses 4 cards and has a best case of 4 steps.

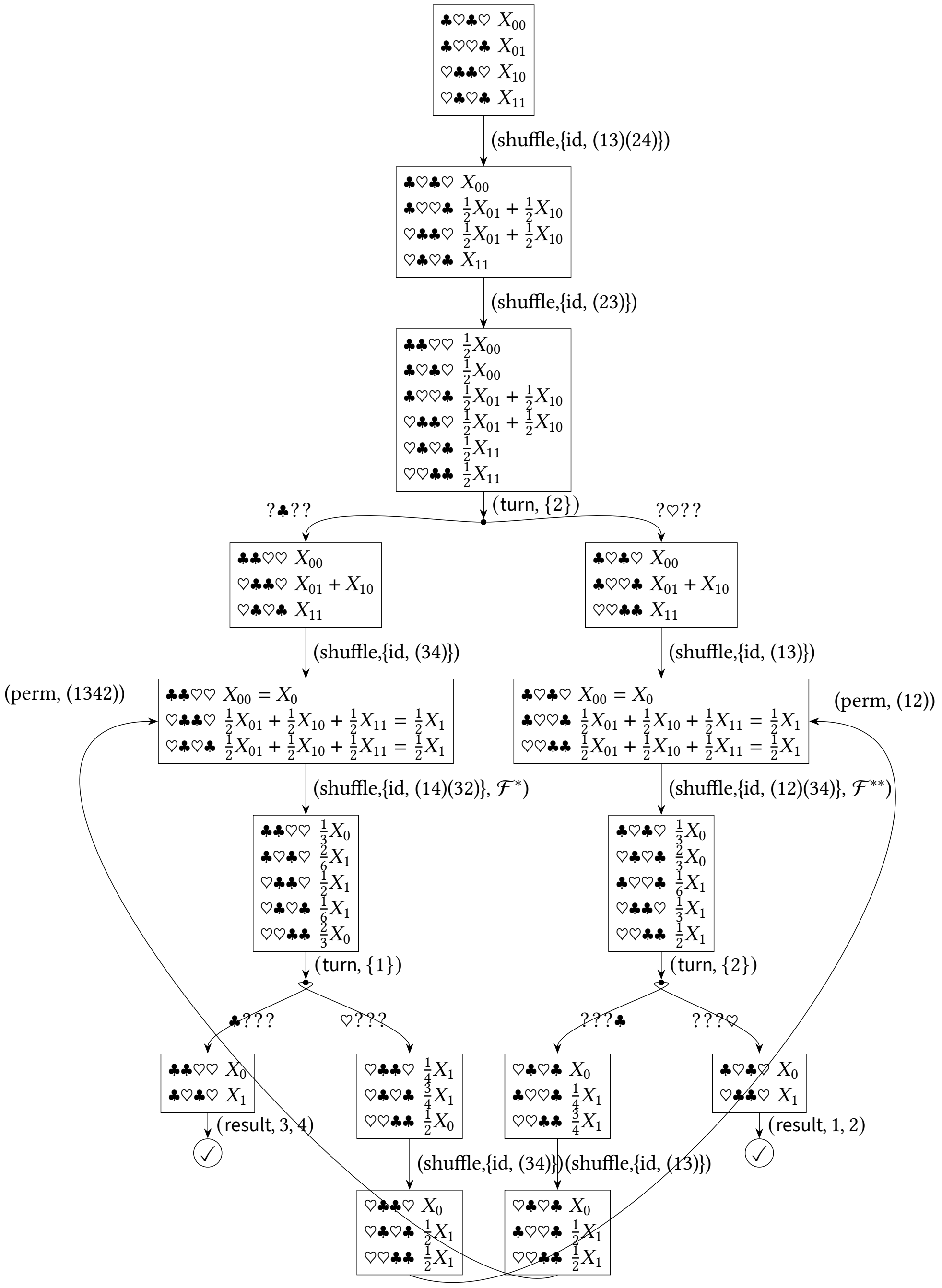


Fig. 3.4: A Las Vegas protocol for *OR* with probabilistic security and closed shuffles. It uses 4 cards and has a best case of 6 steps.
$\mathcal{F}^* = id \rightarrow \frac{1}{3}, (14)(32) \rightarrow \frac{2}{3}$
$\mathcal{F}^{**} = id \rightarrow \frac{1}{3}, (12)(34) \rightarrow \frac{2}{3}$

3.2.3.3 Protocols for *COPY*

#Cards	#Steps	Closed?	Security	Protocol	Runtime
4	2	no	output-possibilistic	×	
4	3	no	output-possibilistic	×	
4	4	no	output-possibilistic	×	
4	5	no	output-possibilistic	×	
5	2	no	probabilistic (*)	✓(see Fig.3.5)	Las Vegas
5	3	yes	output-possibilistic	× (**)	
5	4	yes	output-possibilistic	× (**)	
5	5	yes	output-possibilistic	× (**)	

Table 3.5: Protocols that were found through bounded model checking for the *COPY* function.
(*) the protocol that was found by the bounded model checker had output-possibilistic security, we completed it with the specific fractions of the probabilities, proving that it also satisfies the stronger prerequisites for probabilistic security
(**) For these results, we reduced the permutation set size to 8, to avoid an "out-of-memory" error. This reduces the significance of the result, as there might be protocols for this configuration, that have more than 8 permutations in a shuffle.

For *COPY* protocols, input-possibilistic security and output-possibilistic security are identical. This is, because the two possible inputs ♣♡ and ♡♣ and their input-possibility X_{00} and X_{01} match up with the possible outputs and their output-possibility X_0 and X_1. Therefore any protocol with output-possibilistic security also has input-possibilistic security. In table 3.5, we have given the stronger security definition in each case for illustrative purposes, even though they are synonymous in this case. For the non-existence of a protocol, this is output possibilistic security. For an existing protocol, this is input possibilistic security (remark 1).

We also focused on protocols that produced one copy of the given input commitment the copy protocol. There are protocols, for example by Mizuki et al. (2006) that can produce any number of copies n (table 2.1). The existing protocol for *COPY* that uses the least amount of cards, is the finite runtime protocol by Mizuki and Sone (2009). It uses $2 * n + 4$ cards to produce n copies. To produce one copy, as we did in our experiments, the protocol by Mizuki et al. (2006) would therefore need six cards (listing 2.1).

With the use of our standardized program representation, we were able to find a protocol that only uses five cards and two steps (see fig. 3.5) and thus uses one card less than the protocol with the least amount of cards from literature. It is however a Las Vegas protocol and the shuffles are not closed but uniform.

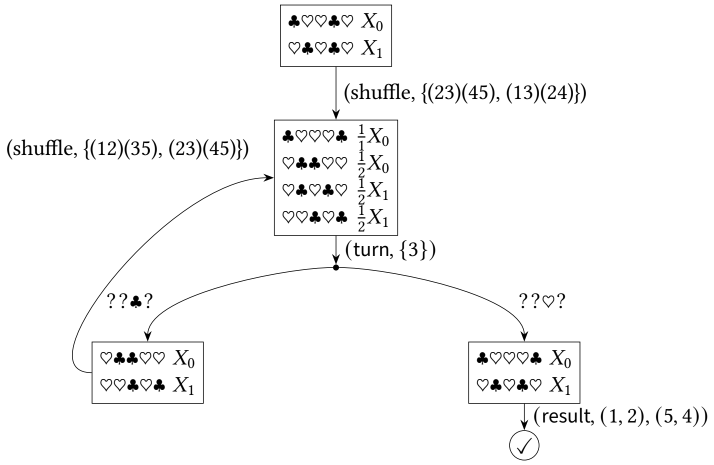


Fig. 3.5: A Las Vegas protocol for *COPY* with probabilistic security and non-closed shuffles. It uses 5 cards and has a best case of 2 steps.

3.2.3.4 Protocols for the Half Adder

#Cards	#Steps	Closed?	Security	Protocol	Runtime
4	2	no	output-possibilistic	×	
4	3	no	output-possibilistic	×	
4	4	no	output-possibilistic	×	
4	5	no	output-possibilistic	×	
5	2	no	output-possibilistic	×	

Table 3.6: Protocols that were found through bounded model checking for the half adder function.

A half adder is a boolean function that takes two bits and calculates the sum of those two inputs as a two bit output.

Definition 21. *A half adder function is a boolean function. It takes two inputs $b_1, b_2 \in \{0, 1\}$ and produces two outputs $s, c \in \{0, 1\}$ where $s = b_1 \oplus b_2$ is the sum and $c = b_1 \wedge b_2$ is the carry.*

As can be observed in table 3.6, we were not able to find any protocols for the half adder with our method. We could show, that there are no protocols for the half adder using four cards and between two and five steps, as well as using five cards and two steps.

3.2.4 Limits for the Use of the Bounded Model Checker with Our Symbolic Program Representation

When we used the bounded model checker CBMC (section 2.3) with our implementation to find protocols, it did not produce results for certain runs. In general, we could not find protocols, or show that no protocol exists for more than five cards and two steps, when we searched for non-closed protocols (definition 7). For closed protocols we could not find protocols, or show that no protocol exists for five cards and two steps. The biggest number of cards and steps for which the bounded model checker returned a result was five cards and five steps for closed protocols. In the cases where we received no result, the bounded model checker either returned after an "out-of-memory" error occurred or because the timeout threshold we set was reached.

The "out-of-memory" error occurred for example when running the program for the half adder with five cards, two steps and output-possibilistic security. We also limited the search to only closed shuffles. We also got an "out-of-memory" error for the program for the *COPY* function with five cards, two steps output-possibilistic security. We limited the search to only closed shuffles here as well. We can see the trace for the *COPY* function in listing 3.6. Here the last notice was, that CBMC was converting into static single assignment (SSA) form (Kroening and Tautschnig, 2014). After that it produced the error message `SAT checker ran out of memory`. We can not explain the exact cause of this "out-of-memory" error. However we note that the size of the program expression for this run is 8219163 steps (listing 3.6, line 2) while for a program that was too big to produce a result before the set timeout, the size of the program expression was 3596454 steps (listing 3.7, line 2) and with therefore smaller.

Listing 3.6 Excerpt from the trace of CBMC for the five cards two steps *COPY* function. The protocol terminated without a result after the SAT checker ran out of memory.

```
1  Runtime Symex: 1041.01s
2  size of program expression: 8219163 steps
3  simple slicing removed 5 assignments
4  Generated 1 VCC(s), 1 remaining after simplification
5  Runtime Postprocess Equation: 2.41528s
6  Passing problem to propositional reduction
7  converting SSA
8  SAT checker ran out of memory
9  Out of memory
```

We also ran protocols, where the bounded model checker did not reach a solution before it was terminated because of the timeout we set. The timeout threshold for the runs in this section was set at 5 days. We reached such a timeout threshold for the program for the half adder with five cards, two steps, output-possibilistic security and non-closed shuffles. As we can see from the trace in listing 3.7, the built-in solver MiniSat was solving the formula when the bounded model checker was terminated. The SAT formula had 124330664 variables and 448826210 clauses.

Listing 3.7 Excerpt from the trace of CBMC for the five cards three steps half adder function. The protocol terminated without a result after the set timeout threshhold of 5 days (432004s) was reached.

```
Runtime Symex: 735.22s
size of program expression: 3596454 steps
simple slicing removed 5 assignments
Generated 1 VCC(s), 1 remaining after simplification
Runtime Postprocess Equation: 1.08506s
Passing problem to propositional reduction
converting SSA
Runtime Convert SSA: 377.169s
Running propositional reduction
Post-processing
Runtime Post-process: 0.000110975s
Solving with MiniSAT 2.2.1 with simplifier
124330664 variables, 448826210 clauses
```

Based on our experiments, the bounded model checker has constraints finding protocols or proving that there are no valid protocols for certain tasks. If the tasks get too complex, it either returns an "out-of-memory" error or does not terminate within a reasonable timeout threshold. For our implementations of the standardized program representation, this threshold was reached for five cards and two steps if we searched for closed protocols and five cards and three steps if we searched for protocols that were not closed.

4 A Standard Program Representation with Nested Structure for Composite Protocols

In section 4.1 we introduce a nested structure of the symbolic program. We define a new action and extend KWH-trees (section 2.2.3) to incorporate protocol operations in section 4.1.1. In sections 4.1.2 and 4.1.3 we motivate its use within our programs to reduce the complexity and size of the search space of the bounded model checking program while ensuring correctness and security of the composited protocols. Then we present an implementation of the new protocol action, that can apply a given protocol to a state within a symbolic program, in section 4.2. To evaluate the effectiveness, we perform a series of experiments and present our findings in section 4.3.

4.1 Introduction of a Nested Structure and Consideration of its Possible Benifits

Functions that perform more complex computations than for example boolean operators, tend to use more cards and also require more steps. One example might be the *COPY* protocol by Mizuki and Sone (2009) which uses six cards (section 2.2.4). This is more than required for boolean operators like *XOR* (fig. A.2), *AND* (fig. A.1) and *OR* (fig. 3.4), that all use five or fewer cards. Another example of a function that uses a lot of cards and steps is the half adder (definition 21). As can be observed in table 3.6, we were not able to find a protocol with our method from chapter 3. There were no protocols for small amounts of cards and steps. For larger numbers of cards and steps the bounded model checker either terminated with an "out-of-memory" error or could not produce a protocol within our set timeout of five days (section 3.2.4).

4.1.1 Defining the Protocol Action and Extending KWH-Trees

To try to make our standard program representation and finding protocols with bounded model checking (sections 2.3 and 2.4) more effective for those protocols with more cards and steps, we propose to integrate another action type, in addition to the already existing permutation (definition 5), shuffle (definition 6) and turn actions (definition 9). This new action will apply protocols instead of shuffles and turns, to the current state. These protocols can for example be from the literature (e.g. table 2.1), or they can be protocols that have been found by using the bounded model checking program (e.g. section 3.2.3).

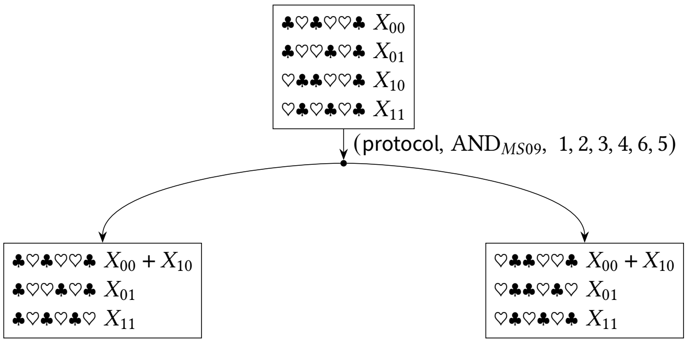
Definition 22. *We define another action, that applies a protocol as (protocol, $\mathcal{P}$, p_1, p_2, ..., p_r). It has a protocol $\mathcal{P}$ and positions p_1, p_2, ..., p_r $\in$ $\{1, ..., l\}$. Given a sequence Γ we perform (protocol, $\mathcal{P}$, p_1, p_2, ..., p_r) on the sequence by performing $\mathcal{P}$ on the sequence $\Gamma^* = (\Gamma[p_1], \Gamma[p_2], \ldots, \Gamma[p_r])$. The result is a sequence Γ^{**}, where at the positions p_1, p_2, ..., p_r we have the cards $\Gamma^*[1], \Gamma^*[2], \ldots, \Gamma^*[r]$ and for all other positions p_i we have $\Gamma[p_i]$. If the protocol $\mathcal{P}$ has more than one possible end states, one of these end states is chosen by random.*

As an example we take the sequence $\Gamma = (\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\clubsuit}, \frac{?}{\heartsuit})$ and the protocol operation (protocol, AND$_{MS09}$, 1,2,3,4,5,6). *AND$_{MS09}$* is the *AND* protocol by Mizuki and Sone (2009). As we can see in the KWH-tree in fig. 2.1, the protocol has two possible result states, that are each reached with probability 1/2. If we now apply the protocol to our chosen sequence sequence $\Gamma = (\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\clubsuit}, \frac{?}{\heartsuit})$ and positions $p = 1, 2, 3, 4, 5, 6$, we have two possible resulting sequences. If we perform the *AND* protocol on $\Gamma^* = (\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\clubsuit}, \frac{?}{\heartsuit})$, and the cards we turn in the last step are ♣♡, we receive the sequence $\Gamma'(\frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\clubsuit}, \frac{?}{\heartsuit})$. If the cards we turn in the last step are ♡♣ however, we receive the sequence $\Gamma''(\frac{?}{\heartsuit}, \frac{?}{\clubsuit}, \frac{?}{\clubsuit}, \frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{?}{\clubsuit})$.

The protocol $\mathcal{P}$ in (protocol, $\mathcal{P}$, p_1, p_2, ..., p_r) must meet certain requirements so that (protocol, $\mathcal{P}$, p_1, p_2, ..., p_r) is a valid action. First of all $\mathcal{P}$ must have as many or fewer cards than the sequences on which it should be applied. It is for example impossible to perform a six card protocol on only five cards. There also needs to be a start state in $\mathcal{P}$ for the sequence $\Gamma^* = (\Gamma[p_1], \Gamma[p_2], \ldots, \Gamma[p_r])$ in order to perform the protocol on a sequence Γ.

We have introduced the concept of KWH-trees in section 2.2.3 to visualize protocols. In order for us to visualize protocols using our new action from definition 22, we need to extend the actions of KWH-trees. After the application of a protocol (protocol, $\mathcal{P}$, p_1, p_2, ..., p_r), the visible sequence differs just as with the turn operation. Thus the tree branches. Every branch is the result of a possible end state of the applied protocol. This is, because if the applied protocol produces a visible sequence while, it does so also with the larger protocol it is embedded in. If we for example apply the *AND* protocol by Mizuki and Sone (2009) to the given sequence as shown, we have two possible resulting states:

The probability for observing either of the end states is the same for each possible input. Therefore no information, about the sequence, that was not revealed by executing the protocol, is leaked.



To use a protocol $\mathcal{P}$ within a protocol action (protocol, $\mathcal{P}$, p_1, p_2, ..., p_r), we do not need to know the full protocol. All that is necessary is the start state and the end states. Each state needs to contain the possible sequences and their probabilities (definition 17). We do not need the actual steps within the protocol, as long as we know that there is a path from the input state to all the end states.

Remark 2. *The protocol $\mathcal{P}$ in (protocol, $\mathcal{P}$, p_1, p_2, ..., p_r) must have at least the same or a stronger level of security than the protocol that should be found using it.*

Therefore, if we want to find an input-possibilistic protocol (definition 15), we need to use a protocol for our protocol action, that has at least input-possibilistic security. We cannot use a protocol that has output-possibilistic security (definition 16). If we want to construct a protocol with a finite runtime (definition 11), we can only use protocols that have finite runtime as well. If a Las Vegas protocol is performed as a protocol action, every protocol using this action will also be a Las Vegas protocol regardless of what the other used actions are.

4.1.2 Reducing the Complexity of the Search Space

As shown in section 3.2.3 for more complex functions whose protocols require more cards, and more steps such as the half adder or the *COPY* function, we had difficulties finding protocols with our method presented in section 3.1. We can for example observe, that there did not exist any protocols for the half adder for a small amount of cards and steps (section 3.2.3.4), and for bigger numbers of cards and steps the bounded model checker could not produce a result. Either because it would take too long or because an "out-of-memory" error occurred (section 3.2.4).

One cause could be the size of the search space. As we have described in chapter 3, while searching for a protocol, the bounded model checker has to nondeterministically choose and execute an action for each step. As he exhaustively searches all possible runs to the program, he must also consider all possible turns and shuffles for each action. There are not that many possible turns. For N cards, there are N possible turn positions that have to be considered. With shuffles however, there are significantly more possibilities. For N

cards, there are $N!$ possible permutations. However, for a shuffle we consider permutation sets of any size (not just size 1) apart from the empty set. Thus the number of possible shuffles is $2^{N!} - 1$. For 4 cards we would therefore have $2^{4!} - 1 = 2^{24} - 1$ possibilities to choose a permutation set for these 4 cards. This is over 16 million possibilities. For five cards we already have more than $1.3 * 10^{36}$ possibilities ($2^{120} - 1$) to choose a single permutation set. For four cards but two steps instead of one we would have to consider $(2^{N!} - 1)^2$ permutation sets, because each of the two actions can be a shuffle. That would be more than $2.8 * 10^{14}$ possibilities. These examples motivate that having less actions as well as less cards has a big impact on the amount of possible runs the bounded model checker has to search.

The assumption with the protocol action is that it makes it possible for us to find protocols with less cards and steps. As described in definition 18 protocols can be used as parts of a bigger protocol to calculate boolean functions. These protocols used as an action consist of multiple turn and shuffle operations. Integrating the protocol action would reduce all the steps from such a protocol, to just the one in the protocol action. Thus we would reduce the amount of steps needed to find a protocol.

A protocol action also does not add much complexity to the program, as it is significantly less complex than a turn operation. To apply a protocol action, we nondeterministically choose the protocols that we will be using, and then the cards on which we want to perform the protocol. After applying the protocol to the cards we have to nondeterministically choose which possible output state we want to look at further. The most complex component is choosing which cards to perform the protocol on, as there are normally very few possible output states and the number of protocols used is also limited. For choosing which cards to perform the protocol on, we have to pick k cards that are all different, from N cards in our sequence. The order is important, as a protocol performed on cards that are arranged differently, produces different results. Therefore we have $\frac{N!}{(N-k)!}$ possibilities to choose the cards to perform the protocol on. Let us consider a set of 4 cards again. On these four cards we now want to perform the 4 card *XOR* by Mizuki and Sone (2009). We therefore have $\frac{4!}{(4-4)!} = 24$ different possibilities of choosing which cards to perform the protocol on. Compared with the over 16 million possible permutations for 4 cards, this is negligible.

4.1.3 Calculating any Function While Ensuring Correctness and Security

In section 2.2.4 we have described that any boolean function can be calculated using two colour cards, as long as there are enough additional cards. Provided we have protocols for a functionally complete set of boolean operators like *AND*, *OR* and *NOT* or *AND* and *XOR* and a *COPY* protocol, we can assemble them to a protocol for an arbitrary function. This can be done without the use of bounded model checking, however using bounded model checking offers several advantages.

First of all, the protocols found by assembling boolean operators, might use more cards than necessary. A naive approach of assembling boolean operators is, to make the needed copies of the input commitments and then to execute the chosen boolean operators on these copies. With the use of bounded model checking, boolean operators are taken into account, but at each step turns and shuffles are also considered. This can allow the bounded model checking tool to find shorter protocols both in the terms of the amount of cards as well as the length of protocols, than the naive approach of assembling boolean operators.

Another advantage of using bounded model checking is, that it ensures the correctness and security of the resulting protocol. A protocol returned by the bounded model checking program meets all security and correctness requirements that are required by the program. It also provides a detailed, step-by-step description of all the operations, that have to be performed in order to execute the protocol.

This is not always the case, when protocols are assembled from boolean operators. One example is the descripion of a composite half adder protocol by Mizuki et al. (2013).

Example: A Composite Half Adder Protocol

In their paper, Mizuki et al. (2013) describe a ten card protocol that computes a half adder (definition 21) which uses preexisting protocols. They start with two input commitments and six cards that were arranged as ♣♡♣♡♣♡. They then copy both input commitments once and apply the four card *XOR* protocol from their paper to the first two commitments and the six card *AND* to the last two commitments. Then they instruct the users to perform some rearrangements.

If we take a look at fig. 4.1, we can see an excerpt from the described protocol. The start state of the tree is the state, that the cards are in after the four card protocol, but before the *AND* protocol has been applied. The first four cards have been rearranged in a way, that the first two cards are now the result of the *XOR* protocol and the third and fourth cards are the rearranged helper cards. This is now the exact state that is depicted in Mizuki et al. (2013).

Now, as instructed, we perform the *AND* protocol that is given in Mizuki et al. (2013) on the last six cards. The resulting states of the *AND* protocol should now be rearranged in a way that we have the result of the *AND* and *XOR* protocol in the front and then afterwards six cards that are arranged as ♣♡♣♡♣♡. Mizuki et al. (2013) do not specify the explicit permutations that have to be used.

We provide a possible set of shuffles and permutations in fig. 4.1. First we rearrange the cards in a way, that the result of the *AND* is the first commitment, and the result of the *XOR* is the second commitment. We can then transfer one of the resulting states into the other, by swapping the seventh and eighth card. Now we have only one post state

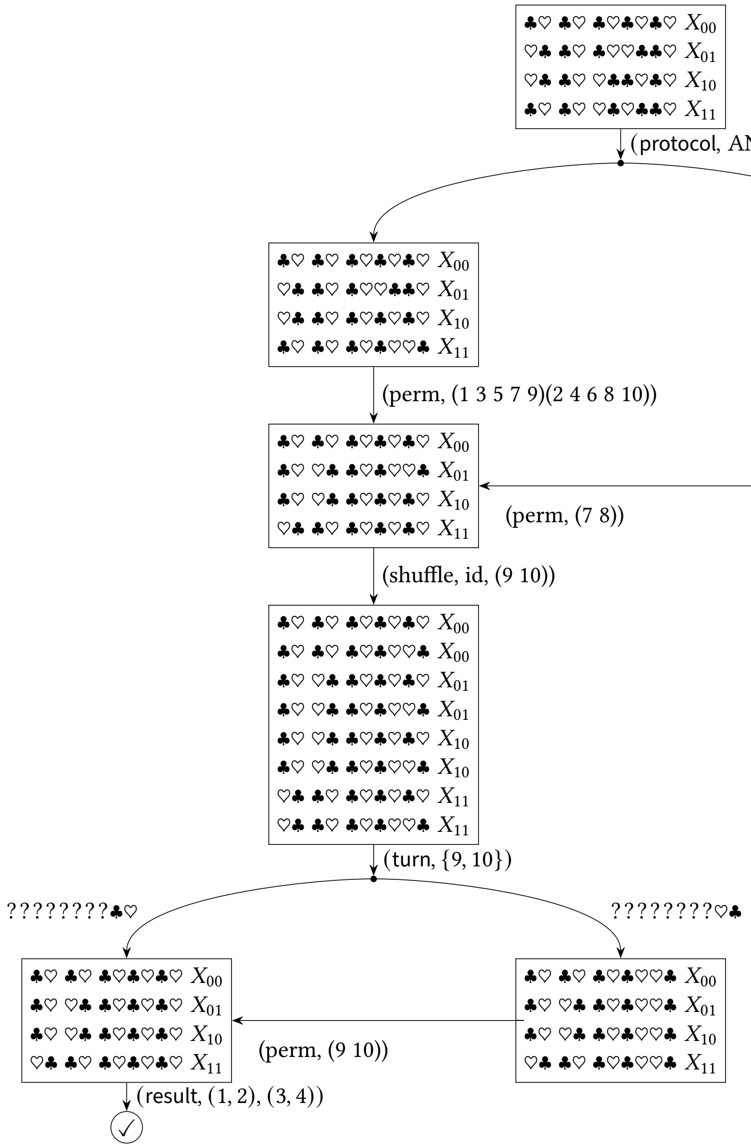


Fig. 4.1: The ten card half adder protocol by Mizuki et al. (2013) using preexisting protocols.

where we only have to put the last six cards in the order ♣♡♣♡♣♡. The first four cards are already correct, but the last two cards are different depending on the input. We thus cannot turn them around without making the protocol insecure. We therefore have to perform a shuffle action first, that shuffles the last two cards, so we can retain security. Then we can turn around the last two cards and rearrange them if necessary.

Remark 3. *In general we can rearrange cards in a way that we can securely turn them over in the following case: Given a state with its possible sequences and k cards at positions $P_1, p_2, \ldots, p_k$ which we want to rearrange. We can securely do so if there are the same number of ♣ cards and the same number of ♡ cards at the chosen positions throughout all the possible sequences.*

As can be seen in fig. 4.1 the actual actions that need to be performed to rearrange the cards are non trivial. Therefore it is useful to specify the operations explicitly. That way there are no ambiguities regarding the correctness and security of the protocol. This highlights the advantage of the use of bounded model checking, as a found protocol will not only mention the need for rearrangements, but also provide the exact turn and shuffle actions necessary.

4.2 Integrating the Nested Structure into the Symbolic Program

We now turn to the implementation of the integration of previously found protocols into the scope of possible actions that can be taken within a protocol (section 2.4). For this integration we design a new possible action in addition to the already existing shuffle and turn actions. The implementation and integration of the protocol action can be found in full in appendix A.6.

As described in section 4.1.1 the protocol action (definition 22) is performed on a state and produces the state, that results from applying a protocol to the input state. For a implementation of the protocol action we would have a function that recieves a state as a input and returns the resulting state on which the protocol has been applied. To apply the protocol, this function executing the protocol action first has to choose a protocol and the cards it wants to execute the protocol on. Afterwards it applies the chosen protocol to the chosen cards and calculates the resulting endstates and the resulting possibilities. It also has to check for the correctness and security of the protocol. Just as with the turn action, the protocol action only returns one of the possible end states.

A protocol action has a set of predefined protocols that it can execute. These protocols must meet the prerequisites described in section 4.1.1. The concrete protocol that is used within the action is picked nondeterministically out of the predefined protocols at the start of the protocol action.

After the protocol is chosen, the appropriate amount of cards has to be selected. We select the cards at certain positions, thus the `com1A`, `com1B`, `com2A` etc. in lines 2-5 in listing 4.1 and the `help1` and `help2` in lines 1-2 in listing 4.2 are all positions of cards that are chosen. These card positions have to be disjoint. That means there cannot be a card that is used twice for the protocol. Therefore after choosing the cards, there are checks that ensure that there is no card, that is used twice for the protocol (lines 7-9 in listing 4.1 and lines 4-5 in listing 4.2). Thus if we want to execute a four card protocol on a state with sequences of six cards, we have to choose four different cards out of the six to execute our protocol on. These selected cards have to form a sequence that fulfills the preconditions of the protocol chosen. Within the input of a protocol, there are generally two types of cards. Cards that are a part of a commitment and cards that are not. Cards that encode a commitment form a complete commitment as a pair of two. In listing 4.1, lines 2-5, there are four cards chosen that represent two commitments in total: `com1A`,`com1B` and `com2A`,`com2B`. They have to be valid commitments which can be checked like in listing 4.1 lines 10-18. Here we check that throughout every possible sequence in the state resulting from the cards we have chosen, we have for our commitments two different symbols.

Listing 4.1 Choosing four cards that represent two commitments and checking if they are valid commitments.

```
 1 // choosing the indices of the cards that will be used in the
       protocol
 2 unsigned int com1A = nondet_uint();
 3 unsigned int com1B = nondet_uint();
 4 unsigned int com2A = nondet_uint();
 5 unsigned int com2B = nondet_uint();
 6 assume(com1A < N && com1B < N&& com2A < N&& com2B < N);
 7 assume(com1A != com1B && com1A != com2A && com1A != com2B);
 8 assume(com1B != com2A && com1B != com2B);
 9 assume(com2A != com2B);
10 for (unsigned int i = 0; i < NUMBER_POSSIBLE_SEQUENCES; i++) {
11     // if the probability/possibility of this state is not 0
12     if (isStillPossible(s.seq[i].probs)) {
13         // check that throughout every possible sequence in the state
               we have chosen two different cards for our commitments
14         assume(s.seq[i].val[com1A] != s.seq[i].val[com1B]);
15         assume(s.seq[i].val[com2A] != s.seq[i].val[com2B]);
16          }
17      }
18 }
```

Cards that do not encode a commitment are additional cards, that have to have a specific colour throughout all possible sequences in the start state. For the six card *AND* protocol by Mizuki and Sone (2009) the additional cards are a ♣ card and a ♡ card in precisely that order. For the protocol action, if it chooses a protocol that contains such additional non-commitment cards, it has to check that the cards are the same throughout all possible sequences in the state. In listing 4.2 this is done for protocols that use two helper cards ♣♡ such as the six card *AND* protocol by Mizuki and Sone (2009). In lines 6-12 we check that the non-commitment cards are `help1=`♣ and `help2=`♡ for all sequences in the state.

Listing 4.2 Choosing two additional cards and checking that they are the same all throughout every possible sequence in the state.

```
1 help1 = nondet_uint();
2 help2 = nondet_uint();
3 assume(help1 < N && help2 < N);
4 assume(help1 != com1A && help1 != com1B && help1 != com2A && help1
      != com2B);
5 assume(help2 != com1A && help2 != com1B && help2 != com2A && help2
      != com2B && help2 != help1);
```

```
for (unsigned int i = 0; i < NUMBER_POSSIBLE_SEQUENCES; i++) {
    // if the probability/possibility of this sequence is not 0
    if (isStillPossible(s.seq[i].probs)) {
        // check that helper cards are the same all throughout every
            possible sequence in the state
        assume(isZero((s.seq[i].val[help1]), s.seq[i].val[help2]));
    }
}
```

The actual application of the protocol to the chosen cards happens in three phases. For each sequence in the state, we determine what input sequence it encodes in the chosen protocol. Afterwards we look up and assign the new values that result after applying the protocol to the cards. Finally we assign the correct probabilities and and calculate the resulting endstate.

Listing 4.3 Application of the chosen protocol. This is performed for each sequence seq.

```
unsigned int idx = 0;
// determine what input sequence is encoded by the chosen cards
if (isZero(seq.val[com1A], seq.val[com1B])) {
    if (isZero(seq.val[com2A], seq.val[com2B])) {
        // 0101
        idx = 0;
    } else if (isOne(seq.val[com2A], seq.val[com2B])) {
        // 0110
        idx = 1;
    }
} else if (isOne(seq.val[com1A], seq.val[com1B])) {
    if (isZero(seq.val[com2A], seq.val[com2B])) {
        // 1001
        idx = 2;
    } else if (isOne(seq.val[com2A], seq.val[com2B])) {
        // 1010
        idx = 3;
    }
for (unsigned int i = 0; i < MAX_PROTOCOL_ENDSTATES; i++) {
    // look up and assign the new values that result after applying
        the protocol to the cards
    seq.val[com1A] = protocolTable[protocolChosen][i][idx][0];
    seq.val[com1B] = protocolTable[protocolChosen][i][idx][1];
    seq.val[com2A] = protocolTable[protocolChosen][i][idx][2];
    seq.val[com2B] = protocolTable[protocolChosen][i][idx][3];
    // only if we have helper cards:
    seq.val[help1] = protocolTable[protocolChosen][i][idx][4];
    seq.val[help2] = protocolTable[protocolChosen][i][idx][5];

    // assign the correct probabilities and and calculate the
        resulting endstate
    result = copyResults(seq, result, i);
}}
```

To apply the chosen protocol to the given state, we first have to match the sequences of the given state to their corresponding input states in the chosen protocols. As the non-commitment cards are the same throughout all sequences in the input states, we only have to consider the output states. As can be observed in listing 4.3 lines 4-19, for protocols with two input commitments we have four different input possibilities. We thus check which of the four different inputs the chosen cards encode for each sequence.

As a protocol performs turn and shuffle operations in its execution, the arrangement of cards in the end state of a protocol is different from that in the start state. For example the six card *AND* protocol shown in fig. 2.1 has the following start and end states:

♣♡♣♡♣♡ X_{00}
♣♡♡♣♣♡ X_{01}
♡♣♣♡♣♡ X_{10}
♡♣♡♣♣♡ X_{11}

♣♡♣♡♣♡ $X_{00} + X_{10}$
♣♡♡♣♣♡ X_{01}
♣♡♣♡♡♣ X_{11}

♡♣♣♡♣♡ $X_{10} + X_{00}$
♡♣♣♡♡♣ X_{01}
♡♣♡♣♣♡ X_{11}

Fig. 4.2: The start and end states of the *AND* protocol by Mizuki and Sone (2009)

If we now pick out the input sequence ♣♡♣♡♣♡ with possibility X_{00} we can see that after the execution of the protocol it will either be rearranged into ♣♡♣♡♣♡, if we find ourselves in the left end state, or ♡♣♣♡♣♡ if we find ourselves in the right end state. The amount of ♣ and ♡ cards are still the same, they have just changed places. As this rearrangement is fixed if we know which input we have, we can assign each input its possible outputs without having to perform the protocol.

Thus what we would want to do in our protocol is determine the sequence, look up how this sequence is rearranged in the end state, and copy the rearranged sequence from our end state. We copy this rearrangement by assigning the correct output values to the card positions. Because we have determined which input sequence corresponds to our sequence, we can look up the new values that this input sequence would have after the execution of the protocol. The new values are stored in a lookup table. An excerpt showing the entry for the six card Finite Runtime *AND* protocol with input possibilistic security by Mizuki and Sone (2009) can be seen in listing 4.4. They contain the representation of the post states that are depicted in fig. 4.2. The protocol has two possible output states and therefore the array has two entries. One entry containing the rearrangements for each one of the output states. Each of these entries contains four entries that are the post states that result from applying the protocols to the given sequence. We have four post states, because we have four possible input sequences. These post state entries now assign to each index, the card that will be located there.

Listing 4.4 Excerpt from the lookup table (`protocolTable[FR_AND]`) for the six card Finite Runtime *AND* protocol with input possibilistic security by Mizuki and Sone (2009).

```
{{{1,2,1,2,1,2}, {1,2,2,1,1,2}, {1,2,1,2,1,2}, {1,2,1,2,2,1}},
 {{2,1,1,2,1,2}, {2,1,1,2,2,1}, {2,1,1,2,1,2}, {2,1,2,1,1,2}}}
```

Let us translate our example from above to this representation. There we had the input sequence ♣♡♣♡ that was rearranged into ♣♡♣♡♣♡ or ♡♣♣♡♣♡. In listing 4.3 lines 4-7 this input sequence ♣♡♣♡ (`{0,1,0,1}`) was assigned to the index 0. Therefore we need to take the first entry of each of the entries for the possible end states. If we for example take the first entry from the array representing the first states, we get `{1,2,1,2,1,2}`. This is the array that represents the sequence ♣♡♣♡♣♡.

As described, all we have to do now is assign the new symbols to the respective card indices. In listing 4.3 this is done in the lines 22-28. In our example for the six card Finite Runtime *AND* protocol lookup table in listing 4.4 and the sequence (`{0,1,0,1}`), the first card index `com1A` will be `1`. The second card index `com1B` will be `2`, the third card index `com2A 1` and the fourth card index `com2B 2`. For the non-commitment cards `help1` and `help2` it will be `1` and `2` respectively.

Security and Correctness

To ensure the security of the resulting protocol, we have to use secure protocols within our protocol action. The protocols that can be used during a protocol action, must generally have the same or a stronger level of security as the protocol that should be found as was described in section 4.1.1. The code and explanations that were presented only apply to protocols that possess input possibilistic security. For output possibilistic security the code and explanations that were presented above do not apply. This is because in an only output possibilistically secure protocol, a possible output state can be reached only by a part of the states. Take the Four-Card *XOR* protocol from fig. 3.2. Here we can observe, that if we have the second output state, only the inputs ♣♡♣♡ and ♡♣♣♡ could have led there. Thus if we want to implement this protocol into our protocol action, the other two possible input states (♣♡♡♣ and ♡♣♡♣) will not have a result in this second output state at all. Unlike in listing 4.3 we would therefore have input sequences that are not relevant for a specific endstate and thus are not assigned new values but ignored.

To ensure correctness of the resulting protocol we have to check whether the protocol action produces a valid resulting state. To be a valid state, a state cannot contain bottom sequences. A sequence is a bottom sequence if it belongs to more than one possible output. A protocol action can produce a bottom sequence, if a resulting sequence from the chosen protocol can result from input sequences that have different outputs in the protocol, that we want to find. Thus we have to check for bottom sequences after applying a protocol. Another prerequisite for a correct protocol as result is that the chosen protocols, that are used in the protocol action, are correct.

4.3 Implementing Concrete Protocols and Evaluating the Implementation

To evaluate the effectiveness of the proposed introduction of a nested structure, we performed a series of experiments. The new action was implemented as described in section 4.2. For the nested protocols we chose to use a set of functionally complete set of boolean operators that contain only finite runtime protocols and Las Vegas protocols each. Las Vegas protocols generally require fewer cards, which makes them more attractive for finding protocols that are as card-minimal as possible. However, if we want to find a finite runtime protocol, we have to use only finite runtime protocols as nested protocols, because if a segment of a protocol has only an expectedly finite number of steps, so does the entire protocol.

For the Las Vegas protocols we chose the functionally complete set of boolean operators *AND*, *OR* and *NOT*. For *AND* we included the six card *AND* protocol by Mizuki and Sone, 2009 which can be seen in fig. 2.1 and for *OR* we chose the four card *OR* protocol discovered through our experiments in section 3.2.3 which is shown in fig. 3.4. We did not implement a nested protocol for the boolean operator *OR*, as it is a simple perm operation, that flips the two cards in a commitment. It is therefore a protocol that only consists of one step and thus does not benefit from being implemented as a nested protocol.

For the finite runtime protocols, we chose the functionally complete set of boolean operators *AND* and *XOR*. For *AND* we chose the five card *AND* protocol by Koch et al., 2021 which is shown in fig. A.1. For *XOR* we included the four card *XOR* protocol by Mizuki and Sone, 2009 that can be seen in fig. A.2.

We also included one more protocol as a nested protocol, which is the *COPY* protocol by Mizuki and Sone, 2009, where it is described as a protocol that can make k copies using $2k + 4$ cards. We chose to implement the protocol for a fixed $k = 1$. Thus the protocol that was implemented is a six card *COPY* protocol with finite runtime.

For our experiments, we chose to test the implementation of our nested structure, by trying to find protocols for two functions, with which we had difficulties finding protocols for with the approach from chapter 3. They are the *COPY* function and the half adder. All the experiments were performed on an AMD Opteron(tm) 6172 CPU at 2.10 GHz with 48 cores and 256 GB of RAM. We used CBMC 5.68 with the built-in solver based on the SAT-solver MiniSat 2.2.1.

Test Results for the *COPY* Function

For the *COPY* function, we included all nested protocols that were implemented, except for the six card *COPY* protocol by Mizuki and Sone (2009). For the runs with five cards the six card *AND* by Mizuki and Sone (2009) was not considered, because it has too many cards. We limited the shuffle set size to 8.

#Cards	#Steps	Security	Protocol	Runtime	Protocols used
5	1	output-possibilistic	×		
5	2	output-possibilistic	×		
5	3	output-possibilistic	×		
5	4	output-possibilistic	×		
5	5	output-possibilistic	×		
6	1	input-possibilistic	✓(see fig. 4.3)	Finite Runtime	6 card *AND* by Mizuki and Sone, 2009
6	2	input-possibilistic	✓	Finite Runtime	4 card *XOR*, 6 card *AND* both by Mizuki and Sone, 2009
6	2	input-possibilistic	✓	Las Vegas	2x 5 Card *AND* by Koch et al. (2021), 6 card *AND* by Mizuki and Sone, 2009

Table 4.1: Protocols that were found through bounded model checking for the *COPY* function. All protocols have closed shuffles only. The permutation set size was limited to 8 for all runs.

As can be observed in table 4.1 we still were not able to find a protocol with closed shuffles and five cards for the *COPY* protocol. For six cards, we already know a protocol for the *COPY* function. It is the six card *COPY* by Mizuki and Sone, 2009 that uses two permutations one shuffle and one turn. Therefore the protocol we found during our experiment for six cards, does not improve on the existing minimal number of cards. As can be seen in fig. 4.3, we could find a protocol for *COPY*, that employs the six card *AND* by Mizuki and Sone, 2009. As the used *AND* protocol also uses two permutations one shuffle and one turn we also did not improve on the number of steps within the protocol.

However our results still provide interesting insights about how an *AND* protocol can be used to copy a given input. This reduces the amount of protocols needed to perform

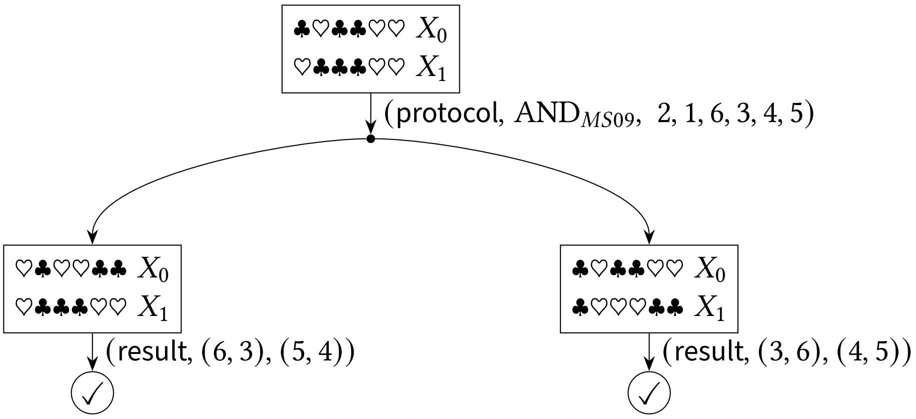


Fig. 4.3: A finite runtime protocol for *COPY* with input-possibilistic security and closed shuffles. It uses 4 cards and an *AND* protocol.

multi-party computation (definition 18). As described in section 2.2.4 both the approaches given by Nishida et al. (2015) and Niemi and Renvall (1998) make use of a functionally complete set of boolean operators and a *COPY* protocol. We can now perform multi-party computation on an arbitrary boolean function with only an *AND* and a *XOR* protocol, reducing the amount of protocols needed. To do so we take the protocol by Nishida et al. (2015) described in section 2.2.4 and replace the *COPY* protocol they use with the protocol from fig. 4.3. The protocol has $2n + 6$ cards for n input commitments.

Test Results for the Half Adder Function

#Cards	#Steps	Perm Set Size	Security	Protocol	Closed?	Runtime
4	1	8	output-possibilistic	×	yes	
4	2	8	output-possibilistic	×	yes	
4	3	8	output-possibilistic	×	yes	
4	4	8	output-possibilistic	×	yes	
4	5	8	output-possibilistic	×	yes	
5	1	8	output-possibilistic	×	yes	
5	2	8	output-possibilistic	×	yes	
5	3	1	output-possibilistic	×	no	
6	1	8	output-possibilistic	×	yes	
6	2	8	output-possibilistic	×	yes	

Table 4.2: Protocols that were found through bounded model checking for the half adder.

As can be observed in table 4.2, we still were not able to find a protocol for the half adder function. We know that there is a protocol for the half adder for ten cards. It is the half adder protocol by Mizuki et al. (2013). However, even when we reduced the permutation set size, we could not reach the necessary number of cards or steps to produce a protocol constructed from the given protocols. This does not mean that there are no protocols for the half adder for the given numbers of cards and steps. Because we limited the permutation set size for all our searches it is theoretically possible that there is a protocol with a bigger permutation set and the same amount of shuffles and turns.

5 Evaluating a Data Structure for Efficient Operations

In this chapter we replace the original data structures implemented with arrays by Koch et al. (2021) described in section 2.4 with an alternative method of implementation, that uses a sequence of bits on which we can perform bitwise operations such as bit shifts. In section 5.1 we describe the implementation of the alternative data structure to represent states and the sequences within. Subsequently we design and execute an experiment in section 5.2 to compare the effect of the two data structures on the runtime of the bounded model checking tool. In sections 5.2.2 and 5.2.3 we present and evaluate the results respectively.

5.1 Integrating the Data Structure into the Symbolic Program

The method of Koch et al. (2021) for finding protocols with the use of bounded model checking (section 2.3) employs a representation of the components of KWH trees (section 2.2.3) within a standardized program. The central aspects of their implementation is described in section 2.4. Here the central data structure that the symbolic program operates on is the representation of the states. A state contains all possible sequences and their probabilities (listing 2.2). Within the implementation by Koch et al. (2021) sequences are represented using arrays (listing 2.3). The shuffles and turns are then performed on these arrays containing the sequences using the standard operations provided by C for arrays (listings 2.5 and 2.6).

In this section we replace the datatype that holds the sequences. Instead of an array we use a single `char` variable. This allows us to use bitwise operations like bit shifts to implement the necessary functionalities of the program, most notably the turn and shuffle operations.

We can implement a card sequence (definition 4) as a sequence of bits. To do so we need to choose a representation for our two symbols ♣ and ♡ as bits. We choose to represent ♣ as a 0 bit and ♡ as a 1 bit.

$$bit = \begin{cases} 0 & \text{if the symbol is a } \clubsuit \\ 1 & \text{if the symbol is a } \heartsuit \end{cases} \tag{5.1}$$

This way the encoding of two bits to form a commitment specified in eq. (2.1) also hold for the bit representation of cards.

$$commitment = \begin{cases} 0 & \text{if we have the bits } \texttt{01} \\ 1 & \text{if we have the bits } \texttt{10} \end{cases} \tag{5.2}$$

Now one sequence can be represented for example by one **`char`** variable. A **`char`** usually has the length of 8 bit. Thus we can represent up to 8 cards in a sequence within one **`char`** variable. For bigger sequences we could use a different data type for example a 32 bit **`int`**, but for this work, 8 bit will be sufficient. If we have sequences with less than 8 cards, we set the remaining bits of the char to 0. For example, if we want to represent the sequence ♣♡♡♣ within a **`char`**, we set the **`char`** to 0000 0101. The leading four zeroes are padding. The remaining four bits represent the cards. Thus, the sequence struct will contain a single **`char`** variable, that will hold all cards (see listing 5.1).

Listing 5.1 Representation of a sequence using the datatype Char

```
struct sequence {
    char val;
    struct fractions probs;
};
```

Using the representation of sequences as chars, we can use the bitwise operations provided by C to execute our operations on sequences. These bitwise operations are shifts and operators like *AND*, *OR*, *XOR* and *NOT*.

First we focus on how the turn operation can be implemented. The goal of this operation is to find the value of the card at a certain position `turnPosition`. Using a right shift, we first shift a **`char`** that has the value 1 to the desired position `turnPosition` (listing 5.2, line 3). Then we calculate the bitwise *AND* of our sequence and the shifted bit (listing 5.2, line 3). The result is equal to zero if the bit at the given index is 0. In that case the card at that index in the sequence is a ♣. If the result is not equal to zero then the bit at the given index is 1. In that case the card at that index in the sequence is a ♡. We can thus set our `turnedCardNumber` accordingly (listing 5.2, line 4).

Listing 5.2 Turning one card at index "turnPosition" in a sequence during the execution of a turn operation using bitwise operations

```
1  char turnedCardNumber = 0;
2  // "true" if != 0 (red card), false if == 0 (black card)
3  if (sequence.val & (1 << turnPosition)) {
4      turnedCardNumber = 1;
5  }
```

If we have for example the sequence 00001001 and we want to know the value of the card at the position 3, we first shift a 1 (0000 0001) by three to the left. This will result in the `char` 0000 1000. We then calculate the bitwise *AND* of the sequence (0000 1001) and the shifted bit (0000 1000). The result of the bitwise *AND* will be 0000 1000. Because 0000 1000 is not equal to 0, we now know that the value of the card at the position 3 is 1 and thus that the card is a ♡. If we instead wanted to know the value of the card at the position 1, we would shift our 1 bit by one to the left and obtain the `char` 0000 0010. If we now calculate the bitwise *AND* of the sequence and the shifted bit we will get the `char` 0000 0000. This `char` is equal to 0 and we thus know that the value of the card at the position 1 is 0 and thus that the card is a ♣.

We can also apply permutations to sequences using bit shifts and other bitwise operations. We iteratively calculate our result and store it as the `char` variable `resultingSeq` (listing 5.3, line 1). First we determine the value of the card at the current index in the same way, as we did for the turn operation (listing 5.3, line 5). Afterwards we determine where we have to shift the card. In order to achieve that, we first determine the value at the right indices in the `permutationSet` (listing 5.3, line 6). These values can be between 0 and $N - 1$ with N being the amount of cards within a sequence. The value n determines that the card should be shifted to position n. So the array `[0,1,2,3]` would be the identity, because it leaves all cards at their original position. To determine how far we have to shift our card, we subtract the position at which the card is currently placed (k) from the permutation value `permutationSet[j][k]` (listing 5.3, line 6). If the resulting value is positive, we shift the card to the left (listing 5.3, line 7). If the resulting value is negative, we shift the card to the right listing 5.3, line 8). After we have performed our shift, we perform a bitwise *XOR* of our result with the resulting sequence (listing 5.3, line 12). This adds the calculated new card onto all the previously calculated cards.

Listing 5.3 Applying permutation j to sequence i during the execution of a shuffle operation using bitwise operations

```
1  char resultingSeq = 0;
2  for (unsigned int k = 0; k < N; k++) {
3      char temp = 0;
4      // Apply permutation j to sequence i.
5      temp = seq.val & (1 << k);
6      int shift = permutationSet[j][k] - k;
7      if (shift >= 0) {
8          temp = temp << (shift);
9      } else {
10         temp = temp >> (-1 * shift);
11     }
12     resultingSeq = resultingSeq | temp;
13 }
```

5.2 Evaluating the Data Structure in an Experiment Setting

In section 5.1 we introduced a new representation of states and sequences based on the encoding of cards as bits. We could then use bitwise operators to perform actions such as turns and shuffles. We now perform an experiment to determine whether our implementation from section 5.1 performs better when we use bounded model checking than the implementation that uses arrays and the standard array operations that we described in section 2.4.

5.2.1 Description of the Experiment Setup

We wrote two symbolic programs executing the same functions. One uses the implementation by Koch et al. (2021) that represents sequences as arrays. The other one uses the implementation proposed in section 5.1 where a sequence is represented by a char. The symbolic programs of the experiment are a partial implementation of the symbolic program described in section 2.4. Their purpose is not to find protocols. Instead they perform a single shuffle on a start state and then check its properties. More precisely, the two symbolic programs first generate the start state such as the program in section 2.4 would do. Then they nondeterministically choose the size of the permutation set and the permutations. Afterwards they apply the chosen permutations to the start state and calculate the resulting sequences and resulting probabilities. Then they call a function that can check for different properties. For our experiment, we aimed at finding a permutation set, that results in all probabilities being not equal to zero. That means that we have to check that for every sequence every probability is greater than zero. This property ensures, that the permutation sets chosen are not too small. For four cards and input-possibilistic security we need a permutation set size of at least five. The full experiment implementation can be found in appendix A.6 and the GitHub repository (appendix A.4).

5.2.2 Experiment Execution and Results

All the experiments were performed on an AMD Opteron(tm) 6172 CPU at 2.10 GHz with 48 cores and 256 GB of RAM. We used CBMC 5.68 with the built-in solver based on the SAT-solver MiniSat 2.2.1.

For both programs, we ran the same experiments. The programs were run for each input possibilistic security and output possibilistic security. For each of these security types, we executed the program for 4 cards, 5 cards and 6 cards respectively. Together this resulted in six different test cases for each the array test program and the bit shift test program. Each of these experiments was performed a total of 5 times, except for the experiments for 5 cards and input possibilistic security, which were only performed 2 times.

#Cards	Security		Array Operations	Bitwise Operations
4	Output Poss.	variables	2 525 596	2 333 279
		clauses	8 167 087	7 256 704
		complete time (s)	382,3	113,3
4	Input Poss.	variables	3 623 257	3 432 778
		clauses	11 633 860	10 723 999
		complete time (s)	1 064,3	521,7
5	Output Poss.	variables	34 216 676	27 384 088
		clauses	123 856 065	95 341 873
		complete time (s)	8 317,7	3 767
5	Input Poss.	variables	47 451 857	40 622 131
		clauses	170 267 718	141 754 048
		complete time	126 316	6 979,5
6	Output Poss.	out of memory after (s)	7 448	3 070
6	Input Poss.	out of memory after (s)	10 098	4 755,7

Table 5.1: The results of the experiments for both the implementation using arrays and operations and arrays, as well as the implementation using `chars` and bitwise operations.

For the experiments that used 4 or 5 cards, the program ended when the SAT solver found the formula to be satisfiable and returned the program trace. We measured the number of clauses and variables that were generated by CBMC and that the SAT solver had to solve. They were the same for each execution of the same experiment. Therefore the table shows the exact amount of clauses and variables used in each experiment. Apart from that we also determined the time that it took the program to execute. Measured from the start of the execution of `runBitShiftTest.sh` up to the end of the execution. These times varied slightly from run to run. For the experiments that we performed five times, we excluded the highest and lowest runtime. The results in table 5.1 are the arithmetic mean of the remaining measured times.

For the experiments that used 6 cards, the SAT solver ran out of memory. Therefore it did not return the amount of clauses and variables. We once again determined the time that it took the program to execute. Measured from the start of the execution of `runBitShiftTest.sh` up to the end of the execution after the program was terminated due to an out of memory error. For the experiments that we performed five times, we excluded the highest and lowest runtime and calculated the arithmetic mean of the remaining values. Please refer to appendix A.1 for the complete experiment results.

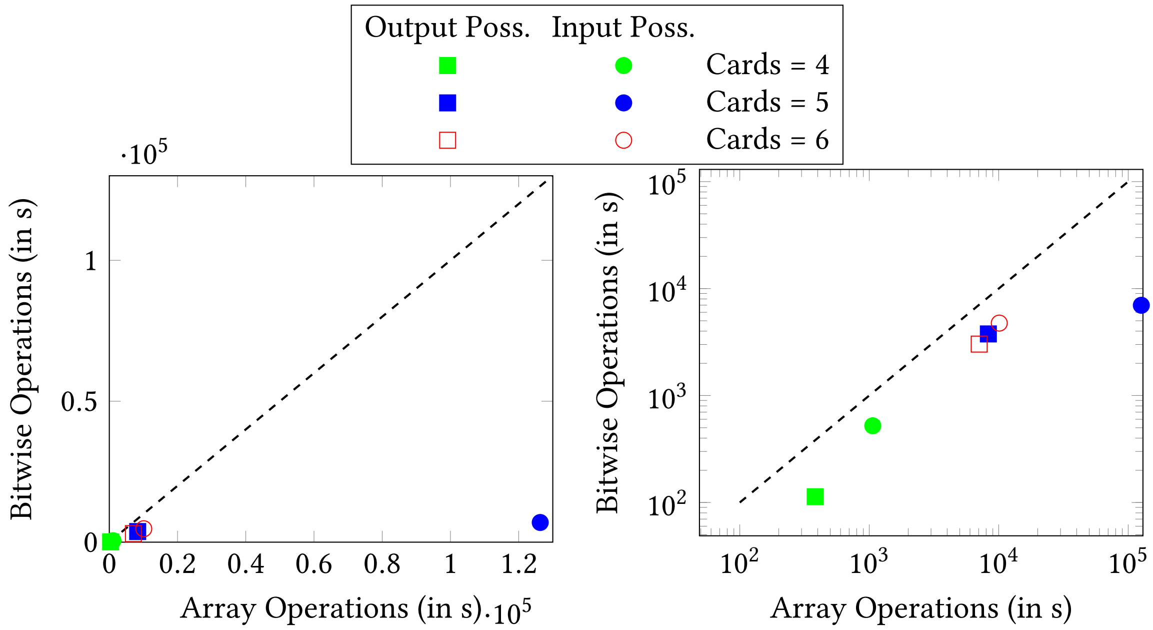


Fig. 5.1: Scatter plots with both a linear and a logarithmic scale. It shows the runtimes for the results from table 5.1 for both the implementation using arrays and operations and arrays, as well as the implementation using `chars` and bitwise operations. For the filled marks, we took the complete time from table 5.1. For the marks that are empty inside we took the time until the "out-of-memory" error from table 5.1.

5.2.3 Discussion of the Experiment Results

The results from our experiments showed, that the implementation using `chars` and bitwise operations was faster than the implementation using arrays. In every experiment performed the time it takes for the bounded model checking tool to produce a solution was less for `chars` and bitwise operations. The highest speedup was observed in the experiment with 5 cards and input possibilistic security. The implementation using bit shifts proved to be about 18 times faster than the implementation employing arrays. The experiment with 5 cards and input possibilistic security, is also the experiment that produces the largest SAT formula and it has the longest running time for both the array implementation as well as the bit shift implementation out of all the tested inputs.

For the experiment with 6 cards, the bounded model checker did not produce a result but ran out of memory instead. As with the experiments where the bounded model checker produced a solution, the implementation employing bit shifts, terminated faster than the implementation using arrays for finding protocols. The advantage of producing an "out-of-memory" error more quickly could be that it will allow for a quicker confirmation that the program is too big and/or complex for the bounded model checker.

If we compare the size of the formula for the implementation with arrays and the implementations with `chars` in table 5.1, we can see that there is not a big difference between them. Both the sets of variables and clauses are slightly smaller for the implementations using `chars` throughout all experiments. They are however in the same order of magnitude. The largest deviation is 30% which can be observed for the experiment with 5 cards and output possibilistic security. Generally the deviation is greater for experiments with output possibilistic security than for experiments with input possibilistic security. The deviation is also higher for 5 cards, than it is for 4 cards. However, as the difference in the size of the formula is not proportional to the difference in runtimes when comparing the two data structures, we believe that the size of the formula is not the cause for the different runtimes.

We have demonstrated that implementing sequences as chars and performing bitwise operations such as bit shifts, have the potential of outperforming implementations employing arrays. We were however not able to obtain results for experiments that run out of memory for implementations employing arrays, as they ran out of memory as well for implementations using bitwise operations.

6 Conclusion

6.1 Summary

We were able to successfully generalize the symbolic program by Koch et al., 2021 to find protocols for any boolean function. We applied it to five different functions, to find new protocols for them. For the *OR* function we were able to find two four card Las Vegas protocols with probabilistic security. One of them had uniform but not closed shuffles, the other one had closed but non-uniform shuffles. For the *COPY* function we were able to find a five card Las Vegas protocol with probabilistic security and not closed but uniform shuffles. However we were not able to find protocols for the half adder function, as the bounded model checker would either take too long or there would be an "out-of-memory" error. We tested the use of different SAT solvers for the use within the bounded model checker. However neither CaDiCal [1] nor Glucose [2] performed better than the built-in SAT solver MiniSat for our application. We therefore were not able to increase the number of cards or steps for which we could obtain protocols by changing out the SAT solver. A remaining problem of the presented method that uses bounded model checking to find card based protocols is therefore, that it is only effective for small numbers of cards and steps.

We introduced and defined a new action that applied protocols to a state. We then presented a method of inserting this new protocol action into the symbolic program. With that we were able to use protocols as an operation when finding new protocols. Subsequently we were then able to implement protocols from literature and that were found in section 3.2 as operations into the symbolic program. With that we were able to apply our implementation to a *COPY* protocol and find a protocol. This protocol for the *COPY* function made use of an *AND* protocol. Using this protocol we were able to show, that we could compute any boolean function by using only two protocols. However we were still not able to find a protocol for the half adder.

We introduced a new representation of sequences and states where cards are stored as bits inside a single variable. We were able to show that it is possible to implement the operations within the symbolic program, most notably the turn and shuffle operation, by using the new data structure and bitwise operators. We performed an experiment

[1] http://fmv.jku.at/cadical/

[2] https://www.labri.fr/perso/lsimon/research/glucose/

comparing our new data structure for the representation of states with the original data structure. Our experiments indicated that our new representation improved the runtime of the verification process. However, our tests were too limited to make a concrete statement on whether different data structures actually allow us to find protocols more effectively. Additionally our new data structure did not reduce the occurrence of "out-of-memory" errors.

6.2 Outlook

A pending task is a complete implementation of the standardized program using the new data structure for sequences to definitely judge whether the runtime improvements translate to programs that find full protocols. A large part of methods needed are already implemented in the program from section 5.2.2. However there are still some methods missing.

Another remaining question is how to reduce the occurrence of the "out-of-memory" error. As we have analyzed in section 3.2.4 it occurred during or directly after the conversion into static single assignment (SSA) form. As we have shown neither using CaDiCal [3] nor Glucose [4] offered any improvements. However there could be a SAT or SMT solver that performs better for our implementations, so more tests using different solvers could be performed.

[3] http://fmv.jku.at/cadical/

[4] https://www.labri.fr/perso/lsimon/research/glucose/

Bibliography

Koch, Alexander (2019). “Cryptographic Protocols from Physical Assumptions”. PhD thesis. Karlsruhe Institute of Technology, Germany.

Koch, Alexander, Michael Schrempp, and Michael Kirsten (2021). “Card-Based Cryptography Meets Formal Verification”. In: *New Gener. Comput.* 39.1, pp. 115–158. DOI: 10.1007/s00354-020-00120-0.

Koch, Alexander, Stefan Walzer, and Kevin Härtel (2015). “Card-Based Cryptographic Protocols Using a Minimal Number of Cards”. In: *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I.* Ed. by Tetsu Iwata and Jung Hee Cheon. Vol. 9452. Lecture Notes in Computer Science. Springer, pp. 783–807. DOI: 10.1007/978-3-662-48797-6_32.

Kroening, Daniel and Michael Tautschnig (2014). “CBMC - C Bounded Model Checker - (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings.* Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, pp. 389–391. DOI: 10.1007/978-3-642-54862-8_26.

Mizuki, Takaaki, Isaac Kobina Asiedu, and Hideaki Sone (2013). “Voting with a Logarithmic Number of Cards”. In: *Unconventional Computation and Natural Computation - 12th International Conference, UCNC 2013, Milan, Italy, July 1-5, 2013. Proceedings.* Ed. by Giancarlo Mauri et al. Vol. 7956. Lecture Notes in Computer Science. Springer, pp. 162–173. DOI: 10.1007/978-3-642-39074-6_16.

Mizuki, Takaaki and Hideaki Sone (2009). “Six-Card Secure AND and Four-Card Secure XOR”. In: *Frontiers in Algorithmics, Third International Workshop, FAW 2009, Hefei, China, June 20-23, 2009. Proceedings.* Ed. by Xiaotie Deng, John E. Hopcroft, and Jinyun Xue. Vol. 5598. Lecture Notes in Computer Science. Springer, pp. 358–369. DOI: 10.1007/978-3-642-02270-8_36.

Mizuki, Takaaki, Fumishige Uchiike, and Hideaki Sone (2006). “Securely computing XOR with 10 cards”. In: *Australas. J Comb.* 36, pp. 279–294.

Niemi, Valtteri and Ari Renvall (Jan. 1998). “Secure multiparty computations without computers”. en. In: *Theoretical Computer Science* 191.1, pp. 173–183. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(97)00107-2. URL: https://www.sciencedirect.com/science/article/pii/S0304397597001072 (visited on 09/04/2022).

Nishida, Takuya, Takaaki Mizuki, and Hideaki Sone (2013). “Securely Computing the Three-Input Majority Function with Eight Cards”. In: *Theory and Practice of Natural Computing - Second International Conference, TPNC 2013, Cáceres, Spain, December 3-5,* *2013, Proceedings*. Ed. by Adrian-Horia Dediu et al. Vol. 8273. Lecture Notes in Computer Science. Springer, pp. 193–204. DOI: 10.1007/978-3-642-45008-2_16.

Nishida, Takuya et al. (2015). "Card-Based Protocols for Any Boolean Function". In: *Theory and Applications of Models of Computation - 12th Annual Conference, TAMC 2015, Singapore, May 18-20, 2015, Proceedings*. Ed. by Rahul Jain, Sanjay Jain, and Frank Stephan. Vol. 9076. Lecture Notes in Computer Science. Springer, pp. 110–121. DOI: 10.1007/978-3-319-17142-5_11.

Rastogi, Aseem, Nikhil Swamy, and Michael Hicks (2019). "Wys*: A DSL for Verified Secure Multi-party Computations". In: *Principles of Security and Trust - 8th International Conference, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by Flemming Nielson and David Sands. Vol. 11426. Lecture Notes in Computer Science. Springer, pp. 99–122. DOI: 10.1007/978-3-030-17138-4_5.

Saito, Takahiro et al. (2020). "How to Implement a Non-uniform or Non-closed Shuffle". In: *Theory and Practice of Natural Computing - 9th International Conference, TPNC 2020, Taoyuan, Taiwan, December 7-9, 2020, Proceedings*. Vol. 12494. Lecture Notes in Computer Science. Springer, pp. 107–118. DOI: 10.1007/978-3-030-63000-3_9.

Sasao, Tsutomu (1999). "Logic Functions and Their Representations". en. In: *Switching Theory for Logic Synthesis*. Ed. by Tsutomu Sasao. Boston, MA: Springer US, pp. 35–61. ISBN: 978-1-4615-5139-3. DOI: 10.1007/978-1-4615-5139-3_3. (Visited on 02/23/2023).

Stiglic, Anton (2001). "Computations with a deck of cards". In: *Theor. Comput. Sci.* 259.1-2, pp. 671–678. DOI: 10.1016/S0304-3975(00)00409-6.

A Appendix

A.1 Full Experiment Results

A.1.1 Test Results for the Comparison of the different Data Structures

Protocol								Average
N = 4 SEC = 2	Arrays	Date	4.1.2023 15/21/00	11.1.2023 12/31/21	11.1.2023 12/24/35	11.1.2023 12/38/50	11.1.2023 12/16/51	
		variables	2525596	2525596	2525596	2525596	2525596	
		clauses	8167087	8167087	8167087	8167087	8167087	
		complete time	382	381	381	384	384	382.4
	BitShifts	Date	4.1.2023 12/06/21	11.1.2023 12/51/10	11.1.2023 12/48/75	11.1.2023 12/46/41	11.1.2023 16/23/56	
		variables	2333279	2333279	2333279	2333279	2333279	
		clauses	7256704	7256704	7256704	7256704	7256704	
		complete time	113	114	113	113	114	113.4
N = 4 SEC = 1	Arrays	Date	12.1.2023 15/41/43	12.1.2023 16/04/37	12.1.2023 16/57/05	12.1.2023 17/27/29	31.1.2023 119/25/59	
		variables	3623257	3623257	3623257	3623257	3623257	
		clauses	11633860	11633860	11633860	11633860	11633860	
		complete time	1059	1068	1066	1072	215	896
	BitShifts	Date	12.11.2023 12/07/16	12.11.2023 12/15/59	12.1.2023 12/24/25	12.1.2023 12/38/03	12.1.2023 12/47/11	
		variables	3432778	3432778	3432778	3432778	3432778	
		clauses	10723999	10723999	10723999	10723999	10723999	
		complete time	482	487	589	489	491	507.6
N= 5 SEC = 2	Arrays	Date	4.1.2023 15/27/46	12.1.2023 12/59/00	12.1.2023 23/56/39	13.1.2023 10/45/17	13.1.2023 14/05/25	
		variables	34216676	34216676	34216676	34216676	34216676	
		clauses	123856065	123856065	123856065	123856065	123856065	
		complete time	6954	6650	10432	7758	10241	8407
	BitShifts	Date	4.1.2023 12/10/22	11.1.2023 12/53/18	31.01.2023 12/35/07	31.012023 13/47/06	31.1.2023 14/42/27	
		variables	27384088	27384088	27384088	27384088	27384088	
		clauses	95341873	95341873	95341873	95341873	95341873	
		complete time	3499	3483	4319	3320	4920	3908.2
N = 5 SEC = 1	Arrays	Date	26.01.2023 11/06/57	28/01/2023 07/02/17				
		variables	47451857	47451857				
		clauses	170267718	170267718				
		complete time	158120	94512				126316
	BitShifts	Date	31.01.2023 12/28/10	31.1.2023 14/31/54				
		variables	40622131	40622131				
		clauses	141754048	141754048				
		complete time	7424	6535				6979.5
N = 6 SEC = 2	Arrays	Date	05.01.2023 13/15/45	31.1.2023 06/04/08	31.1.2023 04/07/32	31.1.2023 10/31/32	31.1.2023 19/29/34	
		out of memory after	7158	9138	6996	6906	7042	7448
	Bit Shifts	Date	04.1.2023 22/48/02	31.1.2023 00/42/09	31.1.2023 01/31/46	31.1.2023 01/17/26	31.01.2023 03/17/26	
		out of memory after	3028	2977	3296	3043	3006	3070
N = 6 SEC = 1	Arrays	Date	30.01.2023 09/47/30	30.1.2023 19/19/49	30,.1.2023 13/36/13	30.1.2023 16/26/34	30.1.2023 22/00/50	
		out of memory after	13723	9660	10221	10395	9679	26839
	BitShifts	Date	31.1.2023 21/26/56	31.1.2023 22/46/49	31.1.2023 19/20/28	31.1.2023 20/38/02	31.1.2023 21/58/27	
		out of memory after	4793	4820	4654	4825	4634	4745.2

A.1.2 Test Results for the Comparaison of Different SAT Solvers

Protocol	Used SAT Solver		Average	Test #1	Test #2	Test #3	Test #4	Test #5
XOR, N=4, L=2, SEC = 2	Mini Sat	Date		2023_02_02_11_36_29	2023_02_02_11_42_35	2023_02_02_11_48_43	2023_02_02_11_37_17	2023_02_02_11_43_59
		Total Time	372.8	366	368	368	402	360
		Variables	10829386					
		Clauses	39863588					
	Glucose	Date		2023_02_01_20_14_02	2023_02_01_20_23_35	2023_02_01_20_33_09	2023_02_01_20_15_35	2023_02_01_20_24_55
		Total Time	568.6	573	573	576	560	561
		Variables	10829386					
		Clauses	39863588					
	Cadical	Date		2023_02_03_10_44_28	2023_02_03_10_56_33	2023_02_03_11_08_44	2023_02_03_10_45_15	2023_02_03_10_57_16
		Total Time	725.2	725	731	729	721	720
		Variables	10829386					
		Clauses	39905499					
XOR, N=4, L=2, SEC = 1	Mini Sat	Date		2023_02_02_11_54_51	2023_02_02_12_05_15	2023_02_02_11_49_59	2023_02_02_12_00_16	2023_02_02_12_10_28
		Total Time	618.8	623	628	617	612	614
		Variables	13191171					
		Clauses	47693620					
	Glucose	Date		2023_02_01_20_42_45	2023_02_01_21_08_54	2023_02_01_20_34_16	2023_02_01_20_59_54	2023_02_01_21_42_42
		Total Time	1757.4	1569	1570	1538	2568	1542
		Variables	13191171					
		Clauses	47693620					
	Cadical	Date		2023_02_03_11_20_53	2023_02_03_11_51_34	2023_02_03_11_09_16	2023_02_03_11_39_10	2023_02_03_12_13_55
		Total Time	1867.4	1841	1836	1794	2085	1781
		Variables	13191171					
		Clauses	47736299					
XOR, N=4, L=2, SEC = 0	Mini Sat	Date		2023_02_02_12_15_43	2023_02_02_12_49_45	2023_02_02_13_23_39	2023_02_02_12_20_42	2023_02_02_13_09_18
		Total Time	2203.4	2042	2034	2039	2916	1986
		Variables	18192251					
		Clauses	73827798					
	Glucose	Date		2023_02_01_21_35_04	2023_02_01_23_31_33	2023_02_02_00_58_19	2023_02_01_22_08_24	2023_02_01_23_30_45
		Total Time	5417.4	6989	5206	5005	4941	4946
		Variables	18192251					
		Clauses	73827798					
	Cadical	Date		2023_02_03_12_22_10	2023_02_03_14_38_31	2023_02_03_16_35_59	2023_02_03_12_43_36	2023_02_03_14_39_12
		Total Time	7731.2	8180	7048	8917	6936	7575
		Variables	18192251					
		Clauses	73870727					
OR, N=4, L=2, SEC = 2	Mini Sat	Date		2023_02_02_13_57_38	2023_02_02_14_05_44	2023_02_02_13_42_24	2023_02_02_13_50_23	2023_02_02_13_58_23
		Total Time	481.6	486	485	479	479	479
		Variables	10819303					
		Clauses	39832637					
	Glucose	Date		2023_02_02_02_21_45	2023_02_02_02_39_26	2023_02_02_00_53_11	2023_02_02_01_10_59	2023_02_02_01_28_35
		Total Time	1065	1061	1081	1068	1056	1059
		Variables	10819303					
		Clauses	39832637					
	Cadical	Date		2023_02_03_19_04_37	2023_02_03_20_07_57	2023_02_03_16_45_27	2023_02_03_17_45_44	2023_02_03_18_43_51
		Total Time	3444	3800	3308	3617	3487	3008
		Variables	10819303					
		Clauses	39874504					
OR, N=4, L=2, SEC = 1	Mini Sat	Date		2023_02_02_14_13_49	2023_02_02_14_34_08	2023_02_02_14_54_28	2023_02_02_14_06_22	2023_02_02_14_31_49
		Total Time	1277	1218	1219	1218	1527	1203
		Variables	13185883					
		Clauses	47673732					
	Glucose	Date		2023_02_02_02_57_27	2023_02_02_03_19_44	2023_02_02_03_42_11	2023_02_02_01_46_14	2023_02_02_02_08_07
		Total Time	1332.8	1347	1347	1345	1313	1312
		Variables	13185883					
		Clauses	47673732					
	Cadical	Date		2023_02_03_21_03_05	2023_02_03_21_43_32	2023_02_03_22_19_42	2023_02_03_19_33_59	2023_02_03_20_09_35
		Total Time	2207.8	2427	2170	2169	2136	2137
		Variables	13185883					
		Clauses	47716387					
OR, N=4, L=2, SEC = 0	Mini Sat	Date		2023_02_02_15_14_46	2023_02_02_15_47_00	2023_02_02_14_51_52	2023_02_02_15_23_32	2023_02_02_15_55_09
		Total Time	1919.4	1934	1946	1900	1897	1920
		Variables	18186963					
		Clauses	73807910					
	Glucose	Date		2023_02_02_04_04_36	2023_02_02_05_15_41	2023_02_02_02_29_59	2023_02_02_04_30_10	2023_02_02_05_39_40
		Total Time	5398.4	4265	4932	7209	4170	6416
		Variables	18186963					
		Clauses	73807910					
	Cadical	Date		2023_02_03_22_55_51	2023_02_04_00_22_08	2023_02_03_20_45_12	2023_02_03_22_02_3	2023_02_03_23_20_0
		Total Time	4754	5177	4728	4647	4644	4574
		Variables	18186963					
		Clauses	73850815					
OR N=5, L=2, SEC = 2	Mini Sat	Date		2023_02_02_16_19_26	2023_02_02_16_49_26	2023_02_02_17_20_32	2023_02_02_16_27_09	2023_02_02_16_56_51
		out of memory	2298.4	1799	1863	4251	1780	1799
	Glucose	Date		2023_02_02_06_37_53	2023_02_02_07_05_23	2023_02_02_07_32_40	2023_02_02_07_26_36	2023_02_02_07_53_46
		out of memory	1638	1648	1636	1650	1629	1627
	Cadical	Date		2023_02_04_01_40_56	2023_02_04_02_26_37	2023_02_04_03_16_54	2023_02_04_00_36_17	2023_02_04_01_21_12
		out of memory	2798.4	2740	3014	2769	2694	2775

A.2 Card Protocols from Literature

The KWH trees of the five card AND protocol by Koch et al. (2021) and the four card XOR protocol by Mizuki and Sone (2009).

A.2.1 Five card AND Protocol

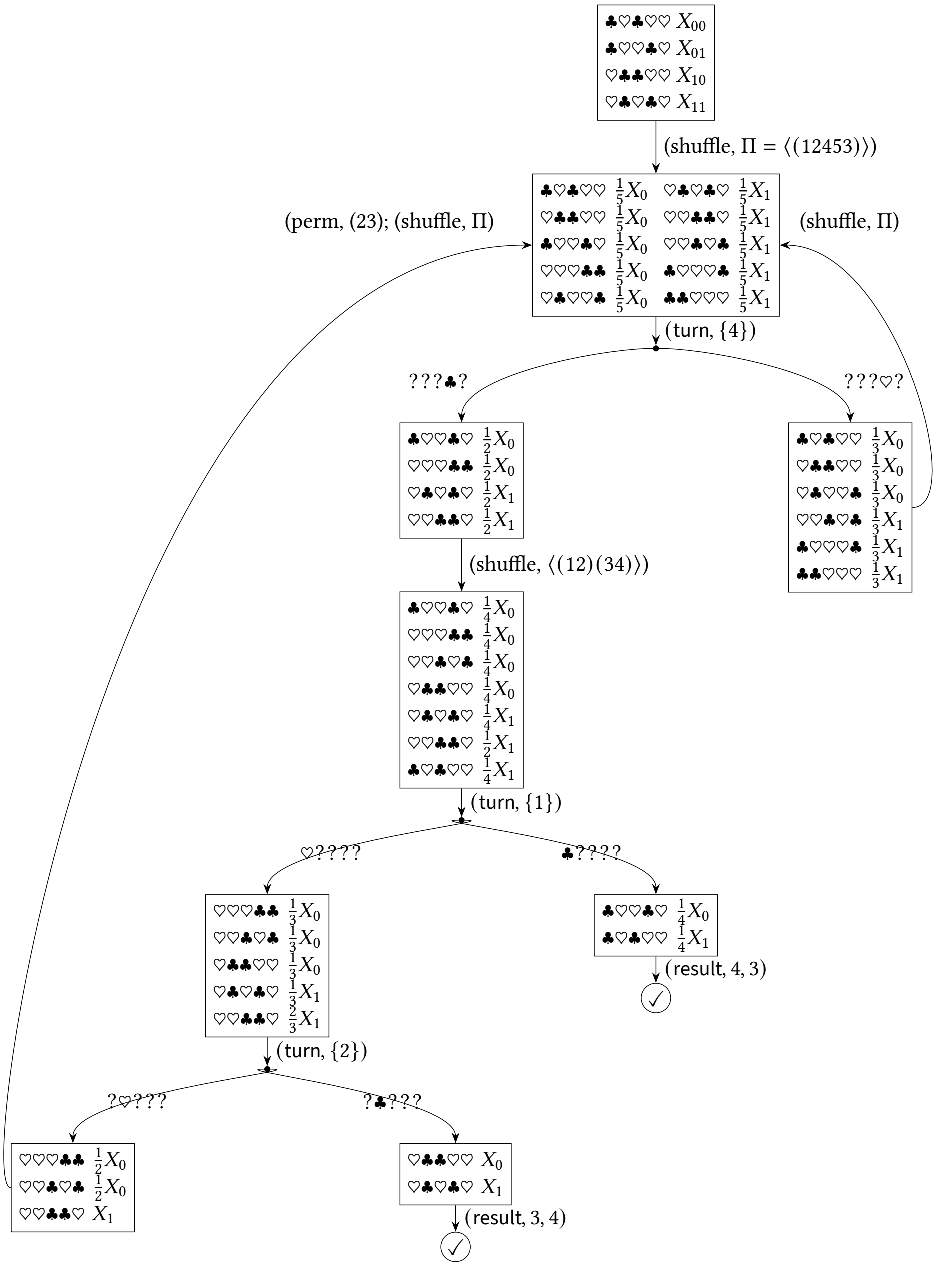


Fig. A.1: Five card AND protocol by Koch et al. (2021)

A.2.2 Four Card XOR Protocol

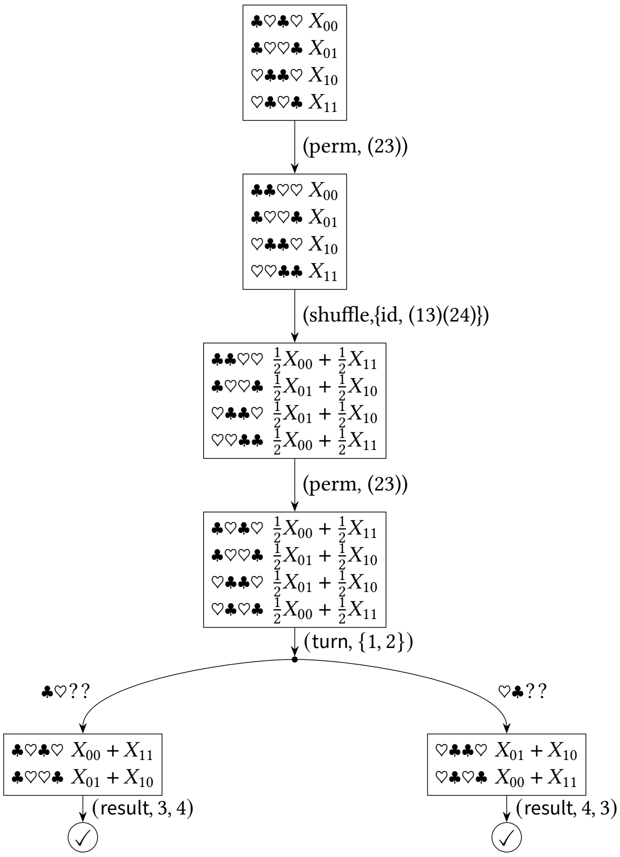

♣♡♣♡ X_{00}
♣♡♡♣ X_{01}
♡♣♣♡ X_{10}
♡♣♡♣ X_{11}
(perm, (23))
♣♣♡♡ X_{00}
♣♡♡♣ X_{01}
♡♣♣♡ X_{10}
♡♡♣♣ X_{11}
(shuffle,{id, (13)(24)})
♣♣♡♡ $\frac{1}{2}X_{00} + \frac{1}{2}X_{11}$
♣♡♡♣ $\frac{1}{2}X_{01} + \frac{1}{2}X_{10}$
♡♣♣♡ $\frac{1}{2}X_{01} + \frac{1}{2}X_{10}$
♡♡♣♣ $\frac{1}{2}X_{00} + \frac{1}{2}X_{11}$
(perm, (23))
♣♡♣♡ $\frac{1}{2}X_{00} + \frac{1}{2}X_{11}$
♣♡♡♣ $\frac{1}{2}X_{01} + \frac{1}{2}X_{10}$
♡♣♣♡ $\frac{1}{2}X_{01} + \frac{1}{2}X_{10}$
♡♣♡♣ $\frac{1}{2}X_{00} + \frac{1}{2}X_{11}$
(turn, {1, 2})
♣♡??
♡♣??
♣♡♣♡ $X_{00} + X_{11}$
♣♡♡♣ $X_{01} + X_{10}$
♡♣♣♡ $X_{01} + X_{10}$
♡♣♡♣ $X_{00} + X_{11}$
(result, 3, 4)
(result, 4, 3)
✓
✓

Fig. A.2: Four card XOR protocol by Mizuki and Sone (2009)

A.3 Additional Protocols found through Bounded Model Checking

In section 3.2.3 we found two protocols that are shown in figs. A.3 and A.4. We completed them with restart operations making their runtime restarting las Vegas. However we do not rule out that there is a way of completing the them to be a restart-free Las Vegas protocol or even a Finite Runtime protocol.

A.3.1 Five Card OR Protocol with input possibilistic security and non-closed shuffles.

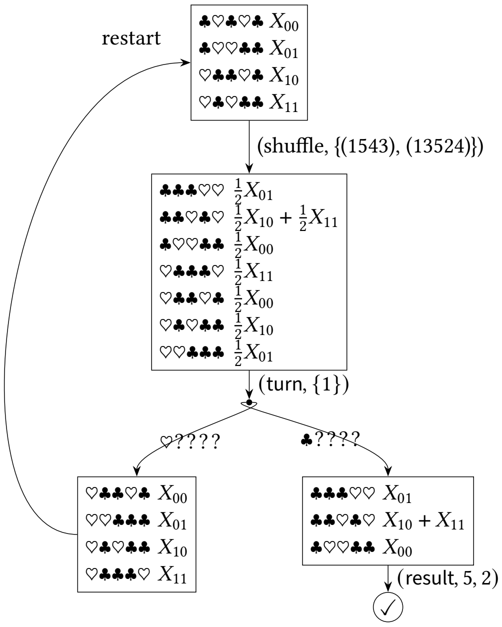

♣♡♣♡♣ X_{00}
♣♡♡♣♣ X_{01}
♡♣♣♡♣ X_{10}
♡♣♡♣♣ X_{11}
restart
(shuffle, {(1543), (13524)})
♣♣♣♡♡ $\frac{1}{2}X_{01}$
♣♣♡♣♡ $\frac{1}{2}X_{10} + \frac{1}{2}X_{11}$
♣♡♡♣♣ $\frac{1}{2}X_{00}$
♡♣♣♣♡ $\frac{1}{2}X_{11}$
♡♣♣♡♣ $\frac{1}{2}X_{00}$
♡♣♡♣♣ $\frac{1}{2}X_{10}$
♡♡♣♣♣ $\frac{1}{2}X_{01}$
(turn, {1})
♡????
♣????
♡♣♣♡♣ X_{00}
♡♡♣♣♣ X_{01}
♡♣♡♣♣ X_{10}
♡♣♣♣♡ X_{11}
♣♣♣♡♡ X_{01}
♣♣♡♣♡ $X_{10} + X_{11}$
♣♡♡♣♣ X_{00}
(result, 5, 2)
✓

Fig. A.3: A restarting Las Vegas protocol for OR with input-possibilistic security. The shuffles are not closed. It uses 5 cards and has a best case of 2 steps.

A.3.2 Input-possibilistic protocol with closed shuffles for OR using 4 cards and 5 steps

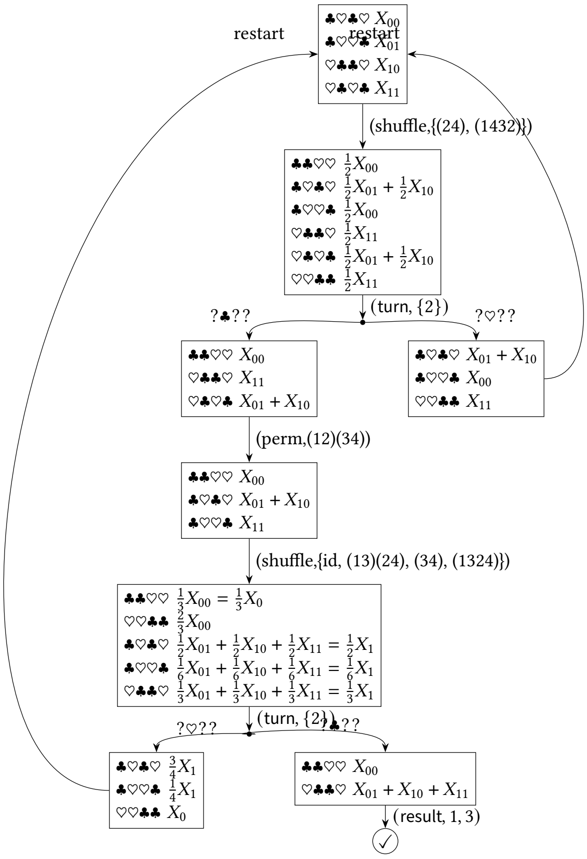


Fig. A.4: A restarting Las Vegas protocol for OR with input-possibilistic security. The shuffles are not closed. It uses 4 cards and has a best case of 5 steps.

A.4 Code Repository

All code written and used for this thesis can be found in the GitHub repository.

A.5 Code Excerpts from the Adapted Standardized Program Representations.

The full implementation of the subsequent programs can be found in the GitHub repository (appendix A.4

A.5.1 XOR

Listing A.1 Excerpt from the preprocessor variables

```
#ifndef COMMIT
#define COMMIT 4
#endif

#ifndef NUMBER_START_SEQS
#define NUMBER_START_SEQS 4
#endif

#if WEAK_SECURITY == 2
    #define NUMBER_PROBABILITIES 2
#else
    #define NUMBER_PROBABILITIES 4
#endif
```

Listing A.2 Test for bottom sequence.

```
unsigned int isBottom(struct fractions probs) {
    unsigned int bottom = 0;

    if (WEAK_SECURITY == 2) {
        bottom = probs.frac[0].num && probs.frac[1].num;
    } else {
        bottom = (probs.frac[0].num || probs.frac[3].num) && (probs.
            frac[1].num || probs.frac[2].num);
    }
    return bottom;
}
```

Listing A.3 Excerpt from the function isFinalState(). It checks whether a given state can be an end state. We only look at the assignment of deciding here, because it is the only thing that is changed from the original Implementation for AND

```
unsigned int isFinalState(struct state s) {
if (WEAK_SECURITY == 2) {
    deciding = (s.seq[i].probs.frac[1].num);
} else {
    deciding = (s.seq[i].probs.frac[1].num) || (s.seq[i].probs.frac
        [2].num);
}
```

A.5.2 OR

Preprocessor variable same as for appendix A.5.1.

Listing A.4 Test for bottom sequence

```
unsigned int isBottom(struct fractions probs) {
    unsigned int bottom = 0;

    if (WEAK_SECURITY == 2) {
        bottom = probs.frac[0].num && probs.frac[1].num;
    } else {
        bottom = (probs.frac[1].num || probs.frac[2].num || probs.
            frac[3].num) && probs.frac[0].num;
    }
    return bottom;
}
```

Listing A.5 Excerpt from the function isFinalState(). It checks whether a given state can be an end state. We only look at the assignment of deciding here, because it is the only thing that is changed from the original Implementation for AND

```
unsigned int isFinalState(struct state s) {
unsigned int deciding = !(s.seq[i].probs.frac[0].num);
```

A.5.3 COPY

Listing A.6 Excerpt from the preprocessor variables

```
* COPY:
* for  COPY we need a 2 cards commitment not 4
*/
#ifndef COMMIT
#define COMMIT 2
#endif

/**
* COPY:
* for COPY we have 2 start sequences 1 and 0
*/
#ifndef NUMBER_START_SEQS
#define NUMBER_START_SEQS 2
#endif

#if WEAK_SECURITY == 2
    #define NUMBER_PROBABILITIES 2
#else
    #define NUMBER_PROBABILITIES 4
#endif
```

Listing A.7 Test for bottom sequence

```
    unsigned int bottom = 0;
    /**
    * COPY:
    * we only have the probabilities/possibilities:
    * X_0 if the input was a 0 (the output will also be a 0)
    * X_1 if the input was a 1
    * if both are != 0 then we have a bottom sequence
    */
    bottom = probs.frac[0].num && probs.frac[1].num;
    return bottom;
}
```

Listing A.8 The function isFinalState() checks whether a given state can be an end state.

```
unsigned int isFinalState(struct state s) {
    unsigned int res = 0;

    if (isValid(s)) { // Non-valid states cannot be final.
        unsigned int a = nondet_uint(); // Index of the first card.
        unsigned int b = nondet_uint(); // Index of the second card.

        assume (a < N && b < N && a != b);
        unsigned int lowerCard = 0;
        unsigned int higherCard = 0;

        unsigned int c = nondet_uint(); // Index of the first card.
        unsigned int d = nondet_uint(); // Index of the second card.

        assume(c < N&& d < N&& c != d);
        assume(a != c && a != d);
        assume(b != c && b != d);

        unsigned int done = 0;
        for (unsigned int i = 0; i < NUMBER_POSSIBLE_SEQUENCES; i++)
             {
            if (!done && isStillPossible(s.seq[i].probs)) {
                unsigned int deciding = s.seq[i].probs.frac[
                    NUMBER_PROBABILITIES - 1].num;
                unsigned int first = s.seq[i].val[a];
                unsigned int second = s.seq[i].val[b];
                unsigned int third = s.seq[i].val[c];
                unsigned int fourth = s.seq[i].val[d];
                assume (first != second && third != fourth);
                assume(first == third);
                assume(second == fourth);
                if (!higherCard && !lowerCard) {
                    higherCard = deciding ? first : second;
                    lowerCard = deciding ? second : first;
                } else {
                    if (   (deciding
                            && !(   first == higherCard
                                 && second == lowerCard))
                        || (!deciding
                            && !(   second == higherCard
                                 && first == lowerCard))) {
                        done = 1;
                        res = 0;
                    }
                }
            }
        }
    }
    return res;
}
```

A.5.4 Half Adder

Listing A.9 Excerpt from the preprocessor variables

```
#ifndef COMMIT
#define COMMIT 4
#endif

#ifndef NUMBER_START_SEQS
#define NUMBER_START_SEQS 4
#endif

/**
* ADDER: IMPORTANT, we have three possible outputs we need to
   distinguish
*/
#if WEAK_SECURITY == 2
#define NUMBER_PROBABILITIES 3
#else
#define NUMBER_PROBABILITIES 4
#endif
```

Listing A.10 Test for bottom sequence

```
unsigned int isBottom(struct fractions probs) {
    unsigned int bottom = 0;

    if (WEAK_SECURITY == 2) {
        bottom = (probs.frac[0].num && probs.frac[1].num) || (probs.
           frac[1].num && probs.frac[2].num) || (probs.frac[2].num
           && probs.frac[0].num);
    }
    else {
        bottom = ((probs.frac[1].num || probs.frac[2].num) && (probs
           .frac[0].num || probs.frac[3].num)) || (probs.frac[0].num
            && probs.frac[3].num);
    }
    return bottom;
}
```

Listing A.11 The function isFinalState() checks whether a given state can be an end state.

```
unsigned int isFinalState(struct state s) {
    unsigned int res = 0;

    if (isValid(s)) { // Non-valid states cannot be final.
        res = 1;
         //SUM
        unsigned int a = nondet_uint(); // Index of the first card -> sum
        unsigned int b = nondet_uint(); // Index of the second card -> sum
        //Carry
        unsigned int c = nondet_uint(); // Index of the third card -> carry
        unsigned int d = nondet_uint(); // Index of the fourth card -> carry

        assume(a < N&& b < N&& a != b);
        assume(c < N&& d < N&& c != d);
        assume(a != c && a != d && b != c && b != d);

        //SUM (XOR)
        unsigned int lowerCardSum = 0;
        unsigned int higherCardSum = 0;

        unsigned int doneSum = 0;
        for (unsigned int i = 0; i < NUMBER_POSSIBLE_SEQUENCES; i++) {
            if (!doneSum && isStillPossible(s.seq[i].probs)) {
                unsigned int decidingSum = 0;
                if (WEAK_SECURITY == 2) {
                    decidingSum = (s.seq[i].probs.frac[1].num);
                }
                else {
                    decidingSum = (s.seq[i].probs.frac[1].num) || (s.seq[i].probs.frac[2].num);
                }
                unsigned int firstSum = s.seq[i].val[a];
                unsigned int secondSum = s.seq[i].val[b];
                assume(firstSum != secondSum);
                if (!higherCardSum && !lowerCardSum) {
                    // In a 1-sequence, the first card is higher, otherwise the second one.
                    higherCardSum = decidingSum ? firstSum : secondSum;
                    lowerCardSum = decidingSum ? secondSum : firstSum;
                }
                else {
                    if ((decidingSum
                        && !(firstSum == higherCardSum
                            && secondSum == lowerCardSum))
                        || (!decidingSum
                            && !(secondSum == higherCardSum
                                && firstSum == lowerCardSum))) {
                        doneSum = 1;
                        res = 0;
                    }
                }
            }
        }

        //CARRY (AND)
        unsigned int lowerCardCarry = 0;
        unsigned int higherCardCarry = 0;

        unsigned int done = 0;
        for (unsigned int i = 0; i < NUMBER_POSSIBLE_SEQUENCES; i++) {
            if (!done && isStillPossible(s.seq[i].probs)) {
                unsigned int decidingCarry = s.seq[i].probs.frac[NUMBER_PROBABILITIES - 1].num;
                unsigned int firstCarry = s.seq[i].val[c];
                unsigned int secondCarry = s.seq[i].val[d];
                assume(firstCarry != secondCarry);
                if (!higherCardCarry && !lowerCardCarry) {
                    // In a 1-sequence, the first card is higher, otherwise the second one.
                    higherCardCarry = decidingCarry ? firstCarry : secondCarry;
                    lowerCardCarry = decidingCarry ? secondCarry : firstCarry;
                }
                else {
                    if ((decidingCarry
                        && !(firstCarry == higherCardCarry
                            && secondCarry == lowerCardCarry))
                        || (!decidingCarry
                            && !(secondCarry == higherCardCarry
                                && firstCarry == lowerCardCarry))) {
                        done = 1;
                        res = 0;
                    }
                }
            }
        }
    }
    return res;
}
```

A.6 Implementation of the Nested Structure

A.6.1 Additions to the Main C Program

We add and edit some parameters of the main symbolic programs that are dependant on the actions that can be performed.

```c
/**
* determines whether the Modules are used or not
* 0: no modules only turns and shuffles
* 1: modules are used
*/
#ifndef MODULES
#define MODULES 1
#endif

/**
* Amount of different action types allowed in protocol, excluding result action.
*/
#if MODULES == 0
#define A 2
#else
#define A 3
#endif

/**
* Number assigned to turn and shuffle action.
*/
#ifndef TURN
#define TURN 0
#endif

#ifndef SHUFFLE
#define SHUFFLE 1
#endif

/**
* Number assigned to protocol execution action.
*/
#ifndef PROTOCOL
#define PROTOCOL 2
#endif
```

We also have to include `modules.c` (appendix A.6.2 with `#include"modules.c"`.

Additionally we add the ptotocol action to the function `performActions`. It chooses an action (turn, shuffle or protocol) and performs it.

```c
unsigned int performActions(struct state s) {
    unsigned int result = 0;
    // All reachable states are stored here.
    struct state reachableStates[MAX_REACHABLE_STATES];
    for (unsigned int i = 0; i < MAX_REACHABLE_STATES; i++) {
        // Begin calculation from start state.
        reachableStates[i] = s;
    }
    for (unsigned int i = 0; i < L; i++) {
        // Choose the action nondeterministically.
        unsigned int action = nondet_uint();
        assume(action < A);
        // If A is greater than 2, we must add cases for additional
           actions below.
        if (MODULES == 0) {
            assume(A == 2);
        }
        else {
            assume(A == 3);
        }
        unsigned int next = i + 1;

        if (action == TURN) {
            //perform shuffle action
        }
        else if (action == SHUFFLE) {
            //perform shuffle
            }
        }
        else if (action == PROTOCOL) {
            reachableStates[next] = applyProtocols(reachableStates[i
               ]);

            // only for not Final Runtime
            if (isFinalState(reachableStates[next])) {
                assume(next == L);
                result = 1;
            }
        }
        else {
            // No valid action was chosen. This must not happen.
            assume(next == L);
        }
    }
    return result;
}
```

A.6.2 Modules.c

This is the complete modules. c program. It contains all necessary implementations for the protocol action

```c
#include <stdlib.h>
#include <stdint.h>
#include <assert.h>

/**
* MODULES:
* The maximum number of possible result states a protocol can have.
* This is defined for all possibly used protocols because protocolStates needs to have a fixed size
* it is currently defined as 2, because the protocol with the most endstates has 2 endstates
*/
#ifndef MAX_PROTOCOL_ENDSTATES
#define MAX_PROTOCOL_ENDSTATES 2
#endif

/**
* whether the protcol
* AND by Takaaki Mizuki and Hideaki Sone (2009) -> Finite Runtime, 6 cards, 2 steps
* (https://doi.org/10.1007/978-3-642-02270-8_36)
* is used (0: not used, 1: used)
*/
#ifndef USE_FR_AND
#define USE_FR_AND 0
#endif

/**
* AND by Takaaki Mizuki and Hideaki Sone (2009) -> Finite Runtime, 6 cards, 2 steps
* (https://doi.org/10.1007/978-3-642-02270-8_36)
*/
#ifndef FR_AND
#define FR_AND 0
#endif

/**
* whether the protcol
* XOR by Takaaki Mizuki and Hideaki Sone(2009) -> Finite Runtime, 4 cards, 2 steps
* (https://doi.org/10.1007/978-3-642-02270-8_36)
* is used (0: not used, 1: used)
*/
#ifndef USE_FR_XOR
#define USE_FR_XOR 0
#endif

/**
* XOR by Takaaki Mizuki and Hideaki Sone(2009) -> Finite Runtime, 4 cards, 2 steps
* (https://doi.org/10.1007/978-3-642-02270-8_36)
*/
#ifndef FR_XOR
#define FR_XOR 1
```

```
#endif

/**
* whether the protcol
* AND by Alexander Koch, Michael Schrempp and Michael Kirsten (2021) -> Las Vegas, 5 cards, 5 steps
* (https://doi.org/10.1007/s00354-020-00120-0)
* is used (0: not used, 1: used)
*/
#ifndef USE_LV_AND
#define USE_LV_AND 0
#endif

/**
* AND by Alexander Koch, Michael Schrempp and Michael Kirsten (2021) -> Las Vegas, 5 cards, 5 steps
* (https://doi.org/10.1007/s00354-020-00120-0)
*/
#ifndef LV_AND
#define LV_AND 2
#endif

/**
* whether the protcol
* OR by Anne Hoff -> Las Vegas, 4 cards, 6 steps
* (https://github.com/a-nne-h/automatedApproachToGeneratingCardProtocols)
* is used (0: not used, 1: used)
*/
#ifndef USE_LV_OR
#define USE_LV_OR 0
#endif

/**
* OR by Anne Hoff -> Las Vegas, 4 cards, 6 steps
* (https://github.com/a-nne-h/automatedApproachToGeneratingCardProtocols)
*/
#ifndef LV_OR
#define LV_OR 3
#endif

/**
* whether the protcol
* COPY by Takaaki Mizuki and Hideaki Sone (2009) with fixed amount of copies = 1 -> Finite Runtime, 6 cards, 2 steps
* (https://doi.org/10.1007/978-3-642-02270-8_36)
* is used(0: not used, 1 : used)
*/
#ifndef USE_FR_COPY
#define USE_FR_COPY 0
#endif

/**
```

```
* COPY by Takaaki Mizuki and Hideaki Sone (2009) with fixed amount of copies = 1 -> Finite Runtime, 6 cards, 2 steps
* (https://doi.org/10.1007/978-3-642-02270-8_36)
*/
#ifndef FR_COPY
#define FR_COPY 4
#endif

/**
* NOT does not have to be a protocol, becaue it is nothing else than a perm operation which is already included
* Whether NOT is used -> Finite Runtime, 2 cards, 1 steps
*/

//unsigned int protocolTable[5][2][4][6] = { FR_AND_TABLE, FR_XOR_TABLE, LV_AND_TABLE, LV_OR_TABLE, FR_COPY_TABLE };
unsigned int protocolTable[5][2][4][6] = { { { { 1,2,1,2,1,2 }, { 1,2,2,1,1,2 },{ 1,2,1,2,1,2 }, { 1,2,1,2,2,1 } }, { { 2,1,1,2,1,2 },{ 2,1,1,2,2,1 }, { 2,1,1,2,1,2 },  { 2,1,2,1,1,2 } } },
{ { { 1,2,1,2,0,0 }, { 2,1,1,2,0,0 }, { 2,1,1,2,0,0 }, { 1,2,1,2,0,0 } }, {{ 2,1,2,1,0,0 }, { 1,2,2,1,0,0 }, { 1,2,2,1,0,0 }, { 2,1,2,1,0,0 }} },
{ { { 1,2,2,1,2,0 }, { 1,2,2,1,2,0 }, { 1,2,2,1,2,0 },  { 1,2,1,2,2,0 } }, { { 2,1,1,2,2,0 },  { 2,1,1,2,2,0 },  { 2,1,1,2,2,0 }, { 2,1,2,1,2,0 }} },
{ { { 1,1,2,2,0,0 }, { 1,2,1,2,0,0 }, { 1,2,1,2,0,0 }, { 1,2,1,2,0,0 } }, {{ 1,2,1,2,0,0 }, { 2,1,1,2,0,0 }, { 2,1,1,2,0,0 }, { 2,1,1,2,0,0 }} },
{ { { 1,2,1,2,1,2 }, { 0,0,0,0,0,0 }, { 1,2,2,1,2,1 }, { 0,0,0,0,0,0 } }, { { 2,1,2,1,2,1 }, { 0,0,0,0,0,0 }, { 2,1,1,2,1,2 }, { 0,0,0,0,0,0 }} } };

/**
* MODULES:
* Analog to turn states, this struct is used to retun arrays of states after a protocol operation.
* There is one state for each possible end state (resut state) of the protocol
* In each usage of a protocol, for each sequence the resulting sequences in each end state are calculated and stored in states.
* isUsed[i] contains if the corresponding state[i] holds a end state or isn't used
*
*/
struct protocolStates {
    struct state states[2];
    unsigned int isUsed[MAX_PROTOCOL_ENDSTATES];
};

/**
* MODULES:
* finds the index of a given sequence (as an array) within a state.
*/
unsigned  int findIndex(struct sequence seq) {

    unsigned int index = nondet_uint();

    assume(index < NUMBER_POSSIBLE_SEQUENCES);
    for (int j = 0; j < N; j++) {
        assume(seq.val[j] == emptyState.seq[index].val[j]);
    }
    return index;
}

/**
* MODULES:
* searches for the endSequence in result.states[resultIdx]
* if found, copy the probabilities/possibilities from seq to result.states[resultIdx] and return new result
*/
struct protocolStates copyResults(struct sequence seq, struct protocolStates result, unsigned int resultIdx) {
    //find index of sequence within state that matches endSequence
    unsigned int index = findIndex(seq);

    // copy the probabilities/possibilities from seq to result.states[resultIdx] (! add the values -> cr shuffle)
    for (unsigned int j = 0; j < NUMBER_PROBABILITIES; j++) {
        struct fraction prob = seq.probs.frac[j];
        // Copy numerator.
        result.states[resultIdx].seq[index].probs.frac[j].num += prob.num;

        if (!WEAK_SECURITY) { // Probabilistic security
            // Copy denominator.
            result.states[resultIdx].seq[index].probs.frac[j].den += prob.den;
        }
    }
    return result;
}

struct protocolStates doProtocols(unsigned int protocolChosen, struct state s, unsigned int com1A, unsigned int com1B, unsigned int com2A, unsigned int com2B, unsigned int help1, unsigned int help2) {
    struct protocolStates result;
    // Initialise N empty states.
    for (unsigned int i = 0; i < MAX_PROTOCOL_ENDSTATES; i++) {
        result.states[i] = emptyState;
        result.isUsed[i] = 0;
    }
    for (unsigned int i = 0; i < MAX_PROTOCOL_ENDSTATES; i++) {
        for (unsigned int j = 0; j < NUMBER_POSSIBLE_SEQUENCES; j++) {
            struct sequence seq = s.seq[j];
            if (isStillPossible(seq.probs)) {
                unsigned int idx = 0;
                if (isZero(seq.val[com1A], seq.val[com1B])) {
                    if (isZero(seq.val[com2A], seq.val[com2B])) {
                        // 0101
                        idx = 0;
                    }
                    else if (isOne(seq.val[com2A], seq.val[com2B])) {
                        // 0110
                        idx = 1;
                    }
                }
                else if (isOne(seq.val[com1A], seq.val[com1B])) {
                    if (isZero(seq.val[com2A], seq.val[com2B])) {
                        // 1001
                        idx = 2;
                    }
                    else if (isOne(seq.val[com2A], seq.val[com2B])) {
                        // 1010
                        idx = 3;
                    }
                }
                seq.val[com1A] = protocolTable[protocolChosen][i][idx][0];
                seq.val[com1B] = protocolTable[protocolChosen][i][idx][1];
                seq.val[com2A] = protocolTable[protocolChosen][i][idx][2];
                seq.val[com2B] = protocolTable[protocolChosen][i][idx][3];

                // if we have one (or more) helper card
                if (protocolChosen == FR_AND || protocolChosen == FR_COPY
                    || protocolChosen == LV_AND) {
                    seq.val[help1] = protocolTable[protocolChosen][i][idx][4];
                    // if we have two helper cards
                    if (protocolChosen == FR_AND || protocolChosen == FR_COPY) {
                        seq.val[help2] = protocolTable[protocolChosen][i][idx][5];
                    }
                }
                result = copyResults(seq, result, i);
                result.isUsed[i] = 1;
            }
        }
    }
    for (unsigned int l = 0; l < MAX_PROTOCOL_ENDSTATES; l++) {
        assume(isBottomFree(result.states[l]));
    }
    return result;
}
```

```

struct state applyProtocols(struct state s) {
    // check that the chosen protocol is actually 'allowed'
    unsigned int protocolChosen = nondet_uint();
    assume(protocolChosen >= 0 && protocolChosen < 5);
    if (USE_FR_AND == 0) {
        assume(protocolChosen != FR_AND);
    }
    if (USE_FR_XOR == 0) {
        assume(protocolChosen != FR_XOR);
    }
    if (USE_LV_AND == 0) {
        assume(protocolChosen != LV_AND);
    }
    if (USE_LV_OR == 0) {
        assume(protocolChosen != LV_OR);
    }
    if (USE_FR_COPY == 0) {
        assume(protocolChosen != FR_COPY);
    }
    // create resulting states
    struct protocolStates resultingStates;
    for (unsigned int i = 0; i < MAX_PROTOCOL_ENDSTATES; i++) {
        resultingStates.states[i] = emptyState;
        resultingStates.isUsed[i] = 0;
    }
    // pick 4 cards that represent the two commitments
    unsigned int com1A = nondet_uint();
    unsigned int com1B = nondet_uint();
    unsigned int com2A = nondet_uint();
    unsigned int com2B = nondet_uint();
    assume(com1A < N&& com1B < N&& com2A < N&& com2B < N);
    assume(com1A != com1B && com1A != com2A && com1A != com2B);
    assume(com1B != com2A && com1B != com2B);
    assume(com2A != com2B);
    unsigned int help1 = 0;
    unsigned int help2 = 0;
    for (unsigned int i = 0; i < NUMBER_POSSIBLE_SEQUENCES; i++) {
        // if the probability/possibility of this state is not 0
        if (isStillPossible(s.seq[i].probs)) {
            // check that througout every possible sequence in the state we have chosen two different cards for our commitments
            assume(s.seq[i].val[com1A] != s.seq[i].val[com1B]);
            assume(s.seq[i].val[com2A] != s.seq[i].val[com2B]);
        }
    }
    // for copy we only have two commitments and four help cards
    if (protocolChosen == FR_COPY) {
        for (unsigned int i = 0; i < NUMBER_POSSIBLE_SEQUENCES; i++) {
            // if the probability/possibility of this state is not 0
            if (isStillPossible(s.seq[i].probs)) {
                // check that helper cards are the same all throughout every possible sequence in the state
                assume(isZero((s.seq[i].val[com2A]), s.seq[i].val[com2B]));
            }
        }
    }
    //protocols with five cards
    if (protocolChosen == LV_AND) {
        help1 = nondet_uint();
        assume(help1 < N);
        assume(help1 != com1A && help1 != com1B && help1 != com2A && help1 != com2B);
        for (unsigned int i = 0; i < NUMBER_POSSIBLE_SEQUENCES; i++) {
            // if the probability/possibility of this state is not 0
            if (isStillPossible(s.seq[i].probs)) {
                // check that helper cards are the same all throughout every possible sequence in the state
                if (protocolChosen == LV_AND) {
                    // for LV_AND the helper card is 2
                    assume(s.seq[i].val[help1] == 2);
                }
            }
        }
    }
    if (protocolChosen == FR_AND || protocolChosen == FR_COPY) {
        help1 = nondet_uint();
        help2 = nondet_uint();
        assume(help1 < N&& help2 < N);
        assume(help1 != com1A && help1 != com1B && help1 != com2A && help1 != com2B);
        assume(help2 != com1A && help2 != com1B && help2 != com2A && help2 != com2B && help2 != help1);
        for (unsigned int i = 0; i < NUMBER_POSSIBLE_SEQUENCES; i++) {
            // if the probability/possibility of this state is not 0
            if (isStillPossible(s.seq[i].probs)) {
                // check that helper cards are the same all throughout every possible sequence in the state
                assume(isZero((s.seq[i].val[help1]), s.seq[i].val[help2]));
            }
        }
    }
    resultingStates = doProtocols(protocolChosen, s, com1A, com1B, com2A, com2B, help1, help2);
    //as with TURN, choose one output nondeterministically to look at further
    unsigned int stateIdx = nondet_uint();
    assume(stateIdx < MAX_PROTOCOL_ENDSTATES);
    assume(resultingStates.isUsed[stateIdx]);
    return resultingStates.states[stateIdx];
}
```

A.7 Programs for the Experiments with Data Structures

The following code excerpt is the full test program `bitShifts.c` for the data structure implemented with `chars` and bitwise operations in chapter 5. This code snippet can be used to reproduce the experimental setup from section 5.2.2. It also shows how we implemented the different operations with bitwise operations. The implementation for the experiment with arrays is analogous to the implementation here. We only use arrays instead of a **`char`** and operations on arrays instead of array operations. The full implementation of both programs can be found in the GitHub repository (appendix A.4

```c
/*
 * definitions of the used static variables
 * NUM_SYM, N, COMMIT, NUMBER_POSSIBLE_SEQUENCES, NUMBER_START_SEQS, NUMBER_POSSIBLE_PERMUTATIONS, WEAK_SECURITY, NUMBER_PROBABILITIES, MAX_PERM_SET_SIZE
 */

struct fraction {
    unsigned int num; // The numerator.
    unsigned int den; // The denominator.
};

struct fractions {
    struct fraction frac[NUMBER_PROBABILITIES];
};

struct sequence {
    char val;
    struct fractions probs;
};

struct state {
    struct sequence seq[NUMBER_POSSIBLE_SEQUENCES];
};

struct permSequence {
    unsigned int val[N];
    struct fractions probs;
};

 // All permutations are remembered here, as seen from left to right, sorted alphabetically.
struct permutationState {
    struct permSequence permSeq[NUMBER_POSSIBLE_PERMUTATIONS];
};

// We store all possible permutations into a seperate state to save resources.
struct permutationState stateWithAllPermutations;

struct state emptyState;

struct narray {
    unsigned int arr[N];
};
struct numsymarray {
    unsigned int arr[NUM_SYM];
};

struct permutationState getStateWithAllPermutations() {
    struct permutationState s;
    for (unsigned int i = 0; i < NUMBER_POSSIBLE_PERMUTATIONS; i++) {
        struct narray taken;
        for (unsigned int j = 0; j < N; j++) {
            taken.arr[j] = 0;
        }
        for (unsigned int j = 0; j < N; j++) {
            s.permSeq[i].val[j] = nondet_uint();
            unsigned int val = s.permSeq[i].val[j];
            assume(0 < val && val <= N);
            unsigned int idx = val - 1;
            assume(!taken.arr[idx]);
            taken.arr[idx]++;
        }
    }
    // Not needed, but to avoid state space explosion
    for (unsigned int i = 0; i < NUMBER_POSSIBLE_PERMUTATIONS; i++) {
        for (unsigned int j = 0; j < NUMBER_PROBABILITIES; j++) {
            s.permSeq[i].probs.frac[j].num = 0;
            s.permSeq[i].probs.frac[j].den = 1;
        }
    }
    for (unsigned int i = 1; i < NUMBER_POSSIBLE_PERMUTATIONS; i++) {
        unsigned int checked = 0;
        unsigned int last = i - 1;
        for (unsigned int j = 0; j < N; j++) {
            // Check lexicographic order
            unsigned int a = s.permSeq[last].val[j];
            unsigned int f = s.permSeq[i].val[j];
            checked |= (a < f);
            assume(checked || a == f);
        }
        assume(checked);
    }
    return s;
}
/**
 * Given an char containing a sequence, we return the index of the given sequence in a state.
 */
unsigned int getSequenceIndexFromArray(char compare, struct state compareState) {
    unsigned int seqIdx = nondet_uint();
    assume(seqIdx < NUMBER_POSSIBLE_SEQUENCES);
    struct sequence seq = compareState.seq[seqIdx];
    assume(!(seq.val ^ compare)); // the chars are equal if XOR is 0
    return seqIdx;
}

/**
 * Update the possibilities of a sequence after a shuffle.
 */
struct fractions recalculatePossibilities(struct fractions probs,
    struct fractions resProbs,
    unsigned int permSetSize) {
    for (unsigned int k = 0; k < NUMBER_PROBABILITIES; k++) {
        struct fraction prob = probs.frac[k];
        unsigned int num = prob.num;
        unsigned int denom = prob.den;
        if (num && WEAK_SECURITY) {
            resProbs.frac[k].num |= num;
        }
        else if (num) {
            /**
             * Only update fractions in case we are in the
             * strong security setup.
             */
             // Update denominator.
            resProbs.frac[k].den = denom * permSetSize;
            // Update numerator.
            resProbs.frac[k].num = (num * permSetSize) + denom;
        }
    }
    return resProbs;
}

/**
 * Calculate the state after a shuffle operation starting from s with the given permutation set.
 * Deleted isStillPossible
 * changed content in 2nd for loop, especially the application of the permutatuin to the sequence
*/
struct state doShuffle(struct state s,
    unsigned int permutationSet[MAX_PERM_SET_SIZE][N],
    unsigned int permSetSize) {
    struct state res = emptyState;
    // For every sequence in the input state.
    for (unsigned int i = 0; i < NUMBER_POSSIBLE_SEQUENCES; i++) {
        struct sequence seq = s.seq[i];
        // For every permutation in the permutation set.
        for (unsigned int j = 0; j < MAX_PERM_SET_SIZE; j++) {
            if (j < permSetSize) {
                char resultingSeq = 0;
                for (unsigned int k = 0; k < N; k++) {
                    char temp = 0;
                    // Apply permutation j to sequence i.
                    temp = seq.val & (1 << k);
                    int index = permutationSet[j][k] - k;
                    if (index >= 0) {
                        temp = temp << (index);
                    }
                    else {
                        temp = temp >> (-1 * index);
                    }
                    resultingSeq = resultingSeq | temp;
                }
                unsigned int resultSeqIndex = // Get the index of the resulting sequence.
                    getSequenceIndexFromArray(resultingSeq, res);
                 //Recalculate possibilities.
                res.seq[resultSeqIndex].probs =
                    recalculatePossibilities(seq.probs,
                        res.seq[resultSeqIndex].probs,
                        permSetSize);
            }
        }
    }
    return res;
}

struct state applyShuffle(struct state s) {
    // Generate permutation set (shuffles are assumed to be uniformly distributed).
    unsigned int permSetSize = nondet_uint();
    assume(0 < permSetSize && permSetSize <= MAX_PERM_SET_SIZE);

    unsigned int permutationSet[MAX_PERM_SET_SIZE][N] = { 0 };
    unsigned int takenPermutations[NUMBER_POSSIBLE_PERMUTATIONS] = { 0 };
    /**
     * Choose permSetSize permutations nondeterministically. To achieve this,
     * generate a nondeterministic permutation index and get the permutation from this index.
     * No permutation can be chosen multiple times.
     */
    unsigned int lastChosenPermutationIndex = 0;
    for (unsigned int i = 0; i < MAX_PERM_SET_SIZE; i++) {
        if (i < permSetSize) { // Only generate permutations up to permSetSize.
            unsigned int permIndex = nondet_uint();
            // This ensures that the permutation sets are sorted lexicographically.
            assume(lastChosenPermutationIndex <= permIndex);
            assume(permIndex < NUMBER_POSSIBLE_PERMUTATIONS);
            assume(!takenPermutations[permIndex]);
            assume(permSetSize != 10);

            takenPermutations[permIndex] = 1;
            lastChosenPermutationIndex = permIndex;
```

```c

            for (unsigned int j = 0; j < N; j++) {
                permutationSet[i][j] = stateWithAllPermutations.permSeq[permIndex].val[j] - 1;
                /**
                 * The '-1' is important. Later, we convert to array indices such as
                 * array[permutationSet[x][y]]. Without the '-1', we would get out-
                 * of-bound errors there.
                 */
            }
        }
    }
    struct state res = doShuffle(s, permutationSet, permSetSize);
    return res;
}

/**
 * Constructor for states. Only use this to create new states.
 */
struct state getEmptyState() {
    struct state s;
    struct numsymarray symbolCount;
    for (unsigned int i = 0; i < NUM_SYM; i++) {
        symbolCount.arr[i] = 0;
    }

    for (unsigned int i = 0; i < NUMBER_POSSIBLE_SEQUENCES; i++) {
        struct numsymarray taken;
        for (unsigned int j = 0; j < NUM_SYM; j++) {
            taken.arr[j] = 0;
        }
        char value = 0;
        for (unsigned int j = 0; j < N; j++) {
            char val = nondet_uint();
            assume(0 <= val && val <= 1);
            taken.arr[val]++;
            assume(taken.arr[val] <= N - 2); // At least two symbols have to be different. Players cannot commit otherwise.
            value = value | (val << (N - 1 - j));
        }
        s.seq[i].val = value;
        for (unsigned int j = 0; j < NUM_SYM; j++) {
            if (i == 0) {
                symbolCount.arr[j] = taken.arr[j];
            }
            else { // We ensure that every sequence consists of the same symbols
                assume(taken.arr[j] == symbolCount.arr[j]);
            }
        }
        // Here we store the numerators and denominators
        for (unsigned int j = 0; j < NUMBER_PROBABILITIES; j++) {
            s.seq[i].probs.frac[j].num = 0;
            s.seq[i].probs.frac[j].den = 1;
        }
    }
    for (unsigned int i = 1; i < NUMBER_POSSIBLE_SEQUENCES; i++) {
        unsigned int checked = 0;
        unsigned int last = i - 1;
        for (unsigned int j = 1; j <= N; j++) {
            // Check lexicographic order
            char a = (s.seq[last].val & (1 << N - j));
            char f = (s.seq[i].val & (1 << N - j));
            checked |= (a < f);
            assume(checked || a == f);
        }
        assume(checked);
    }
    return s;
}

/**
 * Determine if a sequence in the start state belongs to the input possibility (0 0).
 */
unsigned int isZeroZero(char sequence) {
    if(sequence & (1 << (N - 1))) {
        return 0;
    }
    if (!(sequence & (1 << (N - 2)))) {
        return 0;
    }
    if (sequence & (1 << (N - 3))) {
        return 0;
    }
    if (!(sequence & (1 << (N - 4)))) {
        return 0;
    }
    return 1;
}

/**
 * Determine if a sequence in the start state belongs to the input possibility (1 1).
 */
unsigned int isOneOne(char sequence) {
    if (!(sequence & (1 << (N - 1)))) {
        return 0;
    }
    if (sequence & (1 << (N - 2))) {
        return 0;
    }
    if (!(sequence & (1 << (N - 3)))) {
        return 0;
    }
    if (sequence & (1 << (N - 4))) {
        return 0;
    }
    return 1;
}

/**
 * This method constructs the start sequence for a given commitment length COMMIT
 * using nodeterministic assignments. We only consider the case where Alice uses
 * the cards "1" and "2", and Bob uses the cards "3" and "4".
 */
char getStartSequence() {
    assume(N >= COMMIT); // We assume at least as many cards as needed for the commitments.
    struct numsymarray taken;
    for (unsigned int i = 0; i < NUM_SYM; i++) {
        taken.arr[i] = 0;
    }
    char res = 0;
    for (unsigned int i = 0; i < COMMIT; i++) {
        char card = nondet_uint();
        assume(0 <= card && card < COMMIT && card < NUM_SYM);
        assume(taken.arr[card] < COMMIT / NUM_SYM);
        taken.arr[card]++;
        res = res | (card << (N - 1 - i));
    }
    // Here we assume that each player only uses fully distinguishable cards
    assume((res & 1 << (N - 1)) != ((res & 1 << (N - 2))<<1));
    assume((res & 1 << (N - 3)) != ((res & 1 << (N - 4)) << 1));
    // rest of cards
    for (unsigned int i = COMMIT; i < N; i++) {
        char card = nondet_uint();
        assume(0 <= card);
        assume(card < NUM_SYM);
        res = res | (card << (N - 1 - i));
    }
    return res;
}

/**
* This function performs a shuffle and afterwards checks for a specific property of the probabilities
* in this test it is, whether all probabilities in all possible sequences have a value that is not equal to 0
* a correct result needs at least 6 permutations
* therefore the problem is complicated enough to ensure some level of complexity while keeping the code simple
* For other tests, this function can be easiy altered
*/
struct state tryPermutation(struct state s) {
    struct state res = applyShuffle(s);
    // check if every possibility is 1 after shuffle
    for (int i = 0; i < NUMBER_POSSIBLE_SEQUENCES; i++) {
        for (int j = 0; j < NUMBER_PROBABILITIES; j++) {
            assume(res.seq[i].probs.frac[j].num != 0);
        }
    }
    return s;
}

int main() {
    emptyState = getEmptyState();
    struct state startState = emptyState;
    char start[NUMBER_START_SEQS];
    for (unsigned int i = 0; i < NUMBER_START_SEQS; i++) {
        start[i] = getStartSequence();
    }
    assume(isZeroZero(start[0]));
    assume(!((start[0] & (1 << N - 1)) ^ (start[1] & (1 << N - 1))));
    assume((start[1] & (1 << N - 1)) ^ (start[2] & (1 << N - 1)));
    assume(!((start[2] & (1 << N - 1)) ^ (start[3] & (1 << N - 1))));
    assume(!((start[0] & (1 << N - 3)) ^ (start[2] & (1 << N - 3))));
    assume((start[0] & (1 << N - 3)) ^ (start[1] & (1 << N - 3)));
    assume(!((start[1] & (1 << N - 3)) ^ (start[3] & (1 << N - 3))));

    unsigned int arrSeqIdx[NUMBER_START_SEQS];
    for (unsigned int i = 0; i < NUMBER_START_SEQS; i++) {
        arrSeqIdx[i] = getSequenceIndexFromArray(start[i], startState);
    }
    if (WEAK_SECURITY == 2) {
        for (unsigned int i = 0; i < (NUMBER_START_SEQS - 1); i++) {
            startState.seq[arrSeqIdx[i]].probs.frac[0].num = 1;
        }
        unsigned int lastStartSeq = NUMBER_START_SEQS - 1;
        unsigned int arrIdx = arrSeqIdx[lastStartSeq];
        unsigned int lastProbIdx = NUMBER_PROBABILITIES - 1;
        startState.seq[arrIdx].probs.frac[lastProbIdx].num = isOneOne(start[lastStartSeq]);
    } else {
        for (unsigned int i = 0; i < (NUMBER_START_SEQS); i++) {
            startState.seq[arrSeqIdx[i]].probs.frac[i].num = 1;
        }
    }
    stateWithAllPermutations = getStateWithAllPermutations();
    tryPermutation(startState);
    assert(0);
    return 0;
}
```