Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Prateek Sharma

Formally Verified Reduction Proof for the Free XOR Garbling Scheme

Master's thesis in Security and Cloud Computing
Supervisor: Tjerand Silde
Co-supervisor: Russell W. F. Lai, Chris Brzuska, Amirhosein Rajabi, Kirthivaasan Puniamurthy

November 2026

NTNU
Norwegian University of Science and Technology

Aalto University
School of Science

Prateek Sharma

Formally Verified Reduction Proof for the Free XOR Garbling Scheme

Master's thesis in Security and Cloud Computing
Supervisor: Tjerand Silde
Co-supervisor: Russell W. F. Lai, Chris Brzuska, Amirhosein Rajabi, Kirthivaasan Puniamurthy
November 2026

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of Science and Technology

Problem description

Garbled Circuits are a fundamental cryptographic primitive used in secure two-party computation (2PC), enabling two parties to jointly compute a function without revealing their private inputs - only the output is revealed. The concept was introduced by Andrew Yao in 1986 [Yao86], where functions are expressed as Boolean circuits and each wire is labelled with encrypted wire keys. Early constructions required four ciphertexts per gate, resulting in significant communication overhead and limiting practical adoption.

Subsequent optimizations such as Point-and-Permute, Half-Gates and the Free XOR technique greatly reduced the number of ciphertexts needed per gate, as well as reducing the number of gates to be transmitted, leading to substantial performance improvements [BMR90; ZRE15; KS08]. These optimizations form the basis of many state-of-the-art secure computation frameworks.

Formal verification of cryptographic constructions is essential to ensure correctness and prevent subtle vulnerabilities arising from unchecked corner cases. Brzuska and Oechsner [BO23] developed a state-separating pen-and-paper reduction proof for Yao's original garbling scheme. State-separating proofs model protocol components as isolated "packages" with private state and oracle interfaces, allowing for structured security reductions. Their work established security by reducing the garbling scheme to the IND-CPA security of the encryption primitive used for gate tables.

Domino is an automated verification tool specifically designed for state-separating proofs. It has been used to reason about complex real-world protocols such as TLS [Raj25] and WPA2 [BERW26]. Domino offers a high level of automation and readability because proof scripts resemble the pseudocode found in traditional cryptographic proofs.

Furthermore, the above mentioned proof for Yao's garbling scheme by Brzuska and Oechsner [BO23] was later formalized in Domino [BERW26]. It can serve as a good starting point for the thesis to observe how games may be converted from pen-and-paper proofs to Domino code.

The goal of this thesis is to formalize and mechanize a security reduction for the Half-Gates garbling scheme within the Domino framework. This includes:

- Developing a state-separating, pen-and-paper reduction proof for the Free XOR technique.
- Encoding this proof in Domino.

This work extends formal verification techniques to an optimized and widely deployed garbling construction and demonstrates Domino's applicability to modern

cryptographic primitives.

Approved: 2026-03-09 – Tjerand Silde, NTNU (Main supervisor)

Abstract

Garbling schemes are a core building block of Two-party Computation (2PC) protocols, enabling two parties to jointly compute a function while keeping their inputs private. While numerous optimizations to garbling schemes have been proposed since Yao's original construction, verifying the security of these schemes remains challenging, as manual proof review is error-prone and difficult to scale to large, complex arguments. State-separating proofs (SSPs) offer a modular framework for structuring such security arguments, and Domino is a proof assistant that allows SSPs to be encoded and formally verified. Prior work has expressed Yao's original garbling scheme as an SSP, but no such treatment exists for Free XOR, a widely used optimization that underlies later schemes such as Half-Gates and Three-Halves.

This thesis addresses this gap by using Applebaum's LIN-RK-KDM-based proof of the Free XOR garbling scheme to construct a pen-and-paper SSP establishing the privacy of an AND gate, and by encoding this proof in Domino. The assumptions, games, and reduction were successfully formalized and encoded, and Domino was able to validate all but one proofstep. This equivalence proofstep required to complete the proof could not be fully verified due to the complexity of randomness mapping required, leaving part of the security argument unresolved. These results demonstrate the feasibility of translating a Free XOR security proof into the SSP framework and encoding it in Domino, while identifying the remaining verification gap as a direction for future work, and also evaluating Domino as a proof verification tool.

Preface

This thesis was submitted as part of the completion requirements for my Master of Science in Security and Cloud Computing double degree at Norwegian University of Science and Technology (NTNU) and Aalto University. It was supervised by Tjerand Silde at the Department of Information Security and Communication Technology (IIK), NTNU and Russell Lai at Aalto University, and was advised by Chris Brzuska, Amirhosein Rajabi and Kirthivaasan Puniamurthy, also at Aalto University.

Acknowledgements

I would like to express my gratitude to the various people who were responsible for the supervision of this thesis. Chris, I am very thankful for your support throughout my studies and this thesis, and for motivating me to do better. I would also like to thank Tjerand, for being so readily available to answer questions, and for allowing the use of the Crypto-Lab at NTNU. It was a pleasant experience to work around fellow students there. This work also would not have been possible without the guidance I received from Amirhosein and Kirthi, and I am very grateful to them.

I would like to thank the SECCLO programme, for the opportunity to learn among such curious and intelligent people. My thanks also go to the programme staff, who have always been very helpful with administrative matters.

I lack the words to describe how much I appreciate the friends I made at Aalto and NTNU, who have helped, and continue to help me grow as a person, and for all the care and support I have received from them.

Finally, I would like to thank my family and my friends back home for supporting my academic decisions even though it has put me in a place very far away from them.

Contents

List of Figures ix

List of Tables xi

List of Acronyms xiii

1 Introduction 1
- 1.1 Motivation . . . 1
  - 1.1.1 Garbling Schemes . . . 1
  - 1.1.2 Formal Verification . . . 2
  - 1.1.3 State Separating Proofs . . . 3
  - 1.1.4 Bridging the concepts . . . 4
- 1.2 Objectives . . . 4
- 1.3 Sustainability . . . 5
- 1.4 Outline . . . 5

2 Background 7
- 2.1 Garbled Circuits . . . 7
- 2.2 Garbling Schemes . . . 9
  - 2.2.1 Properties . . . 9
  - 2.2.2 Defining Privacy . . . 10
  - 2.2.3 Yao's Garbling Scheme . . . 11
- 2.3 Optimizations to Yao's Garbling Scheme . . . 13
  - 2.3.1 Point and Permute . . . 13
  - 2.3.2 Garbled Row Reduction-3 . . . 14
  - 2.3.3 Free XOR . . . 15
  - 2.3.4 Half-Gates . . . 17

3 Pen-and-Paper State-Separating Proof 19
- 3.1 State Separating Proofs . . . 19
- 3.2 Privacy of the Garbling of Various Gates using Free XOR . . . 20
  - 3.2.1 NOT Gate . . . 20
  - 3.2.2 XOR Gate . . . 21

3.2.3 AND Gate 21

4 Formalizing Free XOR in Domino **29**
4.1 Domino 29
4.1.1 The Domino binary 30
4.1.2 Packages 31
4.1.3 Compositions 32
4.1.4 Theorems 32
4.1.5 Project Structure 34
4.2 Encoding the Free XOR SSP in Domino 34
4.2.1 The Free XOR proof sketch 35
4.2.2 Common parameters 38
4.2.3 Game compositions 38
4.2.4 Packages 43
4.2.5 Theorem 48
4.2.6 Verification Results 60

5 Discussion **63**
5.1 Summary of Results 63
5.2 Challenges in encoding the SSP in Domino 64

6 Conclusion **67**
6.1 Conclusion 67
6.2 Future Work 67

References **69**

Appendix

A Ideal Game Transformations **73**
A.1 Game $\mathcal{R} \to G^1_{\mathsf{LIN-RK-KDM}}$ 73
A.2 Game $G^{1'}_{\mathsf{Sim\text{-}FreeXOR}}$ 74

List of Figures

2.1 Garbled Circuit Protocol 8
2.2 PrvInd and PrvSim Games - adapted from [BHR12] 10
2.3 An AND gate and its garbled form. 12
2.4 Garbled table for an AND gate 13
2.5 Garbled table for an AND gate with Point and Permute 14
2.6 Evaluation of an XOR gate in Free XOR 16

3.1 Wire label mappings before (left) and after (right) a NOT gate. 20
3.2 Code for $G^0_{\mathsf{Sim\text{-}FreeXOR}}$ 22
3.3 Code for $G^b_{\mathsf{LIN-RK-KDM}}$ 22
3.4 Code for $G^1_{\mathsf{Sim\text{-}FreeXOR}}$ 24
3.5 Code of $G^b_{wrapper}$ 25
3.6 Code for Reduction $\mathcal{R}$ 25
3.7 Code for G^b_{Hybrid} 26
3.8 A visualization of the adversary transformation at the end of the proof in Section 3.2.3 28

4.1 Game hops in `freexor.ssp`. The dotted lines represent the planned proof sketch based on the pen-and-paper SSP, whereas the solid lines represent the actual proof sketch. Each game is followed by a list of packages it consists of. 36
4.2 Output of `domino prove` that shows proven steps until the seventh game hop 61
4.3 Output of `domino prove --proofstep 6` that verifies that the seventh game hop fails independently too 61
4.4 Output of `domino prove --proofstep 7` that shows the successful verification of the eighth game hop 61
4.5 Output of `domino prove --proofstep 8` that shows the successful verification of the ninth game hop 61

A.1 Code for $\mathcal{R} \to G^1_{\mathsf{LIN-RK-KDM}}$ 73
A.2 Code for $G^{1\prime}_{\mathsf{Sim\text{-}FreeXOR}}$ 74

List of Tables

4.1 Mapping between the games in the pen-and-paper reduction and Domino packages they are composed of. 35

List of Acronyms

2PC	Two-party Computation
CCR	circular correlation-robust
CR	correlation-robust
FreeXOR	Free XOR
GC	Garbled Circuit
GRR	Garbled Row Reduction
GS	garbling scheme
HG	Half-Gates
LIN-RK-KDM	Linear, Related-Key, Key-Dependent-Message
LPN	Learning Parity with Noise
MPC	Multi-Party Computation
OT	Oblivious Transfer
OTP	one-time pad
P&P	Point and Permute
PPT	probabilistic, polynomial-time
PRF	Pseudo-Random Function
PRG	Pseudo-Random Generator
SDG	Sustainable Development Goal
SMT	Satisfiability Modulo Theory
SSP	state-separating proof
UN	United Nations
Z3	Z3 theorem prover

Chapter 1
Introduction

In Secure Multiparty Computation (also called just Multi-Party Computation) (MPC), two or more parties want to compute a common function using their individual inputs, but do not want to share these inputs with each other. Common use cases of MPC today include delegating computation to cloud servers [Urb24], or querying third party datasets that cannot be shared directly [LPR+21].

Since remote servers are often owned and operated by a separate entity, it is difficult to obtain the same security and privacy guarantees that a user's own machine can provide. Besides user requirements, cybersecurity regulations also require methods to secure the data being processed from the data processor, such as in the case of private healthcare data, or other forms of personal data [Eur16].

1.1 Motivation

Two-party Computation (2PC) is a special case of MPC in which only two parties are involved in the protocol. Often attributed to Andrew Yao's seminal work in 1986 [Yao86], the first approach for 2PC was to represent the function to be computed as a boolean circuit, and to use a Garbled Circuit (GC) to compute the output of the function, while maintaining the secrecy of the inputs of both parties. For more than a decade after this work, Garbled Circuits were considered to not be practical because of the high bandwidth and computation costs of garbling, and due to the fact that garbling an arbitrary function into an optimized boolean circuit is difficult. However, there has been work in the field that has led to optimizations that allow them to be somewhat usable in generic 2PC systems like Fairplay [MNPS04], and in niche applications such as smart contracts [HYLL24].

1.1.1 Garbling Schemes

Yao's version of Garbled Circuits used four ciphertext computations and transmissions per gate in the circuit. It relied on the standard assumption of the existence of

secure Pseudo-Random Functions (PRFs) that could be used to build a secure encryption scheme. Beaver, Micali, and Rogaway [BMR90] published the next well-known optimization to Garbled Circuits, often called Point and Permute (P&P) (we discuss it in more detail in Section 2.3.1), which reduces the computational cost of evaluation of the circuit by up to 75% of Yao's construction, and uses a Pseudo-Random Generator (PRG) security assumption. We observe that while there are improvements, the core assumptions vary. This is a pattern that continues with Naor, Pinkas, and Sumner's Garbled Row Reduction-3 scheme (Section 2.3.2), which offers a flat 25% reduction in bandwidth cost, but returns to a PRF security assumption [NPS99]. The next improvement - the Free XOR scheme by Kolesnikov and Schneider can garble an XOR gate completely free of cost (Section 2.3.3), which can be very efficient if circuits are optimized to maximize the number of XOR gates [KS08]. However, Free XOR relies on an entirely new assumption, 'correlation-robustness' (CR) of a hash function. Furthermore, in their paper, Free XOR was only proven in the stronger Random Oracle model, and CR was later found to be inadequate. Free XOR was later proven secure using other models such as Linear, Related-Key, Key-Dependent-Message (LIN-RK-KDM) security of encryption schemes [App16], and circular correlation-robustness (CCR) of hash functions [CKKZ12]. Both of these assumptions are weaker than the ideal Random Oracle model, but stronger than CR. The Half-Gates garbling scheme was designed under the CCR assumption, and leads to a 50% decrease in bandwidth use over Yao's scheme when garbling AND gates [ZRE15].

1.1.2 Formal Verification

Unlike software verification, proof verification is not concerned with whether an implementation faithfully realizes a protocol, but whether the mathematical argument establishing the security of the proof is logically sound. These concerns are orthogonal: an implementation may be correct while relying on an incorrect security proof, just as a correct proof does not guarantee a secure implementation. Validating a large cryptographic proof, however, is a difficult task, mainly because reasoning about code by simply looking at it relies on intuition, and may hide away corner cases especially in larger proofs. Halevi argued that an alternative is to use computers to enhance the process of reasoning about code [Hal05].

There are two main approaches for automated verification of code - symbolic and computational. Symbolic verification uses an idealized model of cryptography (specifically, the Dolev-Yao model [DY81]) where primitives are treated as perfect black boxes. An adversary algorithm is only allowed to use these black boxes in its operations, limiting the list of operations it can perform to just such functions and their defined properties. This allows symbolic verification to be faster and makes it easier to build formal verification tools. For example, in the symbolic model, an encryption function

may be modeled such that the adversary can derive the message only if they know the key. In contrast, the computational model explicitly models these cryptographic primitives as polynomial-time reductions to some underlying assumption or game. The amount of 'assumed code' is much smaller and the only limit to the adversary is that it must be a polynomial-time algorithm. Computational verifiers reason about the security of cryptographic algorithms under given computational assumptions, as opposed to symbolic verifiers, which assume that cryptographic primitives always satisfy their symbolic equations. However, this leads to proofs that are written for computational verifiers to be more complex, which becomes apparent later in this work as well. Examples of symbolic verification tools include ProVerif [Bla16] and Tamarin [BCDS22], while EasyCrypt [BDG+14], CryptoVerif [Bla23] and Domino [BERW] are examples of computational verification tools.

1.1.3 State Separating Proofs

The security of cryptographic algorithms is often reasoned about as 'games' in an ideal world and a real world. The games consist of 'oracles' which are basically procedures that expose certain functionality and either return an output or modify some state. An adversary interacts with the game in both worlds by calling its oracles and attempts to distinguish between the behavior of the two games. These games were initially expressed as Turing machines with polynomial running time, but Bellare and Rogaway reasoned that pseudocode was also a valid medium to express such games, and was suitable to use in future automated verification systems that Halevi wrote about [BR06]. Going a step further from Bellare and Rogaway's code-based game-playing, Brzuska et al. proposed state-separation, which introduces the concept of games composed of packages with internal states which are modifiable by the oracles contained in those packages [BDF+18]. State-separating proofs (SSPs) are attractive because they allow cryptographers to divide their games into reusable components that have their own state, so a simple state modification does not invalidate game-wide assumptions made during a previous state. Induction-based reasoning also becomes easier, since the amount of code affected by a change of state is smaller when code is divided into packages. Furthermore, having reusable packages allows underlying constructs to be reused in other proofs that rely on similar primitives. It also allows for nice visualisations of package compositions as graphs.

It is worth noting that a proof of Yao's Garbled Circuit construction has already been expressed as an SSP by Brzuska and Oechsner [BO23]. This is a proof of the garbling scheme, rather than the Garbled Circuit protocol. This thesis intends to build on top of this work by converting Kolesnikov and Schneider's proof for the Free XOR garbling scheme under the LIN-RK-KDM assumption into an SSP. The choice is primarily driven by the fact that significant future works in garbling schemes such as Half-Gates [ZRE15] and Three-Halves [RR21] use it as a crucial component. The

improvements in these schemes are not achievable without using Free XOR.

1.1.4 Bridging the concepts

Domino is a highly automated proof assistant for SSPs that allows expression of games, packages and oracles in a syntax intentionally very similar to pen-and-paper SSPs. Currently available as a standalone executable ('crate') in the Rust package manager (cargo), it includes a command-line interface that validates SSP structures, provides type-checking and offers a decent degree of error handling. It also comes with a minimal extension for the Visual Studio Code text editor that provides syntax highlighting. Domino uses an underlying Satisfiability Modulo Theory (SMT) solver to verify equivalence relations between games with different code, and also verifies reduction game-hops. It is worth noting that there exists work in encoding and validating Brzuska and Oechsner's state-separating proof for Yao's garbling scheme in Domino [BERW26]. This work shows the feasibility of encoding the Free XOR garbling scheme in Domino, and therefore, we set this as an additional goal for this thesis. We chose to encode Applebaum's variation of Free XOR in this thesis because of its closeness to Yao's scheme - both of them use encryption to generate the garblings, instead of a hash-based one-time pad (OTP) used in Kolesnikov and Schneider's work. While the hash-based construction is more efficient, it uses a stronger Random Oracle assumption, which is harder to translate to real life, whereas Applebaum's scheme comes with a description of how one can construct a Linear, Related-Key, Key-Dependent-Message secure encryption scheme using an underlying Learning Parity with Noise (LPN) assumption.

It is important to mention that this thesis initially aimed to formally verify the Half-Gates garbling scheme, but ran into the research gap that exists between state-separating proofs for Yao's garbling scheme and Half-Gates. Since Half-Gates builds on top of the Free XOR scheme, this thesis marks the first steps towards formally verifying the Half-Gates scheme, going as far as Free XOR.

1.2 Objectives

The goals of this thesis can be summarized as follows:

1. To develop a state separating pen-and-paper reduction proof for the Free XOR garbling scheme.

2. To encode this proof and verify its correctness in Domino.

1.3 Sustainability

While theoretical cryptography itself does not map well to any of the United Nations (UN) Sustainable Development Goals (SDGs), since the indicators for each subgoal are very distinct, the following arguments can be made for the contributions that this thesis may make to the targets of various subgoals:

- **Target 9.5 - Enhance research and upgrade industrial technologies:** This work pushes the boundaries of research in verified proofs of cryptographic algorithms and protocols that are used to build software technologies.
- **Target 16.6 - Develop effective, accountable and transparent institutions:** This work is in the domain of privacy-preserving technologies, which, when implemented into software, can allow institutions to be more transparent while retaining the privacy of information.

1.4 Outline

The rest of this thesis is organized as follows: Chapter 2 contains background information on garbling schemes, optimizations, SSPs, and Domino. Chapter 3 details the converted pen-and-paper proof for Free XOR. Chapter 4 contains further details about the proof tool Domino, a brief overview on how to set it up, and an encoding of the state-separating proof in Domino. Chapter 5 discusses our results, observations, and challenges of working with Domino as opposed to plain pen-and-paper SSP. Chapter 6 contains concluding remarks and future work.

Chapter 2
Background

In this chapter, we introduce the important cryptographic building blocks needed in this thesis. Section 2.1 introduces Garbled Circuits, while Section 2.2 introduces garbling schemes, their security properties, and Yao's GS. Section 2.3 discusses the GS optimizations that are relevant to the work done in this thesis.

2.1 Garbled Circuits

A Garbled Circuit can be informally defined as a two-party protocol, in which both parties want to evaluate a shared function C together, without revealing their inputs to each other. The two parties participating in the protocol are called the Garbler (or the Garbled Circuit generator), and the Evaluator. The function C needs to be expressed as a boolean circuit if a Garbled Circuit protocol is used. Input hiding is achieved using something called *garbling*, which in practice is some form of encryption. Similarly, the final output is initially garbled, and needs to be ungarbled, which is then implemented using a corresponding decryption function.

A simple run of the protocol starts with both parties possessing their own input bits, respectively (g and e). The protocol proceeds as follows (ref. Figure 2.1):

1. The Garbler garbles the circuit C to obtain the Garbled Circuit C', along with some encoding information (*enc*) and decoding information (*dec*). These will be used to garble and ungarble inputs and outputs.

2. The Garbler garbles its own input g into g'.

3. The Garbler then sends across C', g' and *dec* to the Evaluator.

4. The Evaluator still needs to garble its own input, but does not possess the encoding information, *enc* to do so. Therefore, both parties participate in what is known as a 1-out-of-2 *Oblivious Transfer* (OT) protocol [EGL82]. This

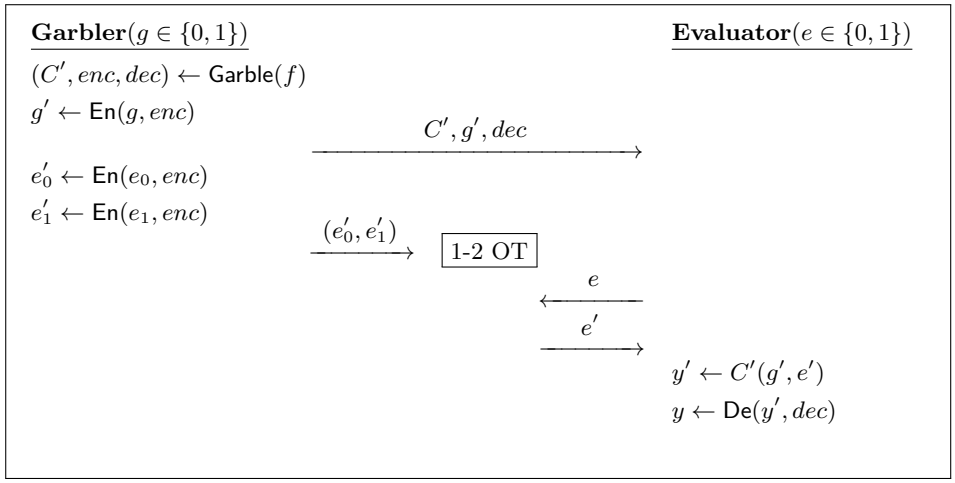


Figure 2.1: Garbled Circuit Protocol

protocol (adapted for GCs) requires the Garbler to garble both possible bit-values that the Evaluator may possess, and provide them to the OT protocol. Then, the Evaluator provides their chosen bit e to the OT protocol and the OT protocol returns the garbling of that bit (e'). The Garbler remains unaware of which garbling was sent to the Evaluator, and the Evaluator remains unaware of the value of the other garbling.

5. At this point, since the Evaluator possesses both garbled inputs (g' and e') and the Garbled Circuit (C'), it can compute the garbled output, y', by running the garbled inputs through the Garbled Circuit.

6. Finally, using the decoding information *dec*, the Evaluator can obtain y by ungarbling y'. This output is equivalent to the output of the function $C(g, e)$.

7. If necessary, the Evaluator can send the final output back to the Garbler.

The protocol can also be scaled up to support larger inputs from both parties. However, this would increase the complexity of the Oblivious Transfer protocol.

Both the Garbler and Evaluator are assumed to be *honest-but-curious*. This means that both parties adhere strictly to the protocol specification. However, they try to learn more than what is intended by using the information received from running the protocol, and from their own randomness and state. So, as an example, the Evaluator could try to obtain the Garbler's bit by ungarbling it, but will not succeed since the algorithm *dec* that the Evaluator receives does not ungarble the inputs.

This kind of adversary is equivalent to the 'Semi-Honest' adversary described by Lindell [Lin17].

2.2 Garbling Schemes

In the previous section, we discussed a 2PC Garbled Circuit protocol. However, it is easier to reason about a part of the security of such a protocol using a more generalized construct called a garbling scheme. A garbling scheme leaves out the Oblivious Transfer sub-protocol and focuses on the correctness and security properties of the garbling of the circuit and the input.

A garbling scheme [BHR12] can be expressed as a 5-tuple of algorithms

$$(\mathsf{Garble}(\mathsf{Gb}), \mathsf{Encode}(\mathsf{En}), \mathsf{Decode}(\mathsf{De}), \mathsf{GarbledEval}(\mathsf{Ev}), \mathsf{Eval}(\mathsf{ev}))$$

1. $\mathsf{Garble}(\mathsf{Gb})$ takes as input the function f and the security parameter 1^λ and returns (F, e, d), where F is the Garbled Circuit, e is the encoding information and d is the decoding information.
2. $\mathsf{Encode}(\mathsf{En})$ takes as input the initial input x and the encoding information e, and returns the garbled input X.
3. $\mathsf{GarbledEval}(\mathsf{Ev})$ takes as input the Garbled Circuit F and the garbled input X, and returns a garbled output Y.
4. $\mathsf{Decode}(\mathsf{De})$ takes input the garbled output Y and the decoding information d, and returns the ungarbled output y.
5. $\mathsf{Eval}(\mathsf{ev})$ takes as input the function f and the plain input x and returns $f(x)$. For the sake of simplicity, we denote $\mathsf{ev}(f, x)$ as $f(x)$, which is more familiar.

Relating this to the representation of the Garbled Circuit protocol in the previous section, the inputs g and e from the previous section are partitions of x. That is, x can be expressed as $x = g \| e$.

2.2.1 Properties

The correctness property of a garbling scheme describes its intended behavior. It says, $\forall f$, if $(F, e, d) \leftarrow\!\$\ \mathsf{Gb}(1^\lambda, f)$,

$$\forall x, \mathsf{De}(\mathsf{Ev}(F, \mathsf{En}(x, e)), d) = f(x)$$

Besides correctness, a garbling scheme can have one or more of the following security properties [BHR12]:

1. **Privacy**: Knowledge of (F, X, d) does not reveal anything beyond $f(x)$. This means that it is computationally infeasible for an Evaluator to obtain x, the initial input of the Garbler.

2. **Obliviousness**: Knowledge of (F, X) does not reveal anything beyond itself. This means that despite being able to solve the Garbled Circuit, the solver would not be able to efficiently map the garbled output to the real output without d.

3. **Authenticity**: Knowledge of (F, X) does not allow one to efficiently find a valid $Y' \neq \mathsf{Ev}(F, X)$.

Of these properties, *Privacy* is the one that is most important to us in this work, and it is what we will continue working with in the rest of the thesis.

2.2.2 Defining Privacy

Bellare, Hoang and Rogaway define privacy as two different cryptographic games: an *indistinguishability*-based one (Game $\mathsf{PrvInd}_{G,\Phi}$), and a *simulation*-based one (Game $\mathsf{PrvSim}_{G,\Phi,S}$) as described in Figure 2.2 [BHR12]. In these notations, and elsewhere, Φ refers to the side information function, which describes how much information about the circuit is public. In common usage, it is often either the whole circuit (denoted as $\Phi_{circ}(C)$), the topology of the circuit ($\Phi_{topo}(C)$), or the size of the circuit ($\Phi_{size}(C)$).

$G^b_{\mathsf{PrvInd}_{G,\Phi}}$

$\mathsf{GARBLE}(f_0, f_1, x_0, x_1)$

if $\Phi(f_0) \neq \Phi(f_1)$ **then**
  return $\perp$
if $f_0(x_0) \neq f_1(x_1)$ **then**
  return $\perp$
$(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f_b)$
$X \leftarrow \mathsf{En}(x_b, e)$
return (F, X, d)

$G^b_{\mathsf{PrvSim}_{G,\Phi,S}}$

$\mathsf{GARBLE}(f, x)$

if $b = 0$ **then**
  $(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f)$
  $X \leftarrow \mathsf{En}(x, e)$
else
  $y \leftarrow \mathsf{ev}(f, x)$
  $(F, X, d) \leftarrow \mathcal{S}(1^\lambda, y, \Phi(f))$
return (F, X, d)

Figure 2.2: PrvInd and PrvSim Games - adapted from [BHR12]

In the indistinguishability-based game (Game $\mathsf{PrvInd}^{\mathsf{b}}_{G,\Phi}$), the GARBLE oracle takes as parameters two pairs of functions and corresponding inputs, (f_0, x_0) and (f_1, x_1) such that $f_0(x_0) = f_1(x_1)$, and the bit b that defines which (f, x) pair GARBLE will use. A distinguishing adversary $\mathcal{D}$ is a polynomial-runtime program that tries to

guess the bit-value b that was used to generate the (F, e, d) tuple. An INITIALIZE procedure samples the value of b, and it is unknown to the adversary. GARBLE then uses the selected (f, x) to generate F, X and d and returns it to the adversary. The advantage of the distinguishing adversary here is defined as the difference in the probability that the adversary guesses the correct bit and the probability of guessing randomly ($P = 1/2$).
If b' is $\mathcal{D}$'s guess,

$$\mathsf{Adv}_{\mathcal{D},G^0_{\mathsf{PrvInd}},G^1_{\mathsf{PrvInd}}}(\lambda) = |\Pr[b' = b] - \frac{1}{2}|$$

However, many follow-up works such as Half-Gates [ZRE15] and Free XOR [KS08; App16] use the simulation-based paradigm (the PrvSim game from Figure 2.2). Since privacy here means that running the garbling and encoding algorithms Gb and En does not reveal anything about f or x, we let the simulator $\mathcal{S}$ access everything that a semi-honest party participating in the protocol would be able to see, i.e., the security parameter (1^λ), the output of the algorithm y, and the side information function of the circuit $\Phi(f)$. Then, this simulator tries to generate a $(\bar{F}, \bar{X}, \bar{d})$ tuple whose distribution is computationally indistinguishable from the one generated by actually running Gb and En. The advantage of an adversary $\mathcal{A}$ against the simulation game $G_{\mathsf{PrvSim}^{\mathsf{b}}_{G,\Phi,S}}$ would be defined as

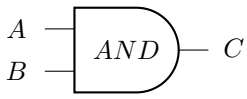
$$\mathsf{Adv}_{\mathcal{A},G^0_{\mathsf{PrvSim}},G^1_{\mathsf{PrvSim}}}(\lambda) = |\Pr\big[1 = \mathcal{A} \to G^0_{\mathsf{PrvSim}}\big] - \Pr\big[1 = \mathcal{A} \to G^1_{\mathsf{PrvSim}}\big]|$$

If the distributions of $(\bar{F}, \bar{X}, \bar{d})$ and (F, X, d) are indistinguishable to any adversary $\mathcal{A}$, the adversary cannot claim to have recovered anything more than what the simulator knew, and hence the privacy of f (excluding $\Phi(f)$) and x is preserved. For further reading on simulators, Lindell provides a comprehensive overview of simulators for beginners [Lin17].

2.2.3 Yao's Garbling Scheme

Before we get into the optimizations for garbling schemes, we must first define a base. In this work, we consider the garbling scheme model defined by Lindell and Pinkas [LP09], and will refer to it as Yao's garbling scheme. The garbling of a circuit f consists of n garbled gates, where n is the number of gates that compose the original circuit before garbling. Each (garbled and ungarbled) gate has two input wires (A and B) and one output wire (C). Each of these ungarbled wires can have two possible values on each wire - 0 or 1 (corresponding to real-life boolean circuits). We will use a, b, and c to refer to the actual bit-values of each wire. Figure 2.3(a) shows this as a diagram and the truth table of an AND gate.

Now, we map each of the possible values on each wire to a random string of length λ denoted by the wire name and the bit-value it was mapped to. This random



(a) AND gate

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

(b) Truth table

Input labels	Ciphertext
(A_0, B_0)	X_1
(A_0, B_1)	X_2
(A_1, B_0)	X_3
(A_1, B_1)	X_4

(c) Generating a garbled table

Figure 2.3: An AND gate and its garbled form.

value is called the wire label for that particular wire and bit-value combination. For example, the random value mapped to the 0-input on wire A would be called A_0. After repeating this process to get all six wire labels $(A_0, A_1, B_0, B_1, C_0, C_1)$, we can generate the garbled table. It is worth noting that this mapping from the wire's bit-value to the wire label corresponds to the encoding information e used in the En function in the description of a garbling scheme in Section 2.2.

The garbled table contains 4 rows, like the truth table, and it contains encryptions of the wire label of the output value of each row ($C_{a \wedge b}$), under two keys - the wire labels of the corresponding inputs (A_a,B_b) that generated the output $a \wedge b$ in that row of the truth table. For example, if we consider row 1 of Figure 2.3(c), we would have

$$X_1 = \mathsf{Enc}(A_0, \mathsf{Enc}(B_0, C_0)).$$

These rows are then permuted so that the relation between the wire labels and the actual bit-values remains hidden.

We know from the protocol description in Section 2.1 that besides the garbling of the circuit, the Garbler also sends the 'garbling' of the inputs to the Evaluator. These garblings of the input refer to the wire labels corresponding only to the inputs that the parties want to evaluate using the circuit. Thus, the Evaluator receives the permuted garbled table, as well as the wire labels of the inputs (which, for a single wire, can either be 0 or 1, but not both simultaneously). We observe from Figure 2.4 that decrypting any row requires possession of a specific A wire label and a specific B wire label. This means that if the Evaluator receives only one A label and one B label, they cannot decrypt any row other than the one that was encrypted using those labels since at least one of the two decryptions will fail. This preserves the privacy of the original bit-value of wire C. However, this requires that the Garbler provides a mapping from the final output wire labels to the real bit-value of that wire. This would correspond to the decoding information d used in the function De.

In this garbling scheme, we assume that when trying to decrypt a ciphertext without the right keys, the decryption will fail and the decryptor will know that it has failed. This can be achieved by concatenating some specific, publicly known padding in the wire labels for wire C. This is related to a property called special correctness where attempting to decrypt a ciphertext without a valid key will result in a decryption failure message with overwhelming probability. The character $\perp$ is commonly used to show such a result in cryptographic notation. Special correctness is a property used later in our work as well.

Ciphertext	Value
X_1	$\mathsf{Enc}(A_0, \mathsf{Enc}(B_0, C_0))$
X_2	$\mathsf{Enc}(A_0, \mathsf{Enc}(B_1, C_0))$
X_3	$\mathsf{Enc}(A_1, \mathsf{Enc}(B_0, C_0))$
X_4	$\mathsf{Enc}(A_1, \mathsf{Enc}(B_1, C_1))$

Figure 2.4: Garbled table for an AND gate

For a larger circuit, garbled tables are generated for each garbled gate in the original circuit, and all of them are transmitted to the Evaluator. In addition to this, a larger circuit would have a larger input as well, so the wire labels corresponding to every input bit are also sent.

2.3 Optimizations to Yao's Garbling Scheme

In the last two decades, there have been many optimizations that improve the efficiency of Yao's garbling scheme, in terms of the computational costs of garbling, evaluating, and also in terms of the bandwidth consumption when transmitting the Garbled Circuit. In this section, we discuss the optimizations that are relevant to our work, namely - Point and Permute (P&P) [BMR90], Garbled Row Reduction (GRR) [NPS99], Free XOR [KS08], and Half-Gates [ZRE15].

2.3.1 Point and Permute

On observing Yao's garbling scheme, it quickly becomes obvious that the Evaluator may have to decrypt 4 rows of the garbled table in order to receive the correct output wire label. Yao's garbling scheme also requires special correctness, which is not difficult to implement, but there is a way to avoid the whole problem in the first place. This technique, now called Point and Permute, is based on the value λ_{W-1}

defined in the Garbled Circuit protocol of Beaver, Micali, and Rogaway [BMR90]. Note that λ continues to serve as the security parameter in this thesis.

To put it simply, for each wire W in the circuit, with wire labels W_0 and W_1, each of length λ, an extra bit is sampled (called the permute or color bit). Let us call this bit l_W. Then, for each wire W,

$$W'_0 = W_0 \| (l_W), \text{ and}$$

$$W'_1 = W_1 \| (l_W \oplus 1)$$

Now, when generating a garbled table, the ciphertexts are ordered based on the permute bit of the wire labels used to encrypt them. For example, if we say that the encryption generated by wire labels A'_a and B'_b would be in row $2l_a + l_b + 1$, and if A'_0 and B'_0 both have the permute bit 1, the garbled table would be ordered as in Figure 2.5. Then, the Evaluator can use the last bit (the permute bit) of both input wire

l_a	corresp. A'_a	l_b	corresp. B'_b	$C'_c = C'_{a \wedge b}$	Ciphertext
0	A'_1	0	B'_1	C'_1	$\mathsf{Enc}(A'_1, \mathsf{Enc}(B'_1, C'_1))$
0	A'_1	1	B'_0	C'_0	$\mathsf{Enc}(A'_1, \mathsf{Enc}(B'_0, C'_0))$
1	A'_0	0	B'_1	C'_0	$\mathsf{Enc}(A'_0, \mathsf{Enc}(B'_1, C'_0))$
1	A'_0	1	B'_0	C'_0	$\mathsf{Enc}(A'_0, \mathsf{Enc}(B'_0, C'_0))$

Figure 2.5: Garbled table for an AND gate with Point and Permute

labels to identify which row to decrypt in order to obtain the output wire label. This process continues until the end of the circuit is processed since each output wire label contains a permute bit as well, and all the garbled tables are ordered based on these permute bits. However, this requires that the Evaluator and Garbler use the same method of permuting the rows, which can be defined in the protocol specification.

This method reduces the number of ciphertexts to be decrypted by the Evaluator to 1, which makes the evaluation faster. Point and Permute also removes the need for the special correctness property that Yao's Garbled Circuit had. It is also of minor note that the length of each wire label increases by one bit, but the security parameter λ remains the same.

2.3.2 Garbled Row Reduction-3

Garbled Row Reduction, also known as Garbled Row Reduction-3, refers to a scheme first introduced by Naor, Pinkas, and Sumner [NPS99, Section 2.5] under the heading

'Optimizing the circuit size'. As discussed briefly in Chapter 1, this optimization allows the number of rows in the garbled table of a boolean gate with two inputs to be reduced to *three* instead of *four*, thus reducing the bandwidth overhead by 25%. The 3 in the name of the scheme refers to the three rows in the garbled table.

This technique works by setting one row of the garbled table of each gate to 0, and therefore assuming it implicitly, instead of sending it. Let us say that row $X_{a,b}$ of the garbled table of a gate is the one being omitted. Here, a and b are the bit-values of the input wires A and B for that gate. Since we would normally compute that row as

$$X_{a,b} = \mathsf{Enc}(A_a, \mathsf{Enc}(B_b, C_c)), \text{ and}$$

$$\text{since } X_{a,b} = 0^\lambda,$$

it would mean that the Evaluator could compute

$$C_c = \mathsf{Enc}^{-1}(B_b, \mathsf{Enc}^{-1}(A_a, 0)).$$

Initially, it might seem that this is insecure since the wire label C_c depends on A_a and B_b instead of being randomly sampled. However, this is not actually the case as the Evaluator cannot see the value for C_c unless it has A_a and B_b, and if it has those two wire labels, it cannot have the wire labels needed to view the $C_{c\oplus 1}$ present in the other rows of the garbled table.

This technique can also be combined with the Point and Permute (P&P) optimization described in Section 2.3.1, where the deleted row would correspond to the permute bits $(0, 0)$. However, in this work, we do not deal with the Point and Permute optimization to reduce the complexity of the proof, especially in Domino.

2.3.3 Free XOR

The Free XOR optimization was a big step in the optimization of Garbled Circuits, allowing the transmission of an XOR gate for free, i.e., without the need to generate, send, or evaluate a garbled table for an XOR gate. This is a big improvement when we consider that circuits can be optimized to maximize the number of XOR gates. This technique was introduced by Kolesnikov and Schneider [KS08].

Instead of sampling each of the wire labels for every wire independently, the Free XOR scheme samples a global offset $\Delta \in \{0,1\}^\lambda$ such that for each wire W,

$$W_1 = W_0 \oplus \Delta, \text{ where } W_0 \leftarrow\!\!\$\ \{0,1\}^\lambda$$

Then, for each XOR gate with input wires A, B and output wire C, the Evaluator can compute the value of the active wire label of C as $C = A \oplus B$ (see Figure

A_a	B_b	$C_c = A_a \oplus B_b$
A_0	B_0	$A_0 \oplus B_0$
A_0	$B_0 \oplus \Delta$	$A_0 \oplus B_0 \oplus \Delta$
$A_0 \oplus \Delta$	B_0	$A_0 \oplus B_0 \oplus \Delta$
$A_0 \oplus \Delta$	$B_0 \oplus \Delta$	$A_0 \oplus B_0$

Figure 2.6: Evaluation of an XOR gate in Free XOR

2.6). Since this calculation of the active wire label of C is always true for XOR gates, the Garbler can just indicate the presence of an XOR gate, omitting the cost of encrypting the output wire labels. These garblings also do not reveal the wire bit-values they correspond to, so privacy remains intact. However, this method of garbling gates requires that the hash or encryption function used to garble the output wire labels satisfy a property called 'Correlation Robustness', since the wire labels used as keys and the output wire label are all in relation to the circuit's global offset Δ.

The notation used in Kolesnikov and Schneider's Free XOR paper is different from the usual encryption under two keys we have seen so far [KS08]. Their scheme uses a hash-based notation to generate a one-time pad, which is then XOR'ed with the output wire label. As mentioned in Chapter 1, this scheme was shown to be secure under the 'Random Oracle' (RO) model [BR93], with the authors also claiming security under some version of correlation-robustness of the hash function (the authors do not describe the exact kind of assumption), which is a weaker assumption than the RO model. They defined the ciphertext corresponding to input bit-values a and b in the garbled table as follows: Given a Random Oracle $H : \{0,1\}^* \mapsto \{0,1\}^{\lambda+1}$, and a gate at index i with operation $op \in \{AND, OR, ...\}$,

$$X_{a,b} = H(A_a \| B_b \| i) \oplus C_c$$

Choi et al. then went on to show that general correlation-robustness was insufficient, and described the exact form of the assumption needed and called it circular correlation-robustness [CKKZ12]. This new assumption is still weaker than the RO model, but stronger than correlation-robustness. Later, Applebaum also proved the security of the Free XOR scheme under a different assumption called 'Linear, Related-Key, Key-Dependent-Message' (LIN-RK-KDM) security for encryption schemes, which uses the following definition for encryption of each row.

$$R_{a,b} \leftarrow\$ \{0,1\}^{\lambda}$$

$$X_{a,b} = (\mathsf{Enc}(A_a, R_{a,b}), \mathsf{Enc}(B_b, R_{a,b} \oplus C_{op(a,b)}))$$

In Applebaum's scheme, the Evaluator decrypts both ciphertexts using the input wire labels it receives, and XORs them together to obtain the active output label C_c. We observe that there are two encryption operations per garbled table row, namely $\mathsf{Enc}(A_a, R_{a,b})$ and $\mathsf{Enc}(B_b, R_{a,b} \oplus C_{op(a,b)})$ (eight in total per gate), compared to Kolesnikov and Schneider's scheme, which requires one hash operation $H(A_a \| B_b \| i) \oplus C_c$ per row. Despite this reduction in efficiency, Applebaum's scheme is interesting because it was published with a detailed method on how to build a LIN-RK-KDM secure encryption scheme out of the Learning Parity with Noise (LPN) assumption. This, in conjunction with the fact that this construction is based on encryption like in Yao's garbling scheme, is what made it attractive for us to use Applebaum's scheme as the Free XOR proof of choice in this thesis. A detailed description of the LPN assumption is available in their paper [App16].

It is also very important to mention that Kolesnikov and Schneider noted that NOT gates can be implemented in a Garbled Circuit for free by simply flipping the bit-value of the wire they serve as input to [KS08, Section 3.1].

2.3.4 Half-Gates

Half-Gates was a novel technique introduced by Zahur, Rosulek, and Evans [ZRE15], which uses the splitting of an AND gate into two half-AND gates and XORing the results to reduce the number of rows to be transmitted. While splitting the gates may seem counterintuitive, the inputs to one of the half-gates are known by the Garbler and the inputs to the other half-gate are known by the Evaluator, and therefore, the outputs of those gates turn into a function of these known values, reducing each AND half-gate to two rows. Then, using the GRR optimization, each of these two-row half-gates are reduced to one row each. The XOR operation itself is free since this scheme builds on top of Free XOR, and we arrive at 2 total ciphertexts per AND gate. The detailed treatment of this scheme is beyond our scope; see [ZRE15].

Chapter 3
Pen-and-Paper State-Separating Proof

This chapter introduces the notion of a State Separating Proof (SSP), and how it differs from a traditional code-based game-playing proof. Then, we define our pen-and-paper state-separating proof for the Free XOR garbling scheme here.

3.1 State Separating Proofs

State separating proofs are a style of proofs built upon code-based game-playing proofs discussed by Bellare and Rogaway [BR06]. They define code-based game-playing as proofs that encompass an adversary's interaction with its environment using code (could be pseudocode or any other formalized programming language), and these interactions are referred to as a game. The game contains subroutines that initialize the state of the game, then run the adversary program. The adversary can call oracles, which are functions exposed by the game, and the game concludes by running a finalize subroutine which outputs a bit from the adversary. The adversary has a similar interaction between two different versions of the game, one in the real world and one in the ideal world, whose behaviors are parameterized by a bit $b = 0$ and $b = 1$, respectively. The success of the adversary is characterized by its 'advantage', defined as the difference in the probability of the adversary returning the same bit in each game. In cryptographic notation, given two games G_{real} and G_{ideal},

$$\mathsf{Adv}_{\mathcal{A},G^0_{real},G^1_{ideal}}(\lambda) = |\Pr[A \to G_{real} = 1] - \Pr[A \to G_{ideal} = 1]|$$

Games and Oracles are constructs of a game-playing proof.

- **Oracles** are defined as pieces of code that provide some functionality. They usually either return a value or modify the state of the Game.
- **Games** are a complete set of oracles that do not call other oracles, an initialize procedure, and a finalize procedure that returns a bit. An adversary interacts with a game.

However, state-separating proofs differ by introducing the notion of a package. A package has its own local state, and it functions as an intermediate layer between a

game and an oracle. Games contain packages, and packages contain oracles. In a state-separating proof [BDF+18], they are defined as follows:

- **Oracles** are pieces of executable code. They usually either return a value or modify the state of the package they are contained in. Oracles in a package cannot call each other to prevent recursion.
- **Packages** are defined as collections of related oracles with a shared state.
- **Games** are defined as packages with no dependencies on oracles of other packages ($[G \rightarrow] = \varnothing$).
- An **adversary** can be defined as a special package that is not a dependency of another package. It always outputs a single bit on termination ($[\rightarrow A] = \varnothing$).

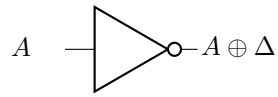
State separating proofs allow us to restrict access to the state of each package to oracles in that package, and only provide the oracles themselves as an interface to other packages. This lets them be highly modular and composable, enabling us to reuse a package used in one reduction in another, while also allowing changes in state that do not make changes to the whole game. If these changes cascaded to the whole game, it would require the cryptographer to once again verify the correctness of the game in the new state. [BDF+18] formally defines the constructs we use here, and the mathematical properties of packages such as associativity and compositions. We slightly diverge from their notation of sequential composition by using the $\rightarrow$ operator instead of $\circ$.

3.2 Privacy of the Garbling of Various Gates using Free XOR

In our work, we are concerned with the garbling of NOT, XOR, and AND gates, since every boolean circuit can be represented with that combination of gates (functional completeness of boolean gates). Despite the fact that NOT and AND gates by themselves are functionally complete, we are interested in XOR gates since they are the basis on which the Free XOR optimization improves the efficiency of evaluating a Garbled Circuit. We discuss these gates in order:

3.2.1 NOT Gate

Bit-value	Wire label
0	W
1	$W \oplus \Delta$



Bit-value	Wire label
0	$W \oplus \Delta$
1	W

Figure 3.1: Wire label mappings before (left) and after (right) a NOT gate.

Kolesnikov and Schneider note that NOT gates do not need to be garbled separately at all [KS08, Section 3.1]. A NOT gate simply inverts the mapping between the bit-value of a wire and its wire label, i.e., if the wire label W_b corresponds to bit b before encountering a NOT gate, the wire label W_b will correspond to bit $b \oplus 1$ after the NOT gate (Figure 3.1 describes this inversion). Since NOT gates are not garbled, they are not part of the security proof of the privacy of the garbling of a circuit.

3.2.2 XOR Gate

Similarly to NOT gates, XOR gates are garbled for 'free' in the Free XOR scheme. What this means in practice is that the Evaluator directly computes the output wire label of an XOR gate using the input wire labels (as discussed in Section 2.3.3). There is no garbled table that the Evaluator receives, and no decryption that takes place, and the security of the XOR gate depends directly on the secrecy of the circuit offset Δ. As we will see in Section 3.2.3, AND gates generate their garbled tables using encryption schemes that are LIN-RK-KDM secure. This means that the relation caused between wire labels by deriving the output wire label of an XOR gate by XORing the input wire labels that may both be functions of the offset Δ is accounted for in the underlying assumption we use to prove the security of an AND gate. That is to say that even in this related-key and key-dependent message setting, there is no privacy loss from an XOR gate in the Free XOR scheme, if Δ remains secret.

3.2.3 AND Gate

The privacy of the garbling of AND gates in the Free XOR scheme becomes the main focus of this work. As mentioned in Section 2.3.3, we base our state-separating proof on Applebaum's proof in [App16].

Since we use the simulation-based definition of privacy (discussed in Section 2.2.2), we define the garbling of an AND gate in the Free XOR scheme as the security game $G^0_{\mathsf{Sim\text{-}FreeXOR}}$ (defined in Figure 3.2). The game exposes a GARBLE oracle that provides the core functionality of garbling an AND gate. It first obtains the evaluation of the circuit f with the input x to determine the output it has to garble. The input x is partitioned into the bit-values for wires A and B. Then, it samples a random offset value, and wire labels for the two active input wires and active output wire. It also samples four random values, one for each row of the garbled table. The order in which these values are used is irrelevant since they are not used across multiple rows of the garbled table; therefore, they can be used independently of the index of the garbled table, as long as they are not reused. The garbled table is then generated as described in Section 2.3.3. The garbled inputs are the wire labels sampled corresponding to the active wire bit-values. The decoding information d is

$G^0_{\mathsf{Sim\text{-}FreeXOR}}$

$\underline{\underline{\mathsf{GARBLE}(f, x)}}$

$c \leftarrow f(x)$

$(a, b) \leftarrow x$

$\Delta \leftarrow\$ \{0,1\}^\lambda$

$A_a, B_b, C_c \leftarrow\$ \{0,1\}^\lambda$

$A_{a\oplus 1} \leftarrow A_a \oplus \Delta; B_{b\oplus 1} \leftarrow B_b \oplus \Delta; C_{c\oplus 1} \leftarrow C_c \oplus \Delta$

$R_{00}, R_{01}, R_{10}, R_{11} \leftarrow\$ \{0,1\}^\lambda$

F : Table

$F[a][b] \leftarrow\$ (\mathsf{Enc}(A_a, R_{00}), \mathsf{Enc}(B_b, R_{00} \oplus C_{a\wedge b}))$

$F[a][b \oplus 1] \leftarrow\$ (\mathsf{Enc}(A_a, R_{01}), \mathsf{Enc}(B_{b\oplus 1}, R_{01} \oplus C_{a\wedge(b\oplus 1)}))$

$F[a \oplus 1][b] \leftarrow\$ (\mathsf{Enc}(A_{a\oplus 1}, R_{10}), \mathsf{Enc}(B_b, R_{10} \oplus C_{(a\oplus 1)\wedge b}))$

$F[a \oplus 1][b \oplus 1] \leftarrow\$ (\mathsf{Enc}(A_{a\oplus 1}, R_{11}), \mathsf{Enc}(B_{b\oplus 1}, R_{11} \oplus C_{(a\oplus 1)\wedge(b\oplus 1)}))$

$X \leftarrow (A_a, B_b)$

$d \leftarrow \{C_c : c\}$

return (F, X, d)

Figure 3.2: Code for $G^0_{\mathsf{Sim\text{-}FreeXOR}}$

set as a map from the active wire label for the output wire to the bit-value of the output wire.

$G^b_{\mathsf{LIN-RK-KDM}}$

$\underline{\text{State:}}$

Δ : global circuit offset

$\underline{\underline{\mathsf{ENC}(key, msg, z \in \{0,1\})}}$

if $\Delta = \perp$: // If Δ is unset

  $\Delta \leftarrow\$ \{0,1\}^\lambda$ // Sample a fresh Δ

if $b = 0$:

  $ct \leftarrow\$ \mathsf{Enc}(key \oplus \Delta,\ msg \oplus z \cdot \Delta)$

if $b = 1$:

  $ct \leftarrow\$ \mathsf{Enc}(key \oplus \Delta,\ 0^{|msg|})$

return ct

Figure 3.3: Code for $G^b_{\mathsf{LIN-RK-KDM}}$

Applebaum defines the notion of LIN-RK-KDM security of a symmetric encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ as the computational indistinguishability of the two games Real_s

and Fake_s [App16, Section 3.2], where

$$\mathsf{Real}_s : (\Delta, M, b) \mapsto \mathsf{Enc}_{S+\Delta}(M + bS), \text{ and,}$$

$$\mathsf{Fake}_s : (\Delta, M, b) \mapsto \mathsf{Enc}_{S+\Delta}(0^{l \times N}).$$

We redefine the same security notion to better fit our SSP setting:

Definition 3.1. An encryption scheme is said to be LIN-RK-KDM secure if any adversary $\mathcal{A}$ interacting with $G^b_{\mathsf{LIN-RK-KDM}}$ (Figure 3.3) in the real and ideal worlds cannot distinguish between them with more than negligible advantage, i.e.,

$$\mathsf{Adv}(\mathcal{A}, G^0_{\mathsf{LIN-RK-KDM}}, G^1_{\mathsf{LIN-RK-KDM}}) \leq \mathsf{negl}(\lambda).$$

Our $G^b_{\mathsf{LIN-RK-KDM}}$ provides an interface ENC that takes the key, message, and an extra parameter z as input, and returns an encryption of the message in the real game, and an encryption of zeroes in the ideal game (Figure 3.3). It is important to note that the real LIN-RK-KDM game does not simply return an encryption under the key used as input to the ENC oracle. It XORs this key by an offset of Δ, and optionally XORs the same offset to the message, depending on the value of z.

Since we now have a base assumption to build from, we try to reduce the security of the real and simulated garbling games (Figures 3.2 and 3.4) to the LIN-RK-KDM assumption.

Theorem 3.2. *If* Enc *is the LIN-RK-KDM secure encryption scheme used in the Free XOR garbling scheme, then for f={AND} and* $\forall x \in \{0,1\}^2$*, there exists a probabilistic, polynomial-time (PPT) simulator* $G^1_{\mathsf{Sim\text{-}FreeXOR}}$ *and reduction* $\mathcal{R}$ *such that for all PPT adversaries* $\mathcal{A}$*,*

$$\mathsf{Adv}(\mathcal{A}, G^0_{\mathsf{Sim\text{-}FreeXOR}}, G^1_{\mathsf{Sim\text{-}FreeXOR}}) \leq \mathsf{Adv}(\mathcal{A} \to \mathcal{R}, G^0_{\mathsf{LIN-RK-KDM}}, G^1_{\mathsf{LIN-RK-KDM}})$$

We now describe the simulation strategy used by $G^1_{\mathsf{Sim\text{-}FreeXOR}}$ (Figure 3.4). The simulator receives as input the output of the circuit evaluation, $f(x)$, which corresponds to the bit-value of the output wire of the AND gate. It then samples random bitstrings for the wire labels corresponding to the zero bit-values on the input and output wires. It also samples the global circuit offset Δ, and random values for each row of the garbled table. The correctness property of garbling requires that when the Evaluator evaluates a garbled table, they are able to decrypt exactly one row of the table with the input wire labels they have. It also requires that the final decrypted output wire label corresponds to the correct function evaluation $f(x)$.

Since the simulator knows $f(x)$, the simulation strategy involves mapping one of the output wire labels (our simulator in Figure 3.4 chooses C_0) to the bit-value $f(x)$,

$G^1_{\mathsf{Sim\text{-}FreeXOR}}$

$\underline{\underline{\mathsf{GARBLE}(f(x), \Phi(f))}}$

$c \leftarrow f(x)$

$\Delta \leftarrow\$ \{0,1\}^\lambda$ // global offset

$A_0, B_0, C_0 \leftarrow\$ \{0,1\}^\lambda$

$A_1 \leftarrow A_0 \oplus \Delta$

$B_1 \leftarrow B_0 \oplus \Delta$

$C_1 \leftarrow C_0 \oplus \Delta$

$R_{00}, R_{01}, R_{10}, R_{11} \leftarrow\$ \{0,1\}^\lambda$

F : Table

$F[0][0] \leftarrow\$ (\mathsf{Enc}(A_0, R_{00}), \mathsf{Enc}(B_0, R_{00} \oplus C_0))$

$F[0][1] \leftarrow\$ (\mathsf{Enc}(A_0, R_{01}), \mathsf{Enc}(B_1, 0^\lambda))$

$F[1][0] \leftarrow\$ (\mathsf{Enc}(A_1, 0^\lambda), \mathsf{Enc}(B_0, R_{10} \oplus C_1))$

$F[1][1] \leftarrow\$ (\mathsf{Enc}(A_1, 0^\lambda), \mathsf{Enc}(B_1, 0^\lambda))$

$X \leftarrow (A_0, B_0)$

$d \leftarrow \{C_0 : c\}$

return (F, X, d)

Figure 3.4: Code for $G^1_{\mathsf{Sim\text{-}FreeXOR}}$

and making sure that the row that the adversary can decrypt actually encrypts the chosen output wire label. We also need to ensure that if the adversary receives wire labels A_0 and B_0 as the garbled input X, then every encryption on the table that uses these as the key is encrypted with real messages, but for the other encryptions, we can forge them by encrypting zeroes. This concludes the generation of the garbled table F. The garbled input X is a tuple of the active wire labels of the first and second wires, which we assumed to be A_0 and B_0 in our simulator. The choice only matters to select which row of the garbled table is available to the adversary for decryption. Finally, the decoding information d is a map from our chosen active output wire label that was encrypted with the active input wire labels, to the real value of $f(x)$ that is available to the adversary. The simulator then returns this (F, X, d) tuple.

Another problem we quickly realize is that the two GARBLE oracles expect different inputs. GARBLE in the real game takes a circuit f, which in this case is a simple AND operation, and it also takes the inputs to the circuit, x. The GARBLE oracle in the ideal game, however, takes $f(x)$ and $\Phi(f)$. To solve this incompatibility, we provide another game $G^b_{wrapper}$ that provides a shared interface GBL that can be

$G^0_{\mathsf{Wrapper},Gbl}$	$G^1_{\mathsf{Wrapper},Sim}$
$\underline{\underline{\mathsf{GBL}(f,x)}}$	$\underline{\underline{\mathsf{GBL}(f,x)}}$
$(F,X,d) \leftarrow\$ \mathsf{GARBLE}(f,x)$	$(F,X,d) \leftarrow\$ \mathsf{GARBLE}(f(x),\Phi(f))$
return (F,X,d)	**return** (F,X,d)

Figure 3.5: Code of $G^b_{wrapper}$

composed with the real and ideal $G^b_{\mathsf{Sim\text{-}FreeXOR}}$ games. However, we do not include this in our proof in order to keep it simple to understand and translate into Domino code.

Lemma 3.3. (Real code equivalence). *Let $\mathcal{R}$ be the reduction defined in Figure 3.6. Then,*

$$\mathcal{R} \rightarrow G^0_{\mathsf{LIN-RK-KDM}} \stackrel{code}{\equiv} G^0_{\mathsf{Sim\text{-}FreeXOR}}.$$

$\mathcal{R}$

$\underline{\underline{\mathsf{GARBLE}(f,x)}}$

$c \leftarrow f(x)$

$(a,b) \leftarrow x$

$A_a, B_b, C_c \leftarrow\$ \{0,1\}^\lambda$

$R_{00}, R_{01}, R_{10}, R_{11} \leftarrow\$ \{0,1\}^\lambda$

F : Table

$F[a][b] \leftarrow\$ (\mathsf{Enc}(A_a, R_{00}), \mathsf{Enc}(B_b, R_{00} \oplus C_0))$

$F[a][b \oplus 1] \leftarrow\$ (\mathsf{Enc}(A_a, R_{01}), \mathsf{ENC}(B_b, R_{01} \oplus C_0, a \wedge (b \oplus 1)))$

$F[a \oplus 1][b] \leftarrow\$ (\mathsf{ENC}(A_a, R_{10} \oplus C_0, (a \oplus 1) \wedge (b)), \mathsf{Enc}(B_b, R_{10}))$

$F[a \oplus 1][b \oplus 1] \leftarrow\$ (\mathsf{ENC}(A_a, R_{11}, (a \oplus 1) \wedge (b \oplus 1)),$
$\mathsf{ENC}(B_b, R_{11} \oplus C_0, (a \oplus 1) \wedge (b \oplus 1)))$

$X \leftarrow (A_a, B_b)$

$d \leftarrow \{C_0 : c\}$

return (F,X,d)

Figure 3.6: Code for Reduction $\mathcal{R}$

We describe the reduction $\mathcal{R}$ which can be composed with $G^b_{\mathsf{LIN-RK-KDM}}$. The interface to the reduction is a GARBLE oracle. It initially computes $f(x)$ and samples three bitstrings A_a, B_b, and C_c of length λ assuming a,b and c are the active wire labels. It also samples four random bitstrings of length λ, just like in $G^0_{\mathsf{Sim\text{-}FreeXOR}}$. However, unlike $G^0_{\mathsf{Sim\text{-}FreeXOR}}$, the reduction does not sample its own Δ. This is

because Δ is part of the state of the LIN-RK-KDM game, and therefore the reduction must rely on calling the ENC oracle of $G^b_{\mathsf{LIN-RK-KDM}}$ in order to generate any of the ciphertexts that require inactive wire labels. The rest of the reduction follows the same steps as $G^0_{\mathsf{Sim\text{-}FreeXOR}}$, and inlining the ENC oracle gives us code that is equivalent ($\overset{code}{\equiv}$) to $G^0_{\mathsf{Sim\text{-}FreeXOR}}$, except for one important difference. For the calculation of $F[a \oplus 1][b]$, $\mathcal{R}$ encrypts the output wire label in the first ciphertext, as opposed to the second ciphertext, where $G^0_{\mathsf{Sim\text{-}FreeXOR}}$ encrypts it. This is done because only the first encryption calls the ENC oracle, which can add an offset of Δ to the message. The correctness of the garbling scheme remains intact since XORing the messages still returns the correct value of the output wire label.

G^b_{Hybrid}

$\mathsf{GETRANDOMS}(w)$

$R \leftarrow\!\$\ \{0,1\}^\lambda$
if b **then**
  $r_1 \leftarrow R \oplus w; r_2 \leftarrow R$
else
  $r_1 \leftarrow R; r_2 \leftarrow R \oplus w$
return (r_1, r_2)

G^b_{Hybrid}

$\mathsf{GETRANDOMS2}(w)$

$R \leftarrow\!\$\ \{0,1\}^\lambda$
if b **then**
  $r_1 \leftarrow 0^\lambda; r_2 \leftarrow R$
else
  $r_1 \leftarrow 0^\lambda; r_2 \leftarrow R \oplus w$
return (r_1, r_2)

Figure 3.7: Code for G^b_{Hybrid}

After inlining $\mathcal{R}$ with $G^0_{\mathsf{LIN-RK-KDM}}$, Lemma 3.4 provides a sufficient condition for the equivalence established in Lemma 3.3 to hold. Specifically,

Lemma 3.4. *Let* Enc *be a LIN-RK-KDM secure encryption scheme. Then,*

$$\begin{array}{rll} & (\mathsf{Enc}(A_{a\oplus 1}, R_{10} \oplus C_{0\oplus((a\oplus 1)\wedge b)}), & \mathsf{Enc}(B_b, R_{10})) \\ \overset{c}{\approx} & (\mathsf{Enc}(A_{a\oplus 1}, R_{10}), & \mathsf{Enc}(B_b, R_{10} \oplus C_{(a\oplus 1)\wedge b})) \end{array}$$

$$\implies \mathcal{R} \to G^0_{\mathsf{LIN-RK-KDM}} \overset{code}{\equiv} G^0_{\mathsf{Sim\text{-}FreeXOR}}$$

It is possible to prove that both sides of the hypothesis of Lemma 3.4 are indeed functionally equivalent. If we define an oracle $\mathsf{GETRANDOMS}$ as in Figure 3.7, we can rewrite both sides of the hypothesis of Lemma 3.4 as variations of $(\mathsf{Enc}(A_{a\oplus 1}, r_1), \mathsf{Enc}(B_b, r_2))$, where $C_{c\oplus((a\oplus 1)\wedge b)}$ is the input to the $\mathsf{GETRANDOMS}$ oracle, and r_1 and r_2 are its outputs. The left-side of the hypothesis is obtained by setting $b = 1$ and the right-side by setting $b = 0$. The codes obtained in this way are functionally equivalent because the respective random values returned from $\mathsf{GETRANDOMS}$ for either value of b are of the same distribution.

Thus, we establish that Lemma 3.4 is unconditionally true, and Lemma 3.3 always holds.

Now, we move on to showing equivalence between the ideal games.

Lemma 3.5. (Ideal code equivalence). *Let $\mathcal{R}$ be the reduction defined in Figure 3.6. Then,*

$$\mathcal{R} \to G^1_{\mathsf{LIN-RK-KDM}} \overset{code}{\equiv} G^1_{\mathsf{Sim\text{-}FreeXOR}}.$$

On inlining oracle ENC of $G^1_{\mathsf{LIN-RK-KDM}}$ into $\mathcal{R}$, we obtain code that is very similar to $G^1_{\mathsf{Sim\text{-}FreeXOR}}$. This is not immediately obvious, so we provide the explicit code of $\mathcal{R} \to G^1_{\mathsf{LIN-RK-KDM}}$ in Appendix A.1. $G^1_{\mathsf{Sim\text{-}FreeXOR}}$ also requires some small transformations to make the equivalence to $\mathcal{R} \to G^1_{\mathsf{LIN-RK-KDM}}$ obvious, and we provide Game $G^{1\prime}_{\mathsf{Sim\text{-}FreeXOR}}$ in Appendix A.2 containing said small transformations.

Here, we would like to discuss the transformations we have made. The composition of $\mathcal{R}$ to oracle ENC is quite straightforward; we inherit a state from $G^1_{\mathsf{LIN-RK-KDM}}$ which stores the global offset Δ. We also add code to $\mathcal{R}$ to sample Δ; however, this can be inlined without the check for an already existing Δ, since ENC is not called from outside of this game, and therefore we will never need to redefine it.

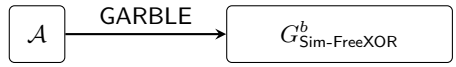
When rearranging $G^1_{\mathsf{Sim\text{-}FreeXOR}}$ into $G^{1\prime}_{\mathsf{Sim\text{-}FreeXOR}}$, we define constant variables a and b which are set to 0, simply to match the code in $\mathcal{R} \to G^1_{\mathsf{LIN-RK-KDM}}$. We also modify all the wire labels so that we refer to their bit-values using a, b, and c instead of zeroes and ones. After making these changes, we can assert the following:

Lemma 3.6. *Let* Enc *be a LIN-RK-KDM secure encryption scheme. Then,*

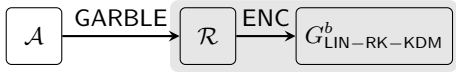
$$\begin{aligned} & (\mathsf{Enc}(A_a \oplus \Delta, 0^\lambda), \quad \mathsf{Enc}(B_b, R_{10})) \\ \overset{c}{\approx} \quad & (\mathsf{Enc}(A_a \oplus \Delta, 0^\lambda), \quad \mathsf{Enc}(B_b, R_{10} \oplus C_c \oplus \Delta)) \end{aligned}$$

$$\implies \mathcal{R} \to G^1_{\mathsf{LIN-RK-KDM}} \overset{code}{\equiv} G^1_{\mathsf{Sim\text{-}FreeXOR}}$$

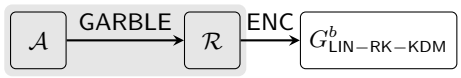
Similar to the earlier case with Lemma 3.4, it is possible to prove that the hypothesis of Lemma 3.6 is always true as well. If we define an oracle $\mathsf{GETRANDOMS2}$ as in Figure 3.7, we can rewrite both sides of the hypothesis of Lemma 3.6 as variations of $(\mathsf{Enc}(A_a \oplus \Delta, r_1), \mathsf{Enc}(B_b, r_2))$, where $C_c \oplus \Delta$ is the input to the $\mathsf{GETRANDOMS}$ oracle, and r_1 and r_2 are its outputs. The left-side of the hypothesis is obtained by setting $b = 1$ and the right-side by setting $b = 0$. The codes obtained in this way are functionally equivalent because the respective random values returned from $\mathsf{GETRANDOMS}$ for either value of b are of the same distribution.



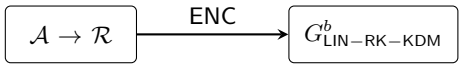
(a) Adversary $\mathcal{A}$ calls oracle GARBLE of Sim-FreeXOR Game



(b) Sim-FreeXOR game $\overset{code}{\equiv} \mathcal{R} \to$ LIN-RK-KDM game



(c) Associativity allows us to compose $\mathcal{A} \to \mathcal{R}$



(d) The composed adversary $\mathcal{A} \to \mathcal{R}$ calls oracle ENC of the LIN-RK-KDM game

Figure 3.8: A visualization of the adversary transformation at the end of the proof in Section 3.2.3

Thus, we establish that Lemma 3.6 is unconditionally true, and Lemma 3.5 always holds.

Now, we make the following assumption to prove Theorem 3.2 by contradiction:

Assumption 3.1. There exists a successful adversary $\mathcal{A}$ that can break the security of $G^b_{\mathsf{Sim\text{-}FreeXOR}}$, i.e.,

$$\mathsf{Adv}(\mathcal{A}, G^0_{\mathsf{Sim\text{-}FreeXOR}}, G^1_{\mathsf{Sim\text{-}FreeXOR}}) > \mathsf{negl}(\lambda)$$

However, since

$$G^0_{\mathsf{Sim\text{-}FreeXOR}} \overset{code}{\equiv} \mathcal{R} \to G^0_{\mathsf{LIN-RK-KDM}} \qquad \text{(Lemma 3.3)}$$
$$\text{and } G^1_{\mathsf{Sim\text{-}FreeXOR}} \overset{code}{\equiv} \mathcal{R} \to G^1_{\mathsf{LIN-RK-KDM}} \qquad \text{(Lemma 3.5)},$$

we can rewrite Assumption 3.1 as

$$\mathsf{Adv}(\mathcal{A}, \mathcal{R} \to G^0_{\mathsf{LIN-RK-KDM}}, \mathcal{R} \to G^1_{\mathsf{LIN-RK-KDM}}) > \mathsf{negl}(\lambda)$$

Package composition is associative, since the states of the packages are separated from each other in a state-separating proof [BO23, Section III(C)]. Therefore, we can rewrite Assumption 3.1 once again, as

$$\mathsf{Adv}(\mathcal{A} \to \mathcal{R}, G^0_{\mathsf{LIN-RK-KDM}}, G^1_{\mathsf{LIN-RK-KDM}}) > \mathsf{negl}(\lambda).$$

The above statement implies that there exists a PPT adversary $\mathcal{A} \to \mathcal{R}$ that achieves non-negligible advantage against the LIN-RK-KDM security game, contradicting Definition 3.1. Therefore, Assumption 3.1 is false. Hence, the advantage of any PPT adversary against $G^b_{\mathsf{Sim\text{-}FreeXOR}}$ is negligible, and Theorem 3.2 follows. □

Chapter 4
Formalizing Free XOR in Domino

This chapter describes the formalization of the SSP introduced in the previous chapter using Domino. We first introduce the aspects of Domino related to the formalization, including its project structure and mechanisms used to represent cryptographic constructions and their security arguments. We then describe how the Free XOR construction was translated from its pen-and-paper description to Domino. While the rest of the proof could be successfully verified by Domino, one equivalence game hop could not be verified. Therefore, we discuss the assumptions and conjectures used, and the invariants and randomness mappings required by the equivalence game hops, as well as the limitations encountered during formalization.

4.1 Domino

Domino is a proof assistant for State-separating proofs (SSPs) (Section 3.1) that allows the user to validate the correctness of a SSP using simple and familiar syntax. To allow this, it converts the SSP-style Domino language code into corresponding code in SMT-LIB, a language used by programs called SMT solvers that use a variety of mathematical theorems to determine if a given problem is satisfiable with some possible combination of inputs under specific constraints. If the problem is satisfiable, the SMT solver can also return the inputs that make the problem satisfiable. Domino requires a SMT solver to be installed on the system - either Z3 theorem prover (Z3) [dMB08], or CVC5, in order to process these SMT-LIB files that are generated.

The SSP-style Domino syntax was designed to use very similar keywords and constructs as those in SSPs – packages, games, and theorems. Packages are composed into games, and these games are used to build a theorem. The explicit relation between the games used in the theorem are defined in the form of game hops. Functional equivalence of pairs of games is argued using invariants that relate their states, and mappings between the randomnesses sampled in either game [BERW26]. These invariants and randomness mappings are written directly in the SMT-LIB language.

An equivalence relation that Domino tries to verify as part of a theorem is referred to as a lemma. In order to establish functional equivalence, games are then checked against one of many in-built and custom lemmata depending on the relations needed in order to prove the correctness of the theorem.

4.1.1 The Domino binary

Domino is published as a Rust crate and is available on GitHub. At the time of writing, Domino is under active development, and its API and language features are subject to frequent changes, including breaking changes between versions.

The main Domino program is exposed as a command line binary. We use two main subcommands in our formalization:

- `domino latex` is used to generate LaTeXcode for the packages, games, theorems and graphs that show how these are related to each other. It uses the underlying SMT solver to generate code that draws the reduction graphs when the output .tex files are compiled. This command also generates pseudocode for every package that is used in the theorem. Z3 is the default SMT solver used in this version of Domino [BERW]. The `.tex` files are stored in the `_build\latex` subdirectory.
- `domino prove` is used to validate each of the game hops that are defined in the theorem file. It also uses the SMT solver to validate the reduction and equivalence game hops. Equivalence game hops require an additional randomness mapping step, and may require a separate invariant file. `domino prove` accepts parameters, and the following are used in this project:
  - `-t` generates transcripts which can be used to review the outputs of the SMT solver.
  - `--proof` selects a specifc theorem to be proven and is useful if the proof code is split into multiple theorems and the user needs to prove a specific one. It is followed by a string parameter with the name of the theorem.
  - `--proofstep` selects the proof step to be proven in the selected theorem. It is followed by a integer which is the index of the proof step to be proven. Indexing starts at 0.
  - `--claim` selects the lemma to be proven in the selected proofstep. It takes a string parameter that is the full name of the lemma to be proven.

Domino has two main dependencies:

- A recent Rust toolchain. Instructions on setting this up can be found at [Rus].
- A functioning Z3 and CVC5 installation and the executables on `PATH`. The installation instructions for these can be found at the respective websites of the projects [CVC; Z3].

It is important to note that the current version of Domino prefers Z3 for running `domino latex` since it is faster at generating the dependency graphs. The `-s cvc5`

option can be used to force it to use CVC5 for graph resolution. However, for `domino prove`, CVC5 is the default SMT solver, and Z3 is only supported using an adaptor script which was not tested by us [BERW26].

After setting up these dependencies, Domino can be installed using the installation instructions in the Domino repository [BERW].

4.1.2 Packages

Packages in Domino are syntactically the same as packages in SSPs. The core of a package are the oracles that it exposes. Each oracle defines what arguments it expects, and its return values. An SSP package can also be composed with other packages, and Domino allows this using an `import oracle` syntax that can be used to declare external oracles that can composed with this package. External oracle calls cannot be inlined, however, and must use an explicit `invoke <ORACLE-NAME>` syntax that assigns the value to some variable. Listing 1 shows a sample Domino package with placeholders.

```
1  package Reduction {
2      params {
3          /*an integer parameter*/
4          n: Integer,
5          /*a function parameter that maps two
6          bitstrings of length n to a bitstring of length m*/
7          fn1: fn Bits(n), Bits(n), -> Bits(m),
8      }
9      /*state of the package*/
10     state {
11         x: Bits(n),
12     }
13     /*external oracles consumed in this package*/
14     import oracles {
15         EXT1(param1: Integer) -> Bits(m),
16     }
17     /*an oracle exposed by this package*/
18     oracle ORA(f: Bits(*), x: (Bool, Bool)) -> (Bits(m),Bits(m))
19     {
20         temp <- invoke EXT1(n);
21         temp2 <- fn1(n);
22         return (temp,temp2);
23     }}
```

Listing 1: Example Domino package

4.1.3 Compositions

Compositions, or games, in Domino are syntactically equivalent to games in SSPs. A game first defines constants that will be passed to package instances. Then, it defines instances of packages that are used in the definition of the game. The compose block is used to define the composition of one or more packages by declaring the oracles of each package instance that are accessible to other package instances in the composition. The compose block also has a special keyword 'adversary' which is used to define the specific package instances that are accessible to an adversary. Listing 2 shows a sample Domino game with placeholders.

```
composition Game_R_Pkg1 {
    /*an integer const parameter passed
     to instances*/
    const n: Integer;
    /*a const function that is passed to
    package instances*/
    const fn1: fn Bits(n), Bits(n), -> Bits(m);
    instance R = Reduction{
       /*instantiate package Reduction
         with the following parameters*/
        params { n: n, fn1: fn1, }
    }
    instance Pkg1 = Package1{
        params { n: n, }
    }
    compose {
        /*Adversary has access to oracle ORA
         of Reduction instance R*/
        adversary: { ORA: R },
        /*R has access to oracle EXT1
         of Package1 instance Pkg1*/
        R: { EXT1: Pkg1 }
    }}
```

Listing 2: Example Domino composition

4.1.4 Theorems

Theorems define the relations between different games in Domino. They first define constants that are passed to game instances defined right after. Then, the games that are used in the theorem are instantiated with these constants. Following this, the assumptions block is used to define the assumptions under which the theorem is supposed to be proven. Finally, the gamehops block defines the game hops used to

relate the games that the theorem is trying to prove functionally equivalent. These game hops can either be reductions, conjectures, or equivalences. Reductions map two games to each other based on some underlying assumption which is defined in the assumption block earlier. Conjectures declare two games to be functionally equivalent without any further reasoning, and Domino accepts their equivalence as-is. Equivalences map games based on lemmata, and are expressed in terms of randomness mappings and invariant relations. Each equivalence is checked for specific lemmata such as `same-output` and `equal-aborts`, which together show that for any given input, the two games output the same values (input-output equivalence), while the `invariant` ensures that some condition remains true before and after the oracles execute. An equivalence proof step may also contains references to other lemmata that need to be verified for that proof step to be valid. These custom definitions for lemmata are written in SMT-LIB. Listing 3 shows a sample Domino theorem with placeholders, and without custom lemmata.

```
theorem Theorem1 {
    /*an integer const parameter passed
     to game instances*/
    const n: Integer;
    /*a const function that is passed to
     game instances*/
    const fn1: fn Bits(n), Bits(n), -> Bits(m);
    /*Game instances used in the theorem*/
    instance LeftGame = Game_R_Pkg1{
        params { n: n, fn1: fn1 }
    }
    instance RightGame = Game2{
        params { n:n, fn1: fn1 }
    }
    instance BaseLGame = Game3{
        params { n:n, fn1: fn1 }
    }
    instance BaseRGame = Game4{
        params { n:n, fn1: fn1 }
    }
    /*... other games...*/
    assumptions {BaseGame: BaseLGame ~ BaseRGame}
    gamehops {
    reduction LeftGame RightGame {
        assumption BaseGame
        map {
        /*map packages of LeftGame and BaseLGame*/
        }
        map {
```

```
30          /*map packages of RightGame and BaseRGame*/
31          }
32      }
33      equivalence GameN GameM {
34          invariant: [
35              ./theorem/invariants/invariant.smt2
36          ]
37          ORACLE: {
38              lemmas {
39                  invariant: [no-abort]
40                  same-output: [no-abort]
41                  equal-aborts: []
42                  /*other lemmata*/
43              }
44          }
45      }
46      /*...*/
47  }}
```

Listing 3: Example Domino theorem

4.1.5 Project Structure

A Domino project directory requires the presence of an empty `ssp.toml` file to indicate the project root. The project root contains three other mandatory folders - `packages`, `games`, and `theorem`, each of which contain their namesakes. Most Domino subcommands also take a `--path` parameter, followed by a directory, to indicate the project root.

Each of the three folders in a Domino project can contain one or more `.ssp` files. Domino checks packages and games for syntactic correctness even if they are not used in a theorem. This is by design, in order to help packages and games be checked for syntax before setting up a complicated proof.

4.2 Encoding the Free XOR SSP in Domino

Before we describe our encoding of the SSP from the previous chapter in Domino, it is important to note that the pen-and-paper SSP was written in pseudocode, which gives us a lot of freedom on how we represent variables and operations, and allows us to get away with not explicitly defining packages. However, since Domino has its own syntax, and is compiled into SMT-LIB, small syntactical changes in the package code can alter the program's behavior and make it difficult to debug. Therefore, we reason a lot more about packages instead of games (or compositions), and clearly

differentiate between them in Domino code. This has the added benefit of allowing us to reflect on the nuances of the Domino programming language and discuss it in Chapter 5. We first discuss our representation of the original pen-and-paper SSP games in Domino, and then describe the other games and game-hops that were used, along with the reasons they were needed.

4.2.1 The Free XOR proof sketch

The theorem file `freexor.ssp` defines all of the assumptions and game hops used in the Domino encoding of the proof, and links to the invariants and randomness mappings used in the game hops. Initially, we started with four packages - `SimFreeXORReal`, `SimFreeXORIdeal`, `Reduction`, and `LINRKKDM`. These packages map cleanly to the pen-and-paper SSP we describe in Section 3, as described in Table 4.1, and are marked with dotted lines in Figure 4.1.

Table 4.1: Mapping between the games in the pen-and-paper reduction and Domino packages they are composed of.

Pen-and-paper game	Domino representation (.pkg.ssp)	Purpose
$G^0_{\mathsf{Sim\text{-}FreeXOR}}$	`SimFreeXORReal`	The real SimFreeXOR game
$G^1_{\mathsf{Sim\text{-}FreeXOR}}$	`SimFreeXORIdeal`	The ideal SimFreeXOR game
$G^b_{\mathsf{LIN-RK-KDM}}$	`LINRKKDM`	Both LIN-RK-KDM games
$\mathcal{R}$	`Reduction`	The reduction that will be composed with $G^b_{\mathsf{LIN-RK-KDM}}$

However, as we went about defining the relations between the variables in the packages involved in each game hop, it became apparent that we would need to use intermediate packages and games due to the complexity of the game hops, and break it down into multiple game hops with less drastic changes.

Figure 4.1 shows an overview of both the planned proof sketch based on the pen-and-paper SSP (in dotted lines), and the actual proof sketch that was used in the Domino encoding.

The first game hop starts with the real Free XOR game, encoded in game `GSimFreeXORReal`. We show that `GSimFreeXORReal` is equivalent to an intermediate game where part of `GSimFreeXORReal`'s code is outlined into the real version of a hybrid randomness sampling game (`GHybrid`). The underlying package, `Hybrid`, contains an oracle `GETRANDOMS` which represents the statistical argument we made in the pen-and-paper proof using the $\mathsf{GETRANDOMS}(w)$ oracle in G^b_{Hybrid} (Figure 3.7).

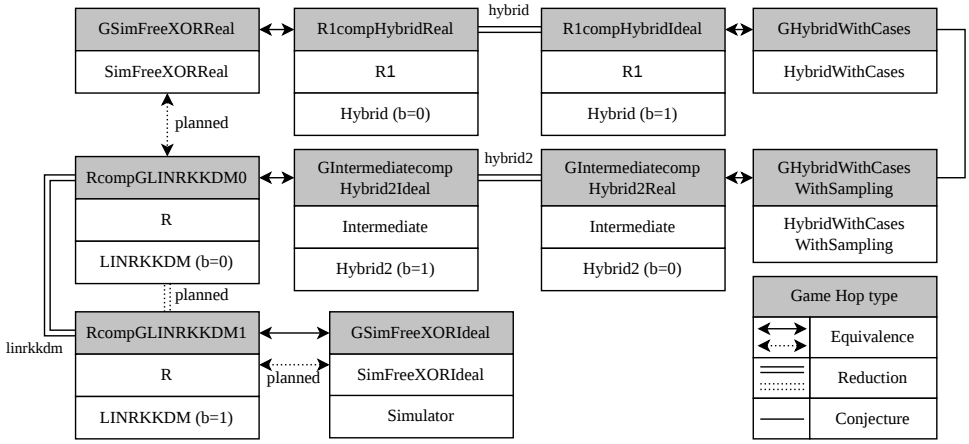


Figure 4.1: Game hops in `freexor.ssp`. The dotted lines represent the planned proof sketch based on the pen-and-paper SSP, whereas the solid lines represent the actual proof sketch. Each game is followed by a list of packages it consists of.

The second game hop is a reduction step from a composition of package `Reduction1` composed with the real `Hybrid` package, to a composition of package `Reduction1` composed with the ideal `Hybrid` package. Both the real and ideal versions of this game are defined in game `R1compHybrid`. The basis for the real and ideal versions of the `GETRANDOMS` oracle of `Hybrid.pkg.ssp` being indistinguishable from each other is the same as the argument made when proving Lemma 3.4, that they follow the same randomness distribution. Since Domino cannot reason about probabilistic arguments at the moment, we declare the real and ideal versions of `GHybrid` as equivalent in our theorem file `freexor.ssp`, as an assumption called `hybrid`.

The third game hop is another equivalence game hop, between the the ideal composition `R1compHybrid` and a new game `GHybridWithCases`. In this game hop, we move a lot of the code from before the conditional branching into the branches. This is a setup for later gamehops that have separate garbled table generation logic for each branch. Domino does not allow $n : 1$ randomness mappings between two games, and this leads to complications, since ideally each of the three random values would be sampled separately in each of the four conditional branches. To work around this, we introduce `tempsample1`, `tempsample2` and `tempsample3`, which are placeholders for the randomness sampling of `A_0`, `B_0` and `C_0`. These still remain outside the conditional branches, however, the assignment of these random values to wire labels, and the assignment of values of the complimentary wire labels are moved inside the branches. In addition, the garbled table generation itself is also moved within the branches.

The fourth game hop is declared as a conjecture between `GHybridWithCases` and `GHybridWithCasesWithSampling`. A conjecture in Domino is a declaration of equivalence that is assumed to be correct by Domino, meaning that it does not perform any checks to verify its correctness. While it is ideal to avoid using them, we use it to move the sampling of the randomness used in wire label generation into the conditional branches. This is a logically sound transformation because only one of the four code branches will run at a time, and therefore, only three of the twelve randomness sampling operations will be executed. The rest of the code remains the exact same as in `GHybridWithCases`.

The fifth game hop is a equivalence game hop between `GHybridWithCases WithSampling` and another compositon (`GIntermediatecompHybrid2`) of games `Intermediate` and (`Hybrid`). This hop outlines the sampling of one or more wire labels that cannot be generated in the adversary-facing packages of later games where the variable `delta` is moved to an inner package.

The sixth game hop is a reduction game hop from a real to ideal sampling of specific wire labels (depending on the input `x`). Similar to the second game hop which was also a reduction, the return values of the real and ideal versions of the oracle `GETRANDOMWIRELABEL` in package `Hybrid2` are indistinguishable because they follow the same distribution. This package corresponds to the pen-and-paper SSP oracle $\mathsf{GETRANDOMS2}(w)$ from G^b_{Hybrid} (Figure 3.7). The assumption `hybrid2` in the theorem file corresponds to this reduction.

The seventh game hop brings us back to the games we dealt with in the pen-and-paper SSP. This game hop is an equivalence between `GIntermediatecompHybrid2Ideal` and the real `RcompGLINRKKDM`. This is the game hop that we were unable to prove fully due to the randomness mapping between the random variables of the games being rather complicated. The mapping becomes complicated for values of `x != (false, false)` since the mapping of wire labels varies for each branch, and it was difficult to determine a suitable mapping for the random values that are used by each encryption operation. However, since the pen-and-paper proof exists, we believe that it should be able to find a mapping that allows us to prove this equivalence.

Finally, the eighth game hop is a equivalence game hop between the ideal `Rcomp GLINRKKDM` and the ideal simulation game `GSimFreeXORIdeal`. The simulation game we use in the pen-and-paper SSP, $G^1_{\mathsf{Sim\text{-}FreeXOR}}$, describes a GARBLE oracle that takes the output of the circuit, and the loss function as inputs. However, these inputs are not usable with our current setup in Domino, since the adversary-exposed oracles of the two games in an equivalence game hop must have the same signature. Therefore, `GSimFreeXORIdeal` consists of a wrapper package `SimFreeXORIdeal` which exposes a `GARBLEAND` oracle with the same signature as the oracle exposed by `Reduction`. It

then computes `f(x)`, and only this value is passed to the actual `Simulator`. This also ensures that the `Simulator` package does not have access to the garbled circuit inputs.

4.2.2 Common parameters

The use of the following parameters remains consistent across many package and game files, and they are also seen in the theorem file, we discuss them before proceeding.

- `n`: Parameter used to represent the security parameter λ.
- `enc`: Parameter used as an abstract function to represent the encryption function Enc. It takes an extra third argument to model a non-deterministic function.
- `m`: Parameter used to represent the output length of the function Enc.
- `b`: Parameter used to switch real and ideal behaviors in a few games.
- `xor_`: Parameter used as an abstract function used to represent the bitwise XOR operation, since Domino does not provide a real representation of the bitwise XOR operation. We discuss this in more detail in Section 5.

4.2.3 Game compositions

In this section, we go over the way game compositons are structured in Domino.

The real SimFreeXOR game is described in the composition `GSimFreeXORReal` (Listing 4). It instantiates the `SimFreeXORReal` package using the common parameters specified in Section 4.2.2 and exposes its `GARBLEAND` oracle to the adversary.

```
composition GSimFreeXORReal {
    const n: Integer;
    const m: Integer;
    const enc: fn Bits(n), Bits(n), Bits(n) -> Bits(m);
    const xor_: fn Bits(n), Bits(n) -> Bits(n);
    instance SimFreeXOR0 = SimFreeXORReal {
        params {
            n: n,
            m:m,
            enc: enc,
            xor_:xor_,
        }
    }
    compose {
        adversary: {
            GARBLEAND: SimFreeXOR0,
        }
    }
```

```
}
```

Listing 4: Composition `GSimFreeXORReal`

Next, the real and ideal versions of the intermediate game `R1compHybrid` are defined as in Listing 5. It instantiates two packages, `Reduction1` and `Hybrid` using the common parameters, and also includes the bit `b` to switch between real and ideal behaviours in package `Hybrid`. It then composes the packages such that the adversary has access to the `GARBLEAND` oracle of the `Reduction1` instance `R1`, and `R1` has access to the `GETRANDOMS` oracle of the `Hybrid` instance.

```
composition R1compHybrid {
    const n: Integer;
    const m: Integer;
    const enc: fn Bits(n), Bits(n), Bits(n) -> Bits(m);
    const b: Bool;
    const xor_: fn Bits(n), Bits(n) -> Bits(n);
    instance GHybrid = Hybrid {
        params {
            n: n,
            b: b,
            xor_:xor_,
        }
    }
    instance R1 = Reduction1 {
        params {
            n: n,
            m: m,
            enc: enc,
            xor_:xor_,
        }
    }
    compose {
        adversary: {
            GARBLEAND: R1,
        },
        R1: {
            GETRANDOMS: GHybrid,
        }
    }
}
```

Listing 5: Composition `R1compHybrid`

Then, the intermediate game `GHybridWithCases` defines a simple composition `GHybridWithCases` as in Listing 6. It consists of just one package, `HybridWithCases`, which exposes a `GARBLEAND` oracle to the adversary.

```
composition GHybridWithCases {
    const n: Integer;
    const m: Integer;
    const enc: fn Bits(n), Bits(n), Bits(n) -> Bits(m);
    const xor_: fn Bits(n), Bits(n) -> Bits(n);
    instance GHybridWithCases = HybridWithCases {
        params {
            n: n,
            m: m,
            enc: enc,
            xor_:xor_,
        }
    }
    compose {
        adversary: {
            GARBLEAND: GHybridWithCases,
        },
    }
}
```

Listing 6: Composition `GHybridWithCases`

Next, the intermediate game `GHybridWithCasesWithSampling` defines a similar composition as the previous one that only differs in the package definition. This game is described in Listing 7.

```
composition GHybridWithCasesWithSampling {
    const n: Integer;
    const m: Integer;
    const enc: fn Bits(n), Bits(n), Bits(n) -> Bits(m);
    const xor_: fn Bits(n), Bits(n) -> Bits(n);
    instance GHybridWithCasesWithSampling = HybridWithCasesWithSampling
    ↪ {
        params {
            n: n,
            m: m,
            enc: enc,
            xor_:xor_,
        }
    }
    compose {
```

```
        adversary: {
            GARBLEAND: GHybridWithCasesWithSampling,
        },
    }
}
```

Listing 7: Composition `GHybridWithCasesWithSampling`

The next composition, `GIntermediateCompHybrid2` composes two packages, `Intermediate`, and `Hybrid2` as in Listing 8. It passes the parameter `b` to `Hybrid2` to switch between real and ideal behaviors. The `GARBLEAND` oracle of the `Intermediate` instance is exposed to the adversary, and the `GETRANDOMWIRELABEL` oracle of the `Hybrid2` instance is exposed to the `Intermediate` instance.

```
composition GIntermediatecompHybrid2 {
    const n: Integer;
    const m: Integer;
    const b: Bool;
    const enc: fn Bits(n), Bits(n), Bits(n) -> Bits(m);
    const xor_: fn Bits(n), Bits(n) -> Bits(n);
    instance GIntermediate = Intermediate {
        params {
            n: n,
            m: m,
            enc: enc,
            xor_:xor_,
        }
    }
    instance GHybrid2 = Hybrid2 {
        params {
            n: n,
            b: b,
            xor_:xor_,
        }
    }
    compose {
        adversary: {
            GARBLEAND: GIntermediate,
        },
        GIntermediate: {
            GETRANDOMWIRELABEL: GHybrid2,
        }
    }
}
```

Listing 8: Composition `GIntermediatecompHybrid2`

Returning to the planned games, `RcompGLINRKKDM` composes two packages, `Reduction` and `LINRKKDM`. It also passes the `b` parameter to `LINRKKDM` to switch between real and ideal behaviours. The `Reduction` instance exposes the `GARBLEAND` oracle to the adversary, and has access to the `ENC` oracle of the `LINRKKDM` instance.

```
composition RcompGLINRKKDM {
    const n: Integer;
    const m: Integer;
    const enc: fn Bits(n), Bits(n), Bits(n) -> Bits(m);
    const b: Bool;
    const xor_: fn Bits(n), Bits(n) -> Bits(n);
    instance GLINRKKDM = LINRKKDM {
        params {
            n: n,
            m:m,
            enc: enc,
            b: b,
            xor_:xor_,
        }
    }
    instance R = Reduction {
        params {
            n: n,
            m: m,
            enc: enc,
            xor_:xor_,
        }
    }
    compose {
        adversary: {
            GARBLEAND: R,
        },
        R: {
            ENC: GLINRKKDM,
        }
    }
}
```

Listing 9: Composition `RcompGLINRKKDM`

Finally, `GSimFreeXORIdeal` is a composition of the simulator interface `SimFreeXOR Ideal` and the simulator itself `Simulator`. The code is described in Listing 10. It

exposes the GARBLEAND oracle from the SimFreeXORIdeal instance, and composes the GARBLE oracle of the Simulator instance with the SimFreeXORIdeal instance.

```
composition GSimFreeXORIdeal {
    const n: Integer;
    const m: Integer;
    const enc: fn Bits(n), Bits(n), Bits(n) -> Bits(m);
    const xor_: fn Bits(n), Bits(n) -> Bits(n);
    instance SimFreeXOR1 = SimFreeXORIdeal {
        params {
            n: n,
            m:m,
        }
    }
    instance Sim = Simulator {
        params {
            n: n,
            m:m,
            enc: enc,
            xor_:xor_,
        }
    }
    compose {
        adversary: {
            GARBLEAND: SimFreeXOR1,
        },
        SimFreeXOR1: {
            GARBLE: Sim,
        }
    }
}
```

Listing 10: Composition GSimFreeXORIdeal

4.2.4 Packages

Since Domino is a strongly typed language, the pseudocode from the pen-and-paper SSP needs to be adapted to be syntactically appropriate. Therefore, we first discuss some common differences in the Domino packages when compared to the pen-and-paper games, and then move on the the specific packages.

Common changes across Domino packages

In all games, delta is declared as a Maybe(Bits(n)) type, which represents an nullable container type (Maybe()), wrapped around a bitstring of length n. This is to

maintain similarity with the type of `delta` used in `linrkkdm.pkg.ssp`, and allows us to avoid typecasting in the SMT-LIB portion of the code, which is preferable. In doing this, we made a tradeoff by having to cast `delta` to its wrapped value whenever we needed to use it in our Domino code.

As briefly mentioned before, `enc` is defined as a three-argument functio, which is different from Enc in the pen-and-paper SSP which only has two arguments. This is because Domino does not have a separate syntax to define a non-deterministic function. The workaround is to have a deterministic function take an argument which is a freshly sampled random value that isn't reused, and this is used as the third extra argument in `enc`.

Since Domino does not allow accessing an array using a boolean variable as the array index in many cases, all of the Domino packages that expose the `GARBLE` oracle use conditional branching on booleans a and b to make varying assignments to the return tuple `(C_,X_,d)`.

Wire labels in all Domino packages, for any wire W, are of the form `W_1`, instead of the W_a notation from the pen-and-paper SSP. The switch from a to `1` is acceptable since the variable names are not reused, and they both refer to a randomly sampled value.

The first item of the return tuple, i.e. F in the pen-and-paper SSP is a table (the garbled table, to be precise). However, in Domino code, the corresponding variable, `C_`, is used as a set. This is by choice. Since Domino does not natively support sets, a workaround is to use the keys of a table as a set-like data structure, since the keys do not repeat themselves. This set-like data structure allows us to remove the relevance of the order or index at which elements exist inside it, and it allows us to compare the entries in the garbled table out-of-order. This is in line with the special correctness property discussed in Section 2.2.3, implying that the Evaluator of the Garbled Circuit can identify the correct entry from this data structure without requiring ordering or indexes.

Other changes

We now discuss other notable differences in the Domino packages compared to their pen-and-paper counterparts. For a full overview of the code, please refer to the code repository [Sha26].

In addition to the package parameters, package `LINRKKDM` also contains a state variable (a bitstring of length `n`) called `delta` that corresponds to Δ in the SSP representation. However, in the Domino package, `delta` is wrapped in `Maybe(...)`. This is because the `Bits(n)` type is not nullable, and therefore cannot be checked

for an 'unassigned' state, as we do in $G^b_{\mathsf{LIN-RK-KDM}}$. Domino provides a nullable type `Maybe(...)` that allows us to set `delta` to a `None` value which we use as a representation for an 'unassigned' state.

We continue to use the `Maybe(Bits(n))` type instead of raw `Bits(n)` in every package that uses `delta`. This choice was made to reduce the number of type casts when comparing and relating game states in SMT-LIB (we discuss this in more detail when discussing invariants and randomness mappings). However, the tradeoff is that we need to use an `Unwrap` operator when trying to access the value of `delta` in the Domino code.

```
oracle ENC(key: Bits(n), msg: Bits(n), z: Bool) -> Bits(m) {
    if (delta == None) {
        tmp <-$ Bits(n) sample-name delta;
        delta <- Some(tmp);
    }
    r <-$ Bits(n) sample-name r;
    if b {return enc(xor_(key, Unwrap(delta)), 0_n, r); }
    else {
        if z {
            return enc(xor_(key, Unwrap(delta)),
            ↪  xor_(msg,Unwrap(delta)), r);
        } else {
            return enc(xor_(key, Unwrap(delta)), msg, r);
}}}
```

Listing 11: Oracle ENC in LINRKKDM.pkg.ssp

The code for the `ENC` oracle in the `LINRKKDM` package also differs from the pen-and-paper SSP as it contains an extra conditional branch using `if-else` on the boolean variable `z` (Listing 11). This is because Domino does not allow us to multiply a boolean value z with a bitstring Δ, and we cannot represent $msg \oplus z \cdot \Delta$ from $G^b_{\mathsf{LIN-RK-KDM}}$ directly.

The `Reduction` package begins with the package parameters and does not use a state variable, which is consistent with its pen-and-paper counterpart. Since $\mathcal{R}$ is composed with $G^b_{\mathsf{LIN-RK-KDM}}$, we must use the `import oracle` syntax in `Reduction` to declare that this package is intended to be composed with another package that exposes a `ENC` oracle with a specific signature (Listing 13). This also allows us to invoke the `ENC` oracle in the code of `Reduction`. The concrete implementation of `ENC` that is invoked from the oracles of `Reduction`, however, is defined in the composition rather than in the `Reduction` package itself (as we see in Listing 9).

```
oracle GARBLEAND(x: (Bool, Bool)) ->
```

```
(Table((Bits(m),Bits(m)), ()), (Bits(n), Bits(n)), Table(Bits(n),
↪ Bool)) {
    /*...*/
    if ((a==false) and (b==false)) {
        c <- false; /* AND gate */
        A_0 <-tempsample1;
        B_0 <-tempsample2;
        C_0 <-tempsample3;
        GarbledRow0 <- (enc(A_0, R_00, r01), enc(B_0, xor_(R_00, C_0),
        ↪ r02));
        temp <- invoke ENC(B_0, xor_(R_01, C_0), false);
        GarbledRow1 <- (enc(A_0, R_01, r11), temp);
        temp <- invoke ENC(A_0, xor_(R_10, C_0), false);
        GarbledRow2 <- (temp, enc(B_0, R_10, r22));
        temp <- invoke ENC(A_0, R_11, false);
        temp2 <- invoke ENC(B_0, xor_(R_11, C_0), true);
        GarbledRow3 <- (temp, temp2);
        C_[GarbledRow0] <- Some(());
        C_[GarbledRow1] <- Some(());
        C_[GarbledRow2] <- Some(());
        C_[GarbledRow3] <- Some(());
        X_ <- (A_0, B_0);
        d[C_0] <- Some(c);
    }
    /*...*/
    return (C_,X_,d);
}
```

Listing 12: Part of oracle GARBLEAND in Reduction.pkg.ssp

`Reduction` exposes one oracle, `GARBLEAND`, which maps to the GARBLE oracle of $\mathcal{R}$, but does not take f as a paramter. This is because Domino does not allow equivalence relations between games that expose oracles with different oracle signatures, and one of the latter games in the equivalence relations we try to establish (specifically $G^1_{\mathsf{Sim\text{-}FreeXOR}}$ and its corresponding package `SimFreeXORIdeal`) require a modified oracle signature. We are able to get around not having f as an input parameter since we already know that we are dealing with the garbling of an AND gate, and can compute $f(x)$ in each conditional branch. Since Domino supports neither assignment operations to arbitrary boolean indexes (such as $A_a \leftarrow\!\$\ \{0,1\}^\lambda$), nor operations between boolean and bitstrings, we got around it using conditional branching on the boolean variables `a` and `b`. This enabled us to compute the value of `c` or $f(x)$ in each branch separately by setting the value of `c` to a precomputed $a \wedge b$ (Listing 12).

```
package Reduction {
```

```
    /*...*/
    import oracles {
        ENC(key: Bits(n), msg: Bits(n), z: Bool) -> Bits(m),}
    /*...*/
    }
```

Listing 13: Importing Oracle `ENC` in `Reduction`

We also define the `GARBLE` oracle which encodes the code from GARBLE in the SSP. While randomly sampling variables, we can assign them a sample name, which is useful when we want to relate these randomnesses to the randomnesses generated in other packages and games.

Aside from the differences listed in Section 4.2.4, we were able to encode `SimFree XORReal` without any significant logical changes.

However, the ideal Free XOR game had to be split into `SimFreeXORIdeal` and `Simulator`, where `Simulator` contains the actual simulation, and the outer wrapper ensures that the simulator does not receive the actual input values. It also allow us to have the required oracle signature for code equivalence, since the `Simulator` package does not expose a `GARBLEAND` oracle 14.

```
package Simulator {
    params {
        n: Integer,
        m: Integer,
        enc: fn Bits(n), Bits(n), Bits(n) -> Bits(m),
        xor_: fn Bits(n), Bits(n) -> Bits(n),
    }
    state {
        delta: Maybe(Bits(n)),
    }
    oracle GARBLE(y: Bool) ->
    (Table((Bits(m),Bits(m)), ()), (Bits(n),Bits(n)),
        Table(Bits(n),Bool))
    {
        if (delta == None) {
            tmp <-$ Bits(n) sample-name delta;
            delta <- Some(tmp);
        }
        A_0 <-$ Bits(n) sample-name A_0;
        B_0 <-$ Bits(n) sample-name B_0;
        C_0 <-$ Bits(n) sample-name C_0;
        A_1 <- xor_(A_0, Unwrap(delta));
        B_1 <- xor_(B_0, Unwrap(delta));
```

```
24          C_1 <- xor_(C_0, Unwrap(delta));
25
26          R_00 <-$ Bits(n) sample-name R_00; /* row 0 randomness */
27          R_01 <-$ Bits(n) sample-name R_01; /* row 1 randomness */
28          R_10 <-$ Bits(n) sample-name R_10; /* row 2 randomness */
29          R_11 <-$ Bits(n) sample-name R_11; /* row 3 randomness */
30          C_ <- new Table((Bits(m),Bits(m)), ());
31          d <- new Table(Bits(n),Bool);
32          c <- y;
33
34          r01 <-$ Bits(n) sample-name randrow0enc1;
35          r02 <-$ Bits(n) sample-name randrow0enc2;
36          GarbledRow0 <- (enc(A_0, R_00, r01), enc(B_0, xor_(R_00, C_0),
            ↪  r02));
37
38          r11 <-$ Bits(n) sample-name randrow1enc1;
39          r12 <-$ Bits(n) sample-name randrow1enc2;
40          GarbledRow1 <- (enc(A_0, R_01, r11), enc(B_1, 0_n, r12));
41
42          r21 <-$ Bits(n) sample-name randrow2enc1;
43          r22 <-$ Bits(n) sample-name randrow2enc2;
44          GarbledRow2 <- (enc(A_1, 0_n, r21), enc(B_0, R_10, r22));
45
46          r31 <-$ Bits(n) sample-name randrow3enc1;
47          r32 <-$ Bits(n) sample-name randrow3enc2;
48          GarbledRow3 <- (enc(A_1, 0_n, r31), enc(B_1, 0_n, r32));
49          C_[GarbledRow0] <- Some(());
50          C_[GarbledRow1] <- Some(());
51          C_[GarbledRow2] <- Some(());
52          C_[GarbledRow3] <- Some(());
53          X_ <- (A_0, B_0);
54          d[C_0] <- Some(c);
55          return (C_,X_,d);
56      }
57  }
```

Listing 14: Package `Simulator`

4.2.5 Theorem

We now go over the `freexor` theorem file. It starts by defining the constant variables that are used to instantiate the games (Figure 15). These parameters are then injected into the games (and then their packages), making their way to the underlying oracles. This is what allow us to ensure that symbolic parameters behave the same way across packages. For example, this is what allows `enc` and `xor` to have exactly the same

behavior in every package, and for `m` and `n` to remain consistent, even though they are never explicitly defined.

```
theorem freexor {
    const n: Integer; /* security parameter */
    const m: Integer; /* message length */
    const enc: fn Bits(n), Bits(n), Bits(n) -> Bits(m);
    const xor_: fn Bits(n), Bits(n) -> Bits(n);
    /* ... */
}
```

Listing 15: Parameters in theorem `freexor`

Then, we instantiate all the games associated with the `freexor` theorem, and intantiate the real and ideal games separately by passing the relevant `b` parameter.

```
theorem freexor {
    /* ... */
    instance GHybridReal = GHybrid {
        params {
            n: n,
            b: false,
            xor_: xor_,
        }}
    instance GHybridIdeal = GHybrid {
        params {
            n: n,
            b: true,
            xor_: xor_,
        }}
    instance GSimFreeXORReal = GSimFreeXORReal {
        params {
            n: n,
            m: m,
            enc: enc,
            xor_: xor_
        }}
    instance R1compHybridReal = R1compHybrid {
        params {
            n: n,
            m: m,
            b: false,
            enc: enc,
            xor_: xor_
        }}
```

```
    instance R1compHybridIdeal = R1compHybrid {
        params {
            n: n,
            m: m,
            b: true,
            enc: enc,
            xor_: xor_
        }}
    instance RcompGLINRKKDM0 = RcompGLINRKKDM {
        params {
            n: n,
            m: m,
            enc: enc,
            b: false,
            xor_: xor_
        }}
    instance GHybridWithCases = GHybridWithCases {
        params {
            n: n,
            m: m,
            enc: enc,
            xor_: xor_
        }}
    instance GHybridWithCasesWithSampling =
    ↪  GHybridWithCasesWithSampling {
        params {
            n: n,
            m: m,
            enc: enc,
            xor_: xor_
        }}
    instance GHybrid2Ideal = GHybrid2 {
        params {
            n: n,
            b: true,
            xor_: xor_
        }}
    instance GHybrid2Real = GHybrid2 {
        params {
            n: n,
            b: false,
            xor_: xor_
        }}
    instance GIntermediatecompHybrid2Real = GIntermediatecompHybrid2 {
        params {
```

```
            n: n,
            m: m,
            b: false,
            enc: enc,
            xor_: xor_
        }}
    instance GIntermediatecompHybrid2Ideal = GIntermediatecompHybrid2 {
        params {
            n: n,
            m: m,
            b: true,
            enc: enc,
            xor_: xor_
        }}
    instance GLINRKKDM0 = GLINRKKDM {
        params {
            n: n,
            m: m,
            enc: enc,
            b: false,
            xor_: xor_,
        }}
    instance GLINRKKDM1 = GLINRKKDM {
        params {
            n: n,
            m: m,
            enc: enc,
            b: true,
            xor_: xor_,
        }}
    instance RcompGLINRKKDM1 = RcompGLINRKKDM {
        params {
            n: n,
            m: m,
            enc: enc,
            b: true,
            xor_: xor_
        }}
    instance GSimFreeXORIdeal = GSimFreeXORIdeal {
        params {
            n: n,
            m: m,
            enc: enc,
            xor_: xor_
        }}
```

```
    /* ... */
}
```

Listing 16: Game instantiations in theorem `freexor`

After creating instances of all the games, we define the assumptions that the subsequent reduction game hops use. In our case, these are `hybrid`, `hybrid2` and `lin_rk_kdm`, which correspond to Definition 3.1 (LIN-RK-KDM security), Lemma 3.4, and Lemma 3.6, respectively.

```
theorem freexor {
    /* ... */
    assumptions {
        hybrid: GHybridReal ~ GHybridIdeal
        hybrid2: GHybrid2Real ~ GHybrid2Ideal
        lin_rk_kdm: GLINRKKDM0 ~ GLINRKKDM1
    }
    /* ... */
}
```

Listing 17: Assumptions used in theorem `freexor`

Finally, the theorem file contains the game hops used in the theorem, along with their randomness mappings and invariants. We go over each game hop individually. It is important to note that game hops in Domino are indexed from 0, and these are mapped accurately to Figure 4.1.

```
theorem freexor {
    /*...*/
    gamehops{
        equivalence GSimFreeXORReal R1compHybridReal {
                invariant: [
                    ./theorem/invariants/hybridreal.smt2
                ]
                GARBLEAND: {
                    lemmas {
                        invariant: [no-abort]
                        equal-aborts: []
                        same-output: [no-abort]

                    }
                }
            }
        /*...*/
```

```
    }
}
```

Listing 18: Game hop 1 in the `freexor` theorem

The first game hop (Listing 18) is a simple equivalence relation between `GSimFree XORReal` and `R1compHybridReal`. The randomness mapping and invariant are found in the `hybridreal2.smt2` file. The randomness mapping is quite simple, it maps the wire label samplings, `delta`, row randomness samplings (`R_00`, `R_01`, `R_10`, `R_11`), and the random values used for `enc` function calls from `GSimFreeXORReal` to `R1compHybridReal`. It is worth noting that Domino allows us to map variables from `SimFreeXORReal` to both the packages that compose `R1compHybridReal`. The `invariant` for this equivalence simply states that the values of `delta` are equivalent in the states of both games. The two other lemmata, `same-output` and `equal-aborts` verify the input-output equivalence property.

```
theorem freexor {
    /*...*/
    gamehops {
    /*...*/
    reduction R1compHybridReal R1compHybridIdeal {
            assumption hybrid
            map GHybridReal R1compHybridReal {
                GHybrid: GHybrid
            }
            map GHybridIdeal R1compHybridIdeal {
                GHybrid: GHybrid
            }
        }
    /*...*/
    }
}
```

Listing 19: Game hop 2 in the `freexor` theorem

The second game hop (Listing 19) is a reduction step, that holds because of the `hybrid` assumption. Domino checks whether the composition graphs for the games we are stating are equivalence actually reduce to the stated assumption or not. However, the assumption itself cannot be reasoned about in Domino.

```
theorem freexor {
    /*...*/
    gamehops{
    /*...*/
```

```
    equivalence R1compHybridIdeal GHybridWithCases {
            invariant: [
                    ./theorem/invariants/hybridideal part1.smt2
                ]
            GARBLEAND: {
                lemmas {
                    invariant: [no-abort]
                    same-output: [no-abort]
                    equal-aborts: []
                }
            }
        }
    /*...*/
    }
}
```

Listing 20: Game hop 3 in the `freexor` theorem

The third game hop (Listing 20) is an equivalence game hop between `R1comp HybridIdeal` and `GHybridWithCases`, and this is quite similar to the first game hop. The randomness mapping is quite straightforward like the first game hop, and the invariant is the same as well.

```
theorem freexor {
    /*...*/
    gamehops {
    /*...*/
    conjecture GHybridWithCases GHybridWithCasesWithSampling
    /*...*/
    }
}
```

Listing 21: Game hop 4 in the `freexor` theorem

The fourth game hop (Listing 21) is a conjecture that `GHybridWithCases` is equivalent to `GHybridWithCasesWithSampling`. As stated before, conjectures are not checked by Domino and the reason we use this is because Domino does not support $n : 1$ randomness mappings even if the first game has conditional branches that ensure that the runtime mapping is $1 : 1$.

```
theorem freexor {
    /*...*/
    gamehops{
    /*...*/
```

```
    equivalence GHybridWithCasesWithSampling
    ↪  GIntermediatecompHybrid2Real {
        invariant: [
            ./theorem/invariants/hybridintermediate.smt2
            ./theorem/invariants/lemma-hybridintermediate.smt2
            ]
        GARBLEAND: {
            lemmas {
                case-x-is-false-false: [no-abort]
                case-x-is-false-true: [no-abort]
                case-x-is-true-false: [no-abort]
                case-x-is-true-true: [no-abort]
                same-output: [no-abort, case-x-is-false-false,
                              case-x-is-false-true,
                              ↪  case-x-is-true-false,
                              case-x-is-true-true]
                invariant: [no-abort]
                abort-case-x-is-false-false: []
                abort-case-x-is-false-true: []
                abort-case-x-is-true-false: []
                abort-case-x-is-true-true: []
                equal-aborts: [abort-case-x-is-false-false,
                           abort-case-x-is-false-true,
                           abort-case-x-is-true-false,
                           abort-case-x-is-true-true]
            }
        }
    }
    /*...*/
    }
}
```

Listing 22: Game hop 5 in the `freexor` theorem

The fifth game hop (Listing 22) is an equivalence game hop where things get a bit more interesting. In this equivalence relation, we needed to define conditional randomness mappings between `GHybridWithCasesWithSampling` and `GIntermediatecompHybrid2Real`, where one or more arbitrary wire labels in the left game would correspond to a sampling in the inner right game `GHybrid2`. Listing 23 shows a sample of how we implemented this. However, conditional branching in SMT-LIB increases the verification time of the program quite significantly.

```
define-fun randomness-mapping-GARBLEAND
((sample-id-left SampleId) (sample-id-right SampleId)
    (sample-offset-left Int) (sample-offset-right Int))
```

```
Bool
(or
    (and
        (= sample-id-left (sample-id "GHybridWithCasesWithSampling"
        ↪  "GARBLEAND" "delta"))
        (= sample-id-right (sample-id "GIntermediate" "GARBLEAND"
        ↪  "delta"))
        (= sample-offset-left 0)
        (= sample-offset-right 0)
    )
    ; ...
    (ite
        (= <arg-GHybridWithCasesWithSampling-GARBLEAND-x>
            ((as mk-tuple2 (Tuple2 Bool Bool)) false false))
        (or
            (and
                (= sample-id-left (sample-id
                ↪  "GHybridWithCasesWithSampling" "GARBLEAND" "A_00"))
                (= sample-id-right (sample-id "GIntermediate"
                ↪  "GARBLEAND" "A_00"))
                (= sample-offset-left 0)
                (= sample-offset-right 0)
            )
            (and
                (= sample-id-left (sample-id
                ↪  "GHybridWithCasesWithSampling" "GARBLEAND" "B_00"))
                (= sample-id-right (sample-id "GIntermediate"
                ↪  "GARBLEAND" "B_00"))
                (= sample-offset-left 0)
                (= sample-offset-right 0)
            )
            (and
                (= sample-id-left (sample-id
                ↪  "GHybridWithCasesWithSampling" "GARBLEAND" "C_00"))
                (= sample-id-right (sample-id "GIntermediate"
                ↪  "GARBLEAND" "C_00"))
                (= sample-offset-left 0)
                (= sample-offset-right 0)
            )
        )'
        ;...
    )
)
```

Listing 23: Conditional randomness mapping in game hop 5 of the `freexor` theorem

In order to offset this performance loss, we declared pairs of lemmata like `case-x-is-false-false` and `abort-case-x-is-false-false`, for each input combination to the `GARBLEAND` oracles. This allowed us to split up the computation for each input, and sped up the verification quite significantly. One sample pair of such lemmata can be seen in Listing 24.

```
(define-lemma case-x-is-false-false GARBLEAND
    (old-state-left old-state-right return-left return-right
        (x (Tuple2 Bool Bool))
    )
    (=>
        (= x ((as mk-tuple2 (Tuple2 Bool Bool)) false false))
        (= return-left.value return-right.value)
    )
)
(define-lemma abort-case-x-is-false-false GARBLEAND
    (
        old-state-left old-state-right return-left return-right
        (x (Tuple2 Bool Bool))
    )
    (=>
        (= x ((as mk-tuple2 (Tuple2 Bool Bool)) false false))
        (= ((_ is mk-abort) return-left.value)
           ((_ is mk-abort) return-right.value))
    )
)
```

Listing 24: One of four lemma pairs in game hop 5 of the `freexor` theorem

We encoded four such pairs, one for each input possible input value to `GARBLEAND`.

```
theorem freexor {
    /*...*/
    gamehops {
    /*...*/
    reduction GIntermediatecompHybrid2Real
    ↪  GIntermediatecompHybrid2Ideal {
        assumption hybrid2
        map GHybrid2Real GIntermediatecompHybrid2Real {
            GHybrid2: GHybrid2
        }
        map GHybrid2Ideal GIntermediatecompHybrid2Ideal {
            GHybrid2: GHybrid2
        }
    }
```

```
    /*...*/
    }
}
```

Listing 25: Game hop 6 in the `freexor` theorem

The sixth game hop is a reduction game hop that holds because of the `hybrid2` assumption.

```
theorem freexor {
    /*...*/
    gamehops {
    /*...*/
    equivalence GIntermediatecompHybrid2Ideal RcompGLINRKKDM0 {
        invariant: [
                ./theorem/invariants/hybridideal part2.smt2
                ./theorem/invariants/lemma-hybridintermediate.smt2
            ]
        GARBLEAND: {
            lemmas {
                case-x-is-false-false: [no-abort]
                case-x-is-false-true: [no-abort]
                case-x-is-true-false: [no-abort]
                case-x-is-true-true: [no-abort]
                same-output: [no-abort,
                            case-x-is-false-false, case-x-is-false-true,
                            case-x-is-true-false, case-x-is-true-true]
                invariant: [no-abort]
                abort-case-x-is-false-false: []
                abort-case-x-is-false-true: []
                abort-case-x-is-true-false: []
                abort-case-x-is-true-true: []
                equal-aborts: [abort-case-x-is-false-false,
                        abort-case-x-is-false-true,
                        abort-case-x-is-true-false,
                        abort-case-x-is-true-true]
            }
        }
    }
    /*...*/
    }
}
```

Listing 26: Game hop 7 in the `freexor` theorem

The seventh game hop is a equivalence step between `GIntermediatecompHybrid2 Ideal` and `RcompGLINRKKDM0`. This game hop uses even larger conditional randomness mappings like in game hop 5, and therefore, we also use the performance optimization used there. However, it also requires the associativity, commutativity, identity and inverse properties of the XOR operation. Since we modelled XOR as an abstract function, with no intrinsic properties of its own, and since Domino does not provide an XOR operation with these properties, there are two ways to work around this.

The first approach is to create a reduction game hop for each property, and then create intermediate games for ever XOR operation that needs any of these properties. However, this would explode in complexity very quickly.

The second approach is to create an assertion that states that two statements are equal (so in this case, we would assert something like $a \oplus b = b \oplus a$), but each such assertion also increases the verification time quite significantly, especially if we make the assertion over the whole domain of the variables. We discuss these approaches in more detail in Chapter 5. Moreover, Domino shows a warning on execution if we include assertions, to prevent the programmer from 'cheating' silently. Since we were unable to get the program to terminate for this game hop (due to time taken to solve), we decided to let it fail and to discuss why this occurs.

```
1  theorem freexor {
2      /*...*/
3      gamehops {
4      /*...*/
5      reduction RcompGLINRKKDM0 RcompGLINRKKDM1 {
6          assumption lin_rk_kdm
7          map GLINRKKDM0 RcompGLINRKKDM0 {
8              GLINRKKDM: GLINRKKDM
9          }
10         map GLINRKKDM1 RcompGLINRKKDM1 {
11             GLINRKKDM: GLINRKKDM
12         }
13     }
14     /*...*/
15     }
16 }
```

Listing 27: Game hop 8 in the `freexor` theorem

The eighth game hop is a reduction between `RcompGLINRKKDM0` and `RcompGLINRKKDM1`, which holds because of the `lin_rk_kdm` assumption.

```
1  theorem freexor {
```

```
    /*...*/
    gamehops {
    /*...*/
    equivalence RcompGLINRKKDM1 GSimFreeXORIdeal {
        invariant: [
            ./theorem/invariants/freexorideal.smt2
            ./theorem/invariants/lemma-hybridintermediate.smt2
        ]
        GARBLEAND: {
            lemmas {
                case-x-is-false-false: [no-abort]
                case-x-is-false-true: [no-abort]
                case-x-is-true-false: [no-abort]
                case-x-is-true-true: [no-abort]
                same-output: [no-abort, case-x-is-false-false,
                         case-x-is-false-true, case-x-is-true-false,
                         case-x-is-true-true]
                invariant: [no-abort]
                abort-case-x-is-false-false: []
                abort-case-x-is-false-true: []
                abort-case-x-is-true-false: []
                abort-case-x-is-true-true: []
                equal-aborts: [abort-case-x-is-false-false,
                 abort-case-x-is-false-true, abort-case-x-is-true-false,
                 abort-case-x-is-true-true]
            }
        }
        }
    /*...*/
    }
}
```

Listing 28: Game hop 9 in the `freexor` theorem

The final game hop is an equivalence relation between `RcompGLINRKKDM1` and `GSimFreeXORIdeal`. This game hop also uses conditional randomness mappings for the row randomness samplings (`R_00`, `R_01`, `R_10`, `R_11`), and the splitting-optimization used in the fifth game hop.

4.2.6 Verification Results

The above discussed formalization was verified using (`domino prove`). We were able to prove all the game hops until the equivalence between `GIntermediatecompHybrid2Ideal` and `RcompGLINRKKDM0`. Figure 4.2 shows the output of `domino prove` until

it terminates at the seventh game hop. Since game hops are indexed from zero by Domino, this corresponds to the `--proofstep 6` syntax used in the following figures.

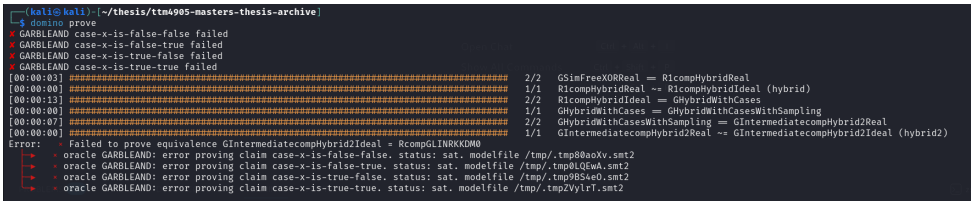


Figure 4.2: Output of `domino prove` that shows proven steps until the seventh game hop

Figure 4.3 shows that it was indeed the seventh game hop, or `proofstep 6` that failed.

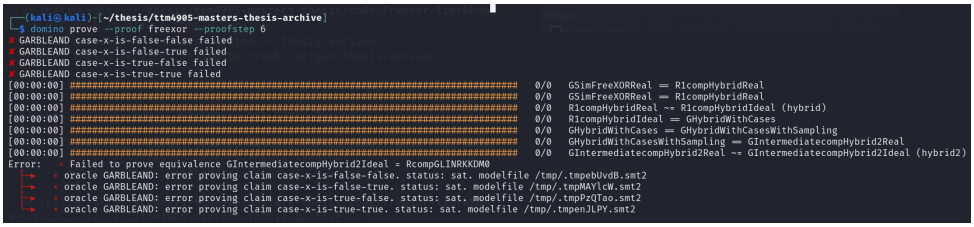


Figure 4.3: Output of `domino prove --proofstep 6` that verifies that the seventh game hop fails independently too

Figure 4.4 and Figure 4.5 show the remaining two game hops being successfully verified.

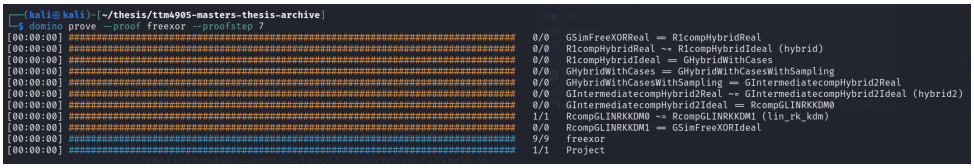


Figure 4.4: Output of `domino prove --proofstep 7` that shows the successful verification of the eighth game hop



Figure 4.5: Output of `domino prove --proofstep 8` that shows the successful verification of the ninth game hop

The full code for this formalization can be found on GitHub [Sha26].

Chapter 5
Discussion

In this chapter, we first discuss our findings from Chapters 3 and 4 very briefly. Then, as the main content of this chapter, we go over some challenges we encountered in using Domino in verifying a pen-and-paper SSP, and the workaround that were available for some of them.

5.1 Summary of Results

In Section 3.1, we first introduced SSPs, and the syntax and constructs they use, as well as how we diverge from the syntax used in existing works. Then, in Section 3.2 we discuss the garbling of various logic gates and establish that the AND gate is of the most importance in this work, based on Kolesnikov and Schneider's work. We represent the privacy of the garbling of AND gates as a cryptographic game, and try to reduce it to a SSP representation of Applebaum's proof of security of the Free XOR garbling scheme. We provide a proof for this reduction, without any unfulfilled assumptions.

Then, in Section 4.1, we introduce the proof verifier tool we use, Domino, and briefly discuss how it works, how a new Domino environment can be set up, and how the files in a Domino project map to a pen-and-paper SSP. Then, in Section 4.2, we provide a high-level overview of our proof sketch, and how it differs from the planned proof that was based on the pen-and-paper SSP. We then go over each individual composition used in the project in hopes to convey the structure of the project and how it maps to our high-level overview. Then, we go into some Domino packages in order to explain how we mapped some pseudocode structures from the pen-and-paper SSP to Domino, and highlight the main differences. Finally, we go over the main proof file, and discuss it step-by-step, concluding with a discussion of the game hops in detail and the associated SMT-LIB code. However, we only briefly discuss features that Domino lacks that increased the complexity of encoding the SSP in Domino, and will continue this discussion in Section 5.2. Finally, we showcased the successes

and shortcomings of this encoding of the SSP in Domino.

5.2 Challenges in encoding the SSP in Domino

Domino aims to be a highly automated tool for verification of state-separating proofs, while also providing quality-of-life features such as syntax that is close to SSPs, and a feature to generate LaTeXand HTML representations of the proof. It also tries to provide verbose error logs to make debugging of code easier.

At the same time, while working on this project, we found it lacking in certain features, that if added, would make the process of encoding and verifying a SSP much simpler. We discuss some of these features below.

The biggest shortcoming we observed during the project was the lack of a in-build XOR operator. While it may be complex to implement this operator since it would have to compile to SMT-LIB, it is a basic mathematical operator available in other languages. SMT-LIB itself provides an XOR operator for certain theories, and this may be a good starting point for partial implementation of the operator for some data types in Domino. While encoding the Free XOR proof in Domino, we ran into a few cases where we could work around the absence of the Free XOR operator using `assert` statements in SMT-LIB. Listing 29 defines an assertion that uses universal quantifiers to define the associativity property for the abstract function `xor_`. However, using universal quantifiers increased the time taken to discharge the proof to over two full days, making it an unrealisitc choice.

```
(assert
    (forall
        ((x Bits_n) (y Bits_n) (z Bits_n))
        (=
            (<<func-xor_>> x (<<func-xor_>> y z))
            (<<func-xor_>> (<<func-xor_>> x y) z)
        )
    )
)
```

Listing 29: An assertion for associativity of the `xor_` function

However, even if we made assertions for fixed relations between variables, like in Listing 30, it is very difficult to determine which variables these relations should exist for in the first place. Often, one has to work the logic out on pen and paper, and then code it in Domino. Upon trying approaches such as this, we had limited success for specific conditional branches, but completely failed on the others - simply because it was difficult to understand the program execution flow on paper. Moreover, these

assertions still slow the proof verification down, though not to an extent as using universal quantifiers.

```
(assert
    (
        let (
            (delta
                (<pkg-state-Intermediate-<$<!m!><!n!>$>-delta>
                (<game-GIntermediatecompHybrid2-<$<!n!><!m!>$>-pkgstate
                ↪ -GIntermediate>
                (<oracle-return-GIntermediatecompHybrid2-<$<!n!><!m!>$>
                ↪ -Intermediate-<$<!m!><!n!>$>-GARBLEAND-game-state>
                <return-GIntermediatecompHybrid2Ideal-GARBLEAND>
                ))))
            (r_11 (__sample-rand-GIntermediatecompHybrid2Ideal-Bits_n
            ↪ (sample-id "GIntermediatecompHybrid2" "GARBLEAND"
            ↪ "R_11") 0 ))
            (c_0 (__sample-rand-GIntermediatecompHybrid2Ideal-Bits_n
            ↪ (sample-id "GIntermediatecompHybrid2" "GARBLEAND"
            ↪ "C_00") 0 ))
            )
        (=
            (<<func-xor_>> r_11 (<<func-xor_>> c_0 (maybe-get delta)))
            (<<func-xor_>> (<<func-xor_>> r_11 c_0) (maybe-get delta))
        )
    )
)
```

Listing 30: An assertion for associativity of the `xor_` function without using universal quantifiers

If the XOR operation, even with limited properties is added to Domino, it would allow proofs that deal with this operation to be signinificantly easier.

Another difficulty we encountered was in creating randomness mappings between two packages, where the first package had a certain number of sampled random values, say n, and the second had $m * n$ sampled random values, but with m branches, where each branch had exactly n values sampledm, and only one of m branches would run per program flow. An example can be seen in the fourth game hop in Listing 21, where we used a conjecture to get around this obstacle.

While this was a minor inconvenience, stronger support for the Boolean data type would be very helpful. Examples include being able to assign values to tables while accessing them based on Boolean indexes. This was the primary reason for the conditional branching we used in the packages that exposed a `GARBLEAND` oracle.

While the conditional branching was technically correct, it led to the code being significantly larger in size, and also more difficult to reason about and compare with the pen-and-paper SSP. Another example is that Domino does not allow multiplication between Booleans and bitstrings, causing the `LINRKKDM` package to have an additional conditional branch.

The flexibility that Domino brings by allowing the user to code their requirements as an invariant also contributes to an unavoidable increase in complexity. The syntax for the SMT-LIB invariant files is not plain SMT-LIB but has additions that are provided for use by the Domino compiler. However, a the time of writing the thesis, it was quite difficult to familiarize oneself with these quickly since the documentation on these is sparse, and the intermediate SMT-LIB files generated by the Domino compiler are verbose and difficult to interpret. Some documentation covering the generated SMT files, the type system in Domino, and the workarounds that we ended up using would be a great addition while features are being developed.

However, not all is terrible in the world of Domino. The project seems to be moving to more opinionated default options recently, like in the case of randomness mappings. Domino now provides automatic randomness mappings for simple games, avoiding the verbosity of writing elaborate mappings for games that have simple mappings like in the case of the first game hop in our Domino encoding. The project continues to receive updates frequently, and new features are added often based on user feedback. It is a project we look forward to seeing develop in the future.

Chapter 6
Conclusion

6.1 Conclusion

This thesis investigated the use of Applebaum's Free XOR proof to construct a pen-and-paper SSP to establish the privacy of an AND gate. Then, it attempted to encode the SSP into the proof verifier, Domino, and formally validate the correctness of the encoding of the SSP.

We were able to successfully express the privacy of an AND gate as an SSP, and show that the privacy held without any specific assumptions. These games and their corresponding packages were successfully encoded in Domino, and the proof was sketched out in the form of equivalences and reductions, which were all validated by Domino, except a singular equivalence game hop.

The work also assessed Domino as a tool for proving SSPs in the form of code. To this end, since most of the proof was successfully verified, we found the tool to be very useful. In addition, we suggested some improvements based on our usage of Domino, and hope that the proof encoding in this thesis can be useful to users of the tool.

Finally, this work demonstrates the feasibility of constructing a state-separating proof from a different proof framework. It also demonstrates that Domino is a suitable tool for the verification of state-separating proofs.

6.2 Future Work

The most immediate direction for future work is to complete the verification of the equivalence relation that remain unresolved in the current formalization. This may be aided by future improvements to the Domino language.

On the subject of the state-separating proof itself, in this work, we chose not to use

the Point and Permute optimization for Garbled Circuits, and assumed the special correctness property. Future work may try to model the privacy of garbling an AND gate with Point and Permute, or without the special correctness property. While it may be ambitious, follow up works to these may continue on to proving the privacy of a whole circuit as a state-separating proof.

References

[App16] B. Applebaum, "Garbling XOR gates "for free" in the standard model", *Journal of Cryptology*, vol. 29, no. 3, pp. 552–576, Jul. 2016.

[BCDS22] D. Basin, C. Cremers, et al., "Tamarin: Verification of Large-Scale, Real-World, Cryptographic Protocols", *IEEE Security & Privacy*, vol. 20, no. 03, pp. 24–32, May 2022. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MSEC.2022.3154689

[BDF+18] C. Brzuska, A. Delignat-Lavaud, et al., "State separation for code-based game-playing proofs", in *Advances in Cryptology – ASIACRYPT 2018, Part III*, (Dec. 2–6, 2018), T. Peyrin and S. Galbraith, Eds., ser. Lecture Notes in Computer Science, vol. 11274, Brisbane, Queensland, Australia: Springer, Cham, Switzerland, Dec. 2018, pp. 222–249.

[BDG+14] G. Barthe, F. Dupressoir, et al., "EasyCrypt: A tutorial", in *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*, A. Aldini, J. Lopez, and F. Martinelli, Eds., Cham: Springer International Publishing, 2014, pp. 146–166. last visited: Jul. 23, 2026. [Online]. Available: https://doi.org/10.1007/978-3-319-10082-1_6

[BERW] C. Brzuska, C. Egger, et al. "Domino". Code last validated against commit c9a1ebaafbd8c414b18078ac5617be8d619375fa, last visited: Nov. 23, 2026. [Online]. Available: https://github.com/domino-lang/domino

[BERW26] C. Brzuska, C. Egger, et al., "Domino: Computer-assisted game-hopping for large-scale protocols", manuscript, 2026.

[BHR12] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of garbled circuits", in *ACM CCS 2012: 19th Conference on Computer and Communications Security*, (Oct. 16–18, 2012), T. Yu, G. Danezis, and V. D. Gligor, Eds., Raleigh, NC, USA: ACM Press, Oct. 2012, pp. 784–796.

[Bla16] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and proverif", *Found. Trends Priv. Secur.*, vol. 1, no. 1–2, pp. 1–135, Oct. 2016. [Online]. Available: https://doi.org/10.1561/3300000004

[Bla23] B. Blanchet, "CryptoVerif: a Computationally-Sound Security Protocol Verifier (Initial Version with Communications on Channels)", Inria Paris, Tech. Rep. RR-9525, Oct. 2023, p. 166. [Online]. Available: https://inria.hal.science/hal-04246199

[BMR90] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols (extended abstract)", in *22nd Annual ACM Symposium on Theory of Computing*, (Baltimore, MD, USA, May 14–16, 1990), ACM Press, 1990, pp. 503–513.

[BO23] C. Brzuska and S. Oechsner, "A state-separating proof for Yao's garbling scheme", in *CSF 2023: IEEE 36th Computer Security Foundations Symposium*, (Dubrovnik, Croatia, Jul. 10–14, 2023), IEEE Computer Society Press, 2023, pp. 137–152.

[BR06] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs", in *Advances in Cryptology – EUROCRYPT 2006*, (May 28–Jun. 1, 2006), S. Vaudenay, Ed., ser. Lecture Notes in Computer Science, vol. 4004, St. Petersburg, Russia: Springer Berlin Heidelberg, Germany, May 2006, pp. 409–426.

[BR93] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols", in *ACM CCS 93: 1st Conference on Computer and Communications Security*, (Nov. 3–5, 1993), D. E. Denning, R. Pyle, et al., Eds., Fairfax, Virginia, USA: ACM Press, Nov. 1993, pp. 62–73.

[CKKZ12] S. G. Choi, J. Katz, et al., "On the security of the "free-XOR" technique", in *TCC 2012: 9th Theory of Cryptography Conference*, (Mar. 19–21, 2012), R. Cramer, Ed., ser. Lecture Notes in Computer Science, vol. 7194, Taormina, Sicily, Italy: Springer Berlin Heidelberg, Germany, Mar. 2012, pp. 39–53.

[CVC] CVC5. "Installation – cvc5 documentation", last visited: May 25, 2026. [Online]. Available: https://cvc5.github.io/docs/cvc5-1.0.0/installation/installation.html

[dMB08] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver", in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[DY81] D. Dolev and A. C.-C. Yao, "On the security of public key protocols (extended abstract)", in *22nd Annual Symposium on Foundations of Computer Science*, (Oct. 28–30, 1981), Nashville, TN, USA: IEEE Computer Society Press, Oct. 1981, pp. 350–357.

[EGL82] S. Even, O. Goldreich, and A. Lempel, "A randomized protocol for signing contracts", in *Advances in Cryptology – CRYPTO'82*, D. Chaum, R. L. Rivest, and A. T. Sherman, Eds., Santa Barbara, CA, USA: Plenum Press, New York, USA, 1982, pp. 205–210.

[Eur16] European Parliament and the Council of the European Union, *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance)*, May 2016. [Online]. Available: http://data.europa.eu/eli/reg/2016/679/oj

[Hal05] S. Halevi, “A plausible approach to computer-aided cryptographic proofs”, 2005. [Online]. Available: https://eprint.iacr.org/2005/181

[HYLL24] N. Haloani, A. Yanai, et al., “COTI V2: Confidential Computing Ethereum Layer 2”, COTI, Soda Labs, Tech. Rep., 2024. [Online]. Available: https://coti.io/files/coti_v2_whitepaper.pdf

[KS08] V. Kolesnikov and T. Schneider, “Improved garbled circuit: Free XOR gates and applications”, in *Automata, Languages and Programming*, L. Aceto, I. Damgård, et al., Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 486–498.

[Lin17] Y. Lindell, “How to simulate it – a tutorial on the simulation proof technique”, in *Tutorials on the Foundations of Cryptography*, Y. Lindell, Ed., Series Title: Information Security and Cryptography. Additional ISBNs: 978-3-319-57048-8, Cham: Springer International Publishing, 2017, pp. 277–346. last visited: May 21, 2026. [Online]. Available: http://link.springer.com/10.1007/978-3-319-57048-8_6

[LP09] Y. Lindell and B. Pinkas, “A proof of security of Yao’s protocol for two-party computation”, *Journal of Cryptology*, vol. 22, no. 2, pp. 161–188, Apr. 2009.

[LPR+21] T. Lepoint, S. Patel, et al., “Private join and compute from PIR with default”, in *Advances in Cryptology – ASIACRYPT 2021, Part II*, (Dec. 6–10, 2021), M. Tibouchi and H. Wang, Eds., ser. Lecture Notes in Computer Science, vol. 13091, Singapore: Springer, Cham, Switzerland, Dec. 2021, pp. 605–634.

[MNPS04] D. Malkhi, N. Nisan, et al., “Fairplay – a secure two-party computation system”, in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, ser. SSYM’04, USA: USENIX Association, Aug. 13, 2004, p. 20. last visited: Jun. 25, 2026.

[NPS99] M. Naor, B. Pinkas, and R. Sumner, “Privacy preserving auctions and mechanism design”, in *Proceedings of the 1st ACM Conference on Electronic Commerce*, ser. EC ’99, Denver, Colorado, USA: Association for Computing Machinery, 1999, pp. 129–139. [Online]. Available: https://doi.org/10.1145/336992.337028

[Raj25] A. Rajabi, “Towards formally verifying the TLS 1.3 Key Schedule Security in SSBee”, *Master thesis, Aalto University*, Jul. 2025. [Online]. Available: https://aaltodoc.aalto.fi/handle/123456789/138153

[RR21] M. Rosulek and L. Roy, “Three halves make a whole? Beating the half-gates lower bound for garbled circuits”, in *Advances in Cryptology – CRYPTO 2021, Part I*, (Aug. 16–20, 2021), T. Malkin and C. Peikert, Eds., ser. Lecture Notes in Computer Science, vol. 12825, Virtual Event: Springer, Cham, Switzerland, Aug. 2021, pp. 94–124.

[Rus] Rust Foundation. “Other Installation Methods - Rust Forge”, last visited: May 25, 2026. [Online]. Available: https://forge.rust-lang.org/infra/other-installation-methods.html

[Sha26] P. Sharma. “Prateek2000/ttm4905-masters-thesis-archive/”, last visited: Nov. 23, 2026. [Online]. Available: https://github.com/Prateek2000/ttm4905-masters-thesis-archive

[Urb24] A. Urban, “Efficient delegated secure multiparty computation”, Theses, Institut Polytechnique de Paris, Dec. 2024. [Online]. Available: https://theses.hal.science/tel-04955820

[Yao86] A. C.-C. Yao, “How to generate and exchange secrets (extended abstract)”, in *27th Annual Symposium on Foundations of Computer Science*, (Toronto, Ontario, Canada, Oct. 27–29, 1986), IEEE Computer Society Press, 1986, pp. 162–167.

[Z3] Z3. “Releases · Z3Prover/z3”, last visited: Jul. 26, 2026. [Online]. Available: https://github.com/Z3Prover/z3/releases

[ZRE15] S. Zahur, M. Rosulek, and D. Evans, “Two halves make a whole - reducing data transfer in garbled circuits using half gates”, in *Advances in Cryptology – EUROCRYPT 2015, Part II*, (Sofia, Bulgaria, Apr. 26–30, 2015), E. Oswald and M. Fischlin, Eds., ser. Lecture Notes in Computer Science, vol. 9057, Springer Berlin Heidelberg, Germany, 2015, pp. 220–250.

Appendix A
Ideal Game Transformations

This appendix describes the transformations to the ideal games in order to clarify the transformations we make to claim code equivalence.

A.1 Game $\mathcal{R} \rightarrow G^1_{\mathsf{LIN-RK-KDM}}$

$\mathcal{R} \rightarrow G^1_{\mathsf{LIN-RK-KDM}}$

State:

Δ : global circuit offset

$\underline{\underline{\mathsf{GARBLE}(f, x)}}$

$c \leftarrow f(x)$

if $\Delta = \perp$: // Always runs at the start once when Δ is unset

  $\Delta \leftarrow\$ \{0,1\}^\lambda$ // Sample a fresh Δ

$(a, b) \leftarrow x$

$A_a, B_b, C_c \leftarrow\$ \{0,1\}^\lambda$

$R_{00}, R_{01}, R_{10}, R_{11} \leftarrow\$ \{0,1\}^\lambda$

F : Table

$F[a][b] \leftarrow\$ (\mathsf{Enc}(A_a, R_{00}), \mathsf{Enc}(B_b, R_{00} \oplus C_c))$

$F[a][b \oplus 1] \leftarrow\$ (\mathsf{Enc}(A_a, R_{01}), \mathsf{Enc}(B_b \oplus \Delta, 0^\lambda))$

$F[a \oplus 1][b] \leftarrow\$ (\mathsf{Enc}(A_a \oplus \Delta, 0^\lambda), \mathsf{Enc}(B_b, R_{10}))$

$F[a \oplus 1][b \oplus 1] \leftarrow\$ (\mathsf{Enc}(A_a \oplus \Delta, 0^\lambda), \mathsf{Enc}(B_b \oplus \Delta, 0^\lambda))$

$X \leftarrow (A_a, B_b)$

$d \leftarrow \{C_c : c\}$

return (F, X, d)

Figure A.1: Code for $\mathcal{R} \rightarrow G^1_{\mathsf{LIN-RK-KDM}}$

Figure A.1 is a composition of the reduction $\mathcal{R}$ to the ideal LIN-RK-KDM game $G^1_{\mathsf{LIN-RK-KDM}}$.

A.2 Game $G^{1'}_{\mathsf{Sim\text{-}FreeXOR}}$

$G^{1'}_{\mathsf{Sim\text{-}FreeXOR}}$

State:

Δ : global circuit offset

$\mathsf{GARBLE}(f(x), \Phi(f))$

$c \leftarrow f(x)$ // This is 0 since we set a and b to 0

$\Delta \leftarrow\$ \{0,1\}^\lambda$ // this is equivalent to the

// if statement in the ideal version of $\mathcal{R} \rightarrow G^1_{\mathsf{LIN-RK-KDM}}$

$(a, b) \leftarrow (0, 0)$

$A_a, B_a, C_a \leftarrow\$ \{0,1\}^\lambda$

$R_{00}, R_{01}, R_{10}, R_{11} \leftarrow\$ \{0,1\}^\lambda$

F : Table

$F[a][b] \leftarrow\$ (\mathsf{Enc}(A_a, R_{00}), \mathsf{Enc}(B_b, R_{00} \oplus C_c))$

$F[a][b \oplus 1] \leftarrow\$ (\mathsf{Enc}(A_a, R_{01}), \mathsf{Enc}(B_b \oplus \Delta, 0^\lambda))$

$F[a \oplus 1][b] \leftarrow\$ (\mathsf{Enc}(A_a \oplus \Delta, 0^\lambda), \mathsf{Enc}(B_b, R_{10} \oplus C_c \oplus \Delta))$

$F[a \oplus 1][b \oplus 1] \leftarrow\$ (\mathsf{Enc}(A_a \oplus \Delta, 0^\lambda), \mathsf{Enc}(B_b \oplus \Delta, 0^\lambda))$

$X \leftarrow (A_a, B_b)$

$d \leftarrow \{C_c : c\}$

return (F, X, d)

Figure A.2: Code for $G^{1'}_{\mathsf{Sim\text{-}FreeXOR}}$

Figure A.2 is a reformulation of the syntax of $G^1_{\mathsf{Sim\text{-}FreeXOR}}$ (Figure 3.4) so that it is closer to the syntax of $\mathcal{R} \rightarrow G^1_{\mathsf{LIN-RK-KDM}}$. It acts as an intermediate transformation step between $\mathcal{R} \rightarrow G^1_{\mathsf{LIN-RK-KDM}}$ and $G^1_{\mathsf{Sim\text{-}FreeXOR}}$.



NTNU
Norwegian University of
Science and Technology

Aalto University
School of Science