J Cryptol (2020) 33:494–566
https://doi.org/10.1007/s00145-019-09341-z

Journal of
CRYPTOLOGY

CryptHOL: Game-Based Proofs in Higher-Order Logic*

David A. Basin · Andreas Lochbihler · S. Reza Sefidgar
Institute of Information Security, Department of Computer Science, ETH Zurich, Universitätstrasse 6, 8092 Zurich, Switzerland
basin@inf.ethz.ch
mail@andreas-lochbihler.de
reza.sefidgar@inf.ethz.ch

Communicated by Ran Canetti.

Received 24 July 2017 / Revised 3 September 2019
Online publication 6 January 2020

Abstract. Game-based proofs are a well-established paradigm for structuring security arguments and simplifying their understanding. We present a novel framework, CryptHOL, for rigorous game-based proofs that is supported by mechanical theorem proving. CryptHOL is based on a new semantic domain with an associated functional programming language for expressing games. We embed our framework in the Isabelle/HOL theorem prover and, using the theory of relational parametricity, we tailor Isabelle's existing proof automation to game-based proofs. By basing our framework on a conservative extension of higher-order logic and providing automation support, the resulting proofs are trustworthy and comprehensible, and the framework is extensible and widely applicable. We evaluate our framework by formalising different game-based proofs from the literature and comparing the results with existing formal-methods tools.

Keywords. Provable security, Game-based proofs, Theorem proving, Higher-order logic, Isabelle/HOL.

1. Introduction

Problem Context. In the 1980s, *Provable Security* emerged as a way of using reduction arguments to put the security of public-key cryptography on a firm scientific footing. But difficulties in applying this idea raised doubts on the credibility of the resulting proofs [38]. For example, in 1994 the OAEP scheme was proposed and proved secure using a reductionist security argument by Bellare and Rogaway [11] and thereafter became part of the SET electronic payment standard of MasterCard and Visa. Seven years later, Shoup found a gap in OAEP's security proof [73]. The reason such proofs are difficult to get right in practice is manifold and includes: the proofs are technical with many corner

*A preliminary version of this paper appeared at ESOP 2016 [48].

cases that are easily overlooked; more careful validation is needed than the current scientific review processes afford [77]; and there is tension between making the proofs comprehensible yet precise.

Various proposals have been made to address the above problems. For instance, different structuring techniques and abstractions were proposed to simplify constructing proofs and understanding the resulting security arguments. Game-based proofs [12,74], Canetti's universal composability framework [23], and Maurer's constructive cryptography [56] are prominent examples of this. Moreover, in response to Halevi's [31] call for more rigour in cryptographic arguments, numerous formal-methods tools were developed to mechanically check game-based security proofs, such as CryptoVerif [18], CertiCrypt [8], Verypto [15], EasyCrypt [6], and FCF [64]. We examine this line of research in more detail in Sect. 7.

We believe the above approaches all have their merits, but they do not go far enough (cf. Sect. 7). An effective methodology for provable security should ideally produce proofs that are trustworthy in the following two respects.

> **Rigour** The methodology should allow only valid proof steps. In practice, this requires formal verification, since humans err.
> **Comprehensibility** The methodology should capture relevant proof ideas in a way that humans can also create, understand, and check. This means that although humans can trust the formally verified proofs, they do not have to, as they can in principle verify the proofs themselves.

With respect to the current state of the art, structuring techniques and abstractions are insufficiently rigorous when they depend on human validation [63,83]. Moreover, existing formal-methods tools have not yet found a satisfactory balance between comprehensibility and rigour. Their emphasis on increased rigour through formal proofs either diminishes their comprehensibility or limits their scope, for example by offering only a fixed set of proof strategies.

We propose the following desiderata for formal-methods tools that are trustworthy and applicable to a wide range of security proofs.

> **Foundational approach** All proof steps are justified by a small number of simple axioms that are known to be consistent.
> **Automation** Proof automation avoids cluttering the main proof ideas with low-level technical details.
> **Naturality** Abstractions enable users to express security arguments using familiar mathematical notions.
> **Extensibility** Users may introduce new proof rules in a trusted way to overcome the incompleteness of existing proof principles.

In Sect. 7, we will use these desiderata to evaluate the shortcomings of existing formal-methods tools as well as the merits of our own proposal.

CryptHOL In this paper, we present CryptHOL, a new framework for game-based proofs that is supported by mechanical theorem proving. To satisfy the aforementioned desiderata, CryptHOL combines the rigour of higher-order logic (HOL), within the Isabelle/HOL [62] theorem prover, with the structuring benefits of game-based proofs.

First, we introduce the new notion of generative probabilistic values (GPVs), which constitute a kind of probabilistic computations with input and output. These provide a compositional semantic foundation for games with oracles. Second, on top of GPVs we define a functional programming language to express relevant concepts in game-based proofs. These include discrete probability distributions, failure events, games, reductions, oracles, and adversaries. Third, we leverage relational parametricity from the theory of programming languages to formally justify congruence reasoning. This provides a theory about relations between (sub)programs where subprograms may be replaced by equivalent programs. This theory plays an important role in allowing computers and humans alike to focus on, and reason about, parts of games in a compositional way. Fourth, we embed this foundation in Isabelle/HOL in a way that leverages Isabelle's existing infrastructure, automation support, and library of formalised mathematics. Finally, we apply our framework to a number of game-based proofs from the literature.

Our framework and all proofs presented in this paper are available online in the Archive of Formal Proofs [49,53]. In particular, the Isabelle/HOL proof assistant has mechanically checked every definition and verified every theorem. The tutorial [51] explains in more detail how to formalise cryptographic constructions and their proofs in CryptHOL.

Contributions CryptHOL meets our desiderata for a trustworthy and widely applicable formal methods tool for game-based proofs. Moreover, in comparison with existing tools, CryptHOL strikes a better balance between rigour and the comprehensibility of provable security results. We achieve this by combining ideas from cryptology, formal-methods, and programming language research, which we now explain in more detail.

Foundational approach CryptHOL ensures that each proof step is a valid application of logical inference rules in HOL. Games are modelled using GPVs, which themselves are constructed from first principles in HOL (i.e. as a conservative extension of HOL). This thereby ensures the consistency of the resulting theory. Moreover, all definitions and proof steps are mechanically checked by Isabelle, guaranteeing soundness and providing the highest degree of rigour.

Automation Our framework's automation avoids cluttering the main proof ideas with low-level technical details. Users specify the main proof steps in a declarative way and automation fill in the formal details, with some directions from the user. Here, we benefit from Isabelle's structuring techniques and proof automation support that have been extensively developed during the past 25 years. This includes Isabelle's language of human-readable proof scripts, module system, and rewriting engine.

We also develop new ways of automating game-based proofs. In particular, we use the theory of relational parametricity to extend Isabelle's existing proof automation to justify game transformations. Moreover, by making the flow of data explicit in CryptHOL's programming language, we support syntactical reasoning about probabilistic independence, which is frequently needed in game-based proofs.

Naturality Naturality concerns the users' ability to construct proofs at the right level of abstraction, using familiar concepts. The use of GPVs leads to a clean embedding of game-based proofs in Isabelle/HOL. Thus, Isabelle's large library of formalised mathematics, e.g. theorems in algebra or number theory, can be used directly in proofs. Moreover, CryptHOL allows users to decompose proofs into small parts. As each such

part can abstract over irrelevant details, it becomes a building block that can be used in other contexts too. This modular approach makes CryptHOL suitable for formalising complex security arguments (Sect. 6).

The use of a functional programming language plays also a large role here. First, the functional syntax is close to the mathematical notation of probability distributions as found in the literature [28,74]. Our syntax additionally provides a monadic notation for games that resembles more familiar imperative pseudo-code. Second, referential transparency and compositionality ensure that programs can be manipulated algebraically. For example, unlike in an imperative setting, a program may be replaced by an equal one within any larger program without checking for state updates that may hide behind procedure calls. Third, CryptHOL programs can abstract over all relevant concepts such as adversaries, oracles, and games. This greatly improves the modularity of the theorems and proofs and of the framework as a whole. For example, we provide higher-order combinators for composing these concepts, which are implemented themselves as functional programs.

Extensibility Embedding our framework in a proof assistant provides a trusted way for users to add new proof principles. Using our framework's proof automation, users may derive new proof principles that are consistent with the formalised game semantics. Furthermore, using CryptHOL's functional programming language, users may introduce new abstractions and operators on the fly (see Sects. 5.7 and 5.8 for examples), which cannot be easily done in imperative programming languages.

To demonstrate our framework's rigour, comprehensibility, and applicability, we used CryptHOL to formalise different provable security results from the literature. We have formalised an IND-CPA secure scheme based on pseudo-random functions in the random oracle model, as well as various examples from Shoup's tutorial on game-based proofs [74]: the IND-CPA security of Elgamal in the standard model and hashed Elgamal in the random oracle model; an extension of a pseudo-random function with a universal hash function; and the IND-CCA security of a symmetric scheme that utilises a pseudo-random function and an unpredictable function. We will present some of these results in the forthcoming sections.

Structure We start by briefly reviewing game-based proofs, higher-order logic, and the functional programming paradigm in Sect. 2. Next in Sect. 3, we illustrate the syntax and usage of our framework by proving IND-CPA security for the Elgamal encryption scheme. In Sect. 4, we delve into the syntax and semantics of CryptHOL's functional programming language, focusing on language support and reasoning principles for games that do not involve oracle interaction. In Sect. 5, we present GPVs as a new semantic domain for games that involve the adversary's interaction with different oracles. We also present CryptHOL's language constructs and reasoning principles for analysing the aforementioned games. In Sect. 6, we demonstrate our framework's rigour, automation, and naturality through a case study, where we formalise the IND-CCA security argument of a symmetric encryption scheme. Finally, in Sects. 7 and 8, we provide a detailed comparison between our framework and existing formal-methods tools and draw conclusions. "Appendices A and B" provide further background and proof details. "Appendix C" provides a small guide to the sources and explains where the paper deviates from the actual Isabelle/HOL syntax and where the examples can be found in the sources.

2. Background

We introduce here the background necessary for this paper. First, in Sect. 2.1, we review the various flavours of game-based proofs in the cryptography literature and position our framework CryptHOL in this spectrum. Then, we introduce higher-order logic and the Isabelle/HOL proof assistant (Sect. 2.2) and review basic notions and notations of functional programming (Sect. 2.3). Impatient readers may want to read this section lightly on a first pass.

2.1. *Game-Based Cryptographic Proofs*

Game hopping was originally proposed as a technique for structuring cryptographic security proofs and taming their complexity [28,84]. Over time, the level of formality has increased; Kilian and Rogaway [36] put forth the idea of games as programs written in a semi-formal language. Bellare and Rogaway [12] suggested that the games be expressed in a probabilistic programming language and the proofs consist of applications of predefined program transformations until the security claim is obvious. In their model, a game consists of three phases: initialisation, running the adversary with access to the oracles, and finalisation. Halevi [31] picked up this idea and envisioned an interactive proof checker that uses static program analysis to apply simple game transformations specified by the user and to verify their correctness; game hops with complicated probabilistic or algebraic reasoning are to be proven by the user on paper and checked by human reviewers instead of the tool.

In contrast to Halevi's proposal, Shoup [74] objected to being restricted to a fixed toolbox of syntactic program manipulations. He considered games to be a convenient notation for probability distributions rather than formal syntactic objects. This lowers the bar for cryptographers, as they can give free reign to their creativity. So, his proofs mix different types of reasoning, including syntactic program transformations, conventional reasoning about conditional probabilities, and algebraic arguments.

Bellare, Rogaway, and Shoup [12,74] all agree that extending the notation, i.e. programming language, with problem-specific conventions is essential, as otherwise, the definitions and proofs become unreadable and hard to check.

In CryptHOL, a game is just a (discrete sub-)probability distribution expressed in CryptHOL's notation, but the notation has a well-defined formal meaning. Thus, CryptHOL combines the best of two worlds: we achieve the extensibility and flexibility that Shoup calls for because a game is just a distribution rather than a program written in a fixed programming language. At the same time, like envisioned by Bellare and Rogaway, we can express program transformations, prove them correct, and apply them to the syntactic objects as the notation is formal. Instead of constructing sequences of games by applying game transformations like in Halevi's vision, CryptHOL users themselves explicitly specify all the intermediate games and then Isabelle/HOL checks that the given justifications for the transformations are correct. This yields declarative proofs, which are in general easier to understand and maintain as experience has shown in other domains [82].

2.2. *Higher-Order Logic in Isabelle*

Higher-order logic (HOL) combines functional programming with logic [61], and, as we will show, functional programming is relevant for cryptographic notions. This section introduces the logic part and its implementation in the proof assistant Isabelle. Functional programming aspects will be reviewed in the next section.

Terms in higher-order logic are expressed in the simply typed λ-calculus with let-polymorphism [67]. That is, a HOL type is either a type variable (we use Greek letters $\alpha, \beta, \ldots$ for type variables) or a type constructor applied to the appropriate number of types as arguments; we usually write type constructors in blackboard bold ($\mathbb{B}, \mathbb{L}, \mathbb{P}, \ldots$) or infix ($+, \times, \Rightarrow, \ldots$). HOL terms are built from variables, constants, function applications, and abstractions. The notation $t : \tau$ means that the HOL term t has type τ.

HOL has a simple set-theoretic semantics [41,42,67], where types are interpreted as sets and terms denote elements of the set corresponding to their type. In particular, the HOL type $A \Rightarrow B$ of functions from A to B is interpreted as the full function space between A's and B's interpretation. A function need not be given a name: $\lambda x : \alpha.\ t(x)$ denotes the anonymous function that maps every value v of type α to the interpretation of t where x is replaced by v. Typically, we omit the type annotation $: \alpha$ and leave type inference to deduce the most general type. Also, if x does not occur in t, we write _ instead of x, like in the constant function $\lambda\_.\ \mathsf{True}$. A HOL proposition is a closed well-typed λ-term of the type $\mathbb{B}$ of the two truth values True and False.

To prove a proposition, one constructs a derivation using the HOL deduction system. Its proof rules include α-, β-, and η-conversion of the λ-calculus, the standard inference rules for implication ($\longrightarrow$) and equality ($=$), and the axioms of excluded middle, of function extensionality, of choice, and of the existence of an infinite type. In short, HOL is a classical logic of total functions with the axiom of choice. Everything else, e.g. quantifiers and induction schemas, can be expressed as HOL terms and derived from these first principles. For example, the universal quantifier $\forall x.\ P(x)$ is defined like in Church's simple theory of types [24] by a constant[1] $\mathsf{All} : (\alpha \Rightarrow \mathbb{B}) \Rightarrow \mathbb{B}$ given by $\mathsf{All} \equiv \lambda P.\ P = (\lambda\_.\ \mathsf{True})$ and $\forall x.\ P(x)$ is syntactic sugar for $\mathsf{All}\ (\lambda x.\ P(x))$. Defining equations are indicated by $\equiv$ and defined constants are written in sans-serif. In theorems, all variables are implicitly universally quantified.

Clearly, mechanical checks are only meaningful if the axioms and proof rules are consistent, which is the case for HOL [41,42,67]. Moreover, when a user adds more axioms, e.g. a theory of the real numbers, he himself has to ensure that a model still exists. To avoid the danger of inconsistency, all concepts in Isabelle/HOL are introduced definitionally. That is, one first defines a new constant in terms of existing HOL terms (e.g. the universal quantifier is expressed in terms of equality of functions) and then derives the desired properties of the new constant as HOL theorems. Importantly, when sufficiently many properties have been formally derived from the construction (e.g. introduction and elimination rules for All), one no longer needs the internal construction for reasoning. Similarly, a new type is introduced by identifying a suitable, non-empty subset of an existing HOL type. These definition principles are conservative, i.e. by

[1] All function symbols in HOL are called constants as function application is primitive.

making definitions, one cannot prove more than what was possible before. This ensures the overall consistency of the formalisation.

The proof assistant Isabelle/HOL is an implementation of HOL written in Standard ML. Its so-called trusted kernel implements the simply typed λ-calculus and the HOL proof rules and checks that definitions adhere to the principles mentioned above [41,43]. Isabelle accepts a HOL term as a theorem only if the theorem has been constructed using the kernel's proof rules. This design ensures a small trusted code base: only implementation errors in the kernel (a few thousand lines of well-tested and scrutinised code) could result in erroneous proofs being accepted.

To alleviate the user from the burden of constructing proofs from primitive derivations, Isabelle/HOL comes with several *proof engines* that compile high-level proof steps down to primitive inferences. Thus, users can write their proofs at a high level of abstraction and call the appropriate proof engine, possibly with some hints.

Isabelle/HOL comes with a library of formalised mathematics that follows these principles. For example, the rationals are constructed from a pair of integers (numerator and denominator) and the reals as Dedekind cuts of rationals. Probability theory, analysis, and many algebraic concepts have been formalised in this way.

We emphasise that our framework is definitional too. Thus, the correctness of proofs done in our framework can be reduced to the consistency of the HOL axioms. No further axioms are required.

2.3. *Functional Programming in HOL*

Analogous to the proof engines, Isabelle's definitional packages alleviate the user from the low-level details of the definitional principles. They take a specification of the new type or constant, internally construct the definition from existing concepts, and derive the user specification as theorems from the definition. This way, users can work in HOL like in a functional programming language with algebraic datatypes and recursive functions, which we explain in this section. We make heavy use of functional programming idioms when formalising cryptographic constructions and in the definitions of our framework.

An algebraic datatype is a disjoint union of (combinations of) HOL types where the embedding into the union is made explicit using constructors. For example, the natural numbers $\mathbb{N}$ form an algebraic datatype as they are the disjoint union of the singleton set $\{0\}$ and the successors of all natural numbers $\{ n + 1 \mid n \in \mathbb{N} \}$, i.e. the constructors are $0 : \mathbb{N}$ and $\mathsf{Suc} : \mathbb{N} \Rightarrow \mathbb{N}$. Algebraic datatypes can be polymorphic. For example, the type of pairs $\alpha \times \beta$ has only one polymorphic constructor $(\_, \_) : \alpha \Rightarrow \beta \Rightarrow \alpha \times \beta$. Datatype values are analysed using case expressions. For example, $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{Suc}(y) \Rightarrow (y, \mathsf{True}) \mid 0 \Rightarrow (0, \mathsf{False})$ analyses $x : \mathbb{N}$ and returns the predecessor y of x and a Boolean to indicate whether x is greater than 0. In Isabelle, the command **datatype** [20] introduces algebraic datatypes according to definitional principles. It also derives an induction schema, e.g. $P(0) \longrightarrow (\forall x.\ P(x) \longrightarrow P(\mathsf{Suc}(x))) \longrightarrow (\forall x.\ P(x))$ for $\mathbb{N}$.

Recursive function definitions also need justification as recursive definitions are not primitive in HOL. For this paper, three kinds of justification suffice: well-founded recursion, least fixpoints, and primitive corecursion, which are explained in "Appendix A.1". They are all supported by Isabelle packages.

We typically curry functions with several arguments, for example, a function f with n arguments has type $\tau_1 \Rightarrow \tau_2 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$, where the function type constructor $\Rightarrow$ associates to the right. So f takes its arguments individually rather than as a single tuple of type $\tau_1 \times \tau_2 \times \ldots \tau_n \Rightarrow \tau$. Function application is written as usual using parentheses as in $f(t_1, t_2, \ldots, t_n)$. Currying has the advantage that functions can be applied partially: if too few arguments are supplied, the result is a specialised function that waits for the missing arguments, e.g. $f(t_1, t_2) = \lambda x_3 \ldots x_n.\ f(t_1, t_2, x_3, x_4, \ldots, x_n)$, when $x_3, \ldots, x_n$ are not free in t_1 and t_2. Tuples nevertheless occur as arguments when the components form a conceptual entity, e.g. a two-part ciphertext (β, ζ). To avoid ambiguities with partial applications, we do not merge the function application parentheses with the tuple constructor: $g((\beta, \zeta))$ applies the unary function g to the tuple (β, ζ) and $h(\beta, \zeta)$ is a two-argument function h applied to β and ζ. Enclosing a binary infix operator in parentheses like in $(=)$ and $(\longrightarrow)$ turns it into a relation or a function.

Other types and operations The singleton type $\mathbb{1}$ has only one element $\circledast$. Pairs (type $\alpha \times \beta$) come with two projection functions π_1 and π_2. Tuples are identified with pairs nested to the right, i.e. (a, b, c) is identical to $(a, (b, c))$ and $\alpha \times \beta \times \gamma$ to $\alpha \times (\beta \times \gamma)$. Dually, the sum type $\alpha + \beta$ models the disjoint union of the types α and β with the injections $\mathsf{Left} :: \alpha \Rightarrow \alpha + \beta$ and $\mathsf{Right} :: \beta \Rightarrow \alpha + \beta$. The predicates $\mathsf{is\text{-}Left}(x)$ and $\mathsf{is\text{-}Right}(x)$ check whether x is of the form $\mathsf{Left}(\_)$ or $\mathsf{Right}(\_)$.

Sets (type $\mathbb{P}(\alpha)$) are isomorphic to predicates (type $\alpha \Rightarrow \mathbb{B}$) via the bijections membership $\in$ and set comprehension $\{ x \mid \_ \}$; the empty set is $\{\}$. Binary relations are sets of pairs and written in infix, for example, $x\ R\ y$ denotes $(x, y) \in R$. We write $R[A] \equiv \{ y \mid \exists x \in A.\ x\ R\ y \}$ for the image of a set A under a relation R.

The datatype $\mathbb{M}(\alpha) = \mathsf{None} \mid \mathsf{Some}\ \alpha$ adjoins a new element None to α while all existing values in α are prefixed by Some. Maps (partial functions) are modelled as functions of type $\alpha \Rightarrow \mathbb{M}(\beta)$, where None represents undefinedness and $f(x) = \mathsf{Some}(y)$ means that f maps x to y. The empty map $\varnothing \equiv (\lambda\_.\ \mathsf{None})$ is undefined everywhere. Map update is defined as $f(a \mapsto b) \equiv (\lambda x.\ \mathsf{if}\ x = a\ \mathsf{then}\ \mathsf{Some}(b)\ \mathsf{else}\ f(x))$.

It is sometimes convenient to write function application forward: $x \rhd f \equiv f(x)$ expresses that x is passed to f and $\rhd$ associates to the left, e.g. $x \rhd f \rhd g = g(f(x))$. Function application can be lifted to most HOL types. Typically, we write the operation that lifts functions to a type constructor as the type constructor with a hat $\widehat{\ }$. For example, $(\widehat{\times})$ on pairs is given by $(x, y) \rhd f\ \widehat{\times}\ g \equiv (f(x), g(y))$, and $\widehat{+}$ for sums by $x \rhd f\ \widehat{+}\ g \equiv \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Left}(y) \Rightarrow \mathsf{Left}(f(y)) \mid \mathsf{Right}(z) \Rightarrow \mathsf{Right}(g(z))$. In case of a unary type constructor, we often attach the type constructor to $\rhd$ when using forward application. For sets, e.g. $A \rhd \widehat{\mathbb{P}}\ f \equiv \{ f(x) \mid x \in A \}$ denotes the image of A under f; and $\widehat{\mathbb{M}} : (\alpha \Rightarrow \beta) \Rightarrow \mathbb{M}(\alpha) \Rightarrow \mathbb{M}(\beta)$ is given by $\widehat{\mathbb{M}}(f, \mathsf{None}) = \mathsf{None}$ and $\widehat{\mathbb{M}}(f, \mathsf{Some}(x)) = \mathsf{Some}(f(x))$.

Effects All HOL functions are pure in that they cannot perform any kind of side effect like probabilistic choice, interaction with an environment, or state updates. Such effects are explicitly modelled using Haskell-style monads [81]. A monad consists of a polymorphic type constructor $\mathbb{T}(\alpha)$ and two polymorphic operations $\mathsf{return} : \alpha \Rightarrow \mathbb{T}(\alpha)$ and $(\gg\!=) : \mathbb{T}(\alpha) \Rightarrow (\alpha \Rightarrow \mathbb{T}(\beta)) \Rightarrow \mathbb{T}(\beta)$. Intuitively, the type constructor $\mathbb{T}$ models the effects and the type variable α represents the computation's result. Then, return embeds an effect-free value into the effectful world, and $(\gg\!=)$ models sequential composition, where the

second computation may depend on the result of the first. Every monad must satisfy the following monad laws:

$$\mathsf{return}\ x \gg\!= f \;=\; f\ x \qquad v \gg\!= (\lambda x.\ \mathsf{return}\ x) \;=\; v \qquad \text{(neutrality)}$$
$$(v \gg\!= f) \gg\!= g \;=\; v \gg\!= (\lambda x.\ f(x) \gg\!= g) \qquad \text{(associativity)}$$

For readability, CryptHOL uses Haskell-style do notation for monadic computations: **do** { $x \leftarrow v$; f } desugars to $v \gg\!= (\lambda x.$ **do** { f }) and **do** { v } to v otherwise. In this paper and our formalisation, we use return, ($\gg\!=$), and **do** for all the different monads and rely on type inference to resolve the overloading.

For example, the type constructor $\mathbb{M}$ models the effect of undefined values as a monad. Here, $\mathsf{return}\ x \equiv \mathsf{Some}(x)$ marks a value as being defined, None denotes undefinedness, and $x \gg\!= f$ applies the partial function $f : \alpha \Rightarrow \mathbb{M}(\beta)$ to the possibly undefined value $x : \mathbb{M}(\alpha)$, propagating undefinedness. Formally, $\mathsf{None} \gg\!= f \equiv \mathsf{None}$ and $\mathsf{Some}(x') \gg\!= f \equiv f(x')$. Hence, given two partial functions $f : \alpha \Rightarrow \mathbb{M}(\beta)$ and $g : \beta \Rightarrow \mathbb{M}(\gamma)$, we can compose them sequentially to a function $h : \alpha \Rightarrow \mathbb{M}(\gamma)$ by $h(x) \equiv f(x) \gg\!= g$, or using **do** notation: $h(x) \equiv$ **do** { $y \leftarrow f(x)$; $g(y)$ }, which is by the neutrality law equivalent to $h(x) \equiv$ **do** { $y \leftarrow f(x)$; $z \leftarrow g(y)$; $\mathsf{return}\ z$ }. Unlike in C-like languages, return does *not* return from a procedure. For example, **do** { $x \leftarrow f(1)$; $y \leftarrow \mathsf{if}\ \mathsf{even}(x)\ \mathsf{then}\ g(x)\ \mathsf{else}\ \mathsf{return}\ 0$; $h(x + y)$ } assigns to the local variable x the result of $f(1)$ and to y the result of $g(x)$ if x is even and 0 otherwise, and returns the result of h applied to $x + y$, propagating undefinedness.

Every monad has a canonical operation $\widehat{\mathbb{T}}$ to lift functions into the monad given by $v \rhd^{\widehat{\mathbb{T}}} f \equiv$ **do** { $x \leftarrow v$; $\mathsf{return}\ f(x)$ }. We typically use $\widehat{\mathbb{T}}$ to post-process the result of an effectful computation. In the $\mathbb{M}$ example, $\widehat{\mathbb{M}}(f, x) = x \rhd^{\widehat{\mathbb{M}}} f$ applies a total function $f : \alpha \Rightarrow \beta$ to a possibly undefined value $x : \mathbb{M}(\alpha)$; the result is undefined if and only if (iff) x is.

3. IND-CPA Security of the Elgamal Encryption Scheme

In this section, we illustrate the principles of our framework by formalising Elgamal encryption [27] and proving indistinguishability under chosen plaintext attacks (IND-CPA) given the decisional Diffie–Hellman (DDH) assumption. Shoup [74] also provides a detailed security proof of this encryption scheme. By using the same example, we explain how the games and the different kinds of reasoning steps in game-hopping proofs are reflected in our framework and where mechanised reasoning differs from rigorous reasoning with pen and paper. The framework and its semantics is formally introduced in Sect. 4. In this section, we first consider the concrete security setting where the fixed security parameter η is implicit to all algorithms of the scheme. In the asymptotic security setting, described at the end of this section, the security parameter will be made explicit.

A public-key encryption scheme consists of three probabilistic algorithms: (i) the key generation algorithm key-gen takes no input and outputs a public/private key pair (pk, sk); (ii) the encryption algorithm aenc takes as input a public key pk and a plaintext

m and outputs a ciphertext c; and (iii) the decryption algorithm adec takes as input a private key sk and a ciphertext c and outputs a message m.

For the Elgamal encryption scheme, we fix a finite cyclic group $\mathcal{G}$ with generator g and order $|\mathcal{G}|$. The public key α is an arbitrary group element g^x, and the private key is the exponent x. Messages are group elements too. To encrypt a message m under the public key α, the algorithm multiplies m with α raised to a random power y; the ciphertext consists of g^y and the product. Decryption inverts the multiplication by exploiting the identity $\alpha^y = (\mathsf{g}^y)^x$. Shoup [74] specifies the algorithms as follows, where the set $\mathbb{Z}_m$ contains the integers between 0 and $m-1$, and $\xleftarrow{\$}$ and $\leftarrow$ denote random sampling and assignment, respectively.

	input	algorithm
key generation		$x \xleftarrow{\$} \mathbb{Z}_{\lvert\mathcal{G}\rvert},\ \alpha \leftarrow \mathsf{g}^x,\ pk \leftarrow \alpha,\ sk \leftarrow x$
encryption	α, m	$y \xleftarrow{\$} \mathbb{Z}_{\lvert\mathcal{G}\rvert},\ \beta \leftarrow \mathsf{g}^y,\ \delta \leftarrow \alpha^y,\ \zeta \leftarrow \delta \cdot m,\ c \leftarrow (\beta, \zeta)$
decryption	$x, (\beta, \zeta)$	$m \leftarrow \zeta / \beta^x$

In our framework, algorithms are probabilistic HOL functions from inputs to outputs. The main difference to Shoup's notation is that they explicitly return the outputs. Thus, all intermediate variables, which are not part of the outputs, are local to the algorithm. Syntactically, we write $\otimes$ for group multiplication, $\hat{}$ for exponentiation, and $\_^{-1}$ for the inverse. Assignment $\leftarrow$ is always probabilistic and takes a probability distribution on the right-hand side. For example, uniform(A) denotes the uniform distribution over the non-empty, finite set A

```
key-gen ≡ do {                 aenc(α, m) ≡ do {                adec(x, (β, ζ)) ≡
  x ← uniform(ℤ_|G|);            y ← uniform(ℤ_|G|);              return ζ ⊗ (β ^ x)^-1
  return (g ^ x, x) }            return (g ^ y, (α ^ y) ⊗ m) }
```

(1)

IND-CPA security is modelled as a game between a challenger and the attacker $\mathcal{A}$ following Goldwasser and Micali [28]. Figure 1 shows our formalisation on the left and Shoup's notation on the right.

1. The challenger generates a new key pair $(pk, sk) \leftarrow$ key-gen (line 2).
2. The adversary is given pk, chooses two plaintexts m_0 and m_1, and gives these to the challenger (line 3).
3. The challenger randomly picks one of the two messages and encrypts it under pk (coin abbreviates uniform$(\{0, 1\})$) (lines 5–6).
4. The adversary receives the challenge ciphertext and outputs a bit b' (line 7).

The adversary tries to guess the random bit b. The IND-CPA advantage is therefore the difference between the probability of $b = b'$ and $1/2$.

There are several points worth discussing. First, the game returns the Boolean $b = b'$ because this is the event we are interested in. Recall that assignments are local in our framework, and hence the game must return the event(s) of interest. Accordingly, the advantage adv$(\mathcal{A})$ checks the probability of whether the game reports a successful guess.

```
1 G((A1, A2)) ≡ try do {
2   (pk, sk) ← key-gen;
3   ((m0, m1), σ) ← A1(pk);
4   assert(valid-plains(m0, m1));
5   b ← coin;
6   c ← aenc(pk, if b then m0 else m1);
7   b' ← A2(c, σ);
8   return b = b'
9 } else coin
```

$x \xleftarrow{\$} \mathbb{Z}_{|\mathcal{G}|}, \alpha \leftarrow \mathsf{g}^x$
$r \xleftarrow{\$} R, (m_0, m_1) \leftarrow \mathcal{A}(r, \alpha)$

$b \xleftarrow{\$} \{0, 1\}$
$y \xleftarrow{\$} \mathbb{Z}_{|\mathcal{G}|}, \beta \leftarrow \mathsf{g}^y, \delta \leftarrow \alpha^y, \zeta \leftarrow \delta \cdot m_b$
$b' \leftarrow \mathcal{A}(r, \alpha, \beta, \zeta)$

$\mathsf{adv}(\mathcal{A}) \equiv |\mathcal{P}[\mathsf{G}(\mathcal{A}) = \mathsf{True}] - 1/2|$

advantage is $|\mathcal{P}[b = b'] - 1/2|$

Fig. 1. The IND-CPA game in our framework (left) and by Shoup [74] (right).

Second, our definitions are in fact parameterised by the encryption scheme. That is, we define the IND-CPA game and the advantage in a module IND-CPA that takes the encryption scheme as a parameter (the grey parts in Fig. 1). Thus, we get the game specialised for Elgamal by instantiating the module with the Elgamal encryption scheme. Modules typically contain definitions and generic theorems, which can be reused for different schemes by instantiation. Note that the locality of assignments is crucial for modularity, as otherwise, e.g. assigning to b within some instantiation of aenc in line 6 would overwrite the coin flip in line 5.

Third, we use assertions (assert) and error handling (try _ else _). Assertions serve two purposes: (i) we use them to validate inputs received from the adversary (line 4). For Elgamal, for example, the predicate valid-plains checks that both plaintext messages indeed belong to the group $\mathcal{G}$. When an assertion fails, the current computation is aborted and the execution continues with the else branch of the closest block surrounding the failure, if any. In Fig. 1, if any step in the game fails, G behaves like a fair coin flip, and the advantage is 0 in that case. With input validation, we can prove a stronger security theorem that does not need the assumption that the adversary always produces valid plaintexts. (ii) Even if we had constrained the adversary to produce only valid plaintexts, this information would not be obvious to Isabelle's reasoning engines. Here, assertions provide a way to include such information syntactically into the programs. The reasoning engines, in particular contextual rewriting, can then pick the asserted properties up from the program text and therefore know that they hold when analysing the remainder of the program. Footnote 5 on page 23 gives an example where the assertion in line 4 helps with proof automation.

Finally, compare our formalisation to Shoup's in Fig. 1 on the right. Apart from input validation, there are three main differences: First, Shoup's definition is tailored to Elgamal instead of being generic. Second, Shoup assumes all assignments are visible globally, i.e. he formulates arbitrary events over program variables. Third, his adversary $\mathcal{A}$ is a deterministic algorithm, which takes a uniformly sampled r as random input. In contrast, we model the adversary $\mathcal{A}$ as a pair $(\mathcal{A}_1, \mathcal{A}_2)$ of two probabilistic algorithms. Our formulation is more suitable for a formal setting because we do not need to specify

a set R that supplies enough randomness to the given adversary.[2] Instead, the adversary can pass an (adversary-specific) state σ from the first invocation to the second. This way, the first part can communicate its random choices to the second.

The security of the Elgamal scheme relies on the hardness of computing the discrete logarithm. In detail, the DDH assumption states that given two random group elements g^x and g^y, it is hard to distinguish $\mathsf{g}^{x \cdot y}$ from a random group element g^z, where $\cdot$ denotes multiplication on numbers. Formally, a DDH adversary $\mathcal{A}$ is a probabilistic computation that takes three group elements and outputs a Boolean. We model the two settings as two games G_0 and G_1 parameterised by the adversary. The DDH advantage $\mathsf{adv}(\mathcal{A})$ captures the difficulty of $\mathcal{A}$ distinguishing the two settings.

$$\begin{array}{ll} \mathsf{G}_0(\mathcal{A}) \equiv \textbf{do}\ \{ & \mathsf{G}_1(\mathcal{A}) \equiv \textbf{do}\ \{ \\ \quad x \leftarrow \mathsf{uniform}(\mathbb{Z}_{|\mathcal{G}|}); & \quad x \leftarrow \mathsf{uniform}(\mathbb{Z}_{|\mathcal{G}|}); \\ \quad y \leftarrow \mathsf{uniform}(\mathbb{Z}_{|\mathcal{G}|}); & \quad y \leftarrow \mathsf{uniform}(\mathbb{Z}_{|\mathcal{G}|}); \\ \quad \mathsf{let}\ z = x \cdot y; & \quad z \leftarrow \mathsf{uniform}(\mathbb{Z}_{|\mathcal{G}|}); \\ \quad \mathcal{A}(\mathsf{g} \hat{}\, x, \mathsf{g} \hat{}\, y, \mathsf{g} \hat{}\, z)\ \} & \quad \mathcal{A}(\mathsf{g} \hat{}\, x, \mathsf{g} \hat{}\, y, \mathsf{g} \hat{}\, z)\ \} \end{array} \tag{2}$$

$$\mathsf{adv}(\mathcal{A}) = \left| \mathcal{P}\big[\mathsf{G}_0(\mathcal{A}) = \mathsf{True}\big] - \mathcal{P}\big[\mathsf{G}_1(\mathcal{A}) = \mathsf{True}\big] \right| \tag{3}$$

Figure 2 shows the Elgamal security proof in our framework expressed in the Isar proof language [16]. We emphasise that this is a faithful pretty print of the Isabelle source code, except for small omissions marked with ⟨—⟩. When such a proof script is given to Isabelle/HOL, it checks that all definitions are type-correct and conservative extensions of HOL and verifies the proofs of all theorems by constructing a formal derivation using the HOL's proof rules, which is guided by the proof engines and the given hints. In case this fails, Isabelle reports the malformed definitions and the non-verified proof steps as errors; there are no such errors for the proof script in Fig. 2. Readers may convince themselves that the formal proof script is readable and at an adequate level of abstraction.

The definitions and proofs are in a module ELGAMAL (a **locale** in Isabelle terminology) parameterised by a group $\mathcal{G}$ that is assumed to be finite and cyclic (lines 1–2). Such assumptions become implicit assumptions for all statements proved within the module. This module also contains the definitions in (1) and imports the modules DDH and IND-CPA as instances with the name qualifiers ddh and ind-cpa, respectively (not shown in Fig. 2). That is, we can refer to the definitions in (2, 3) and in Fig. 1 specialised to the group and the Elgamal encryption scheme using these qualifiers, e.g. $\mathsf{ddh.G}_0$

[2] It is possible to formalise Shoup's adversary model in HOL, but it is harder to reason with. For example, Hurd [35] formalised probabilistic functions as deterministic functions that receive an infinite stream of random bits as input and consume a finite part of it. Being infinite, the stream can never run out of fresh randomness. But this comes at a cost: infinite bitstreams are not discrete, so every function must be proven measurable. Moreover, Hurd must formally prove healthiness conditions for every function, e.g. that the function looks only at the random bits it consumes. Also, this model is not extensional. For example, the function which consumes and returns the first bit mathematically yields the same distribution on bits as the one which returns the negation of the first bit, but in Hurd's model, the two distributions are unequal. Hence, equational reasoning about probability distributions cannot use the equality notion built into the logic, which complicates proofs.

```
1  locale ELGAMAL = fixes 𝒢 : group(α)                                   named context
2    assumes finite-cyclic-group(𝒢) begin                                declaration
3  function elgamal-red_(𝒜1,𝒜2)(α, β, γ) ≡
4    try do {
5      ((m0, m1), σ) ← 𝒜1(α);
6      assert valid-plains(m0, m1);                                      definition
7      b ← coin;                                                         of the
8      let m = if b then m0 else m1;                                     reduction
9      b' ← 𝒜2((β, γ ⊗ m), σ);
10     return b = b'
11   } else coin
12 theorem security: ind-cpa.adv(𝒜) = ddh.adv(elgamal-red_𝒜)             concrete security
13 proof –                                                               statement
14   obtain 𝒜1 and 𝒜2 where 𝒜 = (𝒜1, 𝒜2) by(cases 𝒜)                     preliminaries
15   note [simp] = bind-map-spmf ⟨⟷⟩                                     and proof engine
16   have ddh.G1(elgamal-red_𝒜) = try do {                               configuration
17         x ← uniform(ℤ_|𝒢|);
18         y ← uniform(ℤ_|𝒢|);
19         ((m0, m1), σ) ← 𝒜1(g ^ x);
20         assert valid-plains(m0, m1);                                  step 1:
21         b ← coin;                                                     inline the
22         m ← uniform(ℤ_|𝒢|) ▷̂ (λz. g ^ z ⊗ (if b then m0 else m1));   reduction
23         b' ← 𝒜2((g ^ y, m), σ);                                       in the game
24         return b = b'
25       } else coin
26     by(simp add: ddh.G1-def)
27   also have ... = try do {
28         x ← uniform(ℤ_|𝒢|);
29         y ← uniform(ℤ_|𝒢|);
30         ((m0, m1), σ) ← 𝒜1(g ^ x);                                    step 2: apply
31         assert (valid-plain(m0) ∧ valid-plain(m1));                   one-time-pad
32         m ← uniform(ℤ_|𝒢|) ▷̂ (λz. g ^ z);                             for cyclic groups
33         b' ← 𝒜2((g ^ y, m), σ);                                       and sample
34         coin ▷̂ (λb. b = b')                                           challenge bit b
35       } else coin                                                     after the guess
36     by(simp add: one-time-pad-group ⟨⟷⟩)                             of the adversary
37   also have ... = coin                                                step 3: apply
38     by(simp add: one-time-pad-𝔹 try-bind-lossless)                    one-time-pad
39   also have ddh.G0(elgamal-red_𝒜) = ind-cpa.G(𝒜)                      for booleans
40     by(simp add: ddh.G0-def ind-cpa.G-def nat-pow-pow ⟨⟷⟩)             and eliminate
41   ultimately show ind-cpa.adv(𝒜) = ddh.adv(elgamal-red_𝒜)             dead code
42     by(simp add: ddh.adv-def ind-cpa.adv-def)                         step 4: use DH
43 qed                                                                   group property
44 end                                                                   deduce claim
45 locale ELGAMAL' = fixes 𝒢 : ℕ ⇒ group(α)                              from interme-
46   assumes ∀η. finite-cyclic-group(𝒢(η)) begin                        diate steps
47 sublocale concrete ≡ ELGAMAL(𝒢(η)) for η by(simp add: ⟨⟷⟩)
48 theorem security':                                                   asymptotic
49   assumes negl(λη. ddh.adv(η, concrete.elgamal-red_η,𝒜(η)))          security
50   shows negl(λη. ind-cpa.adv(η, 𝒜(η)))                               statement
51     by(simp add: concrete.security ⟨⟷⟩)                              and proof
52 end
```

Fig. 2. Elgamal security proof.

and ind-cpa.adv. The concrete security statement (line 12) shows that $\mathcal{A}$'s advantage in the IND-CPA game is the same as the DDH advantage of the reduction (defined in lines 3–11). The proof proceeds by several intermediate steps (**have**) in which the game is transformed; the commands **by** ... call Isabelle's proof engines (e.g. *simp* for term rewriting) with appropriate hints. The highlighted parts show the game transformations: the original game and the transformed game are written out declaratively and their equality is expressed as an equality between the probability distributions they denote. Here, ... stands for the right-hand side of the previous **have** statement. The first three steps prove that the game $\mathsf{ddh.G}_1(\mathsf{elgamal\text{-}red}_{\mathcal{A}})$ is identical to a coin flip in a series of equality statements: the first step inlines the reduction $\mathsf{elgamal\text{-}red}_{\mathcal{A}}$ in the $\mathsf{ddh.G}_1$ game (lines 16–26). The second step applies the one-time-pad lemma for cyclic groups, making the challenge ciphertext m independent of the challenge bit b, which can then be sampled after the adversary's guess (lines 27–36). The third step then uses the one-time-pad lemma for Booleans to transform most of the game into dead code, which is then eliminated (lines 37–38). The fourth and last steps show that the other DDH game $\mathsf{ddh.G}_0(\mathsf{elgamal\text{-}red}_{\mathcal{A}})$ is the same as the IND-CPA game for Elgamal (lines 39–40). In summary, $\mathcal{A}$'s IND-CPA advantage is the same as the DDH advantage of $\mathsf{elgamal\text{-}red}_{\mathcal{A}}$ (lines 41–42). We will look closer at this proof in Sects. 4.3 and 4.4 when we describe the reasoning principles of our framework.

So far, the security parameter η has been implicit in our formalisation. Using Isabelle's module system, we make all definitions and statements dependent on the security parameter and derive the conventional asymptotic security statement[3] from the concrete security statement (lines 45–52). In detail, the module ELGAMAL′ now fixes a family $\mathcal{G}$ of finite and cyclic groups (lines 45–46). The command **sublocale** in line 47 imports the ELGAMAL module parametrically for every η with the name qualifier concrete. Imports must immediately discharge the assumptions of the imported module; here, the finite cyclic group assumption of ELGAMAL trivially follows from ELGAMAL′'s assumption. Thus, $\mathcal{A}$'s advantage against the Elgamal scheme is negligible (as a function of η) if $\mathsf{elgamal\text{-}red}_{\mathcal{A}}$'s advantage against the DDH game is (lines 48–51). Here, $\mathsf{negl}(f)$ holds for a function f iff $f \in \mathrm{o}(\lambda x.\ 1/x^c)$ for all $c > 0$.

4. Formal Reasoning about Probability Distributions

Machine-checked proofs need a formal language for expressing probabilistic computations and reasoning principles for the language constructs. We model probabilistic computations as HOL functions from inputs to discrete subprobability distributions over results. For example, a game takes an adversary as input and returns a discrete subprobability distribution over outcomes. Such distributions are formulated as monadic functional programs in HOL (Sect. 4.1).

We choose a functional monadic language over an imperative one for two reasons: First, a functional language provides better abstraction and extension facilities. For example, many operators can be expressed as higher-order functions that abstract over

[3] Typically, asymptotic security statements quantify over all polynomially bounded adversaries and thus eliminate the dependence on the concrete reduction. For further discussion of that point, see Sect. 8.

parts of the program (see Sects. 5.7 and 5.8 for examples). Second, functions are pure and a (part of a) program has the same meaning in every context. Hence, we can replace equal programs in any contexts and all dependencies are explicit. For example, we can read off the independence of random variables syntactically.

Subprobability distributions have two advantages over full probability distributions. First, they form a chain-complete partial order, so we can define recursive computations as least fixpoints (see Sect. 4.2 for examples). Second, the unassigned probability mass can be used to model failures, which serve two purposes: (i) we can transfer control non-locally in error cases, such as invalid data produced by the adversary and (ii) we can use assertions to embed properties into the program such that the proof engines can pick them up easily during proof checking. We choose to restrict our framework to discrete distributions because they are easier to reason about than measure–theoretic distributions, which clutter proofs with measurability requirements. For most cryptographic arguments, discrete subprobability distributions suffice.

We derive proof rules for reasoning about probabilistic programs and illustrate their usage using the Elgamal example from Fig. 2 (Sects. 4.3–4.5).

4.1. *Syntax and Semantics*

Our syntax and semantics is mostly standard [2,68]. The technical novelties are that our fixpoints are more general than in previous work and thus easier to use (they require only monotonicity instead of continuity) and how we use assertions and error handling. We briefly introduce the relevant concepts and notation here and discuss the design choices.

A discrete subprobability distribution is given by its subprobability mass function (spmf), which is a non-negative real-valued function that sums up to *at most* 1. We define the type $\mathbb{D}(\omega)$ of all spmfs over elementary events from the type ω, such as the winning condition in the IND-CPA game in Fig. 1. We use variables p and q for spmfs. We make applications of spmfs explicit using the operator !; so, $p\,!\,x$ denotes the subprobability mass that the spmf p assigns to the elementary event x. An spmf can be considered as a discrete random variable: an event $A : \mathbb{P}(\omega)$ is a set of elementary events whose subprobability $\mathcal{P}[p \in A]$ is given by $\sum_{y \in A} p\,!\,y$. In case of a singleton set $A = \{x\}$, we write $\mathcal{P}[p = x]$ instead of $\mathcal{P}[p \in \{x\}]$. The support $\mathsf{support}(p) = \{x \mid p\,!\,x > 0\}$ is countable by construction. Moreover, the weight $\|p\|$ of p is the total probability mass assigned by p, i.e. $\|p\| = \sum_y p\,!\,y$. If p is a probability distribution, that is, $\|p\| = 1$, then we call p lossless following [8] (notation $\mathsf{lossless}(p)$). Note that many game transformations, e.g. transitions based on failure events (Proposition 2), require that the distributions involved are lossless. So losslessness is a frequent assumption of the security theorems in CryptHOL. We include non-lossless distributions in our semantic domain such that we can formally state and prove that recursive definitions terminate and therefore are lossless (Sect. 4.2).

Subprobability distributions are constructed using four language primitives:

1. the uniform distribution $\mathsf{uniform}(A)$ over a set $A : \mathbb{P}(\omega)$ given by

$$\mathsf{uniform}(A)\,!\,x = \begin{cases} 1/|A| & \text{if } x \in A \text{ and } A \text{ is finite} \\ 0 & \text{otherwise;} \end{cases}$$

2. monadic sequencing $(\gg\!=) : \mathbb{D}(\omega_1) \times (\omega_1 \Rightarrow \mathbb{D}(\omega_2)) \Rightarrow \mathbb{D}(\omega_2)$ given by

$$(p \gg\!= f)\,!\,x = \sum_{y \in \mathsf{support}(p)} (p\,!\,y) \cdot (f(y)\,!\,x),$$

   that is, $p \gg\!= f$ averages over the distributions $f(x)$, weighing each according to the distribution p; from a computational point of view, $p \gg\!= f$ first executes p to obtain a result x and then $f(x)$;
3. error handling $\mathsf{try}\ p\ \mathsf{else}\ q$, which distributes the probability mass not assigned by p according to q—formally $(\mathsf{try}\ p\ \mathsf{else}\ q)\,!\,x = p\,!\,x + (1 - \|p\|) \cdot q\,!\,x$; and
4. least fixpoints of monotone functions (see Sect. 4.2), which give semantics to recursively defined subdistributions.

Any discrete subprobability distribution can be expressed in terms of these four operations as shown by Knuth and Yao [37]. For convenience, our framework provides many derived language elements. This is an advantage of the functional style over the imperative: new language constructs can be integrated easily since everything is compositional—as long as the construct can be expressed in the semantic domain. For example, we define three special cases of the uniform distribution:

- $\mathsf{coin} : \mathbb{D}(\mathbb{B})$ given by $\mathsf{coin} \equiv \mathsf{uniform}(\{\mathsf{True}, \mathsf{False}\})$ samples a random bit;
- $\mathsf{return} : \omega \Rightarrow \mathbb{D}(\omega)$ given by $\mathsf{return}\ x \equiv \mathsf{uniform}(\{x\})$ computes the one-point distribution on x, which is the monadic unit operation; and
- the failure distribution $\bot : \mathbb{D}(\omega)$ given by $\bot \equiv \mathsf{uniform}(\{\})$ does not assign any probability mass at all.

The failure element $\bot$ aborts the current part of the program, as $\bot$ propagates: $\bot \gg\!= f = \bot$. Typically, we do not use $\bot$ in programs directly. It is more natural to define an assertion statement $\mathsf{assert}\ b \equiv \mathsf{if}\ b\ \mathsf{then}\ \mathsf{return} \circledast\ \mathsf{else}\ \bot$. Assertions are useful in validating the inputs received from the adversary. For example, the assertion in the IND-CPA security game in Fig. 1 checks that the adversary $\mathcal{A}_1$ produced valid plaintexts. Failed assertions are handled using the $\mathsf{try}\ \_\ \mathsf{else}\ \_$ statement. For example, the IND-CPA game treats failures as fair coin flips. This is sound as the advantage is the probability of the outcome True less $1/2$.

As spmfs are a monad, the dependent computation $f : \omega_1 \Rightarrow \mathbb{D}(\omega_2)$ in a sequence $p \gg\!= f$ can use all the control structures (e.g. $\mathsf{if\text{-}then\text{-}else}$ and pattern matching) of HOL. Recall that we can process the result of a probabilistic computation $p : \mathbb{D}(\omega_1)$ with a deterministic function $f : \omega_1 \Rightarrow \omega_2$ to get a new distribution $p \rhd^{\widehat{\mathbb{D}}} f : \mathbb{D}(\omega_2)$, namely $p \rhd^{\widehat{\mathbb{D}}} f \equiv p \gg\!= (\lambda x.\ \mathsf{return}\ f(x))$. For example, $\mathsf{uniform}(\mathbb{Z}_m) \rhd^{\widehat{\mathbb{D}}} (\lambda n.\ n \cdot n)$ uniformly samples a square number between 0 and $(m-1)^2$ inclusive.

4.2. *Recursive Definitions as Least Fixpoints*

Sampling, sequencing, and HOL control structures suffice to write straight-line computations. But cryptographic games also require loops and other forms of repetition. As is usual in functional programming, we capture repetitions by recursive definitions of probabilistic computations.

$\mathsf{bernoulli}(r) \equiv \mathbf{do}\ \{$
$\quad b \leftarrow \mathsf{coin};$
$\quad \mathsf{if}\ b\ \mathsf{then\ return}\ r \geq 1/2$
$\quad \mathsf{else\ if}\ r < 1/2\ \mathsf{then}\ \mathsf{bernoulli}(2 \cdot r)$
$\quad \mathsf{else}\ \mathsf{bernoulli}(2 \cdot r - 1)\ \}$

$\mathsf{geometric}(r) \equiv \mathbf{do}\ \{$
$\quad b \leftarrow \mathsf{bernoulli}(r);$
$\quad \mathsf{if}\ b\ \mathsf{then\ return}\ 0$
$\quad \mathsf{else}\ \mathsf{geometric}(r) \rhd^{\widehat{\mathbb{D}}} (\lambda n.\ n + 1)\ \}$

$\mathsf{unif}(n) \equiv \mathsf{unif\text{-}aux}(n, 0, 1)$

$\mathsf{unif\text{-}aux}(n, c, v) \equiv \mathsf{if}\ v \geq n\ \mathsf{then}$
$\quad \mathsf{if}\ c < n\ \mathsf{then\ return}\ c$
$\quad \mathsf{else}\ \mathsf{unif\text{-}aux}(n, v - n, c - n)$
$\mathsf{else}\ \mathbf{do}\ \{$
$\quad b \leftarrow \mathsf{coin};$
$\quad \mathsf{unif\text{-}aux}(n, 2 \cdot v, 2 \cdot c + (\mathsf{if}\ b\ \mathsf{then}\ 1\ \mathsf{else}\ 0))\ \}$

Fig. 3. The Bernoulli, geometric, and uniform distributions built from fair coin flips.

In CryptHOL, we support unbounded recursion and provide proof rules for probabilistic termination. Strictly speaking, bounded recursion would suffice to model the probabilistic computations that appear in cryptographic constructions, as they have bounded run time. But it is more natural and flexible to allow unbounded recursion. For example, the Bernoulli distribution $\mathsf{bernoulli}(r)$, which returns True with probability r for any $0 \leq r \leq 1$, can be sampled from fair coin flips using the recursive computation shown in Fig. 3 [35]. Similarly, we can implement the uniform distribution over $\mathbb{Z}_n$ for any n using coin flips by adapting Lumbroso's sampling algorithm [55] (unif in Fig. 3). This implementation shows that we could have defined $\mathsf{uniform}$ using coin, because $\mathsf{uniform}(\{ a_0, a_1, \ldots, a_n \}) = \mathsf{unif}(n) \rhd^{\widehat{\mathbb{D}}} (\lambda n.\ a_n)$ holds, and accordingly derived all properties of $\mathsf{uniform}$ from coin's. However, our choice of $\mathsf{uniform}$ simplifies the reasoning because it does not need to explicitly enumerate the elements $(a_i)_{i<n}$ in the set A. Moreover, the geometric distribution can be implemented in terms of the Bernoulli distribution using a recursive computation. Note that none of these computations has a bound on the number of iterations, but they all terminate with probability 1. With the proof rules in our framework, it is easy to show that they all are lossless. Note that we could have included all these distributions as primitives in our language and thereby avoided the need for recursive specifications. However, it is clearly more elegant to add one general primitive (unbounded recursion) to the language and derive the distributions as special cases.

By supporting unbounded recursion, we can also reason about repetitions without having to prove termination. This reduces the overall proof effort, because we do not need to prove that every repetition in every probabilistic computation in every (intermediate) game indeed terminates. Unbounded loops, which terminate only probabilistically, do show up in some cryptographic constructions (see [25] for an overview). For example, some protocols for fair two-party computations [1,29] sample the number of rounds in the protocol from a geometric distribution. Thus, their games contain snippets such as

$$\mathbf{do}\ \{\ n \leftarrow \mathsf{geometric}(r);\ \mathsf{for}\ i = 0\ \mathsf{to}\ n\ \mathsf{do} \ldots\ \}$$

where ... typically contains some abort conditions. Rather than sampling the number of iterations up front and then aborting the for loop prematurely, it can be convenient to fuse the implementation of $\mathsf{geometric}$ with the for loop as in

$$\mathsf{do}\ \{\ \ldots;\ b \leftarrow \mathsf{bernoulli}(r)\ \}\ \mathsf{while}\ (b).$$

This shows that such unbounded loops, which are defined by recursion, are useful.

As is usual in programming languages, we interpret a recursively specified spmf as the least fixpoint of the associated functional. In the case of a spmf, the approximation order $\sqsubseteq$ is given by $p \sqsubseteq q \equiv (\forall x.\ p\,!\,x \leq q\,!\,x)$ [2]. In this order, every chain Y has a supremum $\bigsqcup Y$ which is taken point-wise: $(\bigsqcup Y)\,!\,x = \mathsf{SUP}\ \{\ p\,!\,x.\ p \in Y\ \}$, where $\mathsf{SUP}\ A$ denotes the supremum of a bounded set A of real numbers. Thus, the approximation order is a chain-complete partial order (ccpo) with least element $\bot$. In "Appendix B.1", we prove:

Proposition 1. *The approximation order $\sqsubseteq$ is a chain-complete partial order.*

By the Knaster–Tarski fixpoint theorem, every monotone function f on a ccpo has a least fixpoint $\mathsf{fix}(f)$, which is the least upper bound of the transfinite iteration of f starting at the least element. Therefore, we can define recursive probabilistic computations as the least fixpoint of the associated (monotone) functional.

Isabelle's package for recursive monadic function definitions [40] hides the internal construction using fixpoints from the user and automates the monotonicity proof. For example, Fig. 3 exactly reproduces the Isabelle specifications of these functions. The monotonicity proof succeeds as $(\gg\!=)$ is monotone in both arguments. Namely, if $p \sqsubseteq q$ and $f(x) \sqsubseteq g(x)$ for all x, then $p \gg\!= f \sqsubseteq q \gg\!= g$. In contrast, $\mathsf{try}\ \_\ \mathsf{else}\ \_$ is monotone only in the second argument, but not in the first. For example, $\bot \sqsubseteq \mathsf{return}\ 0$, but $\mathsf{try} \bot\ \mathsf{else}\ \mathsf{return}\ 1 = \mathsf{return}\ 1 \not\sqsubseteq \mathsf{return}\ 0 = \mathsf{try}\ \mathsf{return}\ 0\ \mathsf{else}\ \mathsf{return}\ 1$. Therefore, recursion is always possible through $(\gg\!=)$ and else, but not in general through try.

Audebaud and Paulin-Mohring [2] previously studied least fixpoints in the approximation order $\sqsubseteq$. They showed that all *countable* chains have least upper bounds, and defined the fixpoints as the least upper bound of a countable iteration. Consequently, they demand that the functionals for recursive definitions be order-continuous, which is a stronger property than monotonicity. Thus, our fixpoint combinator can handle more recursive functions (see "Appendix B.1" for an example). Apart from expressivity, our monotonicity requirement is easier to automate than continuity: monotonicity proofs in Isabelle are typically automatic, whereas continuity proofs are not.

4.3. *Equational Reasoning*

Our framework provides three kinds of reasoning techniques for probabilistic programs: equational, relational, and via the semantics. In this and the next section, we describe these techniques and their theory and show how they formally capture the reasoning steps that are commonly found in cryptographic proofs. In this section, we focus on equational reasoning.

Equational reasoning establishes that two programs compute the same subprobability distribution. To this end, our framework comes with a large library of identities about the operators in our language. As our semantics is compositional, we can replace one subprogram by an equal one in any program context.

Isabelle's term rewriting engine *simp* automates this process. Term rewriting orients the given equations from left to right and keeps replacing instances of left-hand sides in

the term with the corresponding right-hand sides until a normal form is reached [3]. So the user must just choose the right set of equations.

In game-hopping proofs, equational reasoning is important for applying indistinguishability assumptions and bridging steps. To apply an indistinguishability assumption in a formal proof, one must show that the two games indeed have the right format. To do this, one constructs a distinguisher explicitly and shows that if the assumption is applied to the distinguisher, one obtains the two games of the hop. Typically, the two games in the proof do not syntactically match this format of the games interacting with a distinguisher. Equational reasoning can be used to bring them into the right shape. In this section, we show how this is done in the security proof of Elgamal in Fig. 2; along the way, we encounter the most important identities in our framework.

In the Elgamal example, we want to apply the DDH assumption to the IND-CPA game for the encryption scheme. That is, we want to consider the IND-CPA game $\mathsf{ind\text{-}cpa}.\mathsf{G}(\mathcal{A})$ as an instance of the DDH game $\mathsf{ddh}.\mathsf{G}_0(\mathcal{A}')$ for some distinguisher $\mathcal{A}'$. Indeed, the probabilistic computation $\mathsf{elgamal\text{-}red}_{\mathcal{A}}$ in lines 3–11, which uses the IND-CPA adversary $\mathcal{A}$ as a black box, provides the instance. In lines 39–40, we prove the two distributions equal. Such an identity could be considered obvious in informal, paper-based reasoning, but in a mechanised proof, we must justify that this identity follows from our equations. With our framework, term rewriting automates this justification: a single call to *simp* suffices.

In the remainder of the proof (lines 16–38), the other game $\mathsf{ddh}.\mathsf{G}_1(\mathsf{elgamal\text{-}red}_{\mathcal{A}})$ of the DDH assumption is analysed. We discuss its features in the following.

Sequencing satisfies the monad laws of (neutrality) and (associativity). These two identities ensure that definition unfolding is transparent. For example, unfolding the Elgamal definition of aenc in the IND-CPA game in Fig. 1 results in nested blocks as shown below on the left. The rewriting engine automatically flattens nested blocks by rewriting with neutrality and associativity, the result of which is shown on the right. (If a name clash between variables occurs, the rewriting engine automatically renames them, which is correct as all variables are local.)

```
try do { ...                                try do { ...
  b ← coin;                                   b ← coin;
  c ← do {
    y ← uniform(Z_|G|)                        y ← uniform(Z_|G|);
    return (g ^ y, (α ^ y) ⊗ m) }
  b' ← A2(c, σ); ... }                        b' ← A2((g ^ y, (α ^ y) ⊗ m), σ); ... }
```

Moreover, the order of two independent computations can be swapped, as $\mathbb{D}$ is a commutative monad. Formally,

$$p \gg\!= (\lambda x.\, q \gg\!= (\lambda y.\, f(x,y))) = q \gg\!= (\lambda y.\, p \gg\!= (\lambda x.\, f(x,y))). \tag{4}$$

Also, if the result of a distribution is not used, then it can be removed:

$$p \gg\!= (\lambda \_.\, q) = \|p\| * q, \tag{5}$$

where $r * p$ scales the subprobability masses of p by r, i.e. $(r * p) \,!\, x = r \cdot (p \,!\, x)$ for $0 \leq r \leq 1/\|p\|$. In particular, if p is lossless, then $p \gg\!= (\lambda\_.\ q) = q$.

Equations (4) and (5) form the semantic foundation for rearranging independent parts and dropping unused parts of a probabilistic program. For example, in Fig. 2, the rewriting engine uses commutativity (4) to move the flipping of the random bit b (lines 21 to 34) past the guess b' of the adversary (lines 23 and 33), and (5) is the cornerstone to collapsing the entire game to a coin flip in lines 37–38. (The collapsing additionally requires some equalities about try-else such as *try-bind-lossless* in line 38, which we do not present in detail here; we refer the interested reader to the source code of the formalisation [49,53].)

It is because of these two equations that we chose to not include state management in the sequencing operations. This is why the game explicitly passes the adversary state σ from line 3 to line 7 in Fig. 1. If we had included it (as, e.g. EasyCrypt and CertiCrypt do), every probabilistic computation could possibly access and modify the state. We would then need additional assumptions and dedicated decision procedures whenever we want to rearrange the order of statements. In particular, term rewriting would no longer suffice to reason about rearrangements and dead code, but in the simplest cases. By leaving the state explicit, our formalisation integrates better with Isabelle's existing reasoning infrastructure and term rewriting in particular.

Similarly, many identities hold for the other language elements. For example, the one-time pad for Booleans is expressed by the equation $\mathsf{coin} \rhd^{\widehat{\mathbb{D}}} (\lambda b.\ b = b') = \mathsf{coin}$. It says that if we flip a coin b and check whether it is equal to some (locally) fixed b', then this is the same as randomly flipping a coin. In line 38 in Fig. 2, e.g. *one-time-pad*-$\mathbb{B}$ refers to this equation; rewriting line 34 with this equation turns the guess of the adversary into dead code, whereby the rewrite engine can use (5) to eliminate it from the game.

The final step of the proof summarises the reasoning so far and shows the equality of the advantages by unfolding their definitions (lines 41–42).

4.4. *Relational Reasoning*

Relational reasoning establishes a relation on the outcomes of two probabilistic computations; namely, one can execute them in a coordinated way such that the relation holds on the outcomes, which is called *coupling* [46]. Accordingly, we can relate the probabilities of events in two coupled computations. Relational reasoning generalises equational reasoning as follows: if we establish the identity relation on outcomes, then the distributions are in fact equal.

Some relational reasoning takes place in the Elgamal proof, although we only use term rewriting in Fig. 2. Line 22 multiplies a random group element with the challenge plaintext, whereas line 32 samples a random group element and omits the multiplication. Their executions shall be coordinated such that they always return the same group element, which is possible because multiplication by a fixed group element—$\mathsf{if}\ b\ \mathsf{then}\ m_0\ \mathsf{else}\ m_1$ in this case—is a bijection on the carrier. So in this case, the relation is the graph of the bijection; see (9) for details. This change is the main step in the security proof as it makes the guess b' of the adversary independent of the challenge bit b.

We will return to this example at the end of the section. Before that, we formalise the reasoning infrastructure for such couplings. We establish a connection to the theory

of relational parametricity [60,69,80]. ("Appendix A.3" summarises the relevant background.) This helps us to automate the checking of the proof steps, similar to how the theory of term rewriting supports equational reasoning. We provide more examples of relational reasoning in the case study in Sect. 6.

We first define an operator to lift relations over elementary events to relations over spmfs. With this operator, our primitive operations are relationally parametric. From parametricity, we derive our logic for reasoning about coupled computations.

Lifting The lifting operation $\widetilde{\mathbb{D}} : \mathbb{P}(\omega_1 \times \omega_2) \Rightarrow \mathbb{P}(\mathbb{D}(\omega_1) \times \mathbb{D}(\omega_2))$ transforms a binary relation R over elementary events into a relation $\widetilde{\mathbb{D}}(R)$ on spmfs over these events. For lossless distributions, a number of definitions have appeared in the literature. We generalise the one from [33]. Formally, $\widetilde{\mathbb{D}}(R)$ relates the spmfs $p : \mathbb{D}(\omega_1)$ and $q : \mathbb{D}(\omega_2)$ iff there is an spmf $w : \mathbb{D}(\omega_1 \times \omega_2)$ such that (i) $\mathsf{support}(w) \subseteq R$, (ii) $w \rhd^{\widehat{\mathbb{D}}} \pi_1 = p$, and (iii) $w \rhd^{\widehat{\mathbb{D}}} \pi_2 = q$. We call w an R-coupling of p and q. This definition reformulates the one by Larsen and Skou [45] for lossless spmfs. They consider w as a non-negative weight function on the relation such that the marginals are the original distribution., i.e. w must satisfy (i) $x \; R \; y$ whenever $w \,!\, (x, y) > 0$, (ii) $\sum_y w \,!\, (x, y) = p \,!\, x$ for all x, and (iii) $\sum_x w \,!\, (x, y) = q \,!\, y$ for all y. Using our language for spmfs, our definition expresses the same conditions more succinctly without summations. In previous work [33], our definition led to considerably shorter proofs about $\widetilde{\mathbb{D}}$, e.g. for distributivity over relation composition.

Recently, Sack and Zhang [70] showed that if p and q are lossless, then

$$p \; \widetilde{\mathbb{D}}(R) \; q \quad \text{iff} \quad \forall A.\, \mathcal{P}[p \in A] \leq \mathcal{P}[q \in R[A]].$$

From this characterisation, which we have formalised too [47], we derive the following characterisation of $\widetilde{\mathbb{D}}$ for arbitrary spmfs. The proof adds a new elementary event None that takes all the unassigned probability mass and applies Sack and Zhang's characterisation to $R \cup \{(\mathsf{None}, \mathsf{None})\}$.

Lemma 1. (Characterisation of $\widetilde{\mathbb{D}}$) *The following are equivalent for all R, p, and q:*

(a) $p \; \widetilde{\mathbb{D}}(R) \; q$;
(b) $\mathit{support}(w) \subseteq R$ *and* $w \rhd^{\widehat{\mathbb{D}}} \pi_1 = p$ *and* $w \rhd^{\widehat{\mathbb{D}}} \pi_2 = q$ *for some* w;
(c) $\mathcal{P}[p \in A] \leq \mathcal{P}[q \in R[A]]$ *for all* A *and* $\|p\| \geq \|q\|$.

The lifting operation $\widetilde{\mathbb{D}}$ enjoys a number of useful properties. For example, (i) it generalises equality, namely $p \; \widetilde{\mathbb{D}}((=)) \; q$ iff $p = q$, (ii) it distributes over relation composition, (iii) it is monotone: $p \; \widetilde{\mathbb{D}}(R) \; q$ implies $p \; \widetilde{\mathbb{D}}(R') \; q$ provided that $x \; R \; y$ implies $x \; R' \; y$ for all $x \in \mathsf{support}(p)$ and $y \in \mathsf{support}(q)$, and (iv) it commutes with converses: $\widetilde{\mathbb{D}}(R^{-1}) = (\widetilde{\mathbb{D}}(R))^{-1}$ where $x \; R^{-1} \; y$ iff $y \; R \; x$.

Most importantly, the characterisation allows us to infer bounds on the probabilities of related events for related computations. In particular, if we can find a relation R such that $R[A] \subseteq B$ and $p \; \widetilde{\mathbb{D}}(R) \; q$, then the probability $\mathcal{P}[p \in A]$ of event A in p is bounded by the probability $\mathcal{P}[q \in B]$ of event B in q. Moreover, let $A \; R^? \; B$ denote that R cannot distinguish the events A and B, i.e. $A \; R^? \; B$ iff $x \in A \leftrightarrow y \in B$ for all $x \; R \; y$. Then

their probabilities are the same for R-coupled spmfs. Formally,

$$\frac{p\ \widetilde{\mathbb{D}}(R)\ q \qquad A\ R^{?}\ B}{\mathcal{P}[p \in A] = \mathcal{P}[q \in B]}. \tag{6}$$

This rule plays an important role in Bellare and Rogaway's identical-until-bad lemma [13]: given two programs which are syntactically equal except for code that is executed after a Boolean flag `bad` has been set, the probability of their outputs being different is bounded by the probability of the flag being set. Lacking syntax, we cannot express their syntactic condition of setting a bad flag in CryptHOL. Borrowing ideas from EasyCrypt [6], we instead rephrase the condition in terms of the lifting for random variables.

Lemma 2. (Identical-until-bad lemma [6,13]) *Let A, F_1 and B, F_2 be events of two spmfs p and q, respectively, such that*

$$p\ \widetilde{\mathbb{D}}(\{\,(a,b) \mid (a \in F_1 \leftrightarrow b \in F_2) \wedge (b \notin F_2 \longrightarrow a \in A \leftrightarrow b \in B)\,\})\ q. \tag{7}$$

Then the probability difference between A occurring in p and B in q is bounded by the probability of F_1 in p, which equals F_2's in q.

$$|\mathcal{P}[p \in A] - \mathcal{P}[q \in B]| \leq \mathcal{P}[p \in F_1] = \mathcal{P}[q \in F_2].$$

The precondition (7) that p and q are coupled as required is typically established using equational reasoning and the relational rules presented later. In comparison, Bellare and Rogaway's version replaces the precondition with a syntactic condition on the two programs. So no proof must establish the precondition; it suffices to look at the code. However, before their rule can be applied, the programs must be transformed to meet the syntactic condition, which itself requires equational and sometimes relational reasoning. Our version does not impose syntactic restrictions on the program. In particular, the bad events F_1 and F_2 need not appear syntactically in the programs p and q.

We next derive proof rules with which we can establish $p\ \widetilde{\mathbb{D}}(R)\ q$ for two probabilistic computations p and q that avoid probabilistic reasoning entirely.

Relational Parametricity Our observation is that the lifting operation $\widetilde{\mathbb{D}}$ also serves as the relator for spmfs in relational parametricity. This insight enables us to leverage the existing theory of relational parametricity ("Appendix A.3") to substantially automate proof checking.

For example, the sequencing operation $(\gg\!=) : \mathbb{D}(\omega_1) \times (\omega_1 \Rightarrow \mathbb{D}(\omega_2)) \Rightarrow \mathbb{D}(\omega_2)$ is parametric in the event spaces ω_1 and ω_2. Formally,

$$\forall R_1\ R_2.\ (\gg\!=)\ (\widetilde{\mathbb{D}}(R_1) \mathrel{\widetilde{\Rightarrow}} (R_1 \mathrel{\widetilde{\Rightarrow}} \widetilde{\mathbb{D}}(R_2)) \mathrel{\widetilde{\Rightarrow}} \widetilde{\mathbb{D}}(R_2))\ (\gg\!=),$$

where the relator $R_1 \mathrel{\widetilde{\times}} R_2$ for pairs lifts the relations R_1 and R_2 component-wise to pairs and the relator $\mathrel{\widetilde{\Rightarrow}}$ for the function space is defined by $f\ (R_1 \mathrel{\widetilde{\Rightarrow}} R_2)\ g$ iff $f(x)\ R_2\ g(y)$ whenever $x\ R_1\ y$. Note the similarity between $(\gg\!=)$'s type and the relation, in which type constructors have been replaced by relators.

Similarly, parametricity of the one-point distribution function $\mathsf{return} : \omega \Rightarrow \mathbb{D}(\omega)$ is expressed by the statement $\forall R.\ \mathsf{return}\ (R \mathrel{\widetilde{\Rightarrow}} \widetilde{\mathbb{D}}(R))\ \mathsf{return}$. We prove this statement for return by unfolding the definitions and taking $\mathsf{return}\ (x, y)$ as the coupling for $\mathsf{return}\ x$ and $\mathsf{return}\ y$ for all $x\ R\ y$. The parametricity proof for sequencing is similar. Also, try _ else _ and $\bot$ are parametric.

Parametricity is compositional because function application and function composition preserve parametricity. Thus, any combination of parametric computations is also parametric. This holds in particular for our derived language elements such as assert and $\rhd^{\widehat{\mathbb{D}}}$.

From the parametricity statements, we derive reasoning rules by unfolding some relator definitions. For example, we get the following rules for the monad operations. Note that parametricity dictates the shape of the rules.

$$\frac{x\ R\ y}{(\mathsf{return}\ x)\ \widetilde{\mathbb{D}}(R)\ (\mathsf{return}\ y)} \qquad \frac{p\ \widetilde{\mathbb{D}}(R)\ q \qquad \forall (x, y) \in R.\ f(x)\ \widetilde{\mathbb{D}}(R')\ g(y)}{(p \gg\!= f)\ \widetilde{\mathbb{D}}(R')\ (q \gg\!= g)}.$$

With such rules, we can prove the existence of an appropriate coupling between two computations without having to construct it explicitly. Conceptually, a proof with these rules implicitly constructs the coupling behind the scenes in a compositional way. Moreover, being able to derive these rules from the types acts as a sanity check for our definitions, roughly similar to what type checking achieves for programming ("Type-safe programs cannot go wrong!" [59]).

Unfortunately, not all computations are fully parametric. For example, uniform sampling $\mathsf{uniform} : \mathbb{P}(\omega) \Rightarrow \mathbb{D}(\omega)$ is not parametric in ω because it relies on polymorphic equality, which is not relationally parametric [80]: the cardinality of a set depends on the equality of elements. Hence,

$$\mathsf{uniform}\ (\widetilde{\mathbb{P}}(R) \mathrel{\widetilde{\Rightarrow}} \widetilde{\mathbb{D}}(R))\ \mathsf{uniform} \tag{8}$$

holds if (and only if) the relation R respects equality, i.e. $(=)\ (R \mathrel{\widetilde{\Rightarrow}} R \mathrel{\widetilde{\Rightarrow}} \widetilde{\mathbb{B}})\ (=)$ holds.[4] Here, the relator $\widetilde{\mathbb{P}}(R)$ for sets relates two sets A and B iff $R[A] \subseteq B$ and $R^{-1}[B] \subseteq A$. It turns out that this conditional parametricity property captures the well-known idea of *optimistic sampling* in cryptographic proofs. Namely, if f is injective on A, then

$$\mathsf{uniform}(A) \rhd^{\widehat{\mathbb{D}}} f = \mathsf{uniform}(A \rhd^{\widehat{\mathbb{P}}} f) \tag{9}$$

To see this, choose the graph $G_f = \{\, (x, f(x)) \mid x.\ \mathsf{True} \,\}$ of f for R in (8) and note that lifting the graph of a function is the same as post-processing: $p\ \widetilde{\mathbb{D}}(G_f)\ q$ iff $q = p \rhd^{\widehat{\mathbb{D}}} f$, and $A\ \widetilde{\mathbb{P}}(G_f)\ B$ iff $B = A \rhd \widehat{\mathbb{P}} f$. Injectivity is equivalent to G_f respecting equality.

This is one example of Wadler's free theorems [80] in our context. If we specialise A to bitstrings of a fixed length and f to the bitwise exclusive-or (xor) with a fixed

[4] This restriction on uniform's parametricity stems from our avoiding the enumeration of the set elements (cf. Sect. 4.2). If we had defined uniform on enumerations, i.e. lists rather than sets, then uniform would be parametric. But we would then incur the cost of reasoning about lists, where the order of the elements matters, instead of sets, which are unordered.

bitstring, we obtain the well-known one-time-pad lemma:

$$\mathsf{uniform}(\{0,1\}^n) \rhd^{\widehat{\mathbb{D}}} (\lambda s'.\ s \oplus s') = \mathsf{uniform}(\{0,1\}^n), \tag{10}$$

where s is a bitstring of length n and $\{0,1\}^n$ denotes the set of all bitstrings of length n.

Finally, we now return to the security proof of Elgamal in Fig. 2. It is optimistic sampling that justifies the change from line 22 to line 32. Formally, the reasoning goes as follows (where c is an arbitrary group element):

$$\begin{aligned}\mathsf{uniform}(\mathbb{Z}_{|\mathcal{G}|}) \rhd^{\widehat{\mathbb{D}}} (\lambda z.\ \mathsf{g} \mathbin{\hat{}} z \otimes c) &= \mathsf{uniform}(\mathbb{Z}_{|\mathcal{G}|} \rhd^{\widehat{\mathbb{P}}} (\lambda z.\ \mathsf{g} \mathbin{\hat{}} z \otimes c)) \\ &= \mathsf{uniform}(\mathbb{Z}_{|\mathcal{G}|} \rhd^{\widehat{\mathbb{P}}} (\lambda z.\ \mathsf{g} \mathbin{\hat{}} z)) \\ &= \mathsf{uniform}(\mathbb{Z}_{|\mathcal{G}|}) \rhd^{\widehat{\mathbb{D}}} (\lambda z.\ \mathsf{g} \mathbin{\hat{}} z),\end{aligned}$$

where the first and third steps hold by (9) and the second follows from both sets being the same. Our lemma *one-time-pad-group* captures this reasoning as a reusable rewrite rule with the condition that c is in $\mathcal{G}$'s carrier.[5] Admittedly, this security proof did not involve much relational reasoning. The reason is that we used only the identity relation on elementary events, whereby relational reasoning reduces to equational reasoning. Relational reasoning will become important when oracles enter the stage, for example, for reasoning up to failure events (see Proposition 2 and Sect. 6).

In summary, guided by parametricity, we have essentially rediscovered the functional counterpart to probabilistic relational Hoare logic as implemented in EasyCrypt. But parametricity offers yet another point of view. Mitchell [60] uses parametricity to express representation independence, i.e. one can change the representation of data without affecting the overall result. In Sect. 6, we will exploit representation independence in the bridging steps of the game transformations. With proper configuration, users can write concise proof scripts for the parametricity and representation independence proofs using the existing proof automation in Isabelle [34,44].

Finally, note that parametricity does not stop at probabilistic computations. Observe that (6) merely rephrases $\mathcal{P}[\_ \in \_]$'s parametricity statement

$$\forall R.\ \mathcal{P}[\_ \in \_]\ (\widetilde{\mathbb{D}}(R) \mathrel{\widetilde{\Rightarrow}} R^{?} \mathrel{\widetilde{\Rightarrow}} \widetilde{\mathbb{R}})\ \mathcal{P}[\_ \in \_],$$

where $\widetilde{\mathbb{R}}$ denotes the identity relation on the reals (type $\mathbb{R}$). Thus, parametricity can take us beyond reasoning about probabilistic elements in our language.

Parametricity of Recursively Defined Subdistributions We have not yet covered one building block of our probabilistic language in our parametricity analysis: the fixpoint combinator. It turns out that it preserves parametricity.

[5] This proof step also highlights the use of assertions. To apply the rewrite rule, the rewrite engine must be able to discharge the assumption that $\mathsf{if}\ b\ \mathsf{then}\ m_0\ \mathsf{else}\ m_1$ is indeed a group element. The assertion in line 20 makes precisely this information available because valid-plains checks that both messages are group elements.

Theorem 1. (Parametricity of spmf fixpoints) *If* $f : \mathbb{D}(\omega_1) \Rightarrow \mathbb{D}(\omega_1)$ *and* $g : \mathbb{D}(\omega_2) \Rightarrow \mathbb{D}(\omega_2)$ *are monotone with respect to* $\sqsubseteq$ *and* $f\ (\widetilde{\mathbb{D}}(R) \mathrel{\widetilde{\Rightarrow}} \widetilde{\mathbb{D}}(R))\ g$, *then* $\mathit{fix}(f)\ \widetilde{\mathbb{D}}(R)\ \mathit{fix}(g)$.

The proof is given in "Appendix B.2". Analogues to Theorem 1 hold for fixpoints over $\_ \Rightarrow \mathbb{D}(\omega)$, $\mathbb{D}(\omega_1) \times \mathbb{D}(\omega_2)$, etc.. We use them to show parametricity of (mutually) recursive probabilistic computations (Sect. 5.5) rather than that of distributions.

4.5. *Reasoning Via the Semantics*

The relational proof rules are not complete. For example, we cannot establish

$$\mathsf{uniform}(\{0,1,2\})\ \widetilde{\mathbb{D}}(\{(x,y) \mid x = y \vee x + 1 = y\})\ \mathsf{uniform}(\{0,1,2,3\}) \quad (11)$$

using the proof rules derived from parametricity, in particular (8). Intuitively, the reason is that parametricity looks only at the types, not at the definitions of functions. Hence, the proof rules derived from parametricity cannot be used for arguments that non-trivially involve the semantics.

In such cases, we directly employ the semantic definitions and derive new proof rules or conduct proofs by unfolding these definitions. For example, we derived the following rule from the definition of uniform:

$$\frac{\forall X \subseteq A.\ |B| \cdot |X| \leq |A| \cdot |B \cap R[X]|}{\mathsf{uniform}(A)\ \widetilde{\mathbb{D}}(R)\ \mathsf{uniform}(B)} \quad \text{if } A \text{ and } B \text{ are non-empty and finite.}$$

This rule is strictly stronger than (8) whenever the side condition holds. For example, we use it to establish (11): let $A = \{0,1,2\}$ and $B = \{0,1,2,3\}$ and $R = \{(x,y) \mid x = y \vee x + 1 = y\}$ and $X \subseteq A$. Then, $|B \cap R[X]| \geq |X| + 1$. So, $|B| \cdot |X| = 4 \cdot |X| \leq 3 \cdot (|X| + 1) \leq |A| \cdot |B \cap R[X]|$ holds since $|X| \leq 3$.

Deriving new rules is possible because our formalisation is embedded in a generic-purpose logic like HOL. If we were using a dedicated tool like EasyCrypt, we would have to change its implementation whenever we need a more general rule.

5. Games with Oracles

In many security games, oracles control how the adversary can access information or use cryptographic primitives. For example, in the random oracle model, the adversary can inspect a random function by evaluating it at points of his choosing, but he cannot analyse a representation of the function itself (Sect. 5.1). Similarly, a decryption oracle allows the adversary to decrypt any ciphertext different from the challenge ciphertext, but it prevents the adversary from getting hold of the decryption key (Sect. 6.1). In general, an oracle is a probabilistic function that maintains mutable state across different invocations, but the adversary must not access this state. In other words, the adversary has only black-box access to the oracle.

In this section, we formalise black-box access in higher-order logic. We propose a new semantic domain for probabilistic input–output systems, which we call generative probabilistic values (GPV). We use GPVs in two ways. First, to model adversaries with oracle access (Sect. 5.2): to access an oracle, a GPV can output a query and wait for a response before it continues. Second, we build oracle converters from GPVs to handle how reductions replace oracles, and we define the composition of GPVs, converters, and oracles (Sect. 5.3). We then derive high-level proof principles from the definitions (Sects. 5.4–5.6). Finally, we present operators to modify and impose restrictions on adversaries and we generalise our framework to multiple oracles (Sects. 5.7–5.9). Overall, we obtain a natural formalisation of black-box access, which relies heavily on the higher-order features of our logic.

5.1. *Example: Pseudo-Random Functions*

To illustrate how we model black-box access, we formalise the notion of a pseudo-random function. Informally, a family of functions $(\mathsf{F}_s)_{s\in S}$ from A to B, indexed by a seed $s \in S$, is called *pseudo-random* if—given black-box access—it is hard to distinguish a random function from A to B from a random member of the family.

Formally, we define two games (Fig. 4). In the game G_{RF} on the left, the adversary $\mathcal{A}$ gets black-box access to a random oracle $\mathcal{O}_{\mathsf{RF}}$, which models the random function. In the game $\mathsf{G}_{\mathsf{PRF}}$ on the right, $\mathcal{A}$ interacts with a randomly chosen element of the family. The advantage

$$\mathsf{adv}(\mathcal{A}) \equiv \left|\mathcal{P}\big[\mathsf{G}_{\mathsf{RF}}(\mathcal{A}) = \mathsf{True}\big] - \mathcal{P}\big[\mathsf{G}_{\mathsf{PRF}}(\mathcal{A}) = \mathsf{True}\big]\right|$$

measures the adversary's ability to distinguish whether he interacted with the random oracle or with the pseudo-random function.

The random oracle $\mathcal{O}_{\mathsf{RF}}$ stores all queries and their responses in its state, a map $D : \alpha \Rightarrow \mathbb{M}(\beta)$. When the adversary queries the image of a point x that has not previously been queried (case $D(x) = \mathsf{None}$), the oracle samples a new point y according to the distribution rnd,[6] returns y, and updates the map. Otherwise (case $D(x) = \mathsf{Some}(y)$), the oracle returns the stored value y and leaves the state unchanged. In contrast, the oracle $\mathcal{O}^s_{\mathsf{PRF}}$ merely evaluates the chosen element $F(s)$ of the family F at the point x. As the index s remains unchanged during all queries, we pass s as an initialisation parameter to the oracle instead of storing it in the oracle's state. Hence, the state degenerates to the singleton type, whose only element is $\circledast$.

The oracles must not be passed as arguments to the adversary. Otherwise, the adversary would receive an encoding of the oracle and its state, which he could analyse. This would be white-box access. To ensure black-box access, we define a composition operator $\mathsf{exec}(\mathcal{O}, \mathcal{A}, s)$ in Sect. 5.3. It runs the adversary $\mathcal{A}$ together with the oracle $\mathcal{O}$ from the oracle's initial state s and returns the response of the adversary (a bit b in Fig. 4) and the final state of the oracle. For the random function, G_{RF} runs $\mathcal{A}$ with the random oracle with the empty map $\varnothing$ as initial state. For the pseudo-random function, $\mathsf{G}_{\mathsf{PRF}}$ picks an

[6] The distribution rnd abstracts the distribution on the codomain. Typically, rnd is uniform, but our formulation also supports any other discrete distribution.

$$\mathcal{O}_{\mathsf{RF}}(D, x) \equiv \mathsf{case}\ D(x)\ \mathsf{of}$$
$$\quad \mathsf{None} \Rightarrow \mathbf{do}\ \{\ y \leftarrow \mathsf{rnd};\ \mathsf{return}\ (y, D(x \mapsto y))\ \}$$
$$\quad \mid \mathsf{Some}(r) \Rightarrow \mathsf{return}\ (r, D)$$

$$\mathsf{G}_{\mathsf{RF}}(\mathcal{A}) \equiv \mathbf{do}\ \{$$
$$\quad (b, \_) \leftarrow \mathsf{exec}(\mathcal{O}_{\mathsf{RF}}, \mathcal{A}, \varnothing);$$
$$\quad \mathsf{return}\ b\ \}$$

$$\mathcal{O}^{s}_{\mathsf{PRF}}(\circledast, x) \equiv \mathsf{return}\ (\mathsf{F}(s, x), \circledast)$$

$$\mathsf{G}_{\mathsf{PRF}}(\mathcal{A}) \equiv\ \mathbf{do}\ \{$$
$$\quad s \leftarrow \mathsf{seed\text{-}gen};$$
$$\quad (b, \_) \leftarrow \mathsf{exec}(\mathcal{O}^{s}_{\mathsf{PRF}}, \mathcal{A}, \circledast);$$
$$\quad \mathsf{return}\ b\ \}$$

Fig. 4. Games for formalising pseudo-randomness of F, defined in the module PRF.

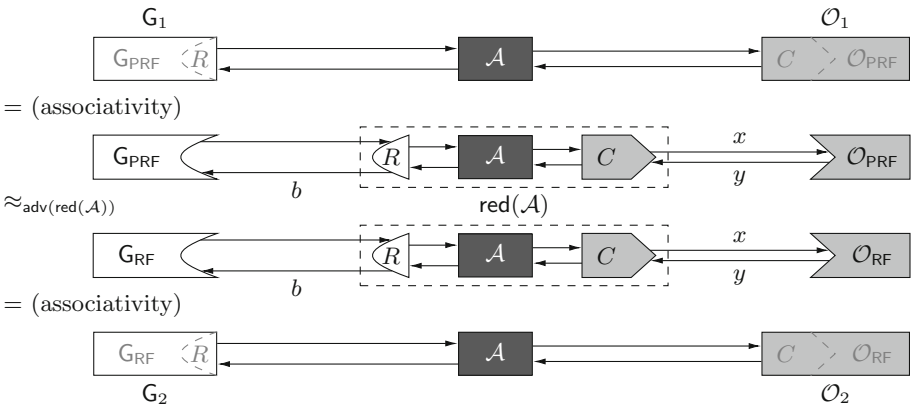


Fig. 5. Game hop from a game G_1–$\mathcal{A}$–$\mathcal{O}_1$ to the game G_2–$\mathcal{A}$–$\mathcal{O}_2$ using the indistinguishability of the pseudo-random function from a random oracle.

index s from the family according to the distribution seed-gen and then runs $\mathcal{A}$ with access to $\mathsf{F}(s)$. Both games return $\mathcal{A}$'s response and discard the final oracle state.

We next illustrate how we apply the indistinguishability assumption between the PRF and a random oracle in a game hop (Fig. 5). Let G_1 be a game that gives an adversary $\mathcal{A}$ access to some oracle $\mathcal{O}_1$ that both use the PRF. We want to obtain a game G_2 that uses the random oracle instead of the PRF. The formal justification of such a transition proceeds in four steps. First, we break the game G_1 into two parts: the game $\mathsf{G}_{\mathsf{PRF}}$ and the remainder R. Similarly, we split the oracle $\mathcal{O}_1$ into $\mathcal{O}_{\mathsf{PRF}}$ and the rest C. Second, we define the reduction $\mathsf{red}(\mathcal{A})$ that runs R (which internally runs $\mathcal{A}$) and answers $\mathcal{A}$'s queries by calling C and querying the oracle $\mathcal{O}_{\mathsf{PRF}}$. We prove that $\mathsf{G}_{\mathsf{PRF}}(\mathsf{red}(\mathcal{A}))$ yields the same distribution as $\mathsf{G}_1(\mathcal{A})$ by associativity of composition. Conceptually, C converts the oracle $\mathcal{O}_{\mathsf{PRF}}$ into $\mathcal{A}$'s oracle $\mathcal{O}_1$. Third, we switch to the game $\mathsf{G}_{\mathsf{RF}}(\mathsf{red}(\mathcal{A}))$, which differs by $\mathsf{red}(\mathcal{A})$'s PRF advantage from $\mathsf{G}_{\mathsf{PRF}}(\mathsf{red}(\mathcal{A}))$. Finally, we apply associativity of composition again and obtain the new oracle $\mathcal{O}_2$ as the composition of C and $\mathcal{O}_{\mathsf{RF}}$ and the new game G_2 built from G_{RF} and R.

In the remainder of this section, we formalise interaction and composition and derive the reasoning principles used. We thus develop the formal counterpart to the pictorial reduction proofs that appear in the literature [75]. In Sect. 6, we show our framework in action by proving an encryption scheme IND-CCA-secure. We thereby express reduction proofs similar to Fig. 5 formally in HOL.

5.2. *Probabilistic Computations with Oracle Access*

We explicitly model the interactions between the adversary and the oracle using resumptions [58], which we combine with subprobabilities in the style of Piróg and Gibbons [66]. The type constructors $\mathbb{G}$ and $\mathfrak{D}$ represent generative and reactive probabilistic values, respectively. They satisfy the following recursion equations[7]:

$$\mathbb{G}(\alpha, \gamma, \rho) \cong \mathbb{D}(\alpha + \gamma \times \mathfrak{D}(\alpha, \gamma, \rho)) \qquad \mathfrak{D}(A, Q, R) \cong R \Rightarrow \mathbb{G}(A, Q, R) \quad (12)$$

Formally, we define $\mathbb{G}$ and $\mathfrak{D}$ as the greatest solution to these equations, namely the algebraic coinductive datatype (codatatype)

codatatype $\mathbb{G}(\alpha, \gamma, \rho) \equiv \mathsf{GPV}\ (\mathsf{un\text{-}GPV} : \mathbb{D}(\alpha + \gamma \times \mathfrak{D}(\alpha, \gamma, \rho)))$
type-synonym $\mathfrak{D}(\alpha, \gamma, \rho) \equiv \rho \Rightarrow \mathbb{G}(\alpha, \gamma, \rho)$

where the bijections $\mathsf{GPV} : \mathbb{D}(\alpha + \gamma \times \mathfrak{D}(\alpha, \gamma, \rho)) \Rightarrow \mathbb{G}(\alpha, \gamma, \rho)$ and $\mathsf{un\text{-}GPV} : \mathbb{G}(\alpha, \gamma, \rho) \Rightarrow \mathbb{D}(\alpha + \gamma \times \mathfrak{D}(\alpha, \gamma, \rho))$ witness the isomorphism of the recursion equation. ("Appendix A.2" provides some background on coinductive definitions.)

Conceptually, each GPV chooses probabilistically between failing, terminating with a result of type α, and continuing by producing a query $q : \gamma$ and transitioning into a reactive probabilistic value (RPV). The RPV waits for a response $r : \rho$ from the environment and then moves to the generative successor state.

We choose the greatest solution to (12) because it allows us to model more adversary behaviours than any other solution. For example, the type $\mathbb{G}$ includes adversaries which terminate only probabilistically. That is, they terminate with probability one (and possibly even within polynomially many steps on average), but if we ignore probabilities and consider them as a non-deterministic system, then there are executions that do not terminate. In distributed computing, such executions are typically considered unfair [30]. Such adversaries naturally arise from reductions in security proofs of protocols that terminate only probabilistically, e.g. for fair two-party computations [1,29].

Computations on GPVs are written in a monadic language similarly to spmfs. The basic operations for GPVs are the monadic functions return and $(\gg\!=)$, calling an oracle call, sampling sample, exceptional termination fail (from which we derive assertions assert similar to Sect. 4.1), and failure handling $\mathsf{try}\ \_\ \mathsf{else}\ \_$. They are implemented as shown in Fig. 6, where $\mathsf{Pure} \equiv \mathsf{Left}$ and $\mathsf{IO}(c, r) \equiv \mathsf{Right}((c, r))$ and id is the identity. For clarity, the return operations are annotated with the monad they operate in. Note that $(\gg\!=)$ and $\mathsf{try}\ \_\ \mathsf{else}\ \_$ are well defined because the corecursive calls on the right-hand sides occur in guarded and friendly contexts [19]. Note further that try behaves slightly differently on spmfs and on GPVs. In the spmf monad, it catches failed assertions and non-termination. In the GPV monad, try catches failed assertions and divergence, i.e. an internal non-terminating computation without queries. It does not, however, prevent a GPV from performing infinitely many queries, i.e. observable non-termination.

GPVs can be visualised as possibly infinite trees of nestings of probability distributions. For example, consider the following process p, which interacts with an environment that takes a number and returns a bit. Before each query, it flips a fair coin. If the

[7] HOL has isorecursive types in the form of algebraic (co)datatypes, but does not support equirecursive types. So, we only demand isomorphism ($\cong$), not equality.

$$
\begin{array}{ll}
\mathsf{return}_{\mathbb{G}}\ x & \equiv \mathsf{GPV}\ (\mathsf{return}_{\mathbb{D}}\ \mathsf{Pure}(x)) \\
\mathsf{call}\ q & \equiv \mathsf{GPV}\ (\mathsf{return}_{\mathbb{D}}\ \mathsf{IO}(q, \mathsf{return})) \\
\mathsf{sample}(p) & \equiv \mathsf{GPV}\ (p \rhd^{\widehat{\mathbb{D}}} \mathsf{Pure}) \\
\mathsf{fail} & \equiv \mathsf{GPV}\ \bot \\
\mathsf{assert}\ b & \equiv \mathsf{if}\ b\ \mathsf{then}\ \mathsf{return}_{\mathbb{G}}\ \circledast\ \mathsf{else}\ \mathsf{fail} \\
\mathsf{try}\ v\ \mathsf{else}\ v' & \equiv \\
\quad \mathsf{GPV}\ (\mathsf{try}\ (\mathsf{un\text{-}GPV}(v) \rhd^{\widehat{\mathbb{D}}} (\mathsf{id} \mathbin{\widehat{+}} (\mathsf{id} \mathbin{\widehat{\times}} (\lambda r\ x.\ \mathsf{try}\ r(x)\ \mathsf{else}\ v')))) \ \mathsf{else}\ \mathsf{un\text{-}GPV}(v'))
\end{array}
$$

$$
\begin{array}{l}
v \gg\!= f \equiv \mathsf{GPV}\ (\mathbf{do}\ \{ \\
\quad x \leftarrow \mathsf{un\text{-}GPV}(v); \\
\quad \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Pure}(y) \Rightarrow \mathsf{un\text{-}GPV}(f(y)) \\
\quad \mid \mathsf{IO}(c, r) \Rightarrow \\
\qquad \mathsf{return}_{\mathbb{D}}\ \mathsf{IO}(c, \lambda w.\ r(w) \gg\!= f)\ \})
\end{array}
$$

Fig. 6. Primitive operations for GPVs.

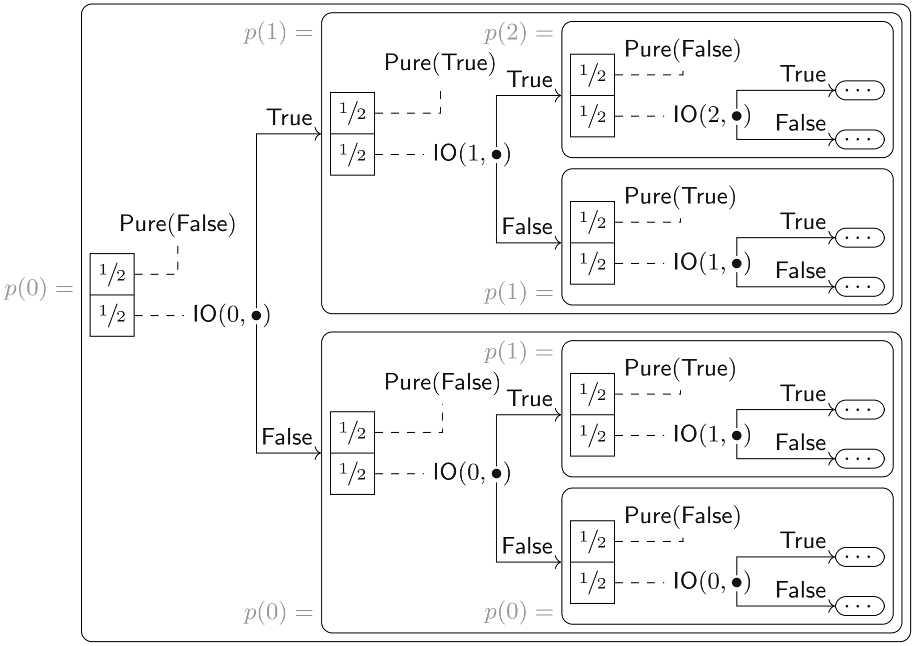


Fig. 7. Visualisation of GPVs as nested boxes of subprobability distributions.

result is heads, then it terminates and returns the parity of all the bits received so far. If it is tails, then it asks another query that is labelled by the number of True responses received so far. Figure 7 visualises this process as follows: A GPV is a box with rounded corners, which contains a subprobability distribution. The distribution is written as a list of probabilities, which add up to at most one. Dashed lines connect the probabilities to the elementary events they are associated to. The distributions inside a GPV contain two kinds of elementary events. First, Pure(x) values denote that the process terminates with result x. Second, IO(q, f) corresponds to a query q, whose response r determines the subtree $f(r)$. In Fig. 7, the function f is written as a $\bullet$ with arrows to the subtrees that are labelled with the response.

This process can also be expressed syntactically as follows, where the state s stores the number of True responses so far and odd(n) returns whether n is odd.

$p(s) =$ **do** {
  $b \leftarrow$ sample coin;
  if b then return (odd(s)) else **do** { $b' \leftarrow$ call s; p(if b' then $s + 1$ else s) } }

In fact, all GPV operations can be interpreted graphically. A box models the constructor GPV, which embeds a subprobability distribution into the type of GPVs. Conversely, the destructor un-GPV unboxes a GPV and returns the contained distribution. Figure 8 depicts four simple operations: (a) return(x) packages the one-point distribution on Pure(x) in a box; (b) fail's box is "empty" in that its subprobability distribution does not assign any mass to any event; (c) sample takes a distribution p, which assigns

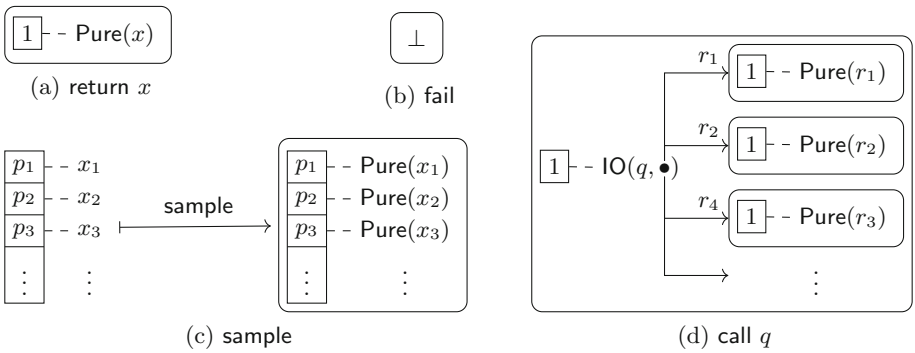


Fig. 8. Visualisation of the primitive GPV operations return, fail, sample, and call.

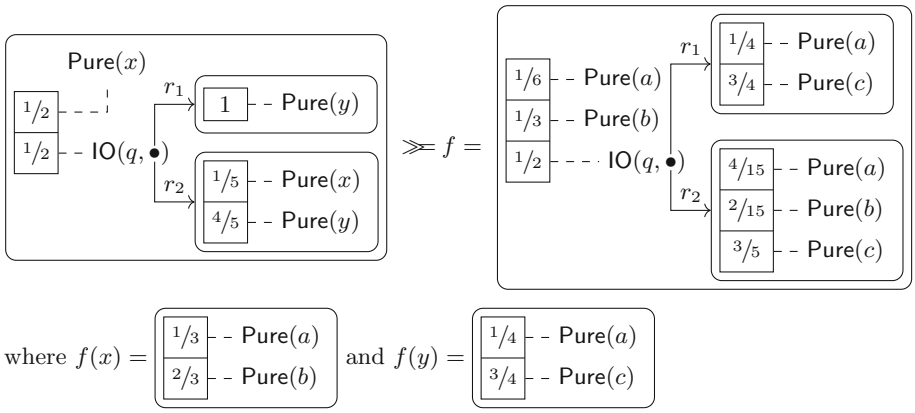


Fig. 9. Visualisation of a sequencing example for GPVs.

probability p_i to x_i, marks all elementary events x_i with Pure, and boxes the resulting distribution as a GPV; and (d) call(q) performs the query q and terminates with the response r_i of the environment.

The sequencing operation $v \gg\!= f$ takes a GPV v and a family f of GPVs indexed by v's possible results. It replaces all occurrences of Pure(x) in any box inside v with the contents of the box $f(v)$ scaled to the probability of Pure(x). Figure 9 shows an example. The probability $1/2$ for Pure(x) on the left is distributed to Pure(a) and Pure(b) on the right according to $f(x)$. When several branches yield identical subtrees, these probabilities are merged, e.g. $4/15 = 1/5 \cdot 1/3 + 4/5 \cdot 1/4$ for Pure(a).

Error handling try v else v' fills all unassigned probability masses in any of v's subdistributions according to the contents of v'. Figure 10 shows an example. As the error handler in the else part contains non-lossless subprobability distributions, so does the result of the error handling. As with sequencing, the probabilities of identical subtrees are merged (e.g. $7/12 = 1/2 + (1 - (1/2 + 1/3)) \cdot 1/2$).

We emphasise that these definitions define only the denotational semantics of the GPV operations. Operationally, these possibly infinite trees are never constructed explicitly

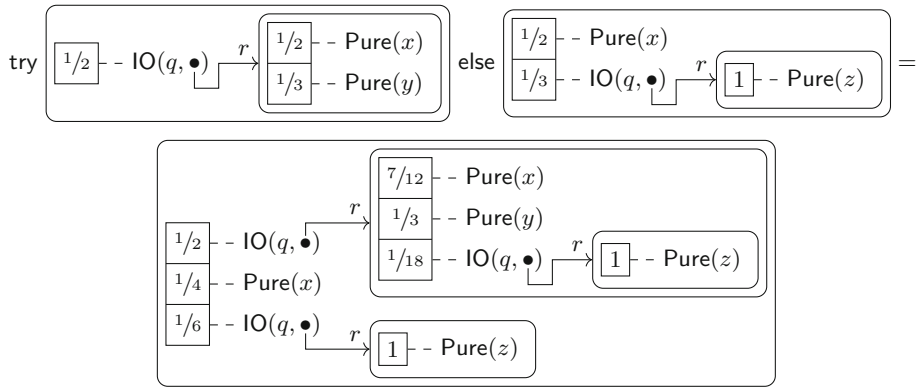


Fig. 10. Visualisation of an error-handling example for GPVs.

and probabilities are need not be merged because execution relies on the syntactic description whereas the reasoning happens at the semantic level (see Sect. 8 for more details).

The operations on GPVs behave as expected. In particular, return and $(\gg\!=)$ satisfy the monad laws, fail propagates, and sample is a monad homomorphism.

$$\begin{gathered}
\mathsf{return}\ x \gg\!= f = f(x) \qquad v \gg\!= (\lambda x.\ \mathsf{return}\ x) = v \\
(v \gg\!= f) \gg\!= g = v \gg\!= (\lambda x.\ f(x) \gg\!= g) \qquad \mathsf{fail} \gg\!= f = \mathsf{fail} \\
\mathsf{sample}(\mathbf{do}\ \{\ x \leftarrow p;\ f(x)\ \}) = \mathbf{do}\ \{\ x \leftarrow \mathsf{sample}(p);\ \mathsf{sample}(f(x))\ \} \qquad (13) \\
\mathsf{sample}(\mathsf{return}\ x) = \mathsf{return}\ x \qquad \mathsf{sample}(\bot) = \mathsf{fail} \qquad \mathsf{sample}(\mathsf{assert}\ b) = \mathsf{assert}\ b \\
\mathsf{sample}(\mathsf{try}\ p\ \mathsf{else}\ q) = \mathsf{try}\ \mathsf{sample}(p)\ \mathsf{else}\ \mathsf{sample}(q).
\end{gathered}$$

Not all properties carry over from the spmf monad. For example, commutativity (4) and cancellation (5) do not hold for $(\gg\!=)$ in general. Therefore, most of our reasoning happens when we compose GPVs with oracles because composition takes the reasoning back to spmfs. We define composition next.

5.3. *Oracle Converters and Composition*

A GPV models a probabilistic computation that interacts with an environment, i.e. the oracle. In this respect, GPVs resemble the models of probabilistic reactive systems from the literature [33,72,76]. When we look at a GPV in isolation, then we consider the oracle as non-deterministic. After we have composed a GPV with an oracle, the nondeterminism has been resolved, so only an spmf remains.

In this section, we formally define composition for GPVs, oracles, and converters. Our definitions are quite technical, but we derive high-level proof rules for reasoning about compositions in Sects. 5.4 and 5.5. Thanks to the proof rules, users of our framework need not understand the technical details, as they only need to use the proof rules. Only when they want to derive additional rules from the semantics must they understand the underlying definitions.

Formally, an oracle is a stateful environment of type $\mathbb{O}(\sigma, \gamma, \rho) \equiv \sigma \Rightarrow \gamma \Rightarrow \mathbb{D}(\rho \times \sigma)$: for each query $q : \gamma$ and local state $s : \sigma$, the oracle probabilistically produces a response $r : \rho$ and updates its local state.

As discussed in Sect. 5.1 and illustrated in Fig. 5, the adversary $\mathsf{red}(\mathcal{A})$ constructed in a reduction proof intercepts the queries of the original adversary $\mathcal{A}$, forwards some of them to its oracles and answers others himself. Thus, the reduction must convert between its oracles and the oracles of the original adversary using a converter C, as the reduction uses the adversary as a black box. In our framework, stateful converters have type $\mathbb{C}(\sigma, \gamma, \rho, \gamma', \rho') \equiv \sigma \Rightarrow \gamma \Rightarrow \mathbb{G}(\rho \times \sigma, \gamma', \rho')$. They intercept queries γ and produce responses ρ for a GPV. In doing so, they interact with an environment (oracle) through queries γ' and responses ρ'. The converter maintains a local state σ, which persists between intercepts.

We first consider composition with converters as composition with an oracle is a special case. Syntactically, composition corresponds to inlining the converter into the adversary. Semantically, we define an operator $\mathsf{inline} : \mathbb{C}(\sigma, \gamma, \rho, \gamma', \rho') \Rightarrow \mathbb{G}(\alpha, \gamma, \rho) \Rightarrow \sigma \Rightarrow \mathbb{G}(\alpha \times \sigma, \gamma', \rho')$. It takes a converter C, a GPV, and the initial state of the converter and returns a GPV that produces a result α and C's new state after interacting with C's environment.

Inlining combines recursion and corecursion as follows: The recursive part goes through the interactions between the GPV and the converter and searches for the next interaction between the converter and its oracles. The corecursive part performs the interaction found and iterates the search.

The search is captured by the recursive function search. If the GPV v terminates with result x, there are no queries and the search terminates; otherwise, the GPV outputs q and becomes the RPV v'. In that case, search analyses C under the query q. If C returns r without issuing a query, the search continues recursively on $v'(r)$. Otherwise, the first query is found and the search terminates.

$$\mathsf{search} : \mathbb{C}(\sigma, \gamma, \rho, \gamma', \rho') \Rightarrow \mathbb{G}(\alpha, \gamma, \rho) \Rightarrow \sigma \\ \Rightarrow \mathbb{D}(\alpha \times \sigma + \gamma' \times \mathfrak{D}(\rho \times \sigma, \gamma', \rho') \times \mathfrak{D}(\alpha, \gamma, \rho))$$

$$\begin{array}{l} \mathsf{search}(C, v, s) \equiv \mathbf{do}\ \{ \\ \quad z \leftarrow \mathsf{un\text{-}GPV}(v); \\ \quad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{Pure}\ x \Rightarrow \mathsf{return}\ \mathsf{Left}((x, s)) \\ \quad \mid \mathsf{IO}(q, v') \Rightarrow \mathbf{do}\ \{ \\ \qquad y \leftarrow \mathsf{un\text{-}GPV}(C(s, q)); \\ \qquad \mathsf{case}\ y\ \mathsf{of}\ \mathsf{Pure}\ (r, s') \Rightarrow \mathsf{search}(C, v'(r), s') \\ \qquad \mid \mathsf{IO}(q', v'') \Rightarrow \mathsf{return}\ \mathsf{Right}((q', v'', v'))\ \}\ \}. \end{array}$$

The function inline first calls the auxiliary function search, which searches for the first query issued by the converter to its oracle during a query of the caller. If search finds none, it returns the caller's response r and the converter's updated state s'. Then, inline terminates with the same outcome. Otherwise, inline issues the call q' and forwards the response r' to the RPV v'' of the converter, which may issue further calls. The result r of the converter is then fed to the RPV v' of the caller and inline corecurses with the updated state s' of the converter.

$$\mathsf{inline} : \mathbb{C}(\sigma, \gamma, \rho, \gamma', \rho') \Rightarrow \mathbb{G}(\alpha, \gamma, \rho) \Rightarrow \sigma \Rightarrow \mathbb{G}(\alpha \times \sigma, \gamma', \rho')$$

$$\begin{aligned}&\mathsf{inline}(C, v, s) \equiv \mathsf{GPV}\ (\mathbf{do}\ \{\\ &\quad z \leftarrow \mathsf{search}(C, v, s);\\ &\quad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{Left}((a, s')) \Rightarrow \mathsf{return}\ \mathsf{Pure}((a, s'))\\ &\quad \mid \mathsf{Right}((q', v'', v')) \Rightarrow \mathsf{return}\ \mathsf{IO}(q', \lambda r'.\ \mathbf{do}\ \{\\ &\qquad (r, s') \leftarrow v''(r');\\ &\qquad \mathsf{inline}(C, v'(r), s')\})\ \}).\end{aligned}$$

As search lives in the spmf monad, it can be defined using the fixpoint operator on spmf (Sect. 4.2). Conversely, inline operates in the GPV monad. So, corecursion is the appropriate definition principle. Accordingly, we prove properties about search by fixpoint induction and about inline by coinduction (Sects. 5.4 and 5.5).

If we compose a GPV with an oracle $\mathcal{O}$ instead of a converter, i.e. an spmf rather than a GPV, the oracle cannot issue further calls. Thus, $\mathsf{search}(\mathcal{O})$ always returns a result of the form $\mathsf{Left}(x, s')$ and the corecursion in inline is not needed. Therefore, we define the execution of a GPV v with $\mathcal{O}$ as follows (where projl is the left inverse to Left).

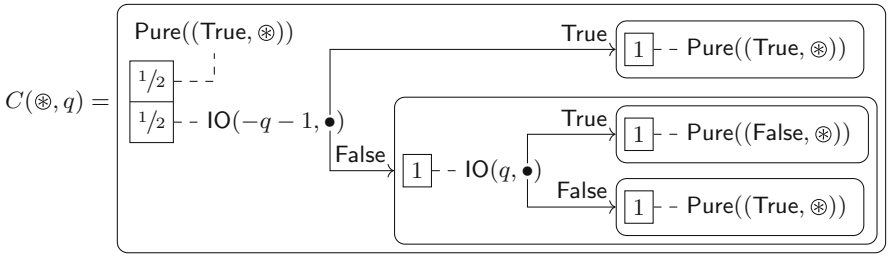
$$\begin{aligned}&\mathsf{exec} : \mathbb{O}(\sigma, \gamma, \rho) \Rightarrow \mathbb{G}(\alpha, \gamma, \rho) \Rightarrow \sigma \Rightarrow \mathbb{D}(\alpha \times \sigma)\\ &\mathsf{exec}(\mathcal{O}, v, s) \equiv \mathsf{search}(\lambda(s, x).\ \mathsf{sample}(\mathcal{O}(s, x)), v, s) \rhd^{\widehat{\mathbb{D}}} \mathsf{projl}.\end{aligned}$$

Composition can also be interpreted graphically. Figure 11a shows a converter C that intercepts integer queries q. This converter does not need any state, so the state $\circledast$ is just of the singleton type $\mathbb{1}$. With probability $1/2$, it immediately responds with True, and with probability $1/2$, it queries its own oracle for $-q-1$. In the latter case, if the oracle's response is True, C also responds with True; otherwise, the converter next queries its oracle with q and returns the negated response. In the following, we compose the GPV $p(0)$ as shown in Fig. 7 with C.

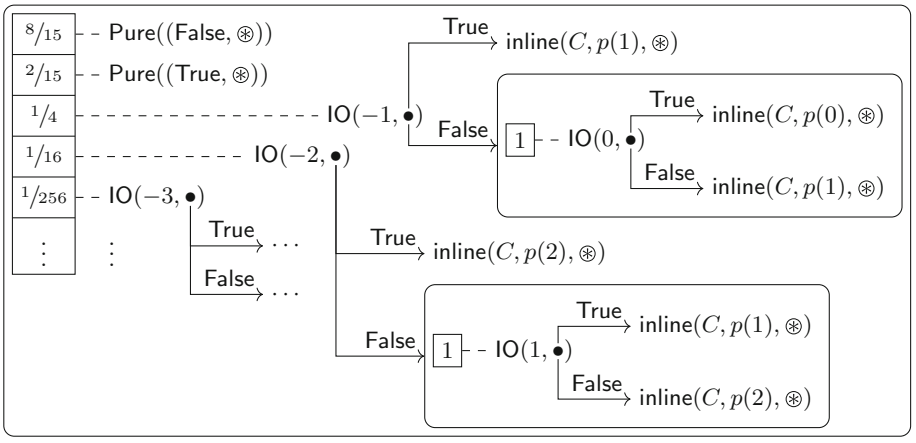
Graphically, composition first replaces each $\mathsf{IO}(q, \bullet)$ in the outer-most box $p(0)$ with the contents of $C(\circledast, q)$ and replaces every $\mathsf{Pure}((r, \circledast))$ in this copy of $C(\circledast, q)$ with $\mathsf{IO}(q, \bullet)$'s subtree with edge label r. Then, the same replacement happens recursively in all these subtrees again.[8] Such replacements create direct nestings of probability distributions (without IO in between), which are then flattened by scaling the probabilities accordingly. The replacement and the flattening up to the first occurrence of $\mathsf{IO}(\_, \bullet)$ in the converter tree is done by the function search, and inline continues the replacement and flattening after the converter has terminated.

The composed GPV is shown in Fig. 11b. The result $\mathsf{Pure}(\mathsf{False})$ of the outer-most box in Fig. 7 is still possible, but it is now formally combined with C's final state $\circledast$. The original $\mathsf{IO}(0, \bullet)$ of the outer-most box has been replaced with $C(\circledast, 0)$'s box contents and the resulting probabilities are merged into the outer-most probability distribution. As $C(\circledast, 0)$ queries its own oracle with -1 with probability $1/2$, $\mathsf{IO}(-1, \bullet)$ has probability $1/2 \cdot 1/2 = 1/4$. Its subtrees are the remainder of $C(\circledast, 0)$ with the Pure subtrees replaced by $p(1)$ and $p(0)$, where inlining continues. But $C(\circledast, 0)$ can also immediately respond with True without performing any queries. Therefore, the True subtree of $\mathsf{IO}(0, \bullet)$ in

[8] In general, composition also takes care of correctly passing the state of the converter, which is not necessary in this simple example.



(a) An example of a state-less converter



(b) Composition of the GPV from Figure 7 with the converter C

Fig. 11. Composition example.

Fig. 7, i.e. $p(1)$, also gets included in the outer-most probability distribution in Fig. 11b. This is the reason why Pure((True, ⊛)) has positive probability. For the same reason, the first query to C's oracle may originate from the second call to C in the upper $p(1)$ box in Fig. 7. Hence, IO(−2, •) has probability $1/16 = 1/2 \cdot 1/2 \cdot 1/2 \cdot 1/2$, as all probabilistic choices are independent. Similarly, Pure((False, ⊛)) is also a possible response in case C immediately responds twice in a row. Therefore, Pure((False, ⊛)) has not probability $1/2$, but $1/2 \cdot \sum_{i=0}^{\infty} (1/2 \cdot 1/2 \cdot 1/2 \cdot 1/2)^i = 8/15$. Analogously, Pure((True, ⊛)) has probability $2/15 = 1/8 \cdot \sum_{i=0}^{\infty} 1/16^i$.

5.4. *Equational Reasoning about Composition*

Cryptographic reasoning primarily happens after the GPV has been composed with oracles, i.e. in the spmf monad with the reasoning principles from Sects. 4.3, 4.4, and 4.5. Therefore, the important principles for reasoning about GPVs concern the composition operators inline and exec.

As with spmf, equational reasoning plays a central role in bringing (intermediate) games into the right syntactic shape for a hardness assumption or a property to apply,

like in the first and last steps in Fig. 5. We have already mentioned in (13) that the usual structural identities hold for GPVs. We now show that the composition operators interact nicely with the rest of the language too. For inline, e.g. the following identities hold (the ones for exec are analogous):

$$\mathsf{inline}(C, \mathsf{call}\ q, s) = C(s, q) \qquad \mathsf{inline}(C, \mathsf{return}\ x, s) = \mathsf{return}\ (x, s)$$
$$\mathsf{inline}(C, v \gg\!= f, s) = \mathsf{inline}(C, v, s) \gg\!= (\lambda(x, s').\ \mathsf{inline}(C, f(x), s'))$$
$$\mathsf{inline}(C, \mathsf{fail}, s) = \mathsf{fail} \qquad \mathsf{inline}(C, \mathsf{sample}(p), s) = \mathsf{sample}(p \rhd^{\widehat{\mathbb{D}}} (\lambda x.\ (x, s)))$$

The reader might wonder why the definition for inline appears so complicated when these equations would have been enough to define inline recursively over GPV programs. This is because our definition relies solely on the semantics of GPV programs, not their syntax. This gives us a strong notion of extensibility: any extension that we or users of our framework add to our language will not interfere with these equations unless the extension requires changes to the semantic domain of GPVs itself. In Sect. 5.7, we will make use of this extensibility.

The most important property for cryptographic proofs is the associativity of composition. As shown in Fig. 5, reductions transform the adversary $\mathcal{A}$ for one game into an adversary for another game, where the oracles of the two games in general differ. So, the reduction emulates the original oracle $\mathcal{O}$ using a converter C, which has access to the new oracle $\mathcal{O}'$. In this way, the new adversary is built from the composition $\mathsf{inline}(C, \mathcal{A})$ and the game in the hardness assumption executes it with access to $\mathcal{O}'$. By the associativity of composition, this is equivalent to executing the original adversary $\mathcal{A}$ with access to the emulated oracle $\mathsf{exec}(\mathcal{O}', C(\_), \_)$. Thus, it suffices to establish that the emulation $\mathsf{exec}(\mathcal{O}', C(\_), \_)$ of $\mathcal{O}$ is good enough, i.e. indistinguishable from $\mathcal{O}$. Formally, the following equations hold

$$\mathsf{inline}(C_1, \mathsf{inline}(C_2, v, s_2), s_1) = \mathsf{inline}(\mathsf{inline}(C_1) \circ\!\circ\, C_2, v, (s_2, s_1)), \tag{14}$$
$$\mathsf{exec}(\mathcal{O}, \mathsf{inline}(C, v, s_2), s_1) = \mathsf{exec}(\mathsf{exec}(\mathcal{O}) \circ\!\circ\, C, v, (s_2, s_1)), \tag{15}$$

where $f \circ\!\circ\, g$ abbreviates $\lambda(x, y)\ z.\ f(g(x, z), y)$ and where we have omitted reassociations of tuples, identifying (x, y, z) with $((x, y), z)$.

Thus, we have captured a fundamental reasoning principle in cryptographic proofs [56] in two simple equations. This supports our thesis that GPVs are at the right level of abstraction for the semantic domain of interaction: it is not too abstract because we can model composition, and it does not contain unnecessary details as otherwise associativity could not be expressed as an equality.

5.5. *Relational Reasoning*

Reasoning up to bisimilarity shows up in many bridging steps of game-hopping proofs. For example, if we change an oracle to keep track of additional state or compute quantities in equivalent ways, it suffices to prove bisimilarity of the old and new oracle (see Sect. 6 for concrete examples). In our model, bisimilarity can be expressed using

relational parametricity. Thus, the theory of representation independence provides the formal underpinning to conclude that the overall results of the game do not change.

Our relational reasoning, which we derive from parametricity, follows the same line as for spmfs. First, we define the canonical relator $\widetilde{\mathbb{G}}$ for GPVs as follows. Given relations A, Q, and R on results, queries, and responses, respectively, $\widetilde{\mathbb{G}}(A, Q, R)$ is the largest relation on GPVs that is consistent with the rule

$$\frac{\mathsf{un\text{-}GPV}(v)\ \widetilde{\mathbb{D}}((A \mathbin{\widetilde{+}} Q) \mathbin{\widetilde{\times}} (R \mathbin{\widetilde{\Rightarrow}} \widetilde{\mathbb{G}}(A, Q, R)))\ \mathsf{un\text{-}GPV}(v')}{v\ \widetilde{\mathbb{G}}(A, Q, R)\ v'}.$$

The double horizontal line indicates that this definition should be interpreted coinductively, i.e. as the largest relation consistent with the rule. As we have chosen the largest solution to (12) for GPVs, we also choose the largest solution for the relator $\widetilde{\mathbb{G}}$. Note that $\widetilde{\mathbb{G}}$ generalises equality, since $v\ \widetilde{\mathbb{G}}((=), (=), (=))\ v'$ iff $v = v'$.

With this relator, all our operators are relationally parametric: since parametricity is compositional, it suffices to note that the operators are all defined in terms of parametric functions and that the definition principles such as primitive corecursion and least fixpoints on spmf preserve parametricity (Theorem 1). For example, the following holds for all relations A, Q, Q', R, R', and S:

$$\begin{gathered}\mathsf{sample}\ (\widetilde{\mathbb{D}}(A) \mathbin{\widetilde{\Rightarrow}} \widetilde{\mathbb{G}}(A, Q, R))\ \mathsf{sample} \\ \mathsf{inline}\ ((S \mathbin{\widetilde{\Rightarrow}} Q \mathbin{\widetilde{\Rightarrow}} \widetilde{\mathbb{G}}(R \mathbin{\widetilde{\times}} S, Q', R')) \mathbin{\widetilde{\Rightarrow}} \widetilde{\mathbb{G}}(A, Q, R) \mathbin{\widetilde{\Rightarrow}} S \mathbin{\widetilde{\Rightarrow}} \widetilde{\mathbb{G}}(A \mathbin{\widetilde{\times}} S, Q', R'))\ \mathsf{inline} \\ \mathsf{exec}\ ((S \mathbin{\widetilde{\Rightarrow}} Q \mathbin{\widetilde{\Rightarrow}} \widetilde{\mathbb{D}}(R \mathbin{\widetilde{\times}} S)) \mathbin{\widetilde{\Rightarrow}} \widetilde{\mathbb{G}}(A, Q, R) \mathbin{\widetilde{\Rightarrow}} S \mathbin{\widetilde{\Rightarrow}} \widetilde{\mathbb{D}}(A \mathbin{\widetilde{\times}} S))\ \mathsf{exec}.\end{gathered} \tag{16}$$

Taking the representation independence point of view on parametricity, we obtain a bisimulation principle from (16): if S is a bisimulation relation between the two oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ and S relates their initial states s_1 and s_2, then an adversary returns the same result when run with both oracles and the resulting oracle states are related again. Formally,

$$\frac{\forall (s_1, s_2) \in S.\ \forall c.\ \mathcal{O}_1(s_1, c)\ \widetilde{\mathbb{D}}((=) \mathbin{\widetilde{\times}} S)\ \mathcal{O}_2(s_2, c) \qquad (s_1, s_2) \in S}{\mathsf{exec}(\mathcal{O}_1, \mathcal{A}, s_1)\ \widetilde{\mathbb{D}}((=) \mathbin{\widetilde{\times}} S)\ \mathsf{exec}(\mathcal{O}_2, \mathcal{A}, s_2)} \tag{17}$$

follows from (16) by choosing $(=)$ for A, Q, and R.

Bisimulation reasoning does not suffice if a failure event occurs during an oracle call because, in such a case, the oracles simply are not bisimilar. Therefore, we provide a different proof rule for reasoning up to failure events. Reasoning up to failure events relies on properties of the composition operators that cannot be encoded into the HOL type of the function. Consequently, we cannot base such reasoning on relational parametricity.

We need two definitions first. We call a GPV *terminating* iff it cannot engage in an infinite interaction with the environment, i.e. with its oracles.[9] A GPV v is *lossless* iff

[9] The terminating GPVs are the least solution to (12), i.e. they form an algebraic *inductive* datatype. Hence, we can prove statements about terminating GPVs by induction. This form of termination is stronger than

all the subprobability distributions in v are lossless; formally,

$$\frac{\mathsf{lossless}(\mathsf{un\text{-}GPV}(v)) \qquad \forall \mathsf{IO}(q, v') \in \mathsf{support}(\mathsf{un\text{-}GPV}(v)).\ \forall r.\ \mathsf{lossless}_{\mathbb{G}}(v'(r))}{\mathsf{lossless}_{\mathbb{G}}(v)}.$$

If a terminating adversary is lossless and we compose it with a lossless oracle, then the resulting subdistribution is again lossless, i.e. executing the adversary with the oracle always yields a result. This property is crucial for our proof rule:

Proposition 2. (Oracle bisimulation up to failure) *Let S be a relation between the states of two oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ and let F_1 and F_2 be predicates on states of the oracles $\mathcal{O}_1$ and $\mathcal{O}_2$, respectively. Define the relation R by*

$$R \equiv \{\, ((x, s_1), (y, s_2)) \mid F_1(s_1) = F_2(s_2) \wedge (s_1, s_2) \in S \wedge (\neg F_2(s_2) \longrightarrow x = y) \,\}.$$

Then $\mathsf{exec}(\mathcal{O}_1, \mathcal{A}, s_1)\ \widetilde{\mathbb{D}}(R)\ \mathsf{exec}(\mathcal{O}_2, \mathcal{A}, s_2)$ *if*

(i) $\mathcal{O}_1(s_1, q)\ \widetilde{\mathbb{D}}(R)\ \mathcal{O}_2(s_2, q)$ *for all q and $(s_1, s_2) \in S$ such that $F_1(s_1) = F_2(s_2) =$ False, and*

(ii) $(s_1, s_2) \in S$ *and $F_1(s_1) = F_2(s_2)$, and*

(iii) $\mathcal{O}_1(s_1, q)$ *and $\mathcal{O}_2(s_2, q)$ are lossless for all q and $(s_1, s_2) \in S$ such that $F_1(s_1) = F_2(s_2) =$ True and all possible successor states s_1' and s_2' in their supports satisfy $F_1(s_1') = F_2(s_2') =$ True and $(s_1', s_2') \in S$.*

(iv) $\mathcal{A}$ *is terminating and lossless.*

The predicates F_1 and F_2 indicate whether a failure event has happened in the oracle $\mathcal{O}_1$ and $\mathcal{O}_2$, respectively. Condition (i) expresses that $\mathcal{O}_1$ and $\mathcal{O}_2$ are bisimilar unless a failure event occurs and failure events occur in both oracles at the same time and (ii) states that they start in bisimilar states. The last two conditions ensure that a failure event persists.

Proof. We explicitly construct the coupling of $\mathsf{exec}(\mathcal{O}_1, \mathcal{A}, s_1)$ and $\mathsf{exec}(\mathcal{O}_1, \mathcal{A}, s_1)$ as follows: By (i), there is an R-coupling $\mathcal{O}_{12}((s_1, s_2), q)$ of $\mathcal{O}_1(s_1, q)$ and $\mathcal{O}_2(s_2, q)$ for all q and $(s_1, s_2) \in S$ with $F_1(s_1) = F_2(s_2) = \mathsf{False}$. First, the coupling runs $\mathcal{A}$ with $\mathcal{O}_{12}$ until the failure event happens. When the failure happens, the coupling continues by running the remainder of $\mathcal{A}$ independently with the two oracles $\mathcal{O}_1$ and $\mathcal{O}_2$. Losslessness and termination of $\mathcal{A}$ and condition (iii) ensure that these independent executions are lossless. Hence, the projections of the coupling to either side can drop the other independent execution (by using (5) on page 19 with scaling factor 1). □

Losslessness and termination of $\mathcal{A}$ in condition (iv) are not just technical side conditions needed to make the proof go through, they are essential (probabilistic termination

Footnote 9 continued
probabilistic termination, which we have also formalised in our framework using weakest pre-expectations. For example, the GPV in Fig. 7 terminates only probabilistically. In this article, we present only non-probabilistic termination. We have formalised most of the statements (about terminating adversaries) for probabilistically terminating adversaries as well. The CryptHOL sources [49] contain the details.

suffices). Otherwise, some of the unassigned probability mass in $\mathsf{exec}(\mathcal{O}_i, \mathcal{A}, s_i)$ may actually correspond to the failure event. Interrupting the execution when the failure event occurs is not an option either: since converters in reductions typically cannot identify the failure event themselves (as they would have to know certain secrets to do so; Fig. 15 in Sect. 6.4 presents an example), they do not know when to abort the interaction. Hence, reasoning up to failures in oracles is in general sound only if the adversary is lossless and terminating.

5.6. *Reasoning Via the Semantics*

Like in the spmf case (Sect. 4.5), the relational reasoning rules are incomplete. For example, it is an open problem [7] how relational reasoning can be used to hoist probabilistic computations out of loops such as exec and inline. In such cases (Sect. 6.5 mentions an example), we derive appropriate proof rules directly from the operator's definitions. In the case of exec and inline, the derivation is typically justified by fixpoint induction and coinduction.

5.7. *Interruptible Adversaries*

Sometimes, a reduction must stop the interactions with the given adversary $\mathcal{A}$ at a certain point. Consider, for example, a reduction that guesses in the beginning at which interaction $\mathcal{A}$ will violate a hardness assumption—say, $\mathcal{A}$ correctly predicts an unpredictable function (Sect. 6.2)—and the reduction then stops the simulation of $\mathcal{A}$ at this query. The composition operator inline, however, runs the simulation until the adversary finishes and produces a result. We could emulate the early termination by having the converter, which processes the queries, change its behaviour: since it cannot stop the simulation right away, it would answer all further queries with dummy values until the adversary terminates. This approach, while possible, would be inconvenient to reason about.

Instead, we exploit our language's extensibility and define by primitive recursion a new operation $\mathsf{interruptible} : \mathbb{G}(\alpha, \gamma, \rho) \Rightarrow \mathbb{G}(\mathbb{M}(\alpha), \gamma, \mathbb{M}(\rho))$ that makes an adversary interruptible:

$$\mathsf{interruptible}(v) \equiv \mathsf{GPV}\ (\mathsf{un\text{-}GPV}(v) \rhd^{\widehat{\mathbb{D}}} (\mathsf{Some} \mathbin{\widehat{+}} \mathsf{id} \mathbin{\widehat{\times}} (\lambda v'.\ \lambda r.\ \mathsf{case}\ r\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{return}\ \mathsf{None} \mid \mathsf{Left}(r') \Rightarrow \mathsf{interruptible}(v'(r'))))).$$

Thus, if a converter C wants to stop interacting with the black-box adversary $\mathcal{A}$, it returns None as a response. Then, $\mathsf{inline}(C, \mathsf{interruptible}(\mathcal{A}), s)$ stops immediately with the result None and the current state of the converter. The same works for an oracle $\mathcal{O}$ instead of a converter: $\mathsf{exec}(\mathcal{O}, \mathsf{interruptible}(\mathcal{A}), s)$ stops immediately with result None and the state of the oracle when $\mathcal{O}$ responds with None. Conversely, if C or $\mathcal{O}$ never request a stop, the execution continues until $\mathcal{A}$ finishes with a result a and then returns $\mathsf{Some}(a)$ together with the state of the converter or oracle.

5.8. *Operators for Oracles and Converters*

So far, the adversary can access at most *one* oracle. However, many cryptographic constructions involve multiple oracles. For example, in the game for symmetric-key IND-CCA security (Sect. 6.1), the adversary has access to a decryption oracle and an encryption oracle, which encrypts one of the two challenge plaintexts. Similarly, a security protocol may use encryptions, signatures, and hashes simultaneously, all of which have their own oracles. In this section, we show that with a bit of functional programming, we cover all these settings with the operations already presented. To this end, we introduce three operators on oracles.

First, suppose we have two oracles $\mathcal{O}_1 : \mathbb{O}(\sigma, \gamma_1, \rho_1)$ and $\mathcal{O}_2 : \mathbb{O}(\sigma, \gamma_2, \rho_2)$, which operate on the same state of type σ, but accept different types of queries (γ_1 and γ_2) with different types of responses (ρ_1 and ρ_2), like the IND-CCA encryption and decryption oracles. As both oracles share the state, we can view them as a single oracle which accepts both kinds of queries. We formalise this change of view with a composition operator $+_{\mathbb{O}}$ for oracles that acts as a query dispatcher: those queries tagged with Left go to $\mathcal{O}_1$ and those tagged with Right go to $\mathcal{O}_2$. Formally,

$$
\begin{aligned}
&(+_{\mathbb{O}}) : \mathbb{O}(\sigma, \gamma_1, \rho_1) \Rightarrow \mathbb{O}(\sigma, \gamma_2, \rho_2) \Rightarrow \mathbb{O}(\sigma, \gamma_1 + \gamma_2, \rho_1 + \rho_2)\\
&(\mathcal{O}_1 +_{\mathbb{O}} \mathcal{O}_2)(s, \mathsf{Left}(q_1)) \equiv \mathcal{O}_1(s, q_1) \rhd^{\widehat{\mathbb{D}}} (\mathsf{Left} \mathbin{\widehat{\times}} \mathsf{id})\\
&(\mathcal{O}_1 +_{\mathbb{O}} \mathcal{O}_2)(s, \mathsf{Right}(q_2)) \equiv \mathcal{O}_2(s, q_2) \rhd^{\widehat{\mathbb{D}}} (\mathsf{Right} \mathbin{\widehat{\times}} \mathsf{id}).
\end{aligned}
$$

Clearly, we can iterate composition and thus combine finitely many oracles to $\mathcal{O}_1 +_{\mathbb{O}} \mathcal{O}_2 +_{\mathbb{O}} \ldots +_{\mathbb{O}} \mathcal{O}_n$, where $+_{\mathbb{O}}$ associates to the right.

Second, suppose that the adversary requires access to oracles of different cryptographic primitives, say encryptions and signatures. Naturally, the oracles of one primitive operate on a state that is disjoint from the oracle state of the other primitive. Similar to $+_{\mathbb{O}}$, this situation can be expressed by a parallel composition operator $||$ that separates the state of the two oracles.

$$
\begin{aligned}
&(||) : \mathbb{O}(\sigma_1, \gamma_1, \rho_1) \Rightarrow \mathbb{O}(\sigma_2, \gamma_2, \rho_2) \Rightarrow \mathbb{O}(\sigma_1 \times \sigma_2, \gamma_1 + \gamma_2, \rho_1 + \rho_2)\\
&(\mathcal{O}_1 \,||\, \mathcal{O}_2)((s_1, s_2), \mathsf{Left}(q_1)) \equiv \mathcal{O}_1(s_1, q_1) \rhd^{\widehat{\mathbb{D}}} (\mathsf{Left} \mathbin{\widehat{\times}} (\lambda s_1'.\ (s_1', s_2)))\\
&(\mathcal{O}_1 \,||\, \mathcal{O}_2)((s_1, s_2), \mathsf{Right}(q_2)) \equiv \mathcal{O}_2(s_1, q_2) \rhd^{\widehat{\mathbb{D}}} (\mathsf{Right} \mathbin{\widehat{\times}} (\lambda s_2'.\ (s_1, s_2')))
\end{aligned}
$$

Similarly, our framework provides operators for families of oracles. For example, they transform oracles for a single-user setting into an oracle for a multi-user setting as used in [10]. Moreover, each of these operators comes with a variant for interruptible adversaries, which we do not distinguish in this paper.

The third operator on oracles deals with state extensions.[10] Many bridging steps require adding additional state information to the oracle. This can be keeping a record of the previous queries or simply a bad flag to indicate whether a failure event has happened. Since this kind of transformation is so common, it is convenient to introduce an operator $\lambda : \mathbb{O}(\sigma, \gamma, \rho) \Rightarrow \mathbb{O}(\sigma' \times \sigma, \gamma, \rho)$ given by $\lambda\mathcal{O}((s^*, s), q) \equiv \mathcal{O}(s, q) \rhd^{\widehat{\mathbb{D}}} (\mathsf{id} \mathbin{\widehat{\times}} (\lambda s'.\ (s^*, s')))$. This operator lets an oracle ignore the new component of the state. It

[10] The modelling of state in our framework is flexible. Here, we use tuples of fields for simplicity, but this does not scale to hundreds of fields. More advanced models are possible; see Schirmer and Wenzel for an overview [71].

is most useful in combination with $+_{\mathbb{O}}$. For example, consider an IND-CCA game with oracles for encryption $\mathcal{O}_{\mathsf{enc}}$ and decryption $\mathcal{O}_{\mathsf{dec}}$ and suppose that we want to add a bad flag to the decryption oracle, producing a new oracle $\mathcal{O}'_{\mathsf{dec}}$. Since the encryption oracle does not care about the bad flag, the new oracle combination is simply $\lambda\mathcal{O}_{\mathsf{enc}} +_{\mathbb{O}} \mathcal{O}'_{\mathsf{dec}}$.

All our operators are relationally parametric. Hence, we can derive reasoning rules for them from relational parametricity in the same way as for our language primitives. Moreover, the proof automation for representation independence can also handle them smoothly.

Analogous operations are available for converters instead of oracles, e.g. $+_{\mathbb{C}}$.

5.9. *Upper Bounds on Interactions*

Bounds on the advantage in the concrete security setting typically depend on the number of queries an adversary can make. For example, in the RP/RF switching lemma, the chance to distinguish a random permutation (RP) from a random function increases quadratically with the number of queries the adversary can ask [12]. In this section, we formalise the concepts needed to bound the number of interactions between the adversary and the oracles.

As we model adversaries using GPVs (instead of, say, a for loop that repeatedly asks the adversary for its next query), we express query bounds as an assumption on the adversary using the HOL function $\mathsf{qbound}_P : \mathbb{G}(\alpha, \gamma, \rho) \Rightarrow \mathbb{N}^\infty$, where $\mathbb{N}^\infty$ denotes the natural numbers extended with infinity. qbound_P returns the least upper bound on the number of queries that the GPV makes and that satisfy the predicate $P : \gamma \Rightarrow \mathbb{B}$. This predicate is useful to classify the adversary's queries. For example, if the adversary $\mathcal{A}$ has access to two oracles $\mathcal{O}_1 +_{\mathbb{O}} \mathcal{O}_2$, $\mathsf{qbound}_{\mathsf{is\text{-}Left}}(\mathcal{A})$ denotes the number of calls made to $\mathcal{O}_1$. We omit P when we want to consider a bound on all calls, i.e. when $P = (\lambda\_.\ \mathsf{True})$. Formally, we define qbound recursively:

$$\begin{aligned}&\mathsf{qbound}_P(v) \equiv \mathsf{SUP}_{x \in \mathsf{support}(\text{un-GPV}(v))}\\ &\quad(\mathsf{case}\ x\ \mathsf{of}\ \mathsf{Pure}(y) \Rightarrow 0\\ &\qquad \mid \mathsf{IO}(c, r) \Rightarrow (\mathsf{if}\ P(c)\ \mathsf{then}\ 1\ \mathsf{else}\ 0) + \mathsf{SUP}_w\ \mathsf{qbound}_P(r(w))).\end{aligned}$$

We use the natural numbers extended with infinity $\mathbb{N}^\infty$ instead of the plain natural numbers as qbound's return type because $\mathbb{N}^\infty$ is a complete lattice, i.e. suprema of infinite sets exist and we can define qbound by recursion.

The choice of $\mathbb{N}^\infty$ also yields simple rules for reasoning about qbound because we need not prove that the suprema exist. In detail, the following identities hold:

$$\begin{gathered}\mathsf{qbound}_P(\mathsf{return}\ x) = 0 \qquad \mathsf{qbound}_P(\mathsf{fail}) = 0 \qquad \mathsf{qbound}_P(\mathsf{sample}\ p) = 0\\ \mathsf{qbound}_P(\mathsf{call}(x)) = (\mathsf{if}\ P(x)\ \mathsf{then}\ 1\ \mathsf{else}\ 0)\\ \mathsf{qbound}_P(\mathsf{interruptible}(v)) = \mathsf{qbound}_P(v)\\ \mathsf{qbound}_P(v \gg\!= f) = \mathsf{qbound}_P(v) + \mathsf{SUP}_{x \in \mathsf{results}(v)}\ \mathsf{qbound}_P(f(x)),\end{gathered}$$

where $\mathsf{results}(v)$ denotes the set of possible values that v can return after interaction with an oracle. There is no simple equation for inline, but we proved the following upper bound:

$$\frac{\mathsf{qbound}_{P'}(v) \le p \qquad \forall s\ q.\ \mathsf{qbound}_P(C(s,x)) \le (\mathsf{if}\ P'(x)\ \mathsf{then}\ q\ \mathsf{else}\ 0)}{\mathsf{qbound}_P(\mathsf{inline}(C,v,s)) \le p \cdot q}.$$

The bound on the oracle calls is typically needed to bound the probability of a failure event F. If we can bound the probability of an event F occurring during one oracle call by k, then we can bound the probability of F anywhere in the game by $k \cdot n$, where n is a bound on the oracle calls in which F may occur. In the RP/RF switching lemma, (a slight generalisation of) the following lemma is the key to obtain the bound on the advantage, which is quadratic in the number of interactions.

Lemma 3. *If* $\mathsf{qbound}_P(v) \le n$, *and* F *is an event on the state space of an oracle* $\mathcal{O}$ *such that* $s_0 \notin F$, *and* $\mathcal{P}\left[\mathcal{O}(s,q) \rhd^{\widehat{\mathbb{D}}} \pi_2 \in F\right] \le k$ *for all states* $s \notin F$ *and all queries* q, *then* $\mathcal{P}\left[\mathsf{exec}(\mathcal{O}, v, s_0) \rhd^{\widehat{\mathbb{D}}} \pi_2 \in F\right] \le k \cdot n$.

6. Case Study: Proving IND-CCA Security

Having presented the main formalisation and reasoning principles of our framework, we now put them to work in a case study. We formalise a symmetric-key encryption scheme taken from Shoup's tutorial [74] (Sect. 6.3), which is built from a pseudo-random function (Sect. 5.1) and an unpredictable function (Sect. 6.2), and prove it IND-CCA secure (Sects. 6.1, 6.4–6.5). In detail, we prove the following statements where the formal definitions of advantages and reductions will be given in the remainder of this section:

Theorem 2. (Concrete IND-CCA security) *Let* $(\mathsf{F}_s)_{s \in S}$ *be pseudo-random with domain* Dom *and* $(\mathsf{H}_k)_{k \in K}$ *be unpredictable and* $\mathcal{A}$ *be a lossless IND-CCA adversary that makes at most* q *encryption queries and* q' *decryption queries. Then,*

$$\mathsf{ind\text{-}cca.adv}(\mathcal{A}) \le \mathsf{prf.adv}(\mathsf{red}_{\mathsf{prf}}(\mathcal{A})) + q' \cdot \mathsf{uf.adv}(\mathsf{red}^{q'}_{\mathsf{uf}}(\mathcal{A})) + \frac{q^2}{|\mathsf{Dom}|}.$$

Corollary 1. (Asymptotic IND-CCA security) *Let* $\mathcal{A}$ *be a lossless IND-CCA adversary that makes at most polynomially many encryption and decryption queries. Suppose that* $(\mathsf{F}_s)_{s \in S}$ *is pseudo-random with domain* Dom *and* $(\mathsf{H}_k)_{k \in K}$ *is unpredictable, so* $\mathsf{prf.adv}(\mathsf{red}_{\mathsf{prf}}(\mathcal{A}))$ *and* $1/|\mathsf{Dom}|$ *and* $\mathsf{uf.adv}(\mathsf{red}_{\mathsf{uf}}(\mathcal{A}))$ *are negligible. Then,* $\mathsf{ind\text{-}cca.adv}(\mathcal{A})$ *is negligible too.*

Our proof loosely follows Shoup's [74] and consists of three main game hops. Here, we focus only on the first hop, as the last two hops use the same techniques and proof principles. The complete formalisation of this cryptographic construct and the proofs are available online [53].

Our formalisation modularises the definitions and proofs using Isabelle's **locale**s. When a section starts by introducing a locale, all the definitions and lemmas in that section are defined in this locale. This yields reusable components that can be specialised to different settings by instantiating the locale's parameters.

locale IND-CCA ≡
  fixes key-gen : $\mathbb{D}(\kappa)$
  and enc : $\kappa \Rightarrow \mu \Rightarrow \mathbb{D}(\chi)$
  and dec : $\kappa \Rightarrow \chi \Rightarrow \mathbb{M}(\mu)$
  and valid-plain : $\mu \Rightarrow \mathbb{B}$

$\mathsf{G}(\mathcal{A}) \equiv$ **do** {
  $k \leftarrow$ key-gen;
  $b \leftarrow$ coin;
  $(b', \_) \leftarrow \mathsf{exec}(\mathcal{O}^{k,b}_{\mathsf{enc}} +_{\mathbb{O}} \mathcal{O}^{k}_{\mathsf{dec}}, \mathcal{A}, \{\})$;
  return $b' = b$ }

$\mathcal{O}^{k,b}_{\mathsf{enc}}(L, (m_1, m_0)) \equiv$ try **do** {
    assert (valid-plain$(m_1) \wedge$ valid-plain(m_0));
    $c \leftarrow \mathsf{enc}(k, \text{if } b \text{ then } m_1 \text{ else } m_0)$;
    return (Some$(c), \{c\} \cup L$)
  } else return (None, L)

$\mathcal{O}^{k}_{\mathsf{dec}}(L, c) \equiv$ return
  (if $c \in L$ then None else dec$(k, c), L$)

$\mathsf{adv}(\mathcal{A}) \equiv |\mathcal{P}[\mathsf{G}(\mathcal{A}) = \mathsf{True}] - 1/2|$

Fig. 12. IND-CCA security game for symmetric-key encryption schemes.

6.1. *IND-CCA Security of Symmetric-Key Ciphers*

A symmetric-key encryption scheme consists of three algorithms key-gen, enc, and dec. The probabilistic algorithm key-gen takes no input (other than the implicit security parameter) and generates a key k. The probabilistic encryption algorithm enc takes as input a key k and a plaintext m and outputs a ciphertext c. Finally, the deterministic decryption algorithm dec takes a key k and a ciphertext c as input and either rejects c or outputs a plaintext m.

IND-CCA security for symmetric encryption schemes is defined using a game between a challenger and an adversary (Fig. 12). In our framework, we leave the encryption scheme abstract by making key-gen, enc, and dec parameters of the module IND-CCA (a **locale** in Isabelle). In this module, the type variables κ, μ, and χ represent the space of keys, plaintexts, and ciphertexts, respectively. Note that the probabilistic algorithms key-gen and enc return a subprobability distribution, whereas the deterministic algorithm dec yields a value of type $\mathbb{M}(\mu)$, where None models rejection and Some(m) corresponds to the decryption succeeding. The additional parameter valid-plain tests whether (an encoding of) a plaintext is valid (e.g. it has the right length or represents an element of an algebraic structure). In contrast, μ models the set of all syntactically correct (encodings of) plaintexts. As explained in Sect. 3, distinguishing the validity of plaintexts from syntactic correctness yields both stronger theorems and better proof automation.

In the IND-CCA game G, the challenger first generates a fresh key k and a random bit $b : \mathbb{B}$. Afterwards, the adversary $\mathcal{A}$ is given access to an encryption oracle $\mathcal{O}^{k,b}_{\mathsf{enc}}$ and a decryption oracle $\mathcal{O}^{k}_{\mathsf{dec}}$, which mediate the access to k and b. Finally, the adversary outputs a bit b' and wins if $b = b'$.

For an encryption query, the adversary submits a pair of messages $(m_1, m_0) : \mu \times \mu$. The encryption oracle $\mathcal{O}_{\mathsf{enc}}$ checks that both messages are valid using the predicate valid-plain and returns the encryption c of m_b under k. The oracles' state L keeps track of all ciphertext responses of the encryption oracle, so $\mathcal{O}_{\mathsf{enc}}$ adds c to L too. If either plaintext is invalid, the query is rejected (response None).

A decryption query consists of a ciphertext c. If the encryption oracle has previously output c (i.e. $c \in L$), the decryption oracle rejects the query. Otherwise, it responds with result of the decryption algorithm dec.

$\mathcal{O}^k_{\mathsf{eval}}(L, x) \equiv$
  return $(\mathsf{H}(k, x), \{x\} \cup L)$

$\mathcal{O}^k_{\mathsf{submit}}((b, L), (x, h)) \equiv$
  return $(\circledast, (b \vee (h = \mathsf{H}(k, x) \wedge x \notin L), L))$

$\mathsf{G}(\mathcal{A}) \equiv$ **do** {
  $k \leftarrow$ key-gen;
  $((x, h), L) \leftarrow \mathsf{exec}(\mathcal{O}^k_{\mathsf{eval}}, \mathcal{A}, \{\})$;
  return $(h = \mathsf{H}(k, x) \wedge x \notin L)$ }

$\mathsf{G}_\mathsf{m}(\mathcal{A}) \equiv$ **do** {
  $k \leftarrow$ key-gen;
  $(\circledast, (b, \_)) \leftarrow \mathsf{exec}(\curlywedge\mathcal{O}^k_{\mathsf{eval}} +_\oplus \mathcal{O}^k_{\mathsf{submit}}, \mathcal{A}, (\mathsf{False}, \{\}))$;
  return b }

$\mathsf{adv}(\mathcal{A}) \equiv \mathcal{P}[\mathsf{G}(\mathcal{A}) = \mathsf{True}]$

$\mathsf{adv}_\mathsf{m}(\mathcal{A}) \equiv \mathcal{P}[\mathsf{G}_\mathsf{m}(\mathcal{A}) = \mathsf{True}]$

Fig. 13. Security game for unpredictability with a single guess (left) or multiple adaptive guesses (right) .

The advantage $\mathsf{adv}(\mathcal{A})$ measures $\mathcal{A}$'s ability to do better than guessing.

6.2. *Unpredictable Functions*

The encryption scheme uses two components: a pseudo-random function as formalised in Sect. 5.1 and an unpredictable function (UF), which we formalise now.

Let $\mathcal{H} := (H_k)_{k \in K}$ be a family of keyed functions. Unpredictability of this family is formalised as a game defined in a locale UF (Fig. 13) with two algorithms: the probabilistic algorithm key-gen generates a key $k : \kappa$ and the deterministic function H represents $\mathcal{H}$.

We formalise two variants of unpredictability, which differ in the number of guesses that the adversary is allowed. In the first setting (game G on the left), the adversary outputs a *single* guess. Formally, the challenger generates a fresh key k and gives the adversary $\mathcal{A}$ access to an oracle $\mathcal{O}^k_{\mathsf{eval}}$ that provides black-box access to the function H. That is, $\mathcal{A}$ can evaluate $\mathsf{H}(k, x)$ by querying $\mathcal{O}_{\mathsf{eval}}$ for x. All queries are recorded in the oracle's state L (initially the empty set). Finally, the adversary makes a guess (x, h) and wins if he correctly predicted $\mathsf{H}(k, x)$, i.e. $h = \mathsf{H}(k, x)$ and x has not been queried ($x \notin L$). This formalises Shoup's description [74].

In the other setting (game G_m on the right), the adversary may submit several guesses through a second oracle $\mathcal{O}_{\mathsf{submit}}$ and may even interleave querying $\mathsf{H}(k)$ and guessing. The oracle $\mathcal{O}_{\mathsf{submit}}$ evaluates the guess with respect to the current history L of guesses, records the result in the winning flag b of the state, and acknowledges the guess. However, its response does not indicate whether the guess was correct. (In G_m, we extend the state of the old oracle $\mathcal{O}_{\mathsf{eval}}$ with the winning flag using $\curlywedge$.) The adversary wins G_m if he has submitted a correct guess before he terminates.[11]

In both settings, the advantage is the probability that the adversary wins the game. In Sect. 6.5, we will reduce the multi-guess setting to the single-guess setting by using a generic many-to-one reduction. Before that, we will reduce the security of our encryption scheme to the multi-guess setting. This separation leads to a cleaner and more abstract proof.

[11] This formulation is not equivalent to the more conventional formulation where the adversary produces a list of guesses that is evaluated at the end of the game. In our setting, a (sub-optimal) adversary may submit a correct guess and then waste a query to see whether he was right. If his guesses were evaluated only at the end of the game, the wasted query would have invalidated his correct guess.

locale CIPHER $\equiv$
  fixes seed-gen : $\mathbb{D}(\sigma)$ **and** F : $\sigma \Rightarrow \mathbb{L}(\mathbb{B}) \Rightarrow \mathbb{L}(\mathbb{B})$ **and** Dom : $\mathbb{P}(\mathbb{L}(\mathbb{B}))$
  and key-gen : $\mathbb{D}(\kappa)$ **and** H : $\kappa \Rightarrow \mathbb{L}(\mathbb{B}) \Rightarrow \omega$
  and dlen : $\mathbb{N}$
  and clen : $\mathbb{N}$
  assumes lossless(seed-gen) **and** lossless(key-gen) **and** finite(Dom) **and** Dom $\neq \{\}$
  and $\forall x \in$ Dom. $\|x\| =$ dlen **and** $\forall s \in$ support(seed-gen). $\forall x \in$ Dom. $\|\mathsf{F}(s,x)\| =$ clen
sublocale prf $\equiv$ PRF(seed-gen, F, uniform($\{0,1\}^{\text{clen}}$))
sublocale uf $\equiv$ UF(key-gen, H)

kg $\equiv$ **do** {
  $s \leftarrow$ seed-gen;
  $k \leftarrow$ key-gen;
  return (s,k) }

enc$((s,k),m) \equiv$ **do** {
  $x \leftarrow$ uniform(Dom);
  let $c = \mathsf{F}(s,x) \oplus m$;
  let $t = \mathsf{H}(k, x +\!\!+ c)$;
  return (x,c,t) }

dec$((s,k),(x,c,t)) \equiv$
  if $\|x\| \neq$ dlen $\vee\ t \neq \mathsf{H}(k, x +\!\!+ c)$
  then None else Some$(\mathsf{F}(s,x) \oplus c)$

valid-plain$(m) \equiv \|m\| =$ clen

Fig. 14. Symmetric encryption scheme built from a pseudo-random function and an unpredictable function.

6.3. *Cipher Construction*

We formalise Shoup's encryption scheme in a module CIPHER, which abstracts over the pseudo-random function F and the unpredictable function H (Fig. 14). The pseudo-random function F maps bitstrings (type $\mathbb{L}(\mathbb{B})$) to bitstrings and we assume that F is used only on bitstrings from a finite, non-empty set Dom, all of which have length dlen ($\|\_\|$ denotes the length of a bitstring), and that if $x \in$ Dom and the seed s has been correctly generated, then $\mathsf{F}(s,x)$ always returns a bitstring of length clen. The unpredictable function H goes from bitstrings to an arbitrary type ω. The seed and key generators for F and H must both be lossless.

The **sublocale** commands import the modules PRF and UF from Sect. 5.1 and Fig. 13, which specify pseudo-randomness and unpredictability, specialising the parameters and types as needed and qualifying the names with the prefixes prf and uf. In particular, the pseudo-random function F's output is assumed to be pseudo-uniform over all bitstrings of length clen.

The symmetric-key encryption scheme is given by the three algorithms shown at the bottom of Fig. 14. A key (s,k) consists of a pseudo-random function's seed s and an unpredictable function's key k, so the key generation algorithm kg runs both seed-gen and key-gen. The encryption algorithm enc works as follows: first, it picks a random element x in Dom, runs it through F and exclusive-ors (notation $\oplus$) the result with the message $m : \mathbb{L}(\mathbb{B})$ to obtain c. Then, it applies the unpredictable function to the concatenation $x +\!\!+ c$ of x and c to obtain t. Finally, it responds with the ciphertext $(x,c,t) : \mathbb{L}(\mathbb{B}) \times \mathbb{L}(\mathbb{B}) \times \omega$. The decryption algorithm dec checks whether the ciphertext (x,c,t) is valid, i.e. t acts as a message authentication code (MAC) for (x,c). If not, it responds with the rejection None. Otherwise, it decrypts the ciphertext using F and xoring.

Clearly, encryption only works if the plaintext m has length clen—otherwise, the computation of exclusive-or is not well defined. Therefore, the predicate valid-plain checks the length.

Finally, we import the IND-CCA game specialised to our cipher using the following command:

sublocale ind-cca ≡ IND-CCA(kg, enc, dec, valid-plain).

But before we analyse the security of the cipher in the next section, we let Isabelle's rewriting engine prove automatically the following correctness statement:

Lemma 4. (Correctness) *Under the assumptions of the module* CIPHER*, if* $sk \in$ *support*(*kg*) *and* $\|m\| =$ *clen, then*

$$\textbf{do}\ \{\ c \leftarrow \mathit{enc}(sk, m);\ \ \mathit{return}\ \mathit{dec}(sk, c)\ \} = \mathit{return}\ \mathit{Some}(m).$$

6.4. *Formalising Security I: First Reduction Step*

To show that the cipher is IND-CCA secure, we first bound the advantage $\mathsf{ind\text{-}cca.adv}(\mathcal{A})$ of the adversary $\mathcal{A}$ by the advantages of two related adversaries. The first adversary is against F's pseudo-randomness and is obtained by the (efficient) reduction $\mathsf{red}_{\mathsf{prf}}$. The second adversary is against H's unpredictability in the multi-guess setting and is obtained by the (efficient) reduction $\mathsf{red}_{\mathsf{uf\text{-}m}}$. Thus, the security of the cipher follows from the assumed security of the two primitives.

Lemma 5. *Under the assumptions of the module* CIPHER*, if* $\mathcal{A}$ *is lossless and sends at most* q *encryption queries (i.e.* $\mathit{qbound}_{\mathit{is\text{-}Left}}(\mathcal{A}) \leq q$*), then*

$$\mathit{ind\text{-}cca.adv}(\mathcal{A}) \leq \mathit{prf.adv}(\mathit{red}_{\mathit{prf}}(\mathcal{A})) + \mathit{uf.adv}_m(\mathit{red}_{\mathit{uf\text{-}m}}(\mathcal{A})) + \frac{q^2}{|\mathit{Dom}|}.$$

Proof. This lemma is proved by defining a sequence of games and showing that the probability of a specific event of interest changes by a small amount between each pair of games in the sequence. The final result is then obtained using the triangle inequality, which sums up differences between successive game steps. All definitions and statements in this section are in the locale CIPHER.

This sequence starts with the game $\mathsf{G}_0 \equiv \mathsf{ind\text{-}cca.G}$. The next game G_1 differs from G_0 only in its decryption oracle rejecting all decryption queries. As the IND-CCA game is defined in the parameterised module IND-CCA, we can specify G_1 simply by importing another instance of IND-CCA:

sublocale $\mathsf{ind\text{-}cca}_1 \equiv$ IND-CCA(kg, enc, $(\lambda\_.\ \mathsf{None})$, valid-plain)
abbreviation $\mathsf{G}_1 \equiv \mathsf{ind\text{-}cca}_1\mathsf{.G}$.

Next, we want to bound the difference in the probability of $\mathcal{A}$ winning G_0 vs. winning G_1. To this end, let F be the failure event that t is a correct MAC for some decryption query (x, c, t). It is clear that G_0 and G_1 proceed identically until the event F happens. To formally prove this, we must be able to express the event F in both games. In our framework, this is possible by adding a Boolean flag *bad* to the oracle states. That is, we define two new games G'_0 and G'_1 that additionally return whether the failure event

has happened and prove that they refine G_0 and G_1, i.e. $\mathsf{G}_0 = \mathsf{G}'_0 \rhd^{\widehat{\mathbb{D}}} \pi_1$ and similarly for G_1.

We only present the necessary steps for G'_0, as those for G'_1 are analogous. First, we modify $\mathsf{ind\text{-}cca}.\mathcal{O}_{\mathsf{dec}}$ to obtain a new decryption oracle $\mathcal{O}_{\mathsf{dec0'}}$, which behaves the same except that the new flag *bad* records whether F has happened. The game G'_0 uses $\mathcal{O}_{\mathsf{dec0'}}$ instead of $\mathsf{ind\text{-}cca}.\mathcal{O}_{\mathsf{dec}}$ and returns the winning condition and the final value of the flag. Note that we do not need to modify the encryption oracle itself; the operator $\lambda\!\!\!\lambda$ suffices to adapt the encryption oracle so that it ignores the new flag *bad*.

$$
\begin{array}{l}
\mathcal{O}^{(s,k)}_{\mathsf{dec0'}}((bad, L), (x, c, t)) \equiv \mathsf{return} \\
\quad (\mathsf{if}\ (x, c, t) \in L \vee \|x\| \neq \mathsf{dlen}\ \mathsf{then}\ (\mathsf{None}, (bad, L)) \\
\quad\ \mathsf{else}\ (\mathsf{dec}((s, k), (x, c, t)), (bad \vee \mathsf{H}(k, x +\!\!+ c) = t, L)))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{G}'_0(\mathcal{A}) \equiv \mathbf{do}\ \{ \\
\quad key \leftarrow \mathsf{kg}; \\
\quad b \leftarrow \mathsf{coin}; \\
\quad (b', (bad, \_)) \leftarrow \mathsf{exec}(\lambda\!\!\!\lambda \mathcal{O}^{key,b}_{\mathsf{enc}} +_{\mathbb{O}} \mathcal{O}^{key}_{\mathsf{dec0'}}, \mathcal{A}, (\mathsf{False}, \{\})); \\
\quad \mathsf{return}\ (b' = b, bad)\ \}
\end{array}
$$

Second, we prove $\mathsf{G}_0 = \mathsf{G}'_0 \rhd^{\widehat{\mathbb{D}}} \pi_1$ using our relational parametricity toolbox. That is, for any pair of results (bad, b') that G'_0 returns, G_0 returns b' and vice versa. In detail, we define the relation $S \equiv \{\, (L_1, (bad, L_2)) \mid L_1 = L_2 \,\}$ and prove that S is a bisimulation relation for $\mathsf{ind\text{-}cca}.\mathcal{O}_{\mathsf{dec}}$ and $\mathcal{O}_{\mathsf{dec0'}}$ and relates the initial states $\{\}$ and $(\mathsf{False}, \{\})$; this step is checked automatically by term rewriting. From this, we derive $\mathsf{G}_0 = \mathsf{G}'_0 \rhd^{\widehat{\mathbb{D}}} \pi_1$ by representation independence, which is automatic too. This shows that equational reasoning and the ideas from relational parametricity do help when conducting formal cryptographic proofs.

We are now ready to state the bound on the difference in probabilities, as G'_0 and G'_1 capture the failure event $F \equiv \{\, \mathsf{True}, \mathsf{False} \,\} \times \{\, \mathsf{True} \,\}$, but neither G_0 nor G_1 do.

Claim 5.1. $\left|\mathcal{P}\left[\mathsf{G}_0(\mathcal{A}) = \mathsf{True}\right] - \mathcal{P}\left[\mathsf{G}_1(\mathcal{A}) = \mathsf{True}\right]\right| \leq \mathcal{P}\left[\mathsf{G}'_1(\mathcal{A}) \in F\right]$.

We apply Prop. 2 to the compound oracles $\lambda\!\!\!\lambda \mathcal{O}^{key,b}_{\mathsf{enc}} +_{\mathbb{O}} \mathcal{O}^{key}_{\mathsf{dec0'}}$ and $\lambda\!\!\!\lambda \mathcal{O}^{key,b}_{\mathsf{enc}} +_{\mathbb{O}} \mathcal{O}^{key}_{\mathsf{dec1'}}$, with the failure predicates $F_1 \equiv F_2 \equiv \pi_2$ and the relation $S \equiv (=)$ to couple G'_0 and G'_1, where $\mathcal{O}_{\mathsf{dec1'}}$ differs from $\mathcal{O}_{\mathsf{dec0'}}$ in that it returns None instead of calling the decryption algorithm dec. Then, the claim follows by the Lemma 2, which formalises the reasoning about failure events, as $\mathsf{G}_0 = \mathsf{G}'_0 \rhd^{\widehat{\mathbb{D}}} \pi_1$ and $\mathsf{G}_1 = \mathsf{G}'_1 \rhd^{\widehat{\mathbb{D}}} \pi_1$.

Finally, we claim that the probability of the event F in G'_1 corresponds to the advantage of an efficient adversary $\mathsf{red}_{\mathsf{uf\text{-}m}}(\mathcal{A})$ against H's unpredictability.

Claim 5.2. $\mathcal{P}\left[\mathsf{G}'_1(\mathcal{A}) \in F\right] = \mathsf{uf.adv}_{\mathsf{m}}(\mathsf{red}_{\mathsf{uf\text{-}m}}(\mathcal{A}))$.

The reduction consists of three parts (Fig. 15): two converters $\mathsf{C}_{\mathsf{enc}}$ and $\mathsf{C}_{\mathsf{dec}}$ answer the encryption and decryption queries of the IND-CCA adversary $\mathcal{A}$ using the evaluation and guessing oracles of the unpredictability game $\mathsf{uf.G}_{\mathsf{m}}$; and the GPV $\mathsf{red}_{\mathsf{uf\text{-}m}}(\mathcal{A})$ initialises

$\mathsf{C}_{\mathsf{enc}}^{(s,b)}((L,X),(m_1,m_0)) \equiv$
  if $\|m_1\| = \mathsf{clen} \wedge \|m_0\| = \mathsf{clen}$ then **do** {
    $x \leftarrow \mathsf{sample}(\mathsf{uniform}(\mathsf{Dom}))$;
    let $c = \mathsf{F}(s,x) +\!\!+ (\mathsf{if}\ b\ \mathsf{then}\ m_1\ \mathsf{else}\ m_0)$;
    $t \leftarrow \mathsf{evaluate}(x +\!\!+ c)$;
    return $(\mathsf{Some}(x,c,t), (\{(x,c,t)\} \cup L, X))$
  } else return $(\mathsf{None}, (L,X))$

$\mathsf{red}_{\mathsf{uf\text{-}m}}(\mathcal{A}) \equiv$ **do** {
  $s \leftarrow \mathsf{sample}(\mathsf{seed\text{-}gen})$;
  $b \leftarrow \mathsf{sample}(\mathsf{coin})$;
  $\mathsf{inline}(\mathsf{C}_{\mathsf{enc}}^{(s,b)} +_{\mathbb{C}} \mathsf{C}_{\mathsf{dec}}, \mathcal{A}, (\{\},\{\}))$;
  return $\circledast$ }

$\mathsf{C}_{\mathsf{dec}}((L,X),(x,c,t)) \equiv$
  if $(x,c,t) \in L \vee \|x\| \neq \mathsf{dlen}$ then
    return $(\mathsf{None}, (L,X))$
  else **do** {
    $\mathsf{submit\text{-}guess}(x +\!\!+ c, t)$;
    return $(\mathsf{None}, (L, \{(x,c,t)\} \cup X))$
  }

$\mathsf{evaluate}(x) \equiv \mathsf{call}(\mathsf{Left}(x))$
$\mathsf{submit\text{-}guess}(y,t) \equiv \mathsf{call}(\mathsf{Right}(y,t))$

Fig. 15. Reduction against the unpredictability in the multi-guess setting.

the converters with a seed for the pseudo-random function and the challenge bit from the IND-CCA game $\mathsf{ind\text{-}cca.G}$ and composes the adversary $\mathcal{A}$ with them.

The converter $\mathsf{C}_{\mathsf{enc}}$ for encryption queries behaves exactly like $\mathsf{ind\text{-}cca}.\mathcal{O}_{\mathsf{enc}}$ except that it does not evaluate H itself, but calls the oracle $\mathsf{uf}.\mathcal{O}_{\mathsf{eval}}$ instead. Hence, it does not take H's key as a parameter. Similarly, $\mathsf{C}_{\mathsf{dec}}$ computes a guess from every decryption query for a ciphertext that has not been produced by $\mathsf{C}_{\mathsf{enc}}$ and submits it to the guessing oracle $\mathsf{uf}.\mathcal{O}_{\mathsf{submit}}$.

The state shared by both converters consists of two sets L and X, which are initially empty. The set L corresponds to the state L of the IND-CCA oracles, which keeps track of all previously generated ciphers. The set X records all the decryption queries that have been submitted as a guess. It is not used by the reduction, only in the proof of Claim 5.2.

The proof of Claim 5.2 consists of two steps which are typical for reduction proofs. Both steps are largely automated. First, we exploit the associativity of composition (15) and consider the composition of the converters with the UF oracles, as illustrated in Fig. 16. Second, we prove that this composition is bisimilar to the UF oracle $\lambda\mathsf{uf}.\mathcal{O}_{\mathsf{eval}} +_{\mathbb{O}} \mathsf{uf}.\mathcal{O}_{\mathsf{submit}}$, where the bisimulation relation enforces among others the condition that X contains a correct guess iff the flag b has been set in $\mathcal{O}_{\mathsf{submit}}$. Here, we prove bisimilarity by term rewriting and resolution and then appeal to representation independence, as we did for G_0 and G_0'.

Claims 5.1 and 5.2 show that the first game hop changes the adversary's advantage by at most $\mathsf{uf.adv_m}(\mathsf{red}_{\mathsf{uf\text{-}m}}(\mathcal{A}))$, which according to Corollary 1 is assumed to be negligible. This completes the first game hop.

The remaining three game hops follow a similar pattern, so we only sketch the idea. The second game hop from G_1 to G_2 changes the encryption oracle such that it uses the random oracle $\mathsf{prf}.\mathcal{O}_{\mathsf{RF}}$ instead of the pseudo-random function F. This game hop is justified by the reduction $\mathsf{red}_{\mathsf{prf}}$ to F being pseudo-random. The reduction follows the same lines as $\mathsf{red}_{\mathsf{uf\text{-}m}}$; Fig. 17 shows the formal definition.

Claim 5.3. $\left|\mathcal{P}[\mathsf{G}_1(\mathcal{A}) = \mathsf{True}] - \mathcal{P}[\mathsf{G}_2(\mathcal{A}) = \mathsf{True}]\right| \leq \mathsf{prf.adv}(\mathsf{red}_{\mathsf{prf}}(\mathcal{A}))$.

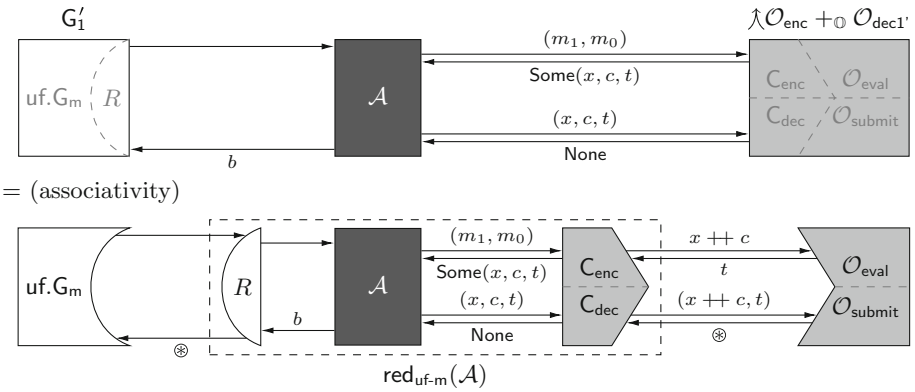


Fig. 16. Illustration of Claim 5.2: the reduction $\mathsf{red_{uf\text{-}m}}$ transforms G_1' into the unpredictability game $\mathsf{uf.G_m}$.

$\mathsf{red_{prf}}(\mathcal{A}) \equiv \mathbf{do}\ \{$
$\quad k \leftarrow \mathsf{sample(key\text{-}gen)};$
$\quad b \leftarrow \mathsf{sample(coin)};$
$\quad (b', \_) \leftarrow \mathsf{inline}(\mathsf{C}_{\mathsf{prf}}^{(k,b)}, \mathcal{A}, \circledast);$
$\quad \mathsf{return}\ b' = b\ \}$

$\mathsf{C}_{\mathsf{prf}}^{(k,b)}(\_, x) \equiv \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Left}(m_1, m_0) \Rightarrow$
$\quad \mathsf{if}\ \|m_1\| = \mathsf{clen} \wedge \|m_0\| = \mathsf{clen}\ \mathsf{then}\ \mathbf{do}\ \{$
$\quad\quad x \leftarrow \mathsf{sample(uniform(Dom))};$
$\quad\quad p \leftarrow \mathsf{call}(x);$
$\quad\quad \mathsf{let}\ c = p \oplus (\mathsf{if}\ b\ \mathsf{then}\ m_1\ \mathsf{else}\ m_0);$
$\quad\quad \mathsf{let}\ t = \mathsf{H}(k, x +\!\!+ c);$
$\quad\quad \mathsf{return}\ (\mathsf{Left}(\mathsf{Some}(x, c, t)), \circledast)$
$\quad \}\ \mathsf{else\ return}\ (\mathsf{Left}(\mathsf{None}), \circledast)$
$\mid \mathsf{Right}(x, c, t) \Rightarrow \mathsf{return}\ (\mathsf{Right}(\mathsf{None}), \circledast)$

Fig. 17. Reduction from games G_1 and G_2 to F's pseudo-randomness.

The third hop makes the random oracle forgetful, whereby it always returns a freshly sampled value even if the same point is queried several times. This yields the game G_3.

Claim 5.4. $\left|\mathcal{P}\left[\mathsf{G}_2(\mathcal{A}) = \mathsf{True}\right] - \mathcal{P}\left[\mathsf{G}_3(\mathcal{A}) = \mathsf{True}\right]\right| \leq \dfrac{q^2}{|\mathsf{Dom}|}.$

The term $q^2/|\mathsf{Dom}|$ bounds the probability of querying the random oracle twice with the same point x, as there are at most q encryption queries.

The last hop applies the one-time-pad for bitstrings (10). Then, the ciphertexts are independent of the plaintexts, so the whole game reduces to a coin flip like in the Elgamal example (Fig. 2, lines 37–38).

Claim 5.5. $\mathcal{P}\left[\mathsf{G}_3(\mathcal{A}) = \mathsf{True}\right] = 1/2.$

Lemma 5 follows directly from Claims 5.1–5.5 by the triangle inequality. □

The bound in Lemma 5 is expressed in the multi-guess setting for the unpredictable function. For comparison, Shoup's cipher-specific reduction goes directly to the single-guess setting [74]. We added the multi-guess setting as an intermediate step because the security proof can be split nicely into two simpler, modular steps. The first step,

$$\textbf{locale } \mathsf{GOM} \equiv$$
$$\quad \textbf{fixes } \mathsf{init} : \mathbb{D}(\mathrm{o} \times \alpha \times \sigma)$$
$$\quad \textbf{and } \mathsf{orc} : \mathrm{o} \Rightarrow \mathbb{O}(\sigma, \theta, \rho)$$
$$\quad \textbf{and } \mathsf{eval} : \mathrm{o} \Rightarrow \alpha \Rightarrow \sigma \Rightarrow \gamma \Rightarrow \mathbb{D}(\mathbb{B})$$

$$\mathsf{G}_{\mathsf{single}}(\mathcal{A}) \equiv \textbf{do } \{$$
$$\quad (c_o, c_a, s_0) \leftarrow \mathsf{init};$$
$$\quad (g, s) \leftarrow \mathsf{exec}(\mathsf{orc}^{c_o}, \mathcal{A}(c_a), s_0);$$
$$\quad \mathsf{eval}(c_o, c_a, s, g)\ \}$$

$$\mathsf{adv}_{\mathsf{single}}(\mathcal{A}) \equiv \mathcal{P}[\mathsf{G}_{\mathsf{single}}(\mathcal{A}) = \mathsf{True}]$$

$$\mathcal{O}_{\mathsf{guess}}^{c_o,c_a}((b, s), g) \equiv \textbf{do } \{$$
$$\quad b' \leftarrow \mathsf{eval}(c_o, c_a, s, g);$$
$$\quad \mathsf{return}\ (\circledast, (b \vee b', s))\ \}$$

$$\mathsf{G}_{\mathsf{multi}}(\mathcal{A}) \equiv \textbf{do } \{$$
$$\quad (c_o, c_a, s_0) \leftarrow \mathsf{init};$$
$$\quad (\_, (b, \_)) \leftarrow$$
$$\quad\quad \mathsf{exec}(\lambda\!\!\lambda \mathsf{orc}^{c_o} +_{\mathbb{O}} \mathcal{O}_{\mathsf{guess}}^{c_o,c_a}, \mathcal{A}(c_a), (\mathsf{False}, s_0));$$
$$\quad \mathsf{return}\ b\ \}$$

$$\mathsf{adv}_{\mathsf{multi}}(\mathcal{A}) \equiv \mathcal{P}[\mathsf{G}_{\mathsf{multi}}(\mathcal{A}) = \mathsf{True}]$$

Fig. 18. Guessing games in the single-guess and multi-guess setting.

which is specific to the cipher construction, remains simple as the IND-CCA game with multiple decryption queries matches well with the multi-guess UF setting (Claim 5.2). The second step in the next section is independent of the cipher and can thus be reused in other proofs. Despite the modularity and generality, we obtain the same overall bound on the advantage as Shoup.

6.5. *Formalising Security II: A Generic Reduction for Games with Multiple Guesses*

We have formalised unpredictability in two ways: the multi-guess setting and the single-guess setting. In this section, we reduce the multi-guess setting to the single-guess setting by picking a random guess of the (multi-guess) adversary, where the bound on the advantage deteriorates linearly in the number of guesses. Randomly picking an output of the adversary is a common trick in reductions, and we demonstrate here how such common steps can be expressed in our framework. Actually, we formalise the reduction abstractly as a reusable module GOM and then instantiate the module for the unpredictable function.

The module GOM (Fig. 18) abstracts over the initialisation part init, the oracle orc of the single-guess adversary, and the evaluation function eval for guesses. The types are left abstract as type variables with the following meaning: o represents secret initialisation data that is given only to the oracle, α the public initialisation data that is given to the adversary, σ the oracle's state space, γ the guesses, θ the queries, and ρ the responses. Evaluating a guess $\mathsf{eval}(c_o, c_a, s, g)$ depends on the initialisation data (c_o, c_a) and the oracle state s and may be probabilistic.

We define two games $\mathsf{G}_{\mathsf{single}}$ and $\mathsf{G}_{\mathsf{multi}}$ for the two settings. They first generate the initialisation data (c_o, c_a) and the initial oracle state s_0 using init. Then, in $\mathsf{G}_{\mathsf{single}}$, the adversary $\mathcal{A}$ produces a guess g after interacting with the oracle orc, and g is evaluated in the final oracle state s using eval. In the multi-guess game $\mathsf{G}_{\mathsf{multi}}$, the adversary $\mathcal{A}$ need not produce a guess at the end, but instead submits his guesses through the additional oracle $\mathcal{O}_{\mathsf{guess}}$. This oracle is an abstract version of $\mathcal{O}_{\mathsf{submit}}$ in Fig. 13: it evaluates the guess using eval, updates the winning flag b in the oracle state, which records whether a correct guess has been submitted, and acknowledges the receipt of the submission to the adversary. Again, the oracle operators $\lambda\!\!\lambda$ and $+_{\mathbb{O}}$ introduced in Sect. 5.8 combine the

$$
\begin{array}{l}
\mathsf{C}_{\mathsf{call}}(s, x) \equiv \mathsf{case}\ s\ \mathsf{of} \\
\quad \mathsf{Left}(g) \ \Rightarrow \mathsf{return}\ (\mathsf{None}, \mathsf{Left}(g)) \\
\quad \mid \mathsf{Right}(j) \Rightarrow \mathbf{do}\ \{\ r \leftarrow \mathsf{call}(x);\ \mathsf{return}\ (\mathsf{Some}(r), \mathsf{Right}(j))\ \}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{C}_{\mathsf{guess}}(s, g) \equiv \mathsf{case}\ s\ \mathsf{of} \\
\quad \mathsf{Left}(g') \quad\ \Rightarrow \mathsf{return}\ (\mathsf{None}, \mathsf{Left}(g')) \\
\quad \mid \mathsf{Right}(0) \quad\ \Rightarrow \mathsf{return}\ (\mathsf{None}, \mathsf{Left}(g)) \\
\quad \mid \mathsf{Right}(j+1) \Rightarrow \mathsf{return}\ (\mathsf{Some}(\circledast), \mathsf{Right}(j))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{red}^q(\mathcal{A}, c_a) \equiv \mathbf{do}\ \{ \\
\quad j_s \leftarrow \mathsf{sample}(\mathbb{Z}_q); \\
\quad (\_, s) \leftarrow \mathsf{inline}(\mathsf{C}_{\mathsf{call}} +_{\mathbb{C}} \mathsf{C}_{\mathsf{guess}}, \mathsf{interruptible}(\mathcal{A}(c_a)), \mathsf{Right}(j_s)); \\
\quad \mathsf{return}(\mathsf{projl}(s))\ \}
\end{array}
$$

Fig. 19. Reduction from the abstract multi-guess setting to the single-guess setting.

two oracles into one. In both settings, $\mathcal{A}$'s advantage is the probability of winning the appropriate game. The next lemma bounds the multi-guess advantage by the single-guess advantage.

Proposition 3. *Let* $\mathsf{orc}^{c_o}(s, x)$ *and* $\mathsf{eval}(c_o, c_a, s, g)$ *be lossless for all* s, x, g, *and* $(c_o, c_a, \_) \in \mathsf{support}(\mathsf{init})$. *Assume that* $\mathcal{A}(c_a)$ *submits at most* q *guesses, i.e.* $\mathsf{qbound}_{\mathsf{is\text{-}Right}}(\mathcal{A}(c_a)) \leq q$ *whenever* $(\_, c_a, \_) \in \mathsf{support}(\mathsf{init})$. *Then,*

$$\mathsf{adv}_{\mathsf{multi}}(\mathcal{A}) \leq q \cdot \mathsf{adv}_{\mathsf{single}}(\mathsf{red}^q(\mathcal{A})),$$

where the reduction red^q *is defined in Fig. 19.*

Proof. The proof consists of three game transformations. First, we express the probability of winning in $\mathsf{G}_{\mathsf{multi}}$ in terms of the winning probabilities of individual guess queries. Let $\{ g_0, g_1, ..., g_q \}$ denote all guesses submitted to $\mathcal{O}_{\mathsf{guess}}$ in $\mathsf{G}_{\mathsf{multi}}$. Let j_0 be the index of the first such query that sets the flag b in $\mathcal{O}_{\mathsf{guess}}$. Also, let E_i be the event that $j_0 = i$ for $i \leq q$. Then, the probability of winning in $\mathsf{G}_{\mathsf{multi}}$ is $\sum_{i=0}^{q} \mathcal{P}[E_i]$ by the total probability law. To show this equivalence in our framework, we define another game $\mathsf{G}_{\mathsf{multi'}}$ that captures the E_i events. Namely, its oracle $\mathcal{O}_{\mathsf{guess'}}$ replaces the flag b in $\mathcal{O}_{\mathsf{guess}}$ with a pair (j, j') where j counts the number of guesses so far and $j' = \mathsf{Some}(j_0)$ if the first correct guess was the j_0th and $j' = \mathsf{None}$ if all guesses were incorrect so far.

$$
\begin{array}{l}
\mathcal{O}^{c_o, c_a}_{\mathsf{guess'}}(((j, j'), s), g) \equiv \mathsf{case}\ j'\ \mathsf{of} \\
\quad \mathsf{Some}(j_0) \Rightarrow \mathsf{return}\ (\circledast, ((j+1, \mathsf{Some}(j_0)), s)) \\
\quad \mid \mathsf{None} \quad\ \Rightarrow \mathbf{do}\ \{ \\
\qquad\qquad b' \leftarrow \mathsf{eval}(c_o, c_a, s, g); \\
\qquad\qquad \mathsf{return}\ (\circledast, ((j+1, \mathsf{if}\ b'\ \mathsf{then}\ \mathsf{Some}(j)\ \mathsf{else}\ \mathsf{None}), s))\ \}
\end{array}
$$

$\mathsf{G}_{\mathsf{multi'}}(\mathcal{A}) \equiv \mathbf{do}\ \{$
$\quad j_s \leftarrow \mathsf{uniform}(\mathbb{Z}_q);$
$\quad (c_o, c_a, s_0) \leftarrow \mathsf{init};$
$\quad (\_, ((\_, j_0), \_)) \leftarrow \mathsf{exec}(\lambda\mathsf{orc}^{c_o} +_{\mathbb{O}} \mathcal{O}^{c_o,c_a}_{\mathsf{guess'}}, \mathcal{A}(c_a), ((0, \mathsf{None}), s_0));$
$\quad \mathsf{return}\ j_0 = \mathsf{Some}(j_s)\ \}$

Claim 5.6. $\mathsf{adv}_{\mathsf{multi}}(\mathcal{A}) = q \cdot \mathcal{P}\big[\mathsf{G}_{\mathsf{multi'}}(\mathcal{A}) = \mathsf{True}\big]$.

We prove this claim in two steps. First, we prove that $\mathcal{O}_{\mathsf{guess}}$ and $\mathcal{O}_{\mathsf{guess'}}$ are bisimilar and extend this result to the two games using representation independence. Second, we use the law of total probability from the Isabelle probability library and elementary facts about summations mostly using term rewriting.

In the second game hop, we change $\mathsf{G}_{\mathsf{multi'}}$ to $\mathsf{G}_{\mathsf{stop}}$ such that the adversary is stopped immediately after j_sth guess query using the operator $\mathsf{interruptible}$ from Sect. 5.7. Accordingly, the new oracles $\mathcal{O}_{\mathsf{o}}$ and $\mathcal{O}_{\mathsf{g}}$ trigger the interrupt by returning None. Their state has two shapes: $\mathsf{Right}(j)$ models that the j_sth guess will be after j more guesses and $\mathsf{Left}(g, s)$ records the j_sth guess g with the oracle state s.

$\mathcal{O}^{c_o}_{\mathsf{o}}((\sigma, s), x) \equiv \mathsf{case}\ \sigma\ \mathsf{of}$
$\quad \mathsf{Left}(\_) \Rightarrow \mathsf{return}\ (\mathsf{None}, (\sigma, s))$
$\mid \mathsf{Right}(\_) \Rightarrow \mathbf{do}\ \{\ (r, s') \leftarrow \mathsf{orc}^{c_o}(s, x);\ \mathsf{return}\ (\mathsf{Some}(r), (\sigma, s'))\ \}$

$\mathcal{O}_{\mathsf{g}}((\sigma, s), g) \equiv \mathsf{case}\ \sigma\ \mathsf{of}$
$\quad \mathsf{Left}(\_) \Rightarrow \mathsf{return}\ (\mathsf{None}, (\sigma, s))$
$\mid \mathsf{Right}(0) \Rightarrow \mathsf{return}\ (\mathsf{None}, (\mathsf{Left}(g, s), s))$
$\mid \mathsf{Right}(i + 1) \Rightarrow \mathsf{return}\ (\mathsf{Some}(\circledast), (\mathsf{Right}(i), s))$

$\mathsf{G}_{\mathsf{stop}}(\mathcal{A}) \equiv \mathbf{do}\ \{$
$\quad j_s \leftarrow \mathsf{uniform}(\mathbb{Z}_q);$
$\quad (c_o, c_a, s_0) \leftarrow \mathsf{init};$
$\quad (\_, (\sigma, \_)) \leftarrow \mathsf{exec}(\mathcal{O}^{c_o}_{\mathsf{o}} +_{\mathbb{O}} \mathcal{O}_{\mathsf{g}}, \mathsf{interruptible}(\mathcal{A}(c_a)), (\mathsf{Right}(j_s), s_0));$
$\quad \mathsf{case}\ \sigma\ \mathsf{of}\ \mathsf{Right}(\_) \Rightarrow \mathsf{return}\ \mathsf{False} \mid \mathsf{Left}(g, s) \Rightarrow \mathsf{eval}(c_o, c_a, s, g)\ \}$

Claim 5.7. $\mathcal{P}\big[\mathsf{G}_{\mathsf{multi'}}(\mathcal{A}) = \mathsf{True}\big] \le \mathcal{P}\big[\mathsf{G}_{\mathsf{stop}}(\mathcal{A}) = \mathsf{True}\big]$.

Claim 5.7 is the crucial step in the proof because eval is now called only at the game's end, but not by the oracles. This step requires reasoning about the semantics of the operator exec as it is an open problem how to move probabilistic computations out of loops (such as exec) using relational reasoning [7]. Instead, we prove a suitable loop transformation using fixpoint induction. Claim 5.7 is an upper bound instead of an equality because we have not bounded the number of $\mathcal{A}$'s queries to orc, so $\mathsf{interruptible}(\mathcal{A}(c_a))$ may terminate more often than $\mathcal{A}(c_a)$.

The third game hop transforms $\mathsf{G}_{\mathsf{stop}}(\mathcal{A})$ into the game $\mathsf{G}_{\mathsf{single}}(\mathsf{red}^q(\mathcal{A}))$ where the converters of the reduction red (Fig. 19) closely follow the oracles in $\mathsf{G}_{\mathsf{stop}}$.

Claim 5.8. $\mathcal{P}\big[\mathsf{G}_{\mathsf{stop}}(\mathcal{A}) = \mathsf{True}\big] \le \mathcal{P}\big[\mathsf{G}_{\mathsf{single}}(\mathsf{red}^q(\mathcal{A})) = \mathsf{True}\big]$.

This claim is proven using a combination of equational and relational reasoning. Again, this claim is not an equality, because the two games differ in case the adversary makes less than j_s queries. Then, the reduction $\mathsf{red}^q(\mathcal{A})$ returns an unspecified value $\mathsf{projl}(s)$ that may happen to be a correct guess.

The statement of Proposition 3 follows from Claims 5.6, 5.7, and 5.8. □

Proposition 3 yields a bound on the advantage in the multi-guess setting for the unpredictable function in terms of the single-guess advantage. To this end, the module UF imports the module GOM as follows.

$$\begin{aligned}
&\mathsf{initialize} && \equiv \mathbf{do}\ \{\ k \leftarrow \mathsf{key\text{-}gen};\ \mathsf{return}\ (k, \circledast, \{\})\ \}\\
&\mathsf{oracle} && \equiv \mathcal{O}_{\mathsf{eval}}\\
&\mathsf{eval}(k, \circledast, L, (x, h)) && \equiv \mathsf{return}\ (h = \mathsf{H}(k, x) \wedge x \notin L)
\end{aligned}$$

sublocale gom ≡ GOM(initialize, oracle, eval)

The following corollary is then proved automatically by term rewriting.

Corollary 2. *In the locale UF, if the lossless adversary $\mathcal{A}$ sends at most q guess queries, then*

$$\mathit{adv}_m(\mathcal{A}) \leq q \cdot \mathit{adv}(\mathit{gom.red}^q(\lambda\_.\ \mathcal{A})).$$

We are now ready to finish the IND-CCA security proof.

Proof of Theorem 2. Recall that the locale CIPHER imports UF and UF imports GOM. Thus, in CIPHER, we can reuse the reduction from Fig. 19 specialised to the unpredictable function H. So, let $\mathsf{red}^{q'}_{\mathsf{uf}}(\mathcal{A}) \equiv \mathsf{uf.gom.red}^{q'}(\lambda\_.\ \mathsf{red}_{\mathsf{uf\text{-}m}}(\mathcal{A}))$. Then, $\mathsf{qbound}_{\mathsf{is\text{-}Right}}(\mathsf{red}_{\mathsf{uf\text{-}m}}(\mathcal{A})) \leq \mathsf{qbound}_{\mathsf{is\text{-}Right}}(\mathcal{A})$, and Theorem 2 follows from Lemma 5 and Corollary 2. □

Corollary 1 follows from Theorem 2 in the same way as the asymptotic statement for Elgamal followed from the concrete one in Sect. 3.

6.6. *Comparison with Shoup's Security Proof*

We now compare our formalised proof with Shoup's informal one. The three game hops and the overall proof structure are the same, except that we factor out the generic many-to-one reduction (Sect. 6.5), which Shoup does on the fly. In particular, our reductions in Fig. 17 and Figs. 15 and 19 are codified versions of his oracle machines $D^{\mathcal{O}}$ and $B^{\mathcal{O}}$, respectively.

The main difference lies in the level of formality. Shoup does not make the games and proofs precise. Indeed, he writes the following [74]:

> Because it would be rather unwieldy, we do not give an explicit, low-level, algorithmic description of these games, but it should by now be clear that this could be done in principle. Rather, we give only a high-level description of Game 0, and brief descriptions of the modifications between successive games.

This suggests that Shoup's semi-formal notation, which he uses in earlier examples like Elgamal, does not scale well to more complicated settings. Apparently, the level of abstraction is too low as Shoup has no good notation for modelling interactions with oracles, which he emulates with for loops. In contrast, our case study shows that CryptHOL's GPVs can handle such settings easily, capturing oracle interactions and reductions in a natural way.

The other major difference is the underlying probability space. In Shoup's model, all games in a sequence of game hops operate on the same probability space, which is left implicit and assumed to be sufficiently large. In contrast, a definitional approach like ours forbids such implicitness. Instead, in CryptHOL, each distribution lives in its own space and there are operators to combine them. The practical implications of this difference can be seen, e.g. in the proof of Claim 5.1. Shoup can simply define the failure event F and reason about it as all random variables are global. In contrast, our proof must first make the relevant information accessible by introducing the games G_0' and G_1'. In return, we get a compositional and modular semantics in which equality coincides with contextual equivalence, that is, we can replace a program by another equal program in any context. This does not hold for Shoup's notation where care is needed to avoid capturing global variables.

7. Comparison and Discussion

The previous sections presented CryptHOL's semantic foundation and demonstrated its usage. Now, we evaluate CryptHOL using the desiderata from Sect. 1 and argue that it strikes a better balance between rigour and comprehensibility than existing frameworks for mechanising game-hopping proofs (Sect. 7.1). We also review other related work (Sect. 7.2).

7.1. *Evaluation and Comparison with Existing Frameworks*

We compare our framework CryptHOL with CertiCrypt [8], EasyCrypt [6], FCF [64], and Verypto [4]. Our comparison is along the four desiderata from the introduction: foundational approach, automation, naturality, and extensibility. Additionally, we highlight technical similarities and differences.

Foundational approach Rigour requires a formal language for formalising the cryptographic constructions, security notions, and proof rules that a trusted proof checker requires to check proofs. The foundational approach demands that these proof rules are rigorously derived from simple axioms that are known to be consistent. To this end, all frameworks assign semantics to the language and justify the proof rules using the semantics. However, they differ in the choice of language, in the semantic domain, and in the justification.

The semantic domain determines what notions and constructions can be expressed in the language. CertiCrypt and EasyCrypt support discrete subprobability distributions expressed using a probabilistic while language. CryptHOL has the same semantic domain and the language is equally expressive, thanks to the fixpoint operator. Verypto's

semantic domain is more general as it builds on measure theory and it therefore supports continuous distributions and higher-order functions at the price of incurring measurability proof obligations. FCF's semantics allows only probability distributions with finite support. The syntax further restricts probabilistic effects to the random sampling of bitstrings and conditional probabilities.

CryptHOL, CertiCrypt, and FCF all construct the semantic domain from first principles of the underlying logic (HOL or the calculus of inductive constructions, CIC) and have the proof assistant (Isabelle or Coq) check the derivation of the proof rules. They thus achieve the highest degree of trustworthiness in the proof rules. (For Isabelle's version of higher-order logic, Kunčar and Popescu proved consistency on paper [41–43].) Verypto falls behind because its measure theory is only axiomatised rather than constructed. For EasyCrypt, we are aware of neither a consistency proof for its underlying logic, in particular for the module system, nor of a derivation of its probabilistic relational Hoare logic.

Checking the individual proof steps, be it a security proof or the derivation of a proof rule, must be done by some program, the so-called trusted code base. Proof assistants like Isabelle and Coq perform this task in a small kernel, which has been carefully scrutinised and tested. Additionally, they can produce proof terms or proof objects that an independent checker can certify. Consequently, CryptHOL, CertiCrypt, FCF, and Verypto achieve high marks here. EasyCrypt has a small kernel, too. Early versions of EasyCrypt could export the relational proofs to CertiCrypt, but this connection has been dropped [78]. So there is no independent checker for EasyCrypt proofs, and when external SMT solvers are used, they must be trusted too.

In summary, CryptHOL, CertiCrypt, and FCF all follow the foundational approach to a large extent. More could be achieved by formally verifying the implementation of the proof checker and possibly even the hardware and software it is running on.

Automation The formalisation effort determines a framework's usability. For this comparison, we estimate the effort by lines of proof that the user must manually write, i.e. the input to the proof checker. Clearly, personal proof styles affect line counts, so the numbers must be taken with a grain of salt (for a discussion on the subtleties of comparing formalisation effort, see [9]). Nevertheless, they roughly indicate the effort required to produce such proofs. Table 1 lists the length (in lines) of the security statement for different cryptographic algorithms and frameworks. The line count includes the statement of the reduction, the concrete security theorem, its proof, and all intermediate games. It does not cover the cryptographic algorithm itself nor the security definition. We obtained the numbers by inspecting the proof scripts distributed with the frameworks. Unfortunately, there are no line counts for Verypto because we could not get access to the sources. For example, the 49 lines of the IND-CPA security proof for Elgamal in CryptHOL correspond to lines 1–44 in Fig. 2; the difference in line counts is due to changes in line breaking and whitespace. In particular, it includes the calls and hints to the proof engines. Hence, the line counts measure the degree of automation as better automation can fill in more details with less hints.

As Petcher and Morrisett have observed [65], shallow embeddings (FCF, CryptHOL) have an advantage over deep ones (CertiCrypt, Verypto), as all the reasoning infrastructure and libraries of the proof assistant can be reused directly; a deep embedding

Table 1. Framework comparison by line counts of the reduction definition and concrete security theorems.

Cryptographic algorithm	Security property	Crypt HOL	Certi Crypt	Easy Crypt[a]	FCF[b]	Shoup [pages]
RP-RF switching lemma		120	225	448		1.3
Elgamal (Sect. 3)	IND-CPA	49	238	68	156	1.8
Hashed Elgamal in the ROM	IND-CPA	253	789	216		
Encryption using a PRF [64]	IND-CPA	357			1167	
Extension of a PRF with a universal hash function [74]	pseudo-random	217				2.2
Encryption using a PRF and a UF (Sect. 6)	IND-CCA	616 + 224[c]				2.9

[a] Version 263740c on github.com/EasyCrypt/easycrypt.git, 19 Dec 2016
[b] Version adb0bdc on github.com/adampetcher/fcf, 11 Feb 2017
[c] The 224 lines formalise the generic reduction from Sect. 6.5, whereas the 616 lines are specific to the encryption scheme

would need to encode the libraries in the syntax of the language. Despite being more general, CryptHOL leads to shorter proofs than FCF. We see two reasons for this. First, our language works directly in the semantic domain, even for effectful programs. Thus, more program equalities hold and all conversions between syntax and semantics become superfluous. For example, the FCF rule for loop fission holds only in the relational logic, but it is a HOL equality in our model. Second, Isabelle's built-in proof automation, in particular conditional rewriting and the support for representation independence, provides a reasonable level of automation, especially for the equational proofs mentioned above. So far, we have not implemented any problem-specific proof engines that could automate the proofs even further. In comparison with EasyCrypt, the state of the art in proof automation, we achieve a similar degree of automation when reasoning about programs. The RP-RF switching lemma involves considerable reasoning about probabilities rather than programs.[12] Here, the CryptHOL proof is much shorter than EasyCrypt's because in CryptHOL we can leverage the richer Isabelle/HOL library and generic proof automation.

The last column in Table 1 lists the pages that Shoup [74] needs for proving the same statements on paper. Clearly, the expansion factor varies greatly. In his introductory example of Elgamal, Shoup is very explicit about every step and writes out the reductions.

[12] In CertiCrypt, the random permutation oracle

$$\mathcal{O}_{\mathsf{RP}}(D, x) \equiv \mathsf{case}\ D(x)\ \mathsf{of}$$
$$\quad \mathsf{None} \Rightarrow \mathbf{do}\ \{\ y \leftarrow \mathsf{repeat}\ \mathsf{uniform}(A)\ \mathsf{until}\ (\lambda y.\ y \notin \mathsf{ran}(D));$$
$$\qquad \mathsf{return}\ (y, D(x \mapsto y))\ \}$$
$$\mid \mathsf{Some}(r) \Rightarrow \mathsf{return}\ (r, D)$$

is formalised using a loop, where $\mathsf{repeat}\ p\ \mathsf{until}\ P$ satisfies

$$\mathsf{repeat}\ p\ \mathsf{until}\ P \equiv \mathbf{do}\ \{\ x \leftarrow p;\ \mathsf{if}\ P(x)\ \mathsf{then}\ \mathsf{return}\ x\ \mathsf{else}\ \mathsf{repeat}\ p\ \mathsf{until}\ P\ \}$$

and $\mathsf{ran}(D)$ denotes the range of the map D. CryptHOL and EasyCrypt use $y \leftarrow \mathsf{uniform}(A - \mathsf{ran}(D))$ instead of the loop. This requires an additional game hop, which accounts for 44 and 93 lines, respectively. These lines are included in the line counts in Table 1.

Therefore, our formal proof (Fig. 2) requires less space, but also omits all informal explanations. Conversely, even our informal presentation of the IND-CCA security proof (Sects. 6.4–6.5) is much longer than his proof simply because Shoup omits many formal details (see Sect. 6.6 for a comparison). Such liberties are not allowed in machine-checked proofs, which are therefore much more detailed than what is standard today for cryptographic security proofs. The flip side, of course, is that one does not have the liberty of making mistakes, either.

Naturality Naturality requires that the language supports recurring cryptographic idioms. One particularly important idiom is the adversary's black-box access to an oracle. CertiCrypt, FCF, and Verypto assume that the adversary is implemented in the language of the framework. Thus, the semantics and proof rules need not distinguish between the unknown code of the adversary and the user-defined games and reductions. For CertiCrypt and Verypto, in particular, the adversary is subject to the restrictions of the language. Access to an oracle can be modelled as an ordinary procedure call as described by Halevi, Bellare, and Rogaway [12,31] because the language ensures that there is only black-box access. In EasyCrypt, the language restrictions on the adversary are not clear as the adversary is formalised as an abstract module, but to our knowledge no formal semantics for the module system has been published. The implicit understanding is that adversaries are written in EasyCrypt's probabilistic while language.

In contrast, one of CryptHOL's distinguishing features is that it is not restricted to a specific language. So, we need other means to enforce black-box access. Our choice of GPVs appears to be at the right level of abstraction as the elegant composition properties indicate (14,15). They are clearly superior to Shoup's approach [74] where the game mediates the oracle access as follows: Upon activation, the adversary either returns its final output or a query that the game then feeds to the oracle. The response is then given to the adversary with the next activation. This approach is not modular, as every game needs to implement the dispatching in a loop, one cannot easily abstract over a game's oracles, and it makes it hard to capture reasoning about oracles in proof rules.

Security definitions should resemble those in the cryptographic literature so that cryptographers can understand and evaluate them. All frameworks achieve good scores in this regard, but in different ways. CertiCrypt and EasyCrypt embed an imperative procedural language in their logics. They closely model Bellare's and Rogaway's idea of a stateful language with oracles as procedures [13]. Verypto deeply embeds a higher-order language with mutable references based on hard-to-read de Bruijn indices in HOL. Readability is regained by reflecting the syntax in HOL's term language using parsing and pretty-printing tricks. In contrast, FCF and CryptHOL shallowly embed a functional language with monadic sequencing in Coq and Isabelle/HOL, respectively, which is close to standard mathematical notation. Like in Shoup's treatment [74], the state of the adversary and the oracles must be passed explicitly. This improves clarity as it makes explicit which party can access which parts of the state. Yet, it can also be a source of errors as the user must ensure that the states are only used linearly.

Mathematical background theories like number theory and algebra are important too. As CryptHOL, CertiCrypt, FCF, and Verypto are integrated with the general-purpose proof assistants Isabelle/HOL or Coq, they immediately benefit from the extensive libraries of formalised mathematics. Here, the shallow embedding of CryptHOL and FCF

has an advantage over CertiCrypt's and Verypto's formalised programming languages because in the latter the theorems from the library must first be transferred to the deep embedding, which can be non-trivial in practice. In comparison with Isabelle/HOL and Coq, EasyCrypt's library is small. As EasyCrypt is designed for reasoning about programs, not general mathematics, its usage in practice follows Halevi's idea that the tool checks only the "boring" game hops and leaves the challenging mathematical problems to human reviewers. Such transitions from formal to informal reasoning and back are particularly subtle, as humans must fully understand the semantics of the formal model to ensure that the informal reasoning is correct. In principle, CryptHOL also supports such mixing as the proofs are written declaratively and (unverified) LaTeX proofs can be embedded into the proof script. The crucial difference is that in CryptHOL, such informal proofs can later be made formal and checked within the same tool, so no translation gap necessarily exists.

Most frameworks support abstraction and reuse via a module system. CryptHOL, FCF, and CertiCrypt build on Isabelle's and Coq's module system. EasyCrypt can clone theories, which is less expressive than Isabelle **locale**s.

Finally, the embedding and the logic determine what kinds of security properties can be formalised. We now discuss this for all of the five frameworks. All frameworks support concrete security proofs in the style of Theorem 2. Thanks to their deep embeddings, CertiCrypt and Verypto also support statements about efficiency. Therefore, statements can quantify over all polynomially bounded adversaries and reductions can be proved efficient. Thus, the reductions, which are explicit in asymptotic security statements like Corollary 1, disappear in the quantifiers. This raises the level of abstraction, as the statement no longer refers to the reduction, which is internal to the proof. FCF comes with an axiomatic cost model for efficiency, which is not formally connected to an operational model. The cost model is only partial in that it just covers the operations needed in the reductions of the case studies. That is, it has been sufficiently developed to eliminate the reductions from the security statements of the case studies. However, the users themselves must change the cost model when they want to prove their own reductions efficient. In these three frameworks, asymptotic bounds are derived from concrete bounds. In EasyCrypt and CryptHOL, the efficiency of computations cannot be expressed formally because their logics identify terms up to computation: for example, the computation $(\lambda x.\ x + x)(1)$ cannot be distinguished from the value 2. However, we have started working on an operational execution model in HOL and connecting it to our shallow embedding using logical relations (Sect. 8).

Extensibility Extensibility demands trusted ways to add both new (stronger) proof rules and new language constructs to a framework. CryptHOL is the best in this category. Users can derive new proof rules from the semantics of language primitives when needed as demonstrated in Sects. 4.5 and 6.5, and Isabelle makes sure that the new rules are sound by checking the derivation. Moreover, users can define new language constructs simply by giving their semantics in the domain of spmfs and GPVs. Crucially, all existing theorems of the other operators, e.g. associativity of composition, remain valid thanks to the shallow embedding.

None of other frameworks supports the two kinds of trusted extensibility discussed above. In EasyCrypt, all proof rules are implemented in OCaml. Any extension thus

must change the implementation of EasyCrypt, so there are no mechanised checks. The language is hard-coded into EasyCrypt's implementation, too. CertiCrypt, FCF, and Verypto support the derivation of stronger proof rules like CryptHOL, but not language extensions due to the deep embedding. Users would have to change the language formalisation itself and then adapt all proofs of all proof rules that are proven by induction over the syntax. Even worse, if different users independently do such language extensions, their modified frameworks are incompatible. This problem does not arise in CryptHOL as it does not rely on syntax.

Technical comparison On the technical level, CryptHOL shares some similarities with EasyCrypt, CertiCrypt, and FCF.

The lifting operator $\widetilde{\mathbb{D}}(\_)$ for relations over probability distributions is also the basis of relational probabilistic Hoare logic pRHL [7] as used in EasyCrypt and CertiCrypt. Regarding its characterisation in Lemma 1, EasyCrypt's proof tactic `bypr` [26] implements a special case of the direction (c) $\longrightarrow$ (a), namely when the relation R is the identity. (In this case, quantifying over all events A simplifies to quantifying over all elementary events.) Similarly, EasyCrypt's tactic `proc B I` implements oracle bisimulation up to failure (Proposition 2). Moreover, `sim` decomposes two structurally similar programs and replaces equivalent parts similar to the representation independence proofs with Isabelle's *transfer* method. The difference is that `sim` handles only conjunctions of equalities, whereas *transfer* deals with arbitrary relations on the intermediate states.

In FCF, composition of GPVs inline is a primitive operation `run` for oracle computations, and our composition operator exec with an oracle is conceptually similar to FCF's semantics operator for oracle computations [64, Fig. 2]. As discussed under extensibility, the difference is that our operators are defined on the semantic domain and therefore work with all elements in the domain, not only with the given syntax.

EasyCrypt and CertiCrypt have global variables to store oracle and adversary state. FCF and CryptHOL provide only local variables and use explicit state passing instead. Consequently, adversaries with oracle access must be written as oracle computations or GPVs rather than procedures. This is because a state monad does not fit together well with *black-box* oracle access in a shallow embedding in HOL, as Petcher and Morrisett observed [64]. The problem is that we cannot give the oracle, represented as a state transformer, as a higher-order argument to an adversary. There is nothing that prevents the adversary from looking at the internals of the oracle, i.e. not using it as a black box. In deep embeddings such as CertiCrypt and EasyCrypt, the syntactic restrictions on adversaries ensure black-box access.

Clearly, we could restrict our semantic domain to those adversaries that use the given oracle only as a black box. Looking at the oracle's state could be prevented by requiring that the adversary be relationally parametric in the state. Relational parametricity expresses that the adversary behaves uniformly for all states, so it cannot inspect the state that it is given as an argument. Yet we must also ensure that the adversary uses the state only linearly, i.e. it does not rewind the oracle. Hofmann, Karbyshev, and Seidl [32] have studied this problem in a deterministic functional setting, without probabilities. If we extend their result, we essentially end up with a domain that is similar to GPVs: GPVs are probabilistic strategy trees. The advantage of our explicit construction as a

codatatype is that we can use this construction in our definitions and proofs, e.g. qbound and inline. This would not possible with a purely semantic characterisation.

7.2. *Other Related Work*

The tool CryptoVerif by Blanchet [18] can prove secrecy and correspondence properties such as authentication of security protocols. It can automatically decompose games into reductions and oracles and even discover the intermediate games themselves. Hence, the tool achieves a much higher degree of automation than any of the frameworks discussed in Sect. 7.1. Unfortunately, it satisfies none of the other desiderata. Users must trust that the whole implementation correctly implements the game transformations that have been proven correct in the transition system semantics just with pen and paper proofs. The tool lacks abstraction and extensibility as it supports only a fixed set of language primitives and game transformations. In particular, reasoning about probabilities or the semantics like in the RP-RF switching lemma (Sect. 5.1) is impossible. The language—a process calculus inspired by the π calculus—distinguishes between a unique output process and possibly many input processes, which communicate via channels. Our GPVs also distinguish between inputs and outputs, but composition works differently. In CryptoVerif, several input processes may be able to receive an output and the semantics picks one uniformly at random. In our setting, the callee represents all input processes and receives all calls. In principle, one could embed Blanchet's calculus in our semantic domain of GPVs using a different composition operator. CryptoVerif's abstractions could then be proven sound in our framework.

The functional programming language F* [14,79] has been used to verify implementations of cryptographic algorithms and protocols [5]. Security properties are formulated as type safety of an annotated, dependently-typed program and the type checker ensures type safety. While this approach scales to larger applications [17], the security properties cannot be stated concisely as the typing assertions are scattered over the whole implementation, and the approach relies on an external soundness result in the style of computational soundness.

Audebaud and Paulin-Mohring [2] formalised the spmf monad in Coq. They also define the approximation order $\sqsubseteq$ on spmfs and show that it forms an ω-complete partial order, i.e. *countable* chains have least upper bounds. Using Kleene's fixpoint theorem, they obtain a fixpoint operator for continuous functions. We generalise their result in that *arbitrary* spmf chains have least upper bounds. Thus, monotonicity (rather than continuity) suffices for the fixpoints.

CertiCrypt [8] uses Audebaud's and Paulin-Mohring's monad as the semantic domain for programs and adds the lifting operator $\widetilde{\mathbb{D}}$. Zanella Béguelin proves a special case of Theorem 1, where the functions f and g are projections of a joint continuous function [85].

Micciancio and Tessaro [57] propose to model multi-party protocols as systems of equations, which are interpreted as order-continuous functions that transform input streams to probability distributions over output streams. Like CryptHOL, their system of equations can be manipulated algebraically as the semantics is compositional. For example, composition is associative in their model, too. Yet, their model is only a notational framework, not a mechanised proof calculus. In particular, the syntax of equations lacks

any formal treatment, which is crucial for mechanised proof checking. Local variables of a process are therefore not really local and users themselves must watch out for unintended variable capture when they perform algebraic transformations, in particular as variable names determine the composition of processes. Moreover, there are no checks that recursive systems as they arise, e.g. from composition, are productive, i.e. finite amounts of outputs require only finite amounts of inputs, and measurable. A coalgebraic approach like ours might be a better semantic foundation than domain theory in this respect. Further, their framework ignores computational aspects and can therefore only be used for information-theoretically secure protocols where a simulator for the ideal functionality can emulate the real functionality exactly instead of indistinguishably.

Our framework reuses the existing infrastructure for relational parametricity in Isabelle/HOL, but in new ways. The Lifting package [34] exploits representation independence to transfer theorems between raw types and their quotients or subtypes. Lammich's tool AutoRef [44] uses transfer rules to refine abstract datatypes and algorithms to executable code. Blanchette et al. [19] use parametricity to express well-formedness conditions for operators under which corecursion may appear in corecursive functions. In contrast, we derive a relational logic for reasoning about shallowly embedded programs from parametricity and apply representation independence to replace oracles in games by bisimilar ones.

8. Conclusions and Future Work

We have presented the semantic foundations of our framework CryptHOL and demonstrated that it is well suited for formalising cryptographic notions and mechanically checking game-hopping proofs. Proofs conducted with CryptHOL in the proof assistant Isabelle/HOL are highly trustworthy as our framework strikes a good balance between rigour and comprehensibility and follows a foundational approach. Apart from the case studies in this paper, CryptHOL has also been used to formalise the security of multiparty computations [21,22].

Our framework cannot yet express efficiency notions such as polynomial run time. Hence, asymptotic reasoning can be used only in limited ways. This is the flipside of the shallow embedding in higher-order logic with all its benefits. For example, the type $\mathbb{D}(\omega)$ of subprobability distributions is extensional, i.e. two subdistributions are equal iff they assign the same probability mass to each elementary event. In particular, it does not matter how the distribution is computed. Accordingly, equational reasoning cannot be used for computationally indistinguishable distributions as one may replace the other only in polynomially bounded contexts. We have recently started to formalise an operational execution model in HOL that will be connected to our framework. This will allow us to make formal statements about the efficiency of HOL functions and GPVs, in particular about their run time being bounded by a polynomial. Then, asymptotic reasoning will be possible and (polynomial) bounds on execution time have to be proven only when needed.[13]

[13] As observed already by Bellare and Rogaway [12], the intermediate games in a sequence of game hops need not meet any computability or efficiency constraints. It is only when we apply a hardness assumption that

Although we have focused on game-hopping proofs, CryptHOL is also an excellent foundation for other structuring techniques like the ideal-real world paradigm, as embodied, e.g. in universal composability [23] and constructive cryptography [56]. For example, constructive cryptography has been formalised in Isabelle/HOL using CryptHOL [52].

Beyond cryptographic arguments, our semantic domain of generative probabilistic values could be applied in different contexts. In previous work [54], we used a (less abstract) precursor to model interactive programs in HOL. The domain could also serve as a basis for formalising and verifying CryptoVerif or as a backend for the new EasyCrypt, as EasyCrypt's logic and module system resemble Isabelle's.

Acknowledgements

Andreas Lochbihler and S. Reza Sefidgar were supported by the Swiss National Science Foundation Grant 153217 "Formalising Computational Soundness for Protocol Implementations".

Appendix A Further Background

This appendix provides further background to make the paper more accessible to cryptographers missing background in languages and logic. It is intended to enable them to convince themselves that cryptographic formalisations using CryptHOL have the indented meaning.

A.1 Justifying Recursive Function Definitions

Recursive function definitions must be justified to avoid inconsistencies. For example, the recursive specification $\mathsf{bad}(\circledast) \equiv 1 + \mathsf{bad}(\circledast)$ for a function $\mathsf{bad} : \mathbb{1} \Rightarrow \mathbb{N}$ must not be admitted. If it were, we could derive $0 = \mathsf{bad}(\circledast) - \mathsf{bad}(\circledast) = (1 + \mathsf{bad}(\circledast)) - \mathsf{bad}(\circledast) = 1 + (\mathsf{bad}(\circledast) - \mathsf{bad}(\circledast)) = 1 + 0 = 1$, a contradiction. So, the justification must prove that the recursion equation has a solution.

The recursive definitions in this paper are justified using three principles: well-founded recursion, least fixpoints, and primitive corecursion. Each definition principle yields a corresponding proof principle.

For well-founded recursion, the specification must be accompanied by a proof that the recursion terminates. Then, there is always a solution. Proofs about functions defined by well-founded recursion proceed by well-founded induction. In Isabelle/HOL, the **function** package [39] is based on well-founded recursion.

Footnote 13 continued
we must show that the reduction runs in polynomial time. Thus, by treating run-time constraints independently of the advantages, we avoid cluttering the proofs with unnecessary details about run time. This demonstrates that it is important that the semantic domains of games, adversaries, and reductions are not artificially restricted, e.g. by termination or efficiency constraints, as these constraints would have to be proven whenever a game is written down formally.

For least fixpoints, a recursive specification $f(x) \equiv \ldots f(\ldots) \ldots x \ldots$ of a function $f : \tau_1 \Rightarrow \tau_2$ is transformed into the associated functional $F(f, x) \equiv \ldots f(\ldots) \ldots x \ldots$, which is not recursive. If f's codomain τ_2 is a chain-complete partial order (i.e. there is a partial order $\sqsubseteq$ on τ_2 in which every totally ordered subset has a least upper bound) and $F : (\tau_1 \Rightarrow \tau_2) \Rightarrow (\tau_1 \Rightarrow \tau_2)$ is monotone, then the transfinite iteration of F starting at the least element leads to the least fixpoint of F, which is taken as the definition of f [40]. Proofs about least fixpoints use fixpoint induction, i.e. the property P must hold for the least element and be preserved by the functional F. Fixpoint induction also demands that P be admissible, i.e. whenever P holds for all elements of a non-empty, totally ordered subset A of $\tau_1 \Rightarrow \tau_2$, then P must also hold for the least upper bound of A.

In Isabelle/HOL, the **partial-function** command [40] is built on the least fixpoint principle. For example, suppose that bad had the type $\mathbb{1} \Rightarrow \mathbb{N}^\infty$ where $\mathbb{N}^\infty$ denotes the natural numbers extended with infinitely. The least fixpoint principle could then justify bad's definition because $\mathbb{N}^\infty$ with the usual order is a chain-complete partial order. We would get the solution $\mathsf{bad} \circledast = \infty$ for the recursive specification. The recursive definitions in Fig. 3 are all justified by the least fixpoint principle.

Primitive corecursion is explained in Appendix A.2.

A.2 Corecursion and Coinduction

CryptHOL heavily uses coinductive definitions and proofs to handle finite and infinite objects uniformly. To make the formalisation and proofs more accessible, we briefly review the coinductive concepts that we use and compare them to their inductive counterparts using a simple example. In a nutshell, coinduction is the dual to induction.

Coinductive Datatypes. To compare inductive and coinductive definitions, consider the construction rules for binary trees with integers at the leaves:

(a) If $n : \mathbb{Z}$, then $\mathsf{Leaf}(n)$ is a tree.
(b) If l and r are trees, then $\mathsf{Node}(l, r)$ is a tree.

If we want to define the set $\mathbb{T}$ of trees inductively, we say that $\mathbb{T}$ is the *smallest* set that is *closed* under the construction rules (a) and (b). This means that every tree in $\mathbb{T}$ is built in finitely many steps according to the above rules. Thus, $\mathbb{T}$ contains all finite binary trees. This yields an (inductive) datatype:

datatype $\mathbb{T} = \mathsf{Leaf}(\mathbb{Z}) \mid \mathsf{Node}(\mathbb{T}, \mathbb{T})$.

Conversely, in a coinductive definition, we say that $\overline{\mathbb{T}}$ is the *largest* set that is *consistent* with the rules (a) and (b). This means that $\overline{\mathbb{T}}$ contains everything where we cannot rule out in finitely many steps that it was not built according to the above rules. Thus, every tree can be decomposed into either a leaf or a node with two child trees. Additionally, we identify all elements in $\overline{\mathbb{T}}$ that cannot be distinguished by finitely many decomposition steps, i.e. bisimilar trees are considered identical. Then, $\overline{\mathbb{T}}$ is the set of all finite and infinite binary trees. This gives a coinductive datatype (or codatatype):

codatatype $\overline{\mathbb{T}} = \mathsf{Leaf}(\mathbb{Z}) \mid \mathsf{Node}(\overline{\mathbb{T}}, \overline{\mathbb{T}})$.

Codatatypes provide a natural way to model computations. One can only observe finite parts of a computation's behaviour, just like one can only decompose codatatypes to a finite depth. In contrast, datatypes model finite objects that can be completely analysed.

Corecursion Recursion is the primary definition principle for functions that take a datatype value as an *argument*, and corecursion is for functions that *return* a codatatype value. For example, the function d below on the left computes the depth of a finite binary tree as a natural number, which is modelled as a datatype in Peano style. The function max (not shown) returns the maximum of its two arguments. This recursive definition is acceptable because the recursion *terminates*: the subtrees l and r given to the recursive calls are strictly smaller than $\mathsf{Node}(l, r)$ since all trees in $\mathbb{T}$ are finite. In fact, it is even primitively recursive, because the recursive calls are on the direct arguments l and r of the constructor Node. In this view, d destructs its argument.

datatype $\mathbb{N} = 0 \mid \mathsf{Suc}(\mathbb{N})$	**codatatype** $\mathbb{N}^\infty = 0 \mid \mathsf{Suc}(\mathbb{N}^\infty)$
$\mathsf{d} : \mathbb{T} \Rightarrow \mathbb{N}$	$\overline{\mathsf{d}} : \overline{\mathbb{T}} \Rightarrow \mathbb{N}^\infty$
$\mathsf{d}(\mathsf{Leaf}(\_)) = 0$	$\overline{\mathsf{d}}(\mathsf{Leaf}(\_)) = 0$
$\mathsf{d}(\mathsf{Node}(l, r)) = \mathsf{Suc}(\mathsf{max}(\mathsf{d}(l), \mathsf{d}(r)))$	$\overline{\mathsf{d}}(\mathsf{Node}(l, r)) = \mathsf{Suc}(\mathsf{max}(\overline{\mathsf{d}}(l), \overline{\mathsf{d}}(r)))$

On the right, $\overline{\mathsf{d}}$ computes the depths of trees in $\overline{\mathbb{T}}$. As these trees may be infinite, $\overline{\mathsf{d}}$'s return type is the natural numbers extended with infinity, which we model as the Peano codatatype (the infinite term $\mathsf{Suc}(\mathsf{Suc}(\mathsf{Suc}(\ldots)))$ represents infinity). This recursive definition cannot be justified by a termination argument because the recursion does not terminate for infinite trees. Instead, corecursion on $\mathbb{N}^\infty$ is the right definition principle. To this end, one must check that $\overline{\mathsf{d}}$ is *productive*, i.e. whenever we want to inspect $\overline{\mathsf{d}}$'s output to some finite depth, $\overline{\mathsf{d}}$ decomposes its argument $t : \overline{\mathbb{T}}$ only to some finite depth. Here, to produce one Suc constructor, $\overline{\mathsf{d}}$ peels off one Node constructor. This definition is not primitively corecursive, because the context of the corecursive calls $\overline{\mathsf{d}}(l)$ and $\overline{\mathsf{d}}(r)$ does not consist of only one Suc constructor, but also includes the max function.

Coinduction. Induction requires an inductive *assumption* like "t is a finite tree": If we can prove that a property P holds for all $\mathsf{Leaf}(n)$ and is preserved by $Node$s, i.e. P is closed under the tree construction rules (a) and (b), then P holds for all finite trees. This proof principle is valid because $\mathbb{T}$ is the smallest set closed under the construction rules.

In contrast, coinduction establishes a coinductive *conclusion* like "two trees are equal". If we can prove that a relation R is consistent with the tree construction rules (a) and (b), then R-related trees are equal. This proof principle is valid because equality is the largest relation consistent with the construction rules (as bisimilar trees are identified).

For example, let mirror be a function which swaps all left and right subtrees of all nodes in a tree. We prove $\forall t : \mathbb{T}.\ \mathsf{d}(\mathsf{mirror}(t)) = \mathsf{d}(t)$ by induction on the inductive assumption $t : \mathbb{T}$. Conversely, $\forall t : \overline{\mathbb{T}}.\ \overline{\mathsf{d}}(\mathsf{mirror}(t)) = \mathsf{d}(t)$ is proven by coinduction on the coinductive equality relation on $\mathbb{N}^\infty$.

A.3 Relational Parametricity and Representation Independence

A function like $\pi_1 : \alpha \times \beta \Rightarrow \alpha$ is polymorphic in a type variable α if its behaviour does not depend on the actual type of the argument, i.e. the function behaves uniformly for all type instances. Reynolds [69] and Wadler [80] formalised this notion of independence

using relational parametricity. They interpret the type of a function as a relation between values instead of a set of values. Type constructors without type arguments like $\mathbb{B}$, $\mathbb{N}$, and $\mathbb{R}$ become the identity relation on the set of values, which we denote by $\widetilde{\mathbb{B}}$, $\widetilde{\mathbb{N}}$, and $\widetilde{\mathbb{R}}$, respectively. Type constructors with type arguments like $\times$ and $\Rightarrow$ become relation transformers, i.e. functions from relations to relations, which we write with a $\widetilde{}$ over the type constructor. For example, $(\widetilde{\times}) : \mathbb{P}(\alpha \times \alpha') \Rightarrow \mathbb{P}(\beta \times \beta') \Rightarrow \mathbb{P}((\alpha \times \beta) \times (\alpha' \times \beta'))$ lifts two relations component-wise to pairs. Formally, $(x, y)\ (A \mathbin{\widetilde{\times}} B)\ (x', y')$ iff $x\ A\ x'$ and $y\ B\ y'$. Similarly, $A \mathbin{\widetilde{\Rightarrow}} B$ for the function space relates two functions f and g iff they transform relatedness in A into relatedness in B. Formally, $f\ (A \mathbin{\widetilde{\Rightarrow}} B)\ g$ iff $f(x)\ B\ g(y)$ whenever $x\ A\ y$. Now, a function is relationally parametric in a type variable α iff it is related to itself in the relation that corresponds to its type where α is interpreted by an arbitrary relation A. For example, π_1 being parametric in α and β is expressed by

$$\forall A\ B.\ \pi_1\ (A \mathbin{\widetilde{\times}} B \mathbin{\widetilde{\Rightarrow}} A)\ \pi_1. \tag{18}$$

Note the similarity between π_1's type and the relation. We list the relators for further type constructors:

> **Sums** $\mathsf{Left}(x)\ (A \mathbin{\widetilde{+}} B)\ \mathsf{Left}(y)$ iff $x\ A\ y$, and $\mathsf{Right}(x)\ (A \mathbin{\widetilde{+}} B)\ \mathsf{Right}(y)$ iff $x\ B\ y$, but $\widetilde{+}$ does not relate $\mathsf{Left}(\_)$ and $\mathsf{Right}(\_)$.
> **Sets** $X\ \widetilde{\mathbb{P}}(R)\ Y$ iff $R[X] \subseteq Y$ and $R^{-1}[Y] \subseteq X$.
> **Maybe** $\widetilde{\mathbb{M}}(A)$ relates None to None, and $\mathsf{Some}(x)$ to $\mathsf{Some}(y)$ whenever $x\ A\ y$, and nothing else.

Wadler [80] proved that all functions definable in the polymorphic lambda calculus are parametric. He also demonstrated that adding polymorphic equality destroys this property. Higher-order logic has polymorphic equality ($=$) and description operators, so not all HOL functions are parametric. Thus, parametricity is not a free theorem in our setting; we must prove it for every constant. Fortunately, function application and function composition preserve parametricity. Hence, functions defined in terms of parametric functions are parametric too.

Wadler also showed that if all types are ω-ccpos, all functions are continuous and all relations are admissible and strict, then the fixpoint operator (defined by countable iteration) is parametric [80]. We do not consider the fixpoint operator as part of the language itself, but as a definition principle for recursive user-specified functions. That is, we assume that fix is always applied to a monotone function. Thus, preservation of parametricity (Theorem 1) suffices and we do not need Wadler's restrictions of the semantic domains. So, monotonicity (instead of continuity as discussed in Sect. 4.2) is expressed as a precondition on the given functions.

Wadler's free theorems express that parametric functions commute with the lift operation $\widehat{\ }$. For example, we get $\pi_1(xy \rhd (f \mathbin{\widehat{\times}} g)) = f(\pi_1(xy))$ from (18) by instantiating A and B with the graphs of the functions f and g, respectively, and by unfolding the definitions of $\widehat{\times}$ and the relators. Wadler calls this a free theorem because we do not need any property of π_1 other than (18); in particular, we do not need to unfold π_1's definition. In this tiny example, it might have been easier to unfold the definition, but

for complicated function definitions, proofs by parametricity are more systematic and therefore automatic.

Representation independence [60] expresses that changing the implementation of a module in a program does not affect the overall result. Mitchell uses parametricity to formalise this notion as follows. If we can find a bisimulation relation for all functions in the interface of the module being changed, and the remainder of the program is parametric in the type defined by the module, then the change does not affect the program. That is, representation independence lifts bisimilarity over contexts using parametricity. For example, consider a deterministic transducer given by its transition function $\delta : \sigma \Rightarrow \alpha \Rightarrow \beta \times \sigma$, which, given a state $s : \sigma$ and a input letter $x : \alpha$, computes an output letter from the output alphabet β and the successor state. This is the interface of the module. The recursive function run_δ given below computes a run of the transducer with transition function δ from state s on the input word $w : \mathbb{L}(\alpha)$.

$$\mathsf{run}_\delta(s, [\,]) \equiv [\,] \qquad \mathsf{run}_\delta(s, x \cdot w) \equiv \mathsf{let}\ (y, s') = \delta(s, x)\ \mathsf{in}\ y \cdot \mathsf{run}_\delta(s', w).$$

This is the context of the transducer module. Note that run is parametric in the state space, i.e.

$$\mathsf{run}\ ((S \mathrel{\widetilde{\Rightarrow}} (=) \mathrel{\widetilde{\Rightarrow}} (=) \mathrel{\widetilde{\times}} S) \mathrel{\widetilde{\Rightarrow}} S \mathrel{\widetilde{\Rightarrow}} (=) \mathrel{\widetilde{\Rightarrow}} (=))\ \mathsf{run}$$

for all S. Now, representation independence says that we can replace the transducer δ_1 by a bisimilar one δ_2 (and the initial state s_1 by s_2) without changing the overall result. Here, bisimilarity means that we find a bisimulation relation S between the states of the two transducers, i.e. (i) S relates the initial states s_1 and s_2, and (ii) whenever S relates two states s_1 and s_2 and for all inputs x, the transition functions $\delta(s_1, x)$ and $\delta(s_2, x)$ return the same output and successor states which S relates. As parametricity is compositional, this result scales to arbitrarily large contexts, not just a small function like run.

In Isabelle, the transfer package [34] implements representation independence as a proof engine. That is, having proven bisimilarity of two particular transition functions, a single invocation of this engine lifts the bisimilarity property over arbitrarily large contexts.

B Proofs

B.1 Subprobabilities are a CCPO

Proof of Proposition 1. We have to show that $\sqsubseteq$ is a partial order and that $\bigsqcup Y$ is well defined and the least upper bound for every chain Y, i.e. every set of spmfs all of whose elements are comparable in $\sqsubseteq$. The difficult part is to show that $\bigsqcup Y$ is well defined. In particular, we must show that the support of $\bigsqcup Y$ is countable even if Y is uncountable. Then, it is not hard to see that $\bigsqcup Y$ sums up to at most 1.

Clearly, we have $\mathsf{support}\left(\bigsqcup Y\right) = \cup_{p \in Y} \mathsf{support}(p)$. Yet, the union of an uncountable sequence of increasing countable sets need not be countable in general. In the following, we show that even for uncountable chains Y of spmfs, the union of the sup-

ports remains countable. To this end, we identify a countable sub-sequence of Y whose lub has the same support. The key idea is that for $\sqsubseteq$-comparable spmfs p and q, the order can be decided by looking only at the assigned probability masses, namely $p \sqsubseteq q$ iff $\|p\| \leq \|q\|$. So suppose without loss of generality that Y does not contain a maximal element (otherwise, the lub is the maximal element and we are done). The set of assigned probability masses $A = \{ \|p\| \mid p \in Y \}$ has a supremum $r \leq 1$, as 1 bounds the set from above. The closure of A contains the supremum r, so A must contain a countable increasing sequence which converges to r. This sequence gives rise to a countable sub-sequence Z of Y, for which we show $(\cup_{p \in Y} \mathsf{support}(p)) \subseteq (\cup_{q \in Z} \mathsf{support}(q))$. For any $p \in Y$, there is a $q \in Z$ such that $\|p\| \leq \|q\|$, as the assigned probability masses in Z converge to r from below and p is not maximal. Hence, $p \sqsubseteq q$ as p and q are related in $\sqsubseteq$, and therefore $\mathsf{support}(p) \subseteq \mathsf{support}(q)$ as $\mathsf{support}$ is monotone. □

The attentive reader might wonder why we need transfinite iteration for the fixpoint despite having shown that uncountable chains can be reduced to countable ones for the purpose of lubs. Countable fixpoint iteration, which defines the least fixpoint as $\bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}$, does not suffice. (Here, function iteration is defined by $f^0 = \mathsf{id}$ and $f^{n+1} = f \circ f^n$.) The reason is that the chain $\{ f^i(\bot) \mid i \in \mathbb{N} \}$ might stop before the least fixpoint is reached. Consider, e.g. the monotone spmf transformer $f : \mathbb{D}(\mathbb{1}) \Rightarrow \mathbb{D}(\mathbb{1})$ given by

$$f(p)\,!\,x = \mathsf{if}\ p\,!\,x < \tfrac{1}{2}\ \mathsf{then}\ \tfrac{2 \cdot p!x+1}{4}\ \mathsf{else}\ 1.$$

The countable iteration of f starting at $\bot$ yields a sequence of spmfs which assign to $\circledast$ the masses 0, 1/4, 3/8, 7/16, 15/32, … The least upper bound of this sequence assigns 1/2 to $\circledast$. That is, the iteration has not yet reached f's fixed point, which assigns the mass 1 to $\circledast$. This is because f is not (chain) continuous, i.e. it does not preserve lubs.

B.2 Fixpoints Preserve Parametricity

A relation $R : \mathbb{P}(A \times B)$ is *admissible* w.r.t. two ccpos iff for any chain $Y \subseteq R$ of pairs in the product ccpo (the ordering is component-wise), R relates the component-wise lubs of Y.

Proof of Theorem 1. We prove Theorem 1 by parallel induction on the two fixpoints. Both inductive cases are trivial. The base case requires $\widetilde{\mathbb{D}}(R)$ to be strict, i.e. it relates the least elements, which holds trivially. The step case is precisely the relatedness condition of f and g, which the theorem assumes. Parallel fixpoint induction is a valid proof principle because $\widetilde{\mathbb{D}}(R)$ is admissible by Proposition 4. □

Proposition 4. $\widetilde{\mathbb{D}}(\_)$ *is admissible.*

Proof. We exploit the characterisation of $\widetilde{\mathbb{D}}$ in terms of $\mathcal{P}[\_ \in \_]$. We must show $(\bigsqcup (Y \rhd \widehat{\mathbb{P}}\ \pi_1))\ \widetilde{\mathbb{D}}(R)\ (\bigsqcup (Y \rhd \widehat{\mathbb{P}}\ \pi_2))$ for all chains Y of pairs in $\widetilde{\mathbb{D}}(R)$. By the characterisation of $\widetilde{\mathbb{D}}$ (Lemma 1), this holds as follows: The first and last steps exploit that the lub commutes with $\mathcal{P}[\_ \in \_]$, and the inequality follows from monotonicity of SUP and

the characterisation of $\widetilde{\mathbb{D}}$ for elements of the chain.

$$\mathcal{P}\Big[\bigsqcup (Y \rhd^{\widehat{\mathbb{P}}} \pi_1) \in A\Big] = \mathsf{SUP}_{(p,\_)\in Y}\, \mathcal{P}[p \in A]$$
$$\leq \mathsf{SUP}_{(\_,q)\in Y}\, \mathcal{P}[q \in R[A]] = \mathcal{P}\Big[\bigsqcup (Y \rhd^{\widehat{\mathbb{P}}} \pi_2) \in R[A]\Big].$$

□

Note that it is not clear how to prove admissibility via the characterisation in terms of couplings. The problem is that the couplings for the pairs in the chain need not form a chain themselves. So we cannot construct the coupling for the lubs as the lub of the couplings.

Admissibility of relators is preserved by the function space (ordered point-wise) and products (ordered component-wise).

C Guide for CryptHOL source syntax

The notation in this article is intended to make the presentation easy to follow. The CryptHOL source code and examples in the Archive of Formal Proofs [49,53] use a somewhat different notation that follows Isabelle/HOL conventions. To help the reader to navigate the sources, this appendix relates the different notations.

Type constructors are written as a single letter in blackboard style in this article, whereas they are spelled out in words in the sources. Table 2 lists the correspondences. The notation for infix type constructors (products $\times$, sums $+$, and function space $\Rightarrow$) remains the same. Moreover, type constructors are written in postfix in Isabelle and type variables start with a prime $'$, and type annotations use a double colon :: instead of a single colon :. For example, what is written $p : \mathbb{D}(\mathbb{L}(\alpha) \times \beta)$ in this article becomes $p :: (\alpha\ \mathsf{list} \times \beta)\ \mathsf{spmf}$ in the sources.

Function application is written without parentheses in Isabelle, but compound arguments must be parenthesised. Forward function application is rarely used. For example,

Table 2. Notation for type constructors in this article and in the sources. The type constructor spmf abbreviates option pmf in Isabelle/HOL .

Description	Article	Sources
Boolean	$\mathbb{B}$	bool
Natural number	$\mathbb{N}$	nat
Real number	$\mathbb{R}$	real
Singleton type	$\mathbb{1}$	unit
Optional value	$\mathbb{M}$	option
List	$\mathbb{L}$	list
Set	$\mathbb{P}$	set
Subprobability mass function	$\mathbb{D}$	spmf
Generative probabilistic value	$\mathbb{G}$	gpv
Reactive probabilistic value	$\mathfrak{I}$	rpv

Table 3. Notation comparison for functions in this article and the CryptHOL sources.

Description	Article	Sources
Singleton value	$\circledast$	()
Projection on pairs	π_1, π_2	fst, snd
Injection into sums	Left, Right	Inl, Inr
Length of a list	$\|\_\|$	length
List concatenation	$(+\!\!+)$	(@)
Bitwise exclusive-or	$(\oplus)$	([⊕])
Bitstrings of length n	$\{0,1\}^n$	nlists UNIV n
Prefix of the natural numbers	$\mathbb{Z}\_$	{0..<_}
Set cardinality	$\|\_\|$	card
Group exponentiation	$(\hat{}\,)$	([^])
Group order	$\|\_\|$	order
One-point distribution	return	return-spmf
Coin flip	coin	coin-spmf
Uniform distribution	uniform	spmf-of-set
Failure distribution	$\bot$	return-pmf None
Assertion distribution	assert	assert-spmf
Weight	$\|\_\|$	weight-spmf
Support	support	set-spmf
Probability of elementary event	(!),$\mathcal{P}[\_ = \_]$	spmf
Probability of event	$\mathcal{P}[\_ \in \_]$	measure (measure-spmf _)
Lossless distribution	lossless	lossless-spmf
Approximation order	$(\sqsubseteq)$	ord-spmf (=)
Fixpoint operator	fix	ccpo.fixp
Pure computation	return	Done
Oracle call	call	Pause _ Done
Sampling	sample	lift-spmf
Assertion computation	assert	assert-spmf
Interaction bound	qbound_	interaction-bound
Lossless computation	$\text{lossless}_{\mathbb{G}}$	colossless-gpv
Inlining GPVs	search, inline	inline1, inline
Running GPVs	exec	exec-gpv

$f(arg_1, arg_2, g(x, y), arg_4) \rhd h\ z$ becomes $h\ z\ (f\ arg_1\ arg_2\ (g\ x\ y)\ arg_4)$. Indices and function parameters in superscripts become ordinary function arguments, too. For instance, $\mathcal{O}_{\mathsf{enc}}^{k,b}(L, (m_1, m_0))$ in Sect. 6.1 is a HOL function applied to four arguments: oracle-encrypt $k\ b\ L\ (m_1, m_0)$, namely a key k, a bit b, a storage L, and a pair of messages (m_1, m_0). In this article, we have moved the first two into superscripts to indicate that they are actually function parameters. Relations are modelled as binary predicates, i.e. functions of type $\alpha \Rightarrow \beta \Rightarrow \mathbb{B}$, rather than sets of pairs, and are usually written prefix instead of infix.

Locale instantiation in Isabelle marks the name prefix with a single colon : instead of $\equiv$ and locale parameters are not parenthesised, like for functions. For example, **sublocale** concrete $\equiv$ ELGAMAL$(\mathcal{G}(\eta))$ **for** η from Fig. 2 is actually written as **sublocale** concrete : ELGAMAL "$\mathcal{G}\ \eta$" **for** η.

Lifting of functions and relations over type constructors is expressed as ordinary HOL functions with the following naming convention. A function map-τ lifts func-

Table 4. File names for the sources for the examples in this article.

Example	File name
RP-RF switching lemma	`RP_RF.thy`
Elgamal (Sect. 3)	`Elgamal.thy`
Hashed Elgamal in the ROM	`Hashed_Elgamal.thy`
Encryption using a PRF [64]	`PRF_IND_CPA.thy`
Extension of a PRF with a universal hash function [74]	`PRF_UHF.thy`
Encryption using a PRF and a UF (Sect. 6)	`PRF_UPF_IND_CCA.thy`
Recursive distribution definitions (Fig. 3)	`Bernoulli.thy`
	`Geometric.thy`
	`Fast_Dice_Roll.thy`

The files are available in the Archive of Formal Proofs [53]. The recursive distribution definitions are in [50]

tions over the type constructor τ, and the operator rel-τ does so for relations. For example, $\widehat{\mathbb{D}}(f)$ and $\widetilde{\mathbb{D}}(R)$ are written as map-spmf f and rel-spmf R, respectively. The exception are lists with map and list-all2, pairs with map-prod and rel-prod, sums with map-sum and rel-sum, and functions with $(-\!\!-\!\!->)$ and $(=\!\!=\!\!=\!\!=>)$. For example, relational parametricity of the first projection on pairs (18) is written as $\forall A\ B.$ (rel-prod $A\ B$ $=\!\!=\!\!=\!\!=>$ A) fst fst, where fst is Isabelle's notation for π_1.

Some functions have different names and notation in the CryptHOL sources. Table 3 lists the most important differences. Changes in capitalisation and for definitions in the examples and case studies are not shown. Table 4 lists the files where the examples can be found in the sources.

References

[1] G. Asharov, A. Beimel, N. Makriyannis, E. Omri, Complete characterization of fairness in secure two-party computation of boolean functions, in Dodis, Y., Nielsen, J.B. (eds.) *TCC 2015*. LNCS, vol. 9014, (Springer, 2015), pp. 199–228

[2] P. Audebaud, C. Paulin-Mohring, Proofs of randomized algorithms in Coq. *Sci. Comput. Program.* **74**(8), 568–589 (2009)

[3] F. Baader, T. Nipkow, Term Rewriting and All That. Cambridge University Press (1998)

[4] M. Backes, M. Berg, D. Unruh, A formal language for cryptographic pseudocode, in *LPAR 2008*. LNCS, vol. 5330, (Springer, 2008), pp. 353–376

[5] G. Barthe, C. Fournet, B. Grégoire, P.Y. Strub, N. Swamy, S. Zanella Béguelin, Probabilistic relational verification for cryptographic implementations. in *POPL 2014*. (ACM, 2014) pp. 193–205

[6] G. Barthe, B. Grégoire, S. Heraud, S. Zanella Béguelin, Computer-aided security proofs for the working cryptographer. in *CRYPTO 2011*. LNCS, vol. 6841, (Springer 2011), pp. 71–90

[7] G. Barthe, B. Grégoire, J. Hsu, P.Y. Strub, Coupling proofs are probabilistic product programs. in *POPL 2017*. (ACM, 2017), pp. 161–174

[8] G. Barthe, B. Grégoire, S. Zanella Béguelin, Formal certification of code-based cryptographic proofs. in *POPL 2009*. (ACM, 2009), pp. 90–101

[9] D. Basin, M. Kaufmann, The Boyer-Moore prover and Nuprl: An experimental comparison. in Huet, G., Plotkin, G. (eds.) *Logical Frameworks*. (Cambridge University Press, 1991), pp. 89–119

[10] M. Bellare, A. Boldyreva, S. Micali, Public-key encryption in a multi-user setting: Security proofs and improvements. in Preneel, B. (ed.) *EUROCRYPT 2000*. LNCS, vol. 1807, (Springer, 2000), pp. 259–274

[11] M. Bellare, P. Rogaway, Optimal asymmetric encryption. in *Workshop on the Theory and Application of Cryptographic Techniques*. (Springer, 1994), pp. 92–111

[12] M. Bellare, P. Rogaway, Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331 (2004), http://eprint.iacr.org/2004/331
[13] M. Bellare, P. Rogaway, The security of triple encryption and a framework for code-based game-playing proofs. in *EUROCRYPT 2006*. LNCS, vol. 4004, (Springer, 2006), pp. 409–426
[14] J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, S. Maffeis, Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.* **33**(2), 8:1–8:45 (2011)
[15] M. Berg, Formal verification of cryptographic security proofs. Ph.D. thesis, Universität des Saarlandes (2013)
[16] S. Berghofer, M. Wenzel, Logic-free reasoning in Isabelle/Isar. in Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) *CICM 2008*. LNCS, vol. 5144, (Springer, 2008), pp. 355–369
[17] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.Y. Strub, Implementing TLS with verified cryptographic security. in *S&P 2013*. (IEEE, 2013), pp. 445–459
[18] B. Blanchet, A computationally sound mechanized prover for security protocols. *IEEE Trans. Dependable Secure Comput.* **5**(4), 193–207 (2008)
[19] J.C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, D. Traytel, Friends with benefits: Implementing corecursion in foundational proof assistants. in Yang, H. (ed.) *ESOP 2017*. LNCS, (Springer 2017), pp. 111–140
[20] J.C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, Popescu, A., D. Traytel, Truly modular (co)datatypes for Isabelle/HOL. in *ITP 2014*. LNCS, vol. 8558, (Springer, 2014), pp. 93–110
[21] D. Butler, D. Aspinall, A. Gascon, How to simulate it in Isabelle: Towards formal proof for secure multi-party computation (2017), accepted at ITP 2017
[22] D. Butler, D. Aspinall, A. Gascón, On the formalisation of Σ-protocols and commitment schemes. in Nielson, F., Sands, D. (eds.) *POST 2019*. LNCS, vol. 11426, (Springer, 2019), pp. 175–196
[23] R. Canetti, Universally composable security: A new paradigm for cryptographic protocols. in *Proceedings of 42nd IEEE Symposium on Foundations of Computer Science, 2001*. (IEEE, 2001), pp. 136–145
[24] A. Church, A formulation of the simple theory of types. *J. Symb. Log.* **5**(2), 56–68 (1940)
[25] R. Cohen, S. Coretti, J. Garay, V. Zikas, Probabilistic termination and composability of cryptographic protocols. in Robshaw, M., Katz, J. (eds.) *CRYPTO 2016*. LNCS, vol. 9816, (Springer ,2016), pp. 240–269
[26] Easycrypt: Reference manual. https://www.easycrypt.info/documentation/refman.pdf (2018), version 1.x, 19 February 2018
[27] T. Elgamal, A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory* **31**(4), 469–472 (1985)
[28] S. Goldwasser, S. Micali, Probabilistic encryption. *Journal of Computer and System Sciences* **28**(2), 270–299 (1984)
[29] S.D. Gordon, C. Hazay, J. Katz, Y. Lindell, Complete fairness in secure two-party computation. *J. ACM* **58**(6), 24:1–24:37 (2011)
[30] O. Grumberg, N. Francez, S. Katz, Fair termination of communicating processes. in *PODC 1984*. (ACM, 1984), pp. 254–265
[31] S. Halevi, A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181 (2005)
[32] M. Hofmann, A. Karbyshev, H. Seidl, What is a pure functional? in Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) *ICALP 2010*. LNCS, vol. 6199, (Springer 2010), pp. 199–210
[33] J. Hölzl, A. Lochbihler, D. Traytel, A formalized hierarchy of probabilistic system types. in *ITP 2015*. LNCS, vol. 9236, (Springer, 2015), pp. 203–220
[34] B. Huffman, O. Kunčar, Lifting and Transfer: A modular design for quotients in Isabelle/HOL. in *CPP 2013*. LNCS, vol. 8307, (Springer, 2013), pp. 131–146
[35] J. Hurd, A formal approach to probabilistic termination. in *TPHOLs 2002*. LNCS, vol. 2410, (Springer, 2002), pp. 230–245
[36] J. Kilian, P. Rogaway, How to protect DES against exhaustive key search (an analysis of DESX). *Journal of Cryptology***14**(1), 17–35 (2001)
[37] D.E. Knuth, A.C. Yao, The complexity of nonuniform random number generation. in Traub, J.F. (ed.) *Algorithms and Complexity—New Directions and Recent Results*. (Academic Press, Inc., 1976), pp. 357–428

[38] N. Koblitz, A.J. Menezes, Another look at "provable security". *Journal of Cryptology* **20**(1), 3–37 (2007)
[39] A. Krauss, Automating Recursive Definitions and Termination Proofs in Higher-Order Logic. Ph.D. thesis, Technische Universität München (2009)
[40] A. Krauss, Recursive definitions of monadic functions. in *PAR 2010*. EPTCS, vol. 43, pp. 1–13 (2010)
[41] O. Kunčar, A. Popescu, A consistent foundation for Isabelle/HOL. in Urban, C., Zhang, X. (eds.) *ITP 2015*. LNCS, vol. 9236, (Springer, 2015), pp. 234–252
[42] O. Kunčar, A. Popescu, Comprehending Isabelle/HOL's consistency. in Yang, H. (ed.) *ESOP 2017*. LNCS, vol. 10201, (Springer, 2017), pp. 724–749
[43] O. Kunčar, A. Popescu, Safety and conservativity of definitions in HOL and Isabelle/HOL. in *POPL 2018*. Proc. ACM Program. Lang., vol. 2, (ACM, 2017), pp. 24:1–24:26
[44] P. Lammich, Automatic data refinement. in *ITP 2013*. LNCS, vol. 7998, (Springer, 2013), pp. 84–99
[45] K.G. Larsen, A. Skou, Bisimulation through probabilistic testing. **Inf. Comp. 94**(1), 1–28 (1991)
[46] T. Lindvall, Lectures on the Coupling Method. Dover Publications, Inc. (2002)
[47] A. Lochbihler, A formal proof of the max-flow min-cut theorem for countable networks. Archive of Formal Proofs (2016), http://isa-afp.org/entries/MFMC_Countable.shtml, Formal proof development
[48] A. Lochbihler, Probabilistic functions and cryptographic oracles in higher order logic. in Thiemann, P. (ed.) *Programming Languages and Systems (ESOP 2016)*. LNCS, vol. 9632, (Springer, 2016), pp. 503–531
[49] A. Lochbihler, CryptHOL. Archive of Formal Proofs (2017), http://isa-afp.org/entries/CryptHOL.shtml, Formal proof development
[50] A. Lochbihler, Probabilistic while loop. Archive of Formal Proofs (2017), http://isa-afp.org/entries/Probabilistic_While.html, Formal proof development
[51] A. Lochbihler, S.R. Sefidgar, A tutorial introduction to CryptHOL. Cryptology ePrint Archive, Report 2018/941 (2018), https://eprint.iacr.org/2018/941
[52] A. Lochbihler, S.R. Sefidgar, D.A. Basin, U. Maurer, Formalizing constructive cryptography using CryptHOL. in *CSF 2019*. (IEEE Computer Society, 2019), pp. 152–166
[53] A. Lochbihler, S.R. Sefidgar, B. Bhatt, Game-based cryptography in HOL. Archive of Formal Proofs (2017), http://isa-afp.org/entries/Game_Based_Crypto.shtml, Formal proof development
[54] A. Lochbihler, M. Züst, Programming TLS in Isabelle/HOL. Isabelle Workshop 2014 (2014)
[55] J. Lumbroso, Optimal discrete uniform generation from coin flips, and applications. CoRR arXiv:1304.1916 (2013)
[56] U. Maurer, Constructive cryptography – a new paradigm for security definitions and proofs. in Moedersheim, S., Palamidessi, C. (eds.) *Theory of Security and Applications (TOSCA 2011)*. LNCS, vol. 6993, (Springer, 2011), pp. 33–56
[57] D. Micciancio, S. Tessaro, An equational approach to secure multi-party computation. in *ITCS 2013*. (ACM, 2013), pp. 355–372
[58] R. Milner, Processes: A mathematical model of computing agents. in Rose, H.E., Shepherdson, J. (eds.) *Logic Colloquium 1973, Studies in Logic and the Foundations of Mathematics*, vol. 80, (Elsevier, 1975), pp. 157–173
[59] R. Milner, A theory of type polymorphism in programming. *Journal of Computer and System Sciences* **17**(3), 348–375 (1978)
[60] J.C. Mitchell, Representation independence and data abstraction. in *POPL 1986*. (ACM, 1986), pp. 263–276
[61] T. Nipkow, G. Klein, Concrete Semantics. Springer (2014)
[62] T. Nipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
[63] R. Pass, E. Shi, F. Tramer, Formal abstractions for attested execution secure processors. Cryptology ePrint Archive, Report 2016/1027 (2016), http://eprint.iacr.org/2016/1027
[64] A. Petcher, G. Morrisett, The foundational cryptography framework. in *POST 2015*. LNCS, vol. 9036, (Springer, 2015), pp. 53–72
[65] A. Petcher, G. Morrisett, A mechanized proof of security for searchable symmetric encryption. in *CSF 2015*. (IEEE 2015), pp. 481–494
[66] M. Piróg, J. Gibbons, The coinductive resumption monad. in Jacobs, B., Silva, A., Staton, S. (eds.) *MFPS 2014*. ENTCS, vol. 308, (2014), pp. 273–288

[67] A.M. Pitts, The HOL logic. in Gordon, M.J.C., Melham, T.F. (eds.) Introduction to HOL: a theorem proving environment for higher order logic, (Cambridge University Press, 1993), pp. 191–232
[68] N. Ramsey, A. Pfeffer, Stochastic lambda calculus and monads of probability distributions. in *POPL 2002*. (ACM, 2002), pp. 154–165
[69] J.C. Reynolds, Types, abstraction and parametric polymorphism. in *IFIP 1983. Information Processing*, vol. 83, (North-Holland/IFIP, 1983), pp. 513–523
[70] J. Sack, L. Zhang, A general framework for probabilistic characterizing formulae. in *VMCAI 2012*. LNCS, vol. 7148, (Springer, 2012), pp. 396–411
[71] N. Schirmer, M. Wenzel, State spaces – the locale way. in Huuck, R., Klein, G., Schlich, B. (eds.) *SSV 2009*. Electronic Notes in Theoretical Computer Science, vol. 254, (2009), pp. 161–179
[72] R. Segala, Modeling and Verification of Randomized Distributed Real-Time Systems. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (1995)
[73] V. Shoup, OAEP reconsidered. in *Annual International Cryptology Conference*. (Springer, 2001), pp. 239–259
[74] V. Shoup, Sequences of games: A tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332 (2004), http://eprint.iacr.org/2004/332
[75] N.P. Smart, Cryptography Made Simple. Information Security and Cryptography, Springer (2016)
[76] A. Sokolova, Coalgebraic Analysis of Probabilistic Systems. Ph.D. thesis, Technische Universiteit Eindhoven (2005)
[77] J. Stern, D. Pointcheval, J. Malone-Lee, N.P. Smart, Flaws in applying proof methodologies to signature schemes. in *Annual International Cryptology Conference*. (Springer, 2002), pp. 93–110
[78] P.Y. Strub, Some questions. Easycrypt Mailing list, post 383. https://lists.gforge.inria.fr/pipermail/easycrypt-club/2016-March/000383.html (2016)
[79] N. Swamy, J. Chen, C. Fournet, P.Y. Strub, K. Bhargavan, J. Yang, Secure distributed programming with value-dependent types. *J. Funct. Program.* **23**(4), 402–451 (2013)
[80] P. Wadler, Theorems for free! in *FPCA 1989*. (ACM, 1989), pp. 347–359
[81] P. Wadler, The essence of functional programming. in *POPL 1992*. (ACM, 1992), pp. 1–14
[82] F. Wiedijk, A synthesis of the procedural and declarative styles of interactive theorem proving. *Logical Methods in Computer Science* **8**(1:30), (2012)
[83] L. Xi, K. Yang, Z. Zhang, D. Feng, DAA-related APIs in TPM 2.0 revisited. in (International Conference on Trust and Trustworthy Computing). (Springer, 2014), pp. 1–18
[84] A.C. Yao, Theory and application of trapdoor functions. in *FOCS 1982*. (IEEE Computer Society, 1982), pp. 80–91
[85] S. Zanella Béguelin, Formal Certification of Game-Based Cryptographic Proofs. Ph.D. thesis, École Nationale Supérieure des Mines de Paris (2010)