Nominal State-Separating Proofs

Markus Krabbe Larsen
Department of Computer Science
IT University of Copenhagen
Copenhagen, Denmark
krml@itu.dk

Carsten Schürmann
Department of Computer Science
IT Univeristy of Copenhagen
Copenhagen, Denmark
carsten@itu.dk

***Abstract*—State-separting proofs are a powerful tool to structure cryptographic arguments, so that they are amenable for mechanization, as has been shown through implementations, such as SSProve. However, the treatment of separation for heaps has never been satisfactorily addressed. In this work, we present the first comprehensive treatment of nominal state separation in state-separating proofs using nominal sets. We provide a Coq library, called Nominal-SSProve, that builds on nominal state separation supporting mechanized proofs that appear more concise and arguably more elegant.**

I. Introduction

State-separating proofs [4] have become a widely accepted tool to express cryptographic games and reductions in the computational model in a formal and precise way to make them palatable for modern verification tools and hereby increase the overall quality of the arguments through formal verification. The central idea in state-separating proofs is to express games, reductions, and the adversary as stateful *packages* that can be combined in modular ways to describe cryptographic security proofs. The state is used to store secret information local to each package.

As an example, consider the Diffie-Hellman key exchange protocol that stores a random value created during the preparation of the first message in the package's local state, and then accesses it again to compute a shared secret in a subsequent message. If an adversary had access to the secret information stored within the state, he would with probability 1 be able to win the cryptographic game, which would render the way cryptographic proofs are done void and meaningless. Thus, the challenge is to protect the state of each package from accidental or malicious access by other packages, for example, by sharing state variable names.

The original formulation of state-separating proofs [4] does not address this challenge adequately, but refers instead to informal on-demand tacit renaming of state variables when packages are composed in order to prevent state variable capture. Similarly, extending Easycrypt by incorporating state-separating proofs [6] does not address the challenge either but leaves it instead to future work. Implementations such as SSProve [10] (in Coq [15]) circumvent this challenge through ad-hoc assumptions about the disjointness of state variables in packages at the expense of modularity.

In this paper, we introduce *nominal state-separating proofs*, extending state-separating proofs by nominal sets [8] to enforce state separation between packages. In nominal state-separating proofs, packages modeling adversaries do not have access to the state of other packages by construction, as each package has its own local state name space. When combining packages, nominal sets ensure that state variable are automatically renamed away from each other so that state variable capture becomes impossible. We demonstrate nominal state-separating proofs by means of showing that the *ElGamal cryptoystem*[7] satisfies the *public key one time secrecy* security property following the proof in [13] (Ch 15.3).

To demonstrate the power of nominal state-separating proofs, we have implemented them in a library that extends SSProve, called Nominal-SSProve.[1] With this library, operations for combining nominal packages – or *modules* as we call them – are semantically transparent, which means that a user does not have to worry about state variable capture, even when quantifying over adversaries. A fully mechanized proof of the running example in Nominal-SSProve can be found here.[2]

We now expand on the challenge and pinpoint the lack of modularity of state-separating proofs for game hopping and cryptographic reductions. The main idea behind state-separating proofs is to reason about the indistinguishability of a pair of games G_1 and G_2 by an adversary $\mathcal{A}$. We may find the two games to be perfectly indistinguishable, meaning that the adversary cannot distinguish between them, written as $\mathrm{Adv}^{\#}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A}) = 0$, where the $\#$ symbol represents a non-nominal advantage definition. We must assume (and make explicit) that the state variables accessible by the adversary are disjoint from the state variables used to define the two games, written as $\mathcal{A} \# \mathrm{G}_1$ and $\mathcal{A} \# \mathrm{G}_2$. *Without nominals, the only way to represent disjointness constraints is to make them explicit.*

Let G_1, G_2, and G_3 be three games, where the pairs of $(\mathrm{G}_1, \mathrm{G}_2)$ and $(\mathrm{G}_2, \mathrm{G}_3)$ are perfectly indistinguishable. To show that $(\mathrm{G}_1, \mathrm{G}_3)$ are perfectly indistinguishable, we have to prove that,

$$\forall \mathcal{A}.\ \mathcal{A} \# \mathrm{G}_1 \wedge \mathcal{A} \# \mathrm{G}_2 \supset \mathrm{Adv}^{\#}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A}) = 0,$$
$$\forall \mathcal{A}.\ \mathcal{A} \# \mathrm{G}_2 \wedge \mathcal{A} \# \mathrm{G}_3 \supset \mathrm{Adv}^{\#}_{\mathrm{G}_2,\mathrm{G}_3}(\mathcal{A}) = 0$$
$$\vdash \forall \mathcal{A}.\ \mathcal{A} \# \mathrm{G}_1 \wedge \mathcal{A} \# \mathrm{G}_3 \supset \mathrm{Adv}^{\#}_{\mathrm{G}_1,\mathrm{G}_3}(\mathcal{A}) = 0$$

[1] See supplementary material.
[2] See supplementary material directory theories/Example/PK.

assuming that the triangle inequality,

$$\mathrm{Adv}^{\#}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A}) + \mathrm{Adv}^{\#}_{\mathrm{G}_2,\mathrm{G}_3}(\mathcal{A}) \geq \mathrm{Adv}^{\#}_{\mathrm{G}_1,\mathrm{G}_3}(\mathcal{A}),$$

holds. This, however, appears to be impossible, since there is no assumption $\mathcal{A} \# \mathrm{G}_2$. We cannot be sure that the adversary's state variables are disjoint from those of game G_2. Making the assumption $\mathcal{A} \# \mathrm{G}_2$ explicit is possible, but it does not generalize well.

The solution is to revert to nominal state-separating proofs, which provides a definition of *nominal advantage* $\mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A})$ that enforces state separation between games $(\mathrm{G}_1, \mathrm{G}_2)$ and the adversary $\mathcal{A}$. As a consequence, there is no longer a need for explicit disjointness assumptions. Therefore, perfect indistinguishability of G_1 and G_3 follows directly from the triangle inequality, as we show in Corollary 37.

$$\begin{aligned} \forall \mathcal{A}.\ \mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A}) = 0, \\ \forall \mathcal{A}.\ \mathrm{Adv}_{\mathrm{G}_2,\mathrm{G}_3}(\mathcal{A}) = 0 \\ \vdash \forall \mathcal{A}.\ \mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_3}(\mathcal{A}) = 0. \end{aligned}$$

A. Contributions

Our contributions consists of the ensuing points.

1) We make precise the informal requirement of tacit variable renaming in [4].
2) We solve the challenge of state variable capture in state-separating proofs.
3) We define a formal semantics of nominal state-separating proofs.
4) We implement Nominal-SSProve as a demonstrator in Coq.
5) We mechanize the reduction from public key one time secrecy for ElGamal to Decisional Diffie–Hellman (DDH). As a result we identify two mistakes in the mechanization of the reduction in SSProve [10].

B. Related Work

The Clutch system [9] derives its power directly from separation logic [12]. It is not designed for state-separating proofs, but can give automatic disjointness guarantees on program contexts similar to nominal state-separating proofs. It is expected, that nominal state-separating proofs can be directly encoded in Clutch, but to our knowledge the work has not been done yet.

How to handle variable names and α-conversion has been studied by many. We refer the reader to the POPLmark challenge [1] for an overview.

State-separating proofs have been used to express the proof of key-schedule security for the TLS1.3 standard on paper [3], and many other examples in the style of state-separating proofs can be found in [13].

C. Overview

This paper is organized as follows. In Section II we define a probabilistic stateful language together with a module system for capturing state-separating proofs. Its static and operational semantics is also described. In Section III, we review the theory of nominal sets and establish that our notion of heap forms a nominal set. In the following Section IV we then show that our notion of adversarial advantage is compatible with nominal sets. In Section V we give a brief overview of Nominal-SSProve and discuss the mistakes we identfied in the development of ElGamal in SSProve. We then conclude in Section VI and assess results.

II. A language for state-separating proofs

We turn our attention to the definition of a simple formal language to express state-separating proofs that extends the simply typed λ-calculus with product types, sum types, a static heap, sampling, and a simple module system that adequately represents sequential and parallel package composition following [4]. The language is inspired by functional programming. It allows us prove correct the nominal constructions that we shall introduce in Section IV in a proof-assistant agnostic way. Packages are encoded as modules and package composition as module composition. Existing tools such as Easycrypt and Coq/SSProve embed this language into higher-order logic and the calculus of inductive constructions, respectively, and inherit additional language features that are irrelevant for this presentation. In Section II-B we will cover syntactic categories and in Section II-C the static semantics. As the language is probabilistic, we introduce the operational semantics and the sub-distribution monad in Sections II-D and II-E, respectively. Finally, in Section II-F we introduce the algebraic equations, that the module system obeys.

A. Base Types and Types

Central to our design is the notion of a sample. As in related work, we capture sample spaces by finite sets, except here, we approximate finite sets proof-theoretically.

$$\alpha \in \mathit{Base} \quad ::= \quad \mathrm{unit} \mid \alpha_1 \times \alpha_2 \mid \alpha_1 + \alpha_2$$

Base types model finite sets to draw samples from: the unit type is inhabited by one element $()$. Products concatenate samples, and sums capture non-deterministic choice. A bit, for example, may be represented as $\mathrm{bit} = \mathrm{unit} + \mathrm{unit}$, which contains two elements **inl** $(())$ and **inr**$(())$. A sample space of size n may be expressed by $\mathrm{fin}\ n$, defined as follows:

$$\begin{array}{lcl} \mathrm{fin}\ 1 & = & \mathrm{unit} \\ \mathrm{fin}\ (2n) & = & \mathrm{bit} \times \mathrm{fin}\ (n) \\ \mathrm{fin}\ (2n+1) & = & \mathrm{bit} \times \mathrm{fin}\ (n) + \mathrm{unit}. \end{array}$$

Given a security parameter λ, we select a group G with a cyclic subgroup of order $q \geq 2^\lambda$ generated by $g \in G$. Formally, G and the subgroup generated by g are denoted by base type $\mathrm{el} = \mathrm{fin}\ q$ and $\mathrm{exp} = \mathrm{fin}\ q$, respectively. We define the cardinality $|\alpha|$ of base type α as follows.

$$|\mathrm{unit}| = 1 \quad |\alpha_1 \times \alpha_2| = |\alpha_1| \cdot |\alpha_2| \quad |\alpha_1 + \alpha_2| = |\alpha_1| + |\alpha_2|$$

By induction on α, it is easy to see that any base type α is inhabited.

Theorem 1 (Inhabitation of base types)**.** *Let α be a base type, then α is inhabited.*

Finally, we declare function types

$$\tau \in \mathit{Type} \quad ::= \quad \alpha \mid \tau_1 \to \tau_2$$

used to type cryptographic algorithms.

B. Values, Expressions, and Modules

Base types and types are inhabited by values which are computed by expressions.

$$\begin{array}{rrl} \mathrm{v} \in \mathit{Val} & ::= & () \mid \mathbf{rec}\ \mathrm{f}\ \mathrm{x} = \mathrm{e} \\ & & \mid (\mathrm{v}_1,\ \mathrm{v}_2) \mid \mathbf{inl}(\mathrm{v}) \mid \mathbf{inr}(\mathrm{v}) \\ \mathrm{e} \in \mathit{Exp} & ::= & \mathrm{x} \mid \mathrm{v} \mid \mathrm{c} \mid \mathrm{e}_1\ \mathrm{e}_2 \mid \mathbf{fst}(\mathrm{e}) \mid \mathbf{snd}(\mathrm{e}) \\ & & \mid \mathbf{case}\ \mathrm{e}\ \mathbf{of}\ \mathbf{inl}(\mathrm{x}_1) \Rightarrow \mathrm{e}_1 \\ & & \qquad \mid \mathbf{inr}(\mathrm{x}_2) \Rightarrow \mathrm{e}_2 \\ & & \mid\ !\mathrm{a} \mid \mathrm{a} := \mathrm{e} \mid \mathbf{sample}(\alpha) \mid \mathrm{F}(\mathrm{e}) \end{array}$$

Most of the constructs are self-explanatory, we only explain those that are not. $\mathrm{x}, \mathrm{f} \in \mathbb{V}$ are local variables, subject to instantiation by substitution. State variables are represented by atoms and allow us to reference the heap.

Definition 1 (Atoms)**.** *The set $\mathbb{A}$ defines a countably infinite set of state variables denoted $\mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3 \ldots$. In this paper, we use atom and state variable interchangeably.*

The set of atoms is infinite, so that in any context of finitely many atoms it is possible to pick a new atom. The nominal constructions that we propose in Section III are built on these atoms.

Finally, $\mathrm{F} \in \mathbb{I}$ are identifiers referring to expressions implemented in other modules, which we define below. The **rec** construct serves as both the fix-point constructor and abstraction according to [10]. The two concepts could have been separated as in other presentations, but nothing is to be gained in our setting. !a resolves an atom $\mathrm{a} \in \mathbb{A}$. The expression a := e binds the result sample of e to state variable a in the heap. There is an expression to sample uniformly from a base type α of size $|\alpha|$, for which we write **sample**(α). Finally, the expression F(e) refers to a call of $\mathrm{F} \in \mathbb{I}$ with argument e. We use parentheses in this case in order to distinguish it from expression application $\mathrm{e}_1\ \mathrm{e}_2$.

We define a few shorthands that will prove useful when writing out expressions in the language. Note that we encode failure by looping indefinitely, and that we omit () when the value is already sorrounded by parentheses. Finally, we allow () in place of a binding occurence of a local variable, when that is the only possbile value.

$$\begin{array}{rcl} \mathrm{bool} & = & \mathrm{unit} + \mathrm{unit} \\ \mathbf{false} & = & \mathbf{inl}() \\ \mathbf{true} & = & \mathbf{inr}() \\ \mathbf{fail} & = & (\mathbf{rec}\ \mathrm{f}\ \mathrm{x} = \mathrm{f}\ \mathrm{x})\ () \\ \lambda \mathrm{x}.\mathrm{e} & = & \mathbf{rec}\ \mathrm{f}\ \mathrm{x} = \mathrm{e} \\ \mathbf{let}\ \mathrm{x} = \mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2 & = & (\lambda \mathrm{x}.\mathrm{e}_2)\ \mathrm{e}_1 \\ \mathbf{let}\ \mathbf{inl}(\mathrm{x}) = \mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2 & = & \mathbf{case}\ \mathrm{e}_1\ \mathbf{of}\ \mathbf{inl}(\mathrm{x}) \Rightarrow \mathrm{e}_2 \\ & & \qquad \mid \mathbf{inr}(\mathrm{y}) \Rightarrow \mathbf{fail} \\ \mathbf{let}\ \mathbf{inr}(\mathrm{x}) = \mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2 & = & \mathbf{case}\ \mathrm{e}_1\ \mathbf{of}\ \mathbf{inl}(\mathrm{y}) \Rightarrow \mathbf{fail} \\ & & \qquad \mid \mathbf{inr}(\mathrm{x}) \Rightarrow \mathrm{e}_2 \\ \mathrm{a} := \mathrm{e}_1\ ;\ \mathrm{e}_2 & = & (\lambda().\mathrm{e}_2)\ (\mathrm{a} := \mathrm{e}_1) \end{array}$$

To explain the mechanism behind identifier resolution, we introduce modules next.

$$\mathrm{M} \in \mathit{Module} \quad ::= \quad \mathbf{module} \mid \mathrm{M}\ \mathbf{fun}\ \mathrm{F}\ \mathrm{x} = \mathrm{e}$$

A *module* groups together the functionality of an application, for example, a cryptosystem, a game, an oracle, or a reduction. Modules are called packages in [4]. Interfaces declare the types of the respective identifiers of a module. We distinguish between import and export interfaces, for which we write I and E, respectively; however, this cannot be observed in all cases, as interfaces may appear both in the import and export position.

C. Static Semantics

Given a fixed heap type for atoms Σ and import interface I for cryptographic algorithms that may be invoked, a set of constants Δ, and a context Γ,

$$\begin{array}{rrl} \Sigma \in \mathit{Heap} & ::= & \cdot \mid \Sigma, \mathrm{a} : \alpha \\ \Gamma \in \mathit{Ctx} & ::= & \cdot \mid \Gamma, \mathrm{x} : \tau \\ \Delta \in \mathit{Const} & ::= & \cdot \mid \Delta, \mathrm{c} : \tau \\ \mathrm{I} \in \mathit{Interface} & ::= & \mathbf{interface} \mid \mathrm{I}\ \mathbf{sig}\ \mathrm{F} : \alpha_1 \to \alpha_2. \end{array}$$

In the case of a non-empty *Heap*, *Ctx*, or *Const*, we omit the leading $\cdot$.

Example 1. *In our running example, constants are group operations. Formally, we write* mult (x,y) *for* $\mathrm{x} \cdot \mathrm{y}$, pow (g,x) *for* g^x *and* powinv (g,x) *for* g^{-x}*. They are declared as follows.*

$$\begin{array}{rcrcl} \Delta & = & \mathrm{mult} & : & \mathrm{el} \times \mathrm{el} \to \mathrm{el}, \\ & & \mathrm{pow} & : & \mathrm{el} \times \mathrm{exp} \to \mathrm{el}, \\ & & \mathrm{powinv} & : & \mathrm{el} \times \mathrm{exp} \to \mathrm{el} \end{array}$$

We define the typing judgments for expressions e in context Γ as follows: $\Sigma; \mathrm{I} \mid \Gamma \vdash_\Delta \mathrm{e} : \tau$. Since Δ is always fixed throughout an argument involving state-separating proofs, we omit it from the judgment and write henceforth $\Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e} : \tau$. The rules for this judgment are given in Figure 1. In the interest of brevity, we omit the definition of well-formedness judgments for *Heap*, *Ctx*, *Const*, and *Interface* but we remark that they are implicitly required in rules unit, ax, deref, sample, and module.

Example 2. *We define the standard three algorithms for the ElGamal cryptosystem as abbreviations in Figure 2.*

Figure 1 also introduces the typing rules for well-typed modules. Informally, module M is well typed of export interface E given a heap typing for Σ an import interface I, and a set of constants Δ, written as judgment $\Sigma; \mathrm{I} \vdash_\Delta \mathrm{M} : \mathrm{E}$ and defined by rules module and fun. For reasons mentioned above, we omit the Δ from this judgment, and simply write $\Sigma; \mathrm{I} \vdash \mathrm{M} : \mathrm{E}$.

Example 3. *We define the modules necessary to capture the DDH assumption in Figure 3. In our example, we use a lazy form of randomness as captured by the two atoms* mga $\in \mathbb{A}$ *and* init $\in \mathbb{A}$. GETA $\in \mathbb{I}$ *and* GETBC $\in \mathbb{I}$ *are identifiers. We*

$$\frac{}{\Sigma; \mathrm{I} \mid \Gamma \vdash () : \mathrm{unit}}\ \text{unit} \qquad \frac{\Gamma(\mathrm{x}) = \tau}{\Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{x} : \tau}\ \text{ax}$$

$$\frac{\Sigma; \mathrm{I} \mid \Gamma, \mathrm{f} : \tau_1 \to \tau_2, \mathrm{x} : \tau_1 \vdash \mathrm{e} : \tau_2}{\Sigma; \mathrm{I} \mid \Gamma \vdash \textbf{rec}\ \mathrm{f}\ \mathrm{x} = \mathrm{e} : \tau_1 \to \tau_2}\ \text{rec} \qquad \frac{\Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e}_1 : \tau_2 \to \tau_1 \quad \Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e}_2 : \tau_2}{\Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e}_1\ \mathrm{e}_2 : \tau_1}\ \text{app}$$

$$\frac{\Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e}_1 : \alpha_1 \quad \Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e}_2 : \alpha_2}{\Sigma; \mathrm{I} \mid \Gamma \vdash (\mathrm{e}_1, \mathrm{e}_2) : \alpha_1 \times \alpha_2}\ \text{pair} \qquad \frac{\Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e} : \alpha_1 \times \alpha_2}{\Sigma; \mathrm{I} \mid \Gamma \vdash \textbf{fst}\,(\mathrm{e}) : \alpha_1}\ \text{fst} \qquad \frac{\Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e} : \alpha_1 \times \alpha_2}{\Sigma; \mathrm{I} \mid \Gamma \vdash \textbf{snd}(\mathrm{e}) : \alpha_2}\ \text{snd}$$

$$\frac{\Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e} : \alpha_1}{\Sigma; \mathrm{I} \mid \Gamma \vdash \textbf{inl}\,(\mathrm{e}) : \alpha_1 + \alpha_2}\ \text{inl} \qquad \frac{\Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e} : \alpha_2}{\Sigma; \mathrm{I} \mid \Gamma \vdash \textbf{inr}(\mathrm{e}) : \alpha_1 + \alpha_2}\ \text{inr}$$

$$\frac{\Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e} : \alpha_1 + \alpha_2 \quad \Sigma; \mathrm{I} \mid \Gamma, x_1 : \alpha_1 \vdash \mathrm{e}_1 : \tau \quad \Sigma; \mathrm{I} \mid \Gamma, x_2 : \alpha_2 \vdash \mathrm{e}_2 : \tau}{\Sigma; \mathrm{I} \mid \Gamma \vdash \textbf{case}\ \mathrm{e}\ \textbf{of}\ \mathrm{x}_1 \Rightarrow \mathrm{e}_1 \mid \mathrm{x}_2 \Rightarrow \mathrm{e}_2 : \tau}\ \text{case}$$

$$\frac{\Sigma(\mathrm{a}) = \alpha}{\Sigma; \mathrm{I} \mid \Gamma \vdash !\mathrm{a} : \alpha}\ \text{deref} \qquad \frac{\Sigma(\mathrm{a}) = \alpha \quad \Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e} : \alpha}{\Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{a} := \mathrm{e} : \mathrm{unit}}\ \text{assign}$$

$$\frac{\mathrm{I}(\mathrm{F}) = \alpha_1 \to \alpha_2 \quad \Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{e} : \alpha_1}{\Sigma; \mathrm{I} \mid \Gamma \vdash \mathrm{F}(\mathrm{e}) : \alpha_2}\ \text{call} \qquad \frac{}{\Sigma; \mathrm{I} \mid \Gamma \vdash \textbf{sample}(\alpha) : \alpha}\ \text{sample}$$

$$\frac{}{\Sigma; \mathrm{I} \vdash \textbf{module} : \textbf{interface}}\ \text{module} \qquad \frac{\Sigma; \mathrm{I} \vdash \mathrm{M} : \mathrm{E} \quad \Sigma; \mathrm{I} \mid \mathrm{x} : \alpha_1 \vdash \mathrm{e} : \alpha_2}{\Sigma; \mathrm{I} \vdash (\mathrm{M}\ \textbf{fun}\ \mathrm{x}\ \mathrm{F} = \mathrm{e}) : (\mathrm{E}\ \textbf{sig}\ \mathrm{F} : \alpha_1 \to \alpha_2)}\ \text{fun}$$

Fig. 1. Static Semantics

```
keygen : exp × el
keygen =
  let sk = sample(exp);
  (sk, g^sk)

enc : el → el → (el × el)
enc = λpk. λm.
  let r = sample(exp);
  (g^r, m · pk^r)

dec : exp → (el × el) → el
dec = λsk. λc.
  snd(c) · fst(c)^-sk
```

Fig. 2. Example: Algorithms of the ElGamal cryptosystem defined as abbreviations.

leave it to the reader to verify that the modules are well-typed, i.e.

$$\mathrm{mga} : \mathrm{unit} + \mathrm{el};\ \cdot \vdash \mathrm{DDH}^0 : \mathrm{I{-}DDH}$$
$$\mathrm{init} : \mathrm{unit} + \mathrm{unit};\ \cdot \vdash \mathrm{DDH}^1 : \mathrm{I{-}DDH}.$$

For the rest of the paper it is useful to introduce the abbreviation $\mathrm{DDH} = (\mathrm{DDH}^0, \mathrm{DDH}^1)$*, also known as a game pair in [4].*

For our final example in this section, we define the interface for adversaries that we use throughout this paper.

Definition 2 (Interface of the Adversary)**.** *An adversary interface is defined as*

$$\mathrm{I{-}ADV} = \textbf{interface}\ \ \textbf{sig}\ \ \mathrm{RUN} : \mathrm{unit} \to \mathrm{bool}$$

In summary, we have presented a simple functional language that is powerful enough to express cryptographic algorithms. If anything, this language is too powerful, because general recursion captures a class of adversaries beyond probabilistic,

```
I–DDH = interface
  sig GETA : unit → el   sig GETBC : unit → el × el

DDH^0 =
  module
  fun GETA () =
    let a = sample(exp) in
    mga := inr(g^a) in
    g^a

  fun GETBC () =
    let inr(x) = !mga in
    mga := inl ();
    let b = sample(exp) in

    (g^b, x^b)

DDH^1 =
  module
  fun GETA () =
    let a = sample(exp) in
    init := inr ();
    g^a

  fun GETBC () =
    let inr() = ! init in
    init := inl ();
    let b = sample(exp) in
    let c = sample(exp) in
    (g^b, g^c)
```

Fig. 3. Example: Definition of DDH interface and games.

polynomial time computable functions. In Section IV, we show that values and expressions form nominal sets. In future work, we might consider extending base types to infinite types, which would complicate working with probability distributions but may offer other benefits, such as greater expressiveness. Another way to extend this language is to generalize base types beyond the sample space, as done, for example, in [9].

D. Operational Semantics

We have chosen to give call-by-value probabilistic operational semantics to our language. It might be possible to experiment with other calling conventions, which we leave to future work. The operational semantics are defined as a small step relation that transforms configurations consisting of an expression e and the current state of the heap σ.

$$\sigma \in \mathit{State} \quad ::= \quad \cdot \mid \sigma, \mathrm{a} := \mathrm{v}$$

As a judgment we write $\langle \mathrm{e}; \sigma \rangle \to_p \langle \mathrm{e}'; \sigma' \rangle$ for a single step, where p is a probability $0 < p \leq 1$, with which this step is taken. For most configurations there is only one deterministic next step the interpreter can take, meaning that $p = 1$, written as $\langle \mathrm{e}; \sigma \rangle \to \langle \mathrm{e}'; \sigma' \rangle$. For sampling, there might be many possible next steps, albeit with a total probability adding to one. There is no step rule for F(e), since calls to an identifier will have been replaced by the corresponding inlined expression during module composition. In the interest of space, we introduce in Figure 4 only the essential reductions and leave the congruence rules to the imagination of the reader. Note that the probabilities of each essential reduction are carried through the congruence closure.

By induction on heap type Σ appealing to Theorem 1, we can show that

Theorem 2. *For all* heap *types* Σ*, there exists a state* σ *that matches* Σ*.*

E. Sub-distribution Monad

Although the operational semantics is probabilistic, a terminating computation will always result in one of finitely many configurations, which means that we can use a sub-distribution monad [9] to capture the probability for each such final configuration.

Definition 3 (Sub-distribution monad)**.** *Let* $\mathcal{D}(X)$ *be the discrete sub-distribution over* X *consisting of functions* $p : X \to [0,1]$ *where* $\sum_{x \in X} p(x) \leq 1$ *with* $\mathrm{unit} : X \to \mathcal{D}(X)$, $\mathrm{bind} : \mathcal{D}(X) \to (X \to \mathcal{D}(Y)) \to \mathcal{D}(Y)$*, and* $\mathrm{zero} : \mathcal{D}(X)$*.*

$$\mathrm{unit}(x)(x') = \begin{cases} 1 & \text{if } x = x' \\ 0 & \text{otherwise} \end{cases}$$
$$\mathrm{bind}(p, f)(y) = \sum_{x \in X} p(x) \cdot f(x)(y)$$
$$\mathrm{zero}(x) = 0$$

The step derivation induces a probability distribution on configurations which we capture formally as the function $\mathrm{step} : \mathit{Cfg} \to \mathcal{D}(\mathit{Cfg})$ given by

$$\mathrm{step}(\mathrm{e}, \sigma)(\mathrm{e}', \sigma') = \begin{cases} p & \text{if } \langle \mathrm{e}, \sigma \rangle \to_p \langle \mathrm{e}', \sigma' \rangle \\ 0 & \text{otherwise} \end{cases}$$

The iterated distribution of executing $n \in \mathbb{N}$ steps results in a probability distribution of values by adding the probabilities for each value while disregarding the state of the configuration. It is defined as $\mathrm{steps}_n : \mathit{Cfg} \to \mathcal{D}(\mathit{Val})$.

$$\mathrm{steps}_0(\mathrm{e}, \sigma) = \mathrm{zero}$$
$$\mathrm{steps}_n(\mathrm{v}, \sigma) = \mathrm{unit}(\mathrm{v})$$
$$\mathrm{steps}_n(\mathrm{e}, \sigma) = \mathrm{bind}(\mathrm{step}(\mathrm{e}, \sigma), \mathrm{steps}_{n-1})$$

Finally, the probability distribution of all values of a computation is defined as taking the following limit

$$\mathrm{steps}(\mathrm{e}, \sigma)(\mathrm{v}) = \lim_{n \to \infty} \mathrm{steps}_n(\mathrm{e}, \sigma)(\mathrm{v}).$$

This limit always exists since it is bounded and monotone. From this construction we obtain the definition of the probability event $\mathrm{v} \leftarrow \mathrm{e}$, which expresses that e evaluates to v in the initial state σ_0 chosen to have the leftmost values for each base type. Existence of such a state is guaranteed by Theorem 2.

Definition 4 (Probability of value)**.** *When* e *has type* $\Sigma; \cdot \mid \cdot \vdash \mathrm{e} : \tau$ *for some* Σ *and* τ *define*

$$\Pr[\mathrm{v} \leftarrow \mathrm{e}] = \mathrm{steps}(\mathrm{e}, \sigma_0)(\mathrm{v})$$

F. Module Algebra

For higher-level cryptographic arguments, where we represent cryptographic systems, oracles, games, and reductions as modules, we need to introduce the algebraic properties of module composition. We start with the identity module ID(I) that implements an interface I by forwarding each call to the composing module.

Definition 5 (Identity Module)**.** *Let*

$$\mathrm{I} = \textbf{interface}\ \ \textbf{sig}\ \mathrm{F}_1 : \alpha_1 \to \alpha_1{}' \ \ \ldots \ \ \textbf{sig}\ \mathrm{F}_n \colon \alpha_n \to \alpha_n{}'$$

be an interface. Then

$$\mathrm{ID(I)} = \textbf{module fun}\ \mathrm{F}_1\mathrm{x} = \mathrm{F}_1(\mathrm{x}) \ \ \ldots \ \ \textbf{fun}\ \mathrm{F}_n\ \mathrm{x} = \mathrm{F}_n(\mathrm{x})$$

is the identity module *for interface* I.

Lemma 3 (Identity module)**.** *Assuming the interface* I *is well-formed, so is the identity module* $\cdot\ ; \mathrm{I} \vdash \mathrm{ID(I)} : \mathrm{I}$.

Before we introduce sequential composition, we define an inlining operation that traverses an expression and replaces every call to a function used in one module, by the definition of the function, declared in a different module. We write this operation as $\mathrm{e} \propto \mathrm{M}$ and is defined to replace $\mathrm{F}(\mathrm{e}_1)$ in e with the corresponding function F from M applied to argument e_1.

$$\overline{\langle (\textbf{rec}\ \mathrm{f}\ \mathrm{x} = \mathrm{e})\ \mathrm{v}; \sigma\rangle \to \langle \mathrm{e}[(\textbf{rec}\ \mathrm{f}\ \mathrm{x} = \mathrm{e})/\mathrm{f}][\mathrm{v}/\mathrm{x}]; \sigma\rangle}$$

$$\overline{\langle \textbf{fst}\,((\mathrm{v}_1,\ \mathrm{v}_2)); \sigma\rangle \to \langle \mathrm{v}_1; \sigma\rangle} \qquad \overline{\langle \textbf{snd}((\mathrm{v}_1,\ \mathrm{v}_2)); \sigma\rangle \to \langle \mathrm{v}_2; \sigma\rangle}$$

$$\overline{\langle \textbf{case}\ \ \textbf{inl}\,(\mathrm{v}_1)\ \ \textbf{of}\ \ \textbf{inl}\,(\mathrm{x}_1)\ \Rightarrow \mathrm{e}_1\ \mid\ \textbf{inr}(\mathrm{x}_2)\ \Rightarrow \mathrm{e}_2; \sigma\rangle \to \langle \mathrm{e}_1[\mathrm{v}_1/\mathrm{x}_1]; \sigma\rangle}$$

$$\overline{\langle \textbf{case}\ \ \textbf{inr}(\mathrm{v}_2)\ \ \textbf{of}\ \ \textbf{inl}\,(\mathrm{x}_1)\ \Rightarrow \mathrm{e}_1\ \mid\ \textbf{inr}(\mathrm{x}_2)\ \Rightarrow \mathrm{e}_2; \sigma\rangle \to \langle \mathrm{e}_2[\mathrm{v}_2/\mathrm{x}_2]; \sigma\rangle}$$

$$\frac{\sigma(\mathrm{a}) = v}{\langle !\mathrm{a}; \sigma\rangle \to \langle \mathrm{v}; \sigma\rangle} \qquad \overline{\langle \mathrm{a}\ :=\ \mathrm{v}; \sigma\rangle \to \langle ();\, \sigma, a := v\rangle} \qquad \frac{\mathrm{v} \in \alpha}{\langle \textbf{sample}(\alpha); \sigma\rangle \to_{1/|\alpha|} \langle \mathrm{v}; \sigma\rangle}$$

Fig. 4. Operational Semantics

Definition 6 (Shared sequential composition)**.** *Define sequential composition* $(\circ) : \mathit{Module} \times \mathit{Module} \to \mathit{Module}$ *on a pair of modules as follows*

$$\textbf{module} \circ \mathrm{M}_2 = \textbf{module}$$
$$(\mathrm{M}_1\,\textbf{val}\ \ \mathrm{f}\ :=\ \mathrm{e}_1) \circ \mathrm{M}_2 = (\mathrm{M}_1 \circ \mathrm{M}_2)\ \textbf{val}\ \ \mathrm{f}\ :=\ \mathrm{e}_1\ \propto\ \mathrm{M}_2$$

To model state-separating proofs adequately, we expect that all calls in the left module are resolved by definitions from the right module. This principle is captured by the following derivable typing rule.

Lemma 4 (Well-typed sequential composition)**.** *Given modules* $\mathrm{M}_1, \mathrm{M}_2$ *with* $\Sigma; \mathrm{I}_1 \vdash \mathrm{M}_1 : \mathrm{E}$ *and* $\Sigma; \mathrm{I}_2 \vdash \mathrm{M}_2 : \mathrm{I}_1$ *we derive* $\Sigma; \mathrm{I}_2 \vdash \mathrm{M}_1 \circ \mathrm{M}_2 : \mathrm{E}$*.*

The interface I_1 is the connection point between M_1 and M_2. It ensures that every call in M_1 has a corresponding implementation in M_2. Note that they all must be typeable in the same state Σ. Separated sequential composition based on nominals that we introduce in Section IV does not have this limitation.

In cases where the import interface does not match, we may need to weaken the imports (expressed by $\mathrm{I}' > \mathrm{I}$) given by the following lemma.

Lemma 5 (Weakening)**.** *Given a module* M *with* $\Sigma; \mathrm{I} \vdash \mathrm{M} : \mathrm{E}$ *and* $\mathrm{I}' > \mathrm{I}$ *then* $\Sigma; \mathrm{I}' \vdash \mathrm{M} : \mathrm{E}$*.*

Parallel composition concatenates the functions of two modules into one bigger module and is defined as follows.

Definition 7 (Shared parallel composition)**.** *Define parallel composition* $(|) : \mathit{Module} \times \mathit{Module} \to \mathit{Module}$ *on a pair of modules as follows*

$$\textbf{module} \mid \mathrm{M}_2 = \mathrm{M}_2$$
$$(\mathrm{M}_1\,\textbf{val}\ \ \mathrm{f}\ :=\ \mathrm{e}) \mid \mathrm{M}_2 = (\mathrm{M}_1 \mid \mathrm{M}_2)\ \textbf{val}\ \ \mathrm{f}\ :=\ \mathrm{e}$$

We can also derive a type for parallel composition of modules; however, we need to disallow cases where identifiers defined by the modules overlap. Otherwise, the export interface is not well-formed.

Lemma 6 (Well-typedness of parallel composition)**.** *Given modules* $\mathrm{M}_1, \mathrm{M}_2$ *and export interfaces* $\mathrm{E}_1, \mathrm{E}_2$ *where the identifiers declared in* E_1 *are disjoint from the identifiers declared in* E_2*. If* $\Sigma; \mathrm{I} \vdash \mathrm{M}_1 : \mathrm{E}_1$ *and* $\Sigma; \mathrm{I} \vdash \mathrm{M}_2 : \mathrm{E}_2$ *we can derive* $\Sigma; \mathrm{I} \vdash (\mathrm{M}_1 \mid \mathrm{M}_2) : \mathrm{E}_1, \mathrm{E}_2$*.*

The following equations allow us to work with the modules reasoning with adversaries, deriving advantages, building oracles, and conducting proofs by game hopping. With these rules we can algebraically manipulate the module expressions to isolate certain parts for reduction.

Lemma 7 (Identities)**.** *For a module* M *with type* $\Sigma; \mathrm{I} \vdash \mathrm{M} : \mathrm{E}$*, the following equations hold.*

$$\mathrm{ID}(\mathrm{E}) \circ \mathrm{M} = \mathrm{M} \qquad \mathrm{ID}(\textbf{interface}) \mid \mathrm{M} = \mathrm{M}$$
$$\mathrm{M} \circ \mathrm{ID}(\mathrm{I}) = \mathrm{M} \qquad \mathrm{M} \mid \mathrm{ID}(\textbf{interface}) = \mathrm{M}$$

Lemma 8 (Associativity)**.** *For modules* $\mathrm{M}_1, \mathrm{M}_2, \mathrm{M}_3 \in \mathit{Module}$ *the sequential and parallel composition operators are associative.*

$$(\mathrm{M}_1 \circ \mathrm{M}_2) \circ \mathrm{M}_3 = \mathrm{M}_1 \circ (\mathrm{M}_2 \circ \mathrm{M}_3)$$
$$(\mathrm{M}_1 \mid \mathrm{M}_2) \mid \mathrm{M}_3 = \mathrm{M}_1 \mid (\mathrm{M}_2 \mid \mathrm{M}_3)$$

Lemma 9 (Interchange)**.** *For modules* $\mathrm{M}_1, \mathrm{M}_2, \mathrm{M}_3, \mathrm{M}_4$ *where* $\Sigma; \mathrm{I}_1 \vdash \mathrm{M}_1 : \mathrm{E}_1$*,* $\Sigma; \mathrm{I}_2 \vdash \mathrm{M}_2 : \mathrm{E}_2$*,* $\Sigma; \mathrm{I} \vdash \mathrm{M}_3 : \mathrm{I}_1$*,* $\Sigma; \mathrm{I} \vdash \mathrm{M}_4 : \mathrm{I}_2$*, and* I_1 *defines different identifiers from* I_2*, then*

$$(\mathrm{M}_1 \mid \mathrm{M}_2) \circ (\mathrm{M}_3 \mid \mathrm{M}_4) = (\mathrm{M}_1 \circ \mathrm{M}_3) \mid (\mathrm{M}_2 \circ \mathrm{M}_4)$$

The well-typedness assumptions for interchange ensure that there are no calls from M_1 to M_4 and from M_2 to M_3.

Looking back at Lemmas 4 and 6, states are shared between modules M_1 and M_2, which is expressed by the use of the same heap type Σ in both typing derivations:$\Sigma; \mathrm{I}_1 \vdash \mathrm{M}_1 : \mathrm{E}$ and $\Sigma; \mathrm{I}_2 \vdash \mathrm{M}_2 : \mathrm{I}_1$. It is this sharing that motivates the rest of the paper. In case that we compose an adversary (of export interface I−ADV) with existing modules DDH^0 and DDH^1 in Example 3, we must prevent the adversary from reading from state variables mga or init , otherwise it could trivially distinguish the games.

The alternative to having one Σ as heap-type, is to use two different heap types Σ_1 and Σ_2, for modules M_1 and M_2, respectively, so that $\Sigma_1; \mathrm{I}_1 \vdash \mathrm{M}_1 : \mathrm{E}$ and $\Sigma_2; \mathrm{I}_2 \vdash \mathrm{M}_2 : \mathrm{I}_1$. When composing M_1 and M_2 (either sequentially or in parallel), the resulting module M will be well-typed in $\Sigma_1, \Sigma_2; \mathrm{I}_2 \vdash \mathrm{M} : \mathrm{I}_1$. If we do this, however, we need to worry about variable capture. This can be avoided by using separated concatenation of heaps $\Sigma_1 * \Sigma_2$ which prevents variable capture as we discuss in the next section.

Our solution is to ensure separation between modules using nominal sets.

III. Nominals

Nominal sets for reasoning about open terms with names were discovered in the seminal paper by Gabbay and Pitts [8] and applied to model languages with binding. α-conversion, which is a conceptually simple but notoriously annoying problem when modelling languages with binding operators, can be seen as an instance of a nominal set, where atoms are automatically renamed to keep the naming of two terms disjoint from one another. The central idea of nominal sets is a permutation model of set theory with names (also known as Fraenkel and Mostowski sets). In this section we briefly review the nominal techniques and their properties following [11], keeping in mind that we apply these techniques to heaps the theory of state-separating proofs. In the following section we show how to keep the internal state of modules disjoint from one another using nominal sets. We introduce $\mathrm{Perm}\ \mathbb{A}$-sets in Section III-A, equivariance in Section III-B, support sets in Section III-C, α-equivalence in Section III-D. The proofs of important lemmas and theorems presented in this section are summarized in Appendix A.

A. $\mathrm{Perm}\ \mathbb{A}$-sets

Recall the definition of atoms $\mathbb{A}$ from Definition 1. Permutations among atoms are central to nominal sets.

Definition 8 ($\mathrm{Perm}\ \mathbb{A}$). *Let $\mathrm{Perm}\ \mathbb{A}$ denote the group of permutations that act on a finite subset of $\mathbb{A}$. $\mathrm{Perm}\ \mathbb{A}$ defines the usual group structure: For two permutations $\pi_1, \pi_2 \in \mathrm{Perm}\ \mathbb{A}$, we write their group product as $\pi_1\pi_2$, while π_1^{-1} denotes the inverse permutation of π_1. Finally, we take id to denote the neural element.*

The following definition gives us a general notion of sets that support applying a permutation of atoms to its elements.

Definition 9 ($\mathrm{Perm}\ \mathbb{A}$-set). *A set X with a group action $(\cdot) : \mathrm{Perm}\ \mathbb{A} \times X \to X$ is called a $\mathrm{Perm}\ \mathbb{A}$-set (read: permutation set). The group action must obey the following equations for all $\pi_1, \pi_2 \in \mathrm{Perm}\ \mathbb{A}$ and $x \in X$*

$$\mathrm{id} \cdot x = x, \qquad \text{(Identity)}$$
$$\pi_1 \cdot \pi_2 \cdot x = (\pi_1 \pi_2) \cdot x. \qquad \text{(Compatibility)}$$

Equations (Identity) and (Compatibility) are standard for group actions. We illustrate the concept of permutation sets by a few examples to give the reader a chance to get acquainted with them.

Lemma 10 ($\mathrm{Perm}\ \mathbb{A}$-sets). *It is easy to see that the following sets are $\mathrm{Perm}\ \mathbb{A}$-sets by checking (Identity) and (Compatibility).*

- *The set $\mathbb{A}$ of atoms is a $\mathrm{Perm}\ \mathbb{A}$-set with group action $\pi \cdot \mathrm{a} = \pi(\mathrm{a})$.*
- *Given $\mathrm{Perm}\ \mathbb{A}$-sets X and Y, their Cartesian product $X \times Y$ is a $\mathrm{Perm}\ \mathbb{A}$-set with group action $\pi \cdot (x, y) = (\pi \cdot x, \pi \cdot y)$.*
- *The set of real numbers $\mathbb{R}$ is a $\mathrm{Perm}\ \mathbb{A}$-set with group action $\pi \cdot r = r$.*
- *The set $\mathrm{P}_{\mathrm{fin}}(X) = \{F \mid F \subset X \text{ and } F \text{ finite}\}$ for any $\mathrm{Perm}\ \mathbb{A}$-set X is also a $\mathrm{Perm}\ \mathbb{A}$-set with group action $\pi \cdot F = \{\pi \cdot x \mid x \in F\}$ for $F \in \mathrm{P}_{\mathrm{fin}}(X)$.*

The following lemma shows how permutations act on heaps.

Lemma 11. *Heap and State are $\mathrm{Perm}\ \mathbb{A}$-sets with group action*

$$\pi \cdot (\Sigma,\ \mathrm{a} : \alpha) = (\pi \cdot \Sigma, \pi(\mathrm{a}) : \alpha)$$
$$\pi \cdot (\sigma,\ \mathrm{a} := \mathrm{v}) = (\pi \cdot \Sigma, \pi(\mathrm{a}) := \mathrm{v})$$

In the last equation, v is invariant under the permutation, since v does not contain any atoms, as it is a sample.

B. Equivariance

We review the concept of equivariant functions. These capture the property of being name invariant in the sense that the result of an equivariant function does not depend on the specific names used.

Definition 10 (Equivariant function). *A function $f : X \to Y$ between $\mathrm{Perm}\ \mathbb{A}$-sets X and Y is* equivariant *if for all $\pi \in \mathrm{Perm}\ \mathbb{A}$ and $x \in X$, $\pi \cdot f(x) = f(\pi \cdot x)$.*

In contrast, if a non-equivariant function f were to create a new atom, it is easy to construct a permutation, so that $\pi \cdot f(x) \neq f(\pi \cdot x)$.

Example 4. *The function $f : \mathbb{A} \to \mathbb{A} \times \mathbb{A}$ given by $f(\mathrm{a}) = (\mathrm{a}, \mathrm{a})$ is equivariant, since for all $\pi \in \mathrm{Perm}\ \mathbb{A}$ and $\mathrm{a} \in \mathbb{A}$,*

$$\pi \cdot f(\mathrm{a}) = \pi \cdot (\mathrm{a}, \mathrm{a}) = (\pi \cdot \mathrm{a}, \pi \cdot \mathrm{a}) = f(\pi \cdot \mathrm{a})$$

On the other hand, the function $g : \mathbb{A} \to \mathbb{A} \times \mathbb{A}$ given by $g(\mathrm{a}) = (\mathrm{a}_1, \mathrm{a})$ is not equivariant, since for $\pi = (\mathrm{a}_1\ \mathrm{a}_2)$ and some $\mathrm{a} \in \mathbb{A}$ different from a_1 and a_2, we have that

$$(\mathrm{a}_1\ \mathrm{a}_2) \cdot g(\mathrm{a}) = (\mathrm{a}_2, \mathrm{a}) \neq (\mathrm{a}_1, \mathrm{a}) = g((\mathrm{a}_1\ \mathrm{a}_2) \cdot \mathrm{a})$$

Lemma 12 (Equivariant set operations). *Intersection and union are equivariant on $\mathrm{P}_{\mathrm{fin}}(X)$ for a $\mathrm{Perm}\ \mathbb{A}$-set X. That is, for all $A, B \in \mathrm{P}_{\mathrm{fin}}(X)$ and $\pi \in \mathrm{Perm}\ \mathbb{A}$ it holds that $\pi \cdot (A \cap B) = (\pi \cdot A) \cap (\pi \cdot B)$ and $\pi \cdot (A \cup B) = (\pi \cdot A) \cup (\pi \cdot B)$.*

From this fact we can infer that subset inclusion and set disjointness are also equivariant in the sense that $A \subseteq B$ if and only if $\pi \cdot A \subseteq \pi \cdot B$ and $A \cap B = \emptyset$ if and only if $\pi \cdot A \cap \pi \cdot B = \emptyset$.

Finally, we prove that shared module compositions are equivariant.

Theorem 13 (Equivariance of shared module compositions)**.** $\mathrm{M}_1 \circ \mathrm{M}_2$ *and* $\mathrm{M}_1 \mid \mathrm{M}_2$ *are equivariant.*

C. Support

We turn to the finiteness criterion of what is called the support of a nominal set.

Definition 11 (Finite support)**.** *Let* X *be a* $\mathrm{Perm}\ \mathbb{A}$*-set and* $S \in \mathrm{P}_{\mathrm{fin}}(\mathbb{A})$ *a finite subset of atoms. We say* S *is a* support set *for* $x \in X$ *if for all permutations* $\pi \in \mathrm{Perm}\ \mathbb{A}$ *so that* $\pi(\mathrm{a}) = \mathrm{a}$ *for all* $\mathrm{a} \in S$*, it holds that* $\pi \cdot x = x$*.*

Finding a support set for an element in a $\mathrm{Perm}\ \mathbb{A}$-set guarantees that any permutation that preserves the atoms in the support set will also preserve the value. In some sense the support set contains at least all of the atoms that are *present* in the element, but we define this without having to say what an atom being *present* means. We expand this concept to a whole $\mathrm{Perm}\ \mathbb{A}$-set with the definition of a nominal set.

Definition 12 (Nominal set)**.** *A* $\mathrm{Perm}\ \mathbb{A}$*-set* X *has a* finite support function $\mathrm{supp} : X \to \mathrm{P}_{\mathrm{fin}}(\mathbb{A})$ *if for all* $x \in X$*,* $\mathrm{supp}(x)$ *is a support set for* x *and for any other support set* S *for* x*, it is the case that* $\mathrm{supp}(x) \subseteq S$*. A* $\mathrm{Perm}\ \mathbb{A}$*-set with a finite support function is called a* nominal set*.*

Henceforth, we will use the names X, Y and Z to denote arbitrary nominal sets. The $\mathrm{Perm}\ \mathbb{A}$-sets that we have seen until this point are also nominal sets.

Lemma 14 (Nominal sets)**.** *The following* $\mathrm{Perm}\ \mathbb{A}$ *sets are nominal sets.*

- *The set* $\mathbb{A}$ *has* $\mathrm{supp}(\mathrm{a}) = \{\mathrm{a}\}$.
- *The set* $X \times Y$ *has* $\mathrm{supp}(x, y) = \mathrm{supp}(x) \cup \mathrm{supp}(y)$.
- *The set* $\mathbb{R}$ *has* $\mathrm{supp}(r) = \emptyset$*. In fact, every* $\mathrm{Perm}\ \mathbb{A}$*-set with* $\pi \cdot x = x$ *has an empty support set and is called a discrete nominal set.*
- *The set* $\mathrm{P}_{\mathrm{fin}}(X)$ *has* $\mathrm{supp}(S) = \bigcup_{x \in S} \mathrm{supp}(x)$

Lemma 15 (Heaps and states are nominal sets)**.** *The following* $\mathrm{Perm}\ \mathbb{A}$ *sets are nominal sets.*

- *The set Heap has* $\mathrm{supp}(\Sigma,\ \mathrm{a}\ :\ \alpha) = \mathrm{supp}(\Sigma) \cup \{\mathrm{a}\}$.
- *The set State has* $\mathrm{supp}(\sigma,\ \mathrm{a}\ :=\ \mathrm{v}) = \mathrm{supp}(\sigma) \cup \{\mathrm{a}\}$.

Note that the supp function maps an element from any nominal set into a finite set of atoms. This allows us to state disjointness generically.

Definition 13 (Disjoint support)**.** *We say that* $x \in X$ *and* $y \in Y$ *are disjoint written* $x \# y$ *when* $\mathrm{supp}(x) \cap \mathrm{supp}(y) = \emptyset$*.*

Lemma 16 (Disjoint support properties)**.** *For elements* $x \in X$*,* $Y \in Y$ *and* $z \in Z$ *and equivariant function* $f : X \to Z$ *it holds that*

- *if* $x \# y$ *then* $y \# x$*,*
- *if* $x \# x$ *then* $\mathrm{supp}(x) = \emptyset$*,*
- *if* $x \# y$ *then* $f(x) \# y$*,*
- *if* $x \# y$ *then* $\pi \cdot x \# \pi \cdot y$ *for* $\pi \in \mathrm{Perm}\ \mathbb{A}$*.*

Now we understand what disjointness means, but we have not proposed a way to render two elements of nominal sets disjoint, which we achieve by active renaming using the permutation fresh.

Definition 14 (Fresh)**.** *Fix some bijection between the atoms and the natural numbers* $\mathrm{idx} : \mathbb{A} \to \mathbb{N}$*. Define* $\mathrm{fresh} : X \times Y \to \mathrm{Perm}\ \mathbb{A}$ *so that* [3]

$$\mathbf{fresh}(x, y)(\mathrm{a}) = \mathrm{idx}^{-1}\left(\mathrm{idx}(\mathrm{a}) + k(x)\right) \text{ for } \mathrm{a} \in \mathrm{supp}(y),$$

where

$$k(x) = \max_{\mathrm{a}' \in \mathrm{supp}(x)} \mathrm{idx}(\mathrm{a}').$$

The purpose of fresh is to choose an element $y' \in Y$ related to $y \in Y$, so that the support of a fixed $x \in X$ is disjoint from the support of the chosen y'. This property is captured in the following lemma.

Lemma 17 (Fresh disjoint)**.** *For all elements* $x \in X$ *and* $y \in Y$*, it holds that* $x \# \mathrm{fresh}(x, y) \cdot y$*.*

To show the usefulness of fresh we introduce separated concatenation of heaps, denoted by $\Sigma_1 * \Sigma_2$, as motivated at the end of Section II-F.

Definition 15 (Separated concatenation)**.** *For* $\Sigma_1, \Sigma_2 \in \mathit{Heap}$ *define separated concatenation so that*

$$\Sigma_1 * \Sigma_2 = \Sigma_1, \mathrm{fresh}(\Sigma_1, \Sigma_2) \cdot \Sigma_2.$$

Separated concatenation enforces support separation by actively renaming atoms in the right argument using fresh. We will re-use this pattern when defining separated compositions.

D. α-equivalence

To capture the nice properties of separated concatenation, we introduce α-equivalence. Two elements are α-equivalent, if they are equal modulo a permutation of atoms. This notion of α-equivalence is defined for any nominal set.

Definition 16 (α-equivalence)**.** *For elements* $x, x' \in X$ *we say that* x *and* x' *are* α*-equivalent written* $x \equiv x'$ *when there exists a permutation* $\pi \in \mathrm{Perm}\ \mathbb{A}$ *so that* $\pi \cdot x = x'$*.*

Theorem 18. α*-equivalence is an equivalence relation.*

When concatenating two heaps, without considering α-equivalence, state variable capture might be unavoidable. When concatenating two heaps using separated concatenation, which takes α-renaming into account, variable capture will be avoided. α-congruence expresses the fact that we can replace heaps by α-equivalent heaps and still avoid variable capture.

Definition 17 (α-congruence)**.** *Assuming* $X_1, \ldots X_n$*, and* Y *are nominal sets. An* n*-ary function* $f : X_1 \times \cdots \times X_n \to Y$ *is an* α-congruence *if whenever* $x_i \equiv x_i'$ *for* $i \in \{1, \ldots, n\}$ *then* $f(x_1, \ldots, x_n) \equiv f(x_1', \ldots, x_n')$*.*

[3] The definition is injective, so we obtain a permutation by mapping elements from the finite set $\mathbf{fresh}(x, y)(\mathrm{supp}(y)) \setminus \mathrm{supp}(y)$ to the finite set of equal size $\mathrm{supp}(y) \setminus \mathbf{fresh}(x, y)(\mathrm{supp}(y))$ injectively.

We show that any unary equivariant function is also an α-congruence.

Example 5. *Let $f : X \to Y$ be given and assume that f is equivariant. For $x, x' \in X$ assume that $x \equiv x'$, thus there exists π so that $\pi \cdot x = x'$. Hence, $f(x') = f(\pi \cdot x) = \pi \cdot f(x)$, so $f(x) \equiv f(x')$ i.e. f is an α-congruence.*

This argument does apply to binary functions, since the support of the two elements may overlap. Swapping elements in the overlap for one argument will result in a different sharing of names; thus the structure of atoms is not preserved in the result. Note that the shared compositions are not α-congruent for this reason. However, separated concatenation enjoys the property of being a binary α-congruence, since state variable capture is impossible.

Theorem 19 (Separated concatenation is an α-congruence)**.** *When $\Sigma_1 \equiv \Sigma_1'$ and $\Sigma_2 \equiv \Sigma_2'$, then $\Sigma_1 * \Sigma_2 \equiv \Sigma_1' * \Sigma_2'$.*

IV. Nominal State-Separating Proofs

We will now demonstrate how to extend the language of state-separating proofs introduced in Section II by nominals, which results in a concept called *nominal state-separating proofs*. We argue that all concepts introduced in that section are nominal sets where atoms are state variables. We describe *nominal expressions* in Section IV-A and *nominal modules* in Section IV-B. In Section IV-C, we introduce the concepts of *nominal advantage* and *nominal indistinguishability* between pairs of games. The proofs for the lemmas and theorems in this section can be found in Appendix A.

A. Nominal Expressions

Recall that we distinguish different kinds of variables: Local variables are declared within each function, state variables – or atoms – are declared globally and shared across functions, and identifiers are used to refer to imported functions. When moving to nominal expressions, we move away from a single shared heap to many separate heaps, as discussed in Section III-D. As a starting point, we show that $Expr$ is a $\mathrm{Perm}\ \mathbb{A}$-set.

Definition 18 (Group action for $Expr$)**.** *Let $\pi \in \mathrm{Perm}\ \mathbb{A}$ be a permutation and $\mathrm{e} \in Expr$. We define the group action $\cdot : \mathrm{Perm}\ \mathbb{A} \times Expr \to Expr$. We only present the base cases here, leaving it to the reader to define the remaining cases by forming the congruence closure. We omit the proofs of identity and compatibility.*

$$\begin{array}{rclrcl} \pi \cdot (\mathrm{x}) & = & \mathrm{x} & \pi \cdot (\mathrm{F(e)}) & = & \mathrm{F}(\pi \cdot \mathrm{e}) \\ \pi \cdot (()) & = & () & \pi \cdot (\textbf{sample}(\alpha)) & = & \textbf{sample}(\alpha) \\ \pi \cdot (!\mathrm{a}) & = & !\pi(\mathrm{a}) & \pi \cdot (\mathrm{a} := \mathrm{e}) & = & \pi(\mathrm{a}) := (\pi \cdot \mathrm{e}) \end{array}$$

Next we define the support function for $Expr$.

Definition 19 (Support function for $Expr$)**.** *The support function $\mathrm{supp} : Expr \to \mathrm{P}_{\mathrm{fin}}(\mathbb{A})$ is defined according to the structure of $Expr$. We only give base cases, since the congruence closure is straightforward and omitted in the interest of space, as is the proof of the well-definedness of the support function.*

$$\begin{array}{rclrcl} \mathrm{supp(x)} & = & \emptyset & \mathrm{supp(F(e))} & = & \mathrm{supp(e)} \\ \mathrm{supp}(()) & = & \emptyset & \mathrm{supp}(\textbf{sample}(\alpha)) & = & \emptyset \\ \mathrm{supp}(!\mathrm{a}) & = & \{\mathrm{a}\} & \mathrm{supp}(\mathrm{a} := \mathrm{e}) & = & \{\mathrm{a}\} \cup \mathrm{supp(e)} \end{array}$$

Finally, we show that $Expr$ is a nominal set.

Lemma 20 (Nominal expressions)**.** *The set of expressions $Expr$, together with the group action $\cdot : \mathrm{Perm}\ \mathbb{A} \times Expr \to Expr$ and the support function $\mathrm{supp} : Expr \to \mathrm{P}_{\mathrm{fin}}(\mathbb{A})$ forms a nominal set.*

We have already seen that $Heap$ is also a nominal set, so we state and prove that well-typedness is preserved under permutation.

Lemma 21 (Type preserved by permutation)**.** *Given $\Sigma; \mathrm{I} \mid \mathrm{e} : \tau$ we have $\pi \cdot \Sigma; \mathrm{I} \mid \pi \cdot \mathrm{e} : \tau$.*

Moving from the static to the dynamic semantics, we show that operational semantics introduced in Section II-D is also equivariant, which means that the probability distribution on the resulting values is invariant under the group action: $\pi \cdot \Pr[\mathrm{v} \leftarrow \mathrm{e}] = \Pr[\pi \cdot \mathrm{v} \leftarrow \pi \cdot \mathrm{e}]$. We structure the argument into smaller steps, starting with the fact that being a value is preserved under permutation.

Lemma 22 (Preservation of values)**.** *If $\mathrm{v} \in Val$, then $\pi \cdot \mathrm{v} \in Val$.*

Recall that the probabilistic small-step operational semantics enabled us to define a $\mathrm{step} : Cfg \to \mathcal{D}(Cfg)$ function, that maps configurations to a probability distribution of configurations. Recall that configurations are defined as products of two nominal sets, which means that they are also nominal sets. We can show that the step-function is equivariant, meaning that the probability distribution remains intact under permutations. With the group action derived from the group action of the configurations, we can show that step is equivariant.

Lemma 23 (Step equivariance)**.** *The function $\mathrm{step} : Cfg \to \mathcal{D}(Cfg)$ is equivariant.*

In Definition 3, we introduced the sub-distribution monad in order to iterate the single-step reductions of the operational semantics. We can show that the sub-distribution monad inherits the nominal set property from the nominal expressions, by the virtue that unit, bind and zero are bound to value (of functional type), for which we have already established in Lemma 20 that they are nominal sets.

Lemma 24 (Sub-Distribution equivariance)**.** *The functions* unit, bind *and* zero *are equivariant.*

Based on these observations, we can generalize the equivariance property of the single-step relation step to the multi-step case steps.

Theorem 25 (steps equivariance)**.** *The function $\mathrm{steps} : Cfg \to \mathcal{D}(Cfg)$ is equivariant.*

This means that the mapping of configurations into the sub-distribution monad is equivariant, and it follows directly that that the probability distribution on result values is also equivariant.

Corollary 26 (Pr equivariance)**.** $\Pr[\cdot \leftarrow \cdot] : \mathit{Val} \times \mathit{Expr} \to \mathbb{R}$ *is equivariant.*

B. Nominal Modules

Recall form Section II-F that two modules can be composed either sequentially, which means that calls to external functions are resolved by inlining, or in parallel, which means that the functions of either module are merged. In this section we show that modules form a nominal set, and that we can define separating module composition based on the results of preceding sections.

As usual, as the first step, we define a group action for modules, for which we write $(\cdot) : \mathrm{Perm}\ \mathbb{A} \times \mathit{Module} \to \mathit{Module}$. Its definition is straightforward.

Definition 20 (Group action for Module)**.**

$$\pi \cdot \textbf{module} = \textbf{module}$$
$$\pi \cdot (\mathrm{M}\ \textbf{fun}\ \mathrm{F}\ \mathrm{x} = \mathrm{e}) = ((\pi \cdot \mathrm{M})\ \textbf{fun}\ \mathrm{F}\ \mathrm{x} = (\pi \cdot \mathrm{e}))$$

Note that F is a function identifier, and x is a local variable; thus the permutation does not affect them. We omit the well-formedness proof of the group action. As a second step, we define the support function $\mathrm{supp} : \mathit{Module} \to \mathrm{P}_{\mathrm{fin}}(\mathbb{A})$ for modules.

Definition 21 (Support function for Module)**.**

$$\mathrm{supp}(\textbf{module}) = \emptyset$$
$$\mathrm{supp}(\mathrm{M}\ \textbf{fun}\ \mathrm{F}\ \mathrm{x} = \mathrm{e}) = \mathrm{supp}(\mathrm{M}) \cup \mathrm{supp}(\mathrm{e})$$

We omit the well-formedness proof of the support function.

Lemma 27 (Nominal modules)**.** *The set* Module *forms a nominal set together with the group action* $\cdot : \mathrm{Perm}\ \mathbb{A} \times \mathit{Module} \to \mathit{Module}$ *and the support function* $\mathrm{supp} : \mathit{Module} \to \mathrm{P}_{\mathrm{fin}}(\mathbb{A})$.

Like the definition for separated heap concatenation we now use fresh to define separated module composition based on the previously defined shared compositions.

Definition 22 (Separated composition)**.** *Define separated sequential composition as*

$$\mathrm{M} \circledcirc \mathrm{N} = \mathrm{M} \circ (\mathrm{fresh}(\mathrm{M}, \mathrm{N}) \cdot \mathrm{N}),$$

and parallel composition as

$$\mathrm{M} \parallel \mathrm{N} = \mathrm{M} \mid (\mathrm{fresh}(\mathrm{M}, \mathrm{N}) \cdot \mathrm{N}).$$

Note the similarity of how we apply fresh here in relation to how we define separating concatenation of heaps in Section III-C.

To express the heap of separated composition, we introduce a pruning operation. To type a separated composition of modules M_1 and M_2, we need to ensure that the separated concatenation of heaps $\Sigma_1 * \Sigma_2$ are aligned. If there are atoms in Σ_1 that do not appear M_1, then the permutations applied internally may not match. To rectify this problem, we define a *pruning* operation that strengthens Σ_1 to Σ_1' which only contains atoms that are also used in M_1. We write $\Sigma_1' = \mathrm{prune}(\Sigma_1, \mathrm{M}_1)$ for pruning.

Lemma 28 (Well-typedness of separated sequential composition)**.** *Given modules* $\mathrm{M}_1, \mathrm{M}_2$ *with* $\Sigma_1; \mathrm{I}_1 \vdash \mathrm{M}_1 : \mathrm{E}$ *and* $\Sigma_2; \mathrm{I}_2 \vdash \mathrm{M}_2 : \mathrm{I}_1$ *we can derive* $\Sigma_3; \mathrm{I}_2 \vdash \mathrm{M}_1 \circledcirc \mathrm{M}_2 : \mathrm{E}$ *where* $\Sigma_3 = \mathrm{prune}(\Sigma_1, \mathrm{M}_1) * \Sigma_2$.

Lemma 29 (Well-typedness of separated parallel composition)**.** *Given modules* $\mathrm{M}_1, \mathrm{M}_2$ *and export interfaces* $\mathrm{E}_1, \mathrm{E}_2$ *where the identifiers declared in* E_1 *are disjoint from the identifiers declared in* E_2*. If* $\Sigma_1; \mathrm{I} \vdash \mathrm{M}_1 : \mathrm{E}_1$ *and* $\Sigma_2; \mathrm{I} \vdash \mathrm{M}_2 : \mathrm{E}_2$ *we can derive* $\Sigma_3; \mathrm{I} \vdash (\mathrm{M}_1 \parallel \mathrm{M}_2) : \mathrm{E}_1, \mathrm{E}_2$ *where* $\Sigma_3 = \mathrm{prune}(\Sigma_1, M_1) * \Sigma_2$.

As a consequence of the separation and the fact that shared compositions are equivariant, functions we are able to derive the following congruence theorem.

Theorem 30 (Separated congruence)**.** *For modules where* $\mathrm{M}_1 \equiv \mathrm{M}_1{}'$ *and* $\mathrm{M}_2 \equiv \mathrm{M}_2{}'$ *we have*

$$\mathrm{M}_1 \circledcirc \mathrm{M}_2 \equiv \mathrm{M}_1{}' \circledcirc \mathrm{M}_2{}',$$
$$\mathrm{M}_1 \parallel \mathrm{M}_2 \equiv \mathrm{M}_1{}' \parallel \mathrm{M}_2{}'.$$

We now show that the desired algebraic rules are preserved under either α-equivalence or equality.

Lemma 31 (Identities)**.** *For a module* M *with type* $\Sigma; \mathrm{I} \vdash \mathrm{M} : \mathrm{E}$*, the following identities hold.*

$$\mathrm{ID}(\mathrm{E}) \circledcirc \mathrm{M} = \mathrm{M} \qquad \mathrm{ID}(\textbf{interface}) \parallel \mathrm{M} = \mathrm{M}$$
$$\mathrm{M} \circledcirc \mathrm{ID}(\mathrm{I}) = \mathrm{M} \qquad \mathrm{M} \parallel \mathrm{ID}(\textbf{interface}) = \mathrm{M}$$

Lemma 32 (Associativity)**.** *For modules* $\mathrm{M}_1, \mathrm{M}_2, \mathrm{M}_3$ *the separated sequential and parallel composition operators are associative up to* α*-equivalence.*

$$(\mathrm{M}_1 \circledcirc \mathrm{M}_2) \circledcirc \mathrm{M}_3 \equiv \mathrm{M}_1 \circledcirc (\mathrm{M}_2 \circledcirc \mathrm{M}_3)$$
$$(\mathrm{M}_1 \parallel \mathrm{M}_2) \parallel \mathrm{M}_3 \equiv \mathrm{M}_1 \parallel (\mathrm{M}_2 \parallel \mathrm{M}_3)$$

We lose the strict equality of the associativity rules; however, as we will see, this is sufficient to be compatible with nominal indistinguishability. Finally, we prove that also interchange holds under α-equivalence.

Lemma 33 (Interchange)**.** *For modules* $\mathrm{M}_1, \mathrm{M}_2, \mathrm{M}_3, \mathrm{M}_4$ *where* $\Sigma_1; \mathrm{I}_1 \vdash \mathrm{M}_1 : \mathrm{E}_1$*,* $\Sigma_2; \mathrm{I}_2 \vdash \mathrm{M}_2 : \mathrm{E}_2$*,* $\Sigma_3; \mathrm{I}_3 \vdash \mathrm{M}_3 : \mathrm{I}_1$*,* $\Sigma_4; \mathrm{I}_4 \vdash \mathrm{M}_4 : \mathrm{I}_2$*, and* I_1 *defines different identifiers from* I_2*, then*

$$(\mathrm{M}_1 \parallel \mathrm{M}_2) \circledcirc (\mathrm{M}_3 \parallel \mathrm{M}_4) \equiv (\mathrm{M}_1 \circledcirc \mathrm{M}_3) \parallel (\mathrm{M}_2 \circledcirc \mathrm{M}_4)$$

C. Nominal Indistinguishability

In cryptography, the security of a cryptographic protocol is captured by an adversary being able to distinguish between two games, M_1 and M_2. After observing an exchange of messages (also called a transcript) that a verifier and a prover play together, the task of the adversary is to guess which of the two

```
I−PK−OTS$ = interface
  sig GET : unit → el   sig QUERY : el → el × el

PK−OTS$^0 =                          PK−OTS$^1 =
  module                               module
  fun GET () =                         fun GET () =
    let inl() = !mpk in                  let inl() = !mpk in
    let pk = snd(keygen) in              let pk = snd(keygen) in
    mpk := inr(pk);                      mpk := inr(pk);
    pk                                   pk

  fun QUERY m =                        fun QUERY m =
    let inr(pk) = !mpk in                let inr(pk) = !mpk in
    let inl() = !flag in                 let inl() = !flag in
    flag := inr();                       flag := inr();
    enc pk m                             sample(el × el)
```

Fig. 5. Example: Definition of games PK−OTS$ specialized for ElGamal.

games was played. If the cryptographic protocol is insecure, the adversary can identity with non-negligible probability the correct game that was played. In our setting, following [4], games and adversaries can be constructed by defining and composing modules.

Definition 23 (Game)**.** *An* E-game *is a module* G *with type* $\Sigma; \cdot \vdash \mathrm{G} : \mathrm{E}$ *for any* Σ*, i.e. it cannot contain any calls of the form* F(e).

As an example, consider the DDH games from Example 3. Both DDH^0 and DDH^1 are I−DDH-games.

Example 6. *As part of our running example, we now introduce the public key one time secrecy games that are presented along with their interface in Figure 5. We leave it to the reader to verify that*

$$\text{mpk} : \text{unit} + \text{el}, \text{flag} : \text{bool}; \cdot \vdash \text{PK−OTS\$}^0 : \text{I−PK−OTS\$}$$
$$\text{mpk} : \text{unit} + \text{el}, \text{flag} : \text{bool}; \cdot \vdash \text{PK−OTS\$}^1 : \text{I−PK−OTS\$}.$$

Thus, both modules are I−PK−OTS$*-games. As a shorthand we refer to the game pair as* PK−OTS$ = (PK−OTS^0, PK−OTS^1).

An adversary is modelled as a module as well, which exports a single method called RUN. Recall that we introduced the I−ADV for the adversary, introduced in Example 2.

Definition 24 (Adversary)**.** *An* E*-adversary is a module* $\mathcal{A}$ *with type* $\Sigma; \mathrm{E} \vdash \mathcal{A} : \text{I−ADV}$ *for any* Σ.

With definitions for games and adversaries in place we define advantage to be the difference in behavior of the adversary given the two games.

Definition 25 (Advantage)**.** *We define an* E*-adversary* $\mathcal{A}$*'s advantage to distinguish between* E*-games* $\mathrm{G}_1, \mathrm{G}_2$ *as*

$$\begin{aligned}\mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A}) = |\ &\Pr[\textbf{true} \leftarrow \mathrm{RUN}() \propto (\mathcal{A} \circledcirc \mathrm{G}_1)] \\ &-\Pr[\textbf{true} \leftarrow \mathrm{RUN}() \propto (\mathcal{A} \circledcirc \mathrm{G}_2)]\ |\end{aligned}$$

From this definition we can easily derive symmetry

$$\mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A}) = \mathrm{Adv}_{\mathrm{G}_2,\mathrm{G}_1}(\mathcal{A}),$$

and the triangle inequality

$$\mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_3}(\mathcal{A}) \leq \mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A}) + \mathrm{Adv}_{\mathrm{G}_2,\mathrm{G}_3}(\mathcal{A}).$$

With these properties, through a game hopping argument, we can bound the adversary's advantage of distinguishing between a pair of games representing the security property.

Finally, based on this notion of advantage, we introduce the notion of perfect indistinguishability between two games.

Definition 26 (Perfect Indistinguishability)**.** E*-games* G_1 *and* G_2 *are said to be perfectly indistinguishable written* $\mathrm{G}_1 \approx_0 \mathrm{G}_2$ *when*

$$\mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A}) = 0$$

for all E*-adversaries* $\mathcal{A}$.

In contrast to related work and thanks to the use of nominals, our version of perfect indistinguishability is based on separated composition, rendering any consideration regarding disjointness assumptions entirely unnecessary. To our knowledge, this is a significant improvement of other existing works on state-separating proofs [4], [6], [10]. The elegance of our approach shines through in our running example: No disjointness assumptions are necessary.

Example 7. *This example demonstrates, how to express* PK−OTS$ *in terms of* DDH *using perfect indistinguishability. For this, we define a reduction*

```
RED = module
  fun GET () =                    fun QUERY m =
    let inl() = !stop;              let rsh := GETBC();
    stop := inr();                  (fst(rsh), m · snd(rhs))
    GETA()
```

that is well-typed stop : unit + unit; I−DDH ⊢ RED : I−PK−OTS$. *We show that* PK−OTS$^0 \approx_0$ RED ⊙ DDH^0 *and* PK−OTS$^1 \approx_0$ RED ⊙ DDH^1.

Let an I−PK−OTS$*-adversary* $\mathcal{A}$ *be given. To show that*

$$\mathrm{Adv}_{\text{PK\_OTS\$}^0,\text{RED}\circledcirc\text{DDH}^0}(\mathcal{A}) = 0$$

we prove that the functions GET *and* QUERY *exhibit the same sub-distribution of answers for each of the two modules.*

This is proven formally in a probabilistic relational Hoare logic (see Section V). In the interest of space we will resort to the following argument.

We define an invariant to act as the relation between the state of the two games. As long as the support of the relation is limited to the support of the games, the adversary cannot break it, as the atoms of the games are inaccessible to the adversary due to seperated composition. In this proof, the

relation encodes the fact that matches of **inl** *and* **inr** *will have the same outcome, and that under certain conditions* mpk *and* mga *contain the same value.*

Likewise, we formally prove that

$$\mathrm{Adv}_{\mathrm{PK\_OTS\$}^1,\mathrm{RED}\circledcirc\mathrm{DDH}^1}(\mathcal{A}) = 0.$$

Next, we confirm what we would expect to hold.

Theorem 34 (α-equivalence implies perfect indistinguishability)**.** *For all* E*-games* $\mathrm{G}_1, \mathrm{G}_2$*, when* $\mathrm{G}_1 \equiv \mathrm{G}_2$*, then* $\mathrm{G}_1 \approx_0 \mathrm{G}_2$*.*

As the main result, we show that advantage respects perfect indistinguishability. Note that advantage between games is real-valued in the interval $[0, 1]$.

Theorem 35 (Advantage congruence)**.** *For* E*-adversaries* $\mathcal{A}$ *and* $\mathcal{A}'$*, so that* $\mathcal{A} \approx_0 \mathcal{A}'$ *and* E*-games* $\mathrm{G}_1, \mathrm{G}_1', \mathrm{G}_2, \mathrm{G}_2'$ *where* $\mathrm{G}_1 \approx_0 \mathrm{G}_1'$ *and* $\mathrm{G}_2 \approx_0 \mathrm{G}_2'$*.*

$$\mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A}) = \mathrm{Adv}_{\mathrm{G}_1',\mathrm{G}_2'}(\mathcal{A}')$$

This theorem is seldom used in its full generality. Usually it is used in a context where $\mathcal{A} = \mathcal{A}'$.

The consequence of this theorem is that we can freely replace perfectly indistinguishable games in a context consisting of module composition and advantage arithmetic, as in the next example. First, we show how to perform a reduction.

Theorem 36 (Reduction)**.** *For an* E*-adversary* $\mathcal{A}$*,* I*-games* $\mathrm{G}_1, \mathrm{G}_2$ *and a module* R *with type* $\Sigma; \mathrm{I} \vdash \mathrm{R} : \mathrm{E}$ *it holds that*

$$\mathrm{Adv}_{\mathrm{R}\circledcirc\mathrm{G}_1,\mathrm{R}\circledcirc\mathrm{G}_2}(\mathcal{A}) = \mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A} \circledcirc \mathrm{R})$$

We finish our running example by showing the reduction.

Example 8 (PK-OTS reduction)**.** *For all* I–PK–OTS\$*-adversaries* $\mathcal{A}$ *it holds that*

$$\mathrm{Adv}_{\mathrm{PK–OTS\$}}(\mathcal{A}) = \mathrm{Adv}_{\mathrm{DDH}}(\mathcal{A} \circledcirc \mathrm{RED})$$

Since

$$\begin{aligned}&\mathrm{Adv}_{\mathrm{PK–OTS\$}^0,\mathrm{PK–OTS\$}^1}(\mathcal{A})\\ &= \mathrm{Adv}_{\mathrm{RED}\circledcirc\mathrm{DDH}^0,\mathrm{RED}\circledcirc\mathrm{DDH}^1}(\mathcal{A})\\ &= \mathrm{Adv}_{\mathrm{DDH}^0,\mathrm{DDH}^1}(\mathcal{A} \circledcirc \mathrm{RED}),\end{aligned}$$

where the first equality holds by Theorem 35 and the second holds by Theorem 36.

Putting all pieces together, we can now solve the problem presented in the introduction: The triangle inequality can be easily expressed in the setting of nominal state-separating proofs. No disjointness requirements are necessary; the renaming of state variables is taken care by the nominal nature of the operators.

Corollary 37 (Transitivity of perfect indistinguishability)**.** *Given* E*-games* $\mathrm{G}_1, \mathrm{G}_2, \mathrm{G}_3$ *where* $\mathrm{G}_1 \approx_0 \mathrm{G}_2$ *and* $\mathrm{G}_2 \approx_0 \mathrm{G}_3$*, then* $\mathrm{G}_1 \approx_0 \mathrm{G}_3$*.*

V. Mechanization

We apply the theory of nominal state-separating proofs to develop Nominal-SSProve[4] as an extension to the Coq framework SSProve. In Nominal-SSProve, when quantifying over modules, those modules can always be assumed to be disjoint from each other. This means that when working within Nominal-SSProve, disjointness assumptions about state variables are not required in contrast to other implementations of non-nominal state-separating proofs. This renders formalizations of security proofs in Nominal-SSProve considerably easier.

We have mechanized the running example of this paper in Nominal-SSProve, which includes a mechanization of the PK–OTS\$ security property, as shown in earlier sections. While developing this example we discovered two mistakes in the existing mechanization of the PK–OTS\$ proof for ElGamal in SSProve [10]. First, the definition of ElGamal decryption was incorrect. The authors would have noticed that, if they had tried to mechanize the proof of correctness of the ElGamal cryptosystem. Second, the definition of PK–OTS\$ is wrong even after an attempt was made to fix it, as noted in footnote 4. The problem as explained still exists: The adversary cannot access the public key until after the adversary has comitted to a message. To be precise, it is keygen that initializes the location pk with a public key, but this function is only evaluated after a call to Challenge has been made; thus the value is out of reach for the adversary.

The curious reader might have noticed our non-standard formulation of the DDH game in Figure 3 in Section II-C. A standard formulation takes all two or three samples in the same function call; however, we rely on this formulation to complete the formal proof. If we could encode the fact that a value stored in the heap is randomly chosen, we could let the DDH games eagerly take all two or three samples, and let the reduction RED save the last two values until they are needed for the encryption. We leave it as future work to resolve the issue of lazy versus eager sampling in the context of state-separating proofs. The problem exists specifically for formulations in the style of state-separating proofs. If we instead retain control flow and selectively invoke the adversary as in [2], we get to use the fact that the value is randomly chosen.

As for implementation, we develop a general theory of nominal sets in Coq using Hierarchy Builder [5]. We encode the definitions of separated composition operators in Coq and and show that they form the module algebra that we discuss in Section IV-B. When doing proofs in Nominal SSProve, we heavily rely on Coq's congruence system [14] to support the proof mechanization process. In conclusion, we have encoded the entire theory discussed in this paper in Coq, and the interested reader is invited to consult the git repository.

VI. Conclusion

We highlight the problem with ad-hoc disjointness assumptions in mechanizations of state-separating proofs. This is solved by introducing nominal state-sepration proofs, where

[4]See supplementary material.

we rely on active renaming to enforce name separation. The benefit of nominal state-separating proofs is succinct equations used to reason about advantage between games without considering separation. Implementing nominal state-separating proofs as Nominal-SSProve has shown actual improvements: Theorems that quantify over an adversary need not include assumptions about state separation, and rewriting using perfect indistinguishability is seamless; thus resulting in shorter proofs.

References

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, pages 50–65, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[2] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of the 31st Annual Conference on Advances in Cryptology*, CRYPTO'11, page 71–90, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. Key-schedule security for the tls 1.3 standard. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022*, pages 621–650, Cham, 2022. Springer Nature Switzerland.

[4] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 222–249, Cham, 2018. Springer International Publishing.

[5] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic hierarchies Made Easy in Coq with Elpi. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:21, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[6] François Dupressoir, Konrad Kohbrok, and Sabine Oechsner. Bringing state-separating proofs to easycrypt a security proof for cryptobox. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, pages 227–242, 2022.

[7] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[8] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2001.

[9] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. Asynchronous probabilistic couplings in higher-order separation logic. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024.

[10] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. Ssprove: A foundational framework for modular cryptographic proofs in coq. *ACM transactions on programming languages and systems*, 45(3):1–61, 2023.

[11] Andrew M. Pitts. Nominal sets. https://people.cs.nott.ac.uk/pszvc/mgs/MGS2011_nominal_sets.pdf, 2011. [Online; accessed 29-Jan-2025].

[12] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[13] Mike Rosulek. The joy of cryptography, 2020. https://joyofcryptography.com.

[14] Matthieu Sozeau. Generalized rewriting, September 2023. https://coq.inria.fr/doc/v8.18/refman/addendum/generalized-rewriting.html.

[15] The Coq Development Team. The coq proof assistant, September 2023. https://doi.org/10.5281/zenodo.11551177.

Appendix A
Omitted proofs

Proof (Theorem 13). First, we prove that inlining is equivariant i.e. $\pi \cdot (\mathrm{e} \propto \mathrm{M}) = (\pi \cdot \mathrm{e}) \propto (\pi \cdot \mathrm{M})$. We proceed by induction on e and consider the F(e) case with **fun** F x = e′ in M.

$$\begin{aligned}
\pi \cdot (\mathrm{F(e)} \propto \mathrm{M}) &= \pi \cdot ((\lambda \mathrm{x},\ \mathrm{e}')\ (\mathrm{e} \propto \mathrm{M})) \\
&= (\lambda \mathrm{x},\ \pi \cdot \mathrm{e}')\ (\pi \cdot (\mathrm{e} \propto \mathrm{M})) \\
&= (\lambda \mathrm{x},\ \pi \cdot \mathrm{e}')\ ((\pi \cdot \mathrm{e}) \propto (\pi \cdot \mathrm{M})) \\
&= (\pi \cdot \mathrm{F(e)}) \propto (\pi \cdot \mathrm{M})
\end{aligned}$$

We finish the proof by induction over M_1. □

Proof (Lemma 15). We prove that $\mathrm{supp}(\Sigma)$ is the minimal support. Let S be a support set for Σ. Assume for contradiction that $\mathrm{a} \in \mathrm{supp}(\Sigma) \setminus S$. Pick an $\mathrm{a}' \notin S \cup \mathrm{supp}(\Sigma)$ and note that $(\mathrm{a}\ \mathrm{a}') \cdot \Sigma \neq \Sigma$, which is shown by induction over Σ using the fact that a is in $\mathrm{supp}(\Sigma)$. Then S cannot be a support set since neither a or a' is in S, which forces a contradiction; thus $\mathrm{supp}(\Sigma) \setminus S = \emptyset$, which lets us conclude that $\mathrm{supp}(\Sigma) \subseteq S$. The proof for *State* is similar. □

Proof (Lemma 17). Let $\mathrm{a}' \in \mathrm{supp}(x)$ and $\mathrm{a} \in \mathrm{supp}(y)$ be given. We have

$$\begin{aligned}\mathrm{idx}(\mathrm{fresh}(x,y)(\mathrm{a})) &= \mathrm{idx}(\mathrm{a}) + k(x) \\ &\geq \mathrm{idx}(\mathrm{a}) + \mathrm{idx}(\mathrm{a}') \\ &> \mathrm{idx}(\mathrm{a}'),\end{aligned}$$

thus $\mathrm{a} \neq \mathrm{fresh}(x,y)(\mathrm{a})$. □

Proof (Theorem 19). Assume that $\pi_1 \cdot \Sigma_1 = \Sigma_1'$ and $\pi_2 \cdot \Sigma_2 = \Sigma_2'$. Define $\pi' = \mathrm{fresh}(\pi_1 \cdot \Sigma_1, \pi_2 \cdot \Sigma_2)\ \pi_2$ and extend

$$\pi(\mathrm{a}) = \begin{cases} \pi_1(\mathrm{a}) & \text{if } \mathrm{a} \in \mathrm{supp}(\Sigma_1) \\ \pi'(\mathrm{a}) & \text{if } \mathrm{a} \in \mathrm{supp}(\mathrm{fresh}(\Sigma_1, \Sigma_2) \cdot \Sigma_2) \end{cases}$$

to define a permutation. Then

$$\begin{aligned}\pi \cdot (\Sigma_1 * \Sigma_2) &= \pi \cdot \Sigma_1, \pi \cdot \mathrm{fresh}(\Sigma_1, \Sigma_2) \cdot \Sigma_2 \\ &= \pi_1 \cdot \Sigma_1, \mathrm{fresh}(\pi_1 \cdot \Sigma_1, \pi_2 \cdot \Sigma_2) \cdot \pi_2 \cdot \Sigma_2 \\ &= \pi_1 \cdot \Sigma_1 * \pi_2 \cdot \Sigma_2 \\ &= \Sigma_1' * \Sigma_2'.\end{aligned}$$

□

Proof (Lemma 23). By induction on small-step derivations we can show, that $\langle \mathrm{e}; \sigma \rangle \to_p \langle \mathrm{e}'; \sigma' \rangle$ if and only if $\langle \pi \cdot \mathrm{e}, \pi \cdot \sigma \rangle \to_p \langle \pi \cdot \mathrm{e}', \pi \cdot \sigma' \rangle$. Then

$$\begin{aligned}\pi \cdot \mathrm{step}(\mathrm{e}, \sigma)(\mathrm{e}', \sigma') &= \pi \cdot p \\ &= p \\ &= \mathrm{step}(\pi \cdot \mathrm{e}, \pi \cdot \sigma)(\pi \cdot \mathrm{e}', \pi \cdot \sigma').\end{aligned}$$

□

Proof (Lemma 24). We show that bind is equivariant.

$$\begin{aligned}\mathrm{bind}(\pi \cdot p, \pi \cdot f)(\pi \cdot y) &= \sum_{x \in X} p(\pi^{-1} \cdot x) \cdot (\pi \cdot f)(x)(\pi \cdot y) \\ &= \sum_{x' \in X} p(x') \cdot (\pi \cdot f)(\pi \cdot x')(\pi \cdot y) \\ &= \sum_{x' \in X} p(x') \cdot f(x')(y) \\ &= \mathrm{bind}(p, f)(y)\end{aligned}$$

□

Proof (Lemma 28). We weaken the heap of M_1 to obtain $\Sigma_3; \mathrm{I}_1 \vdash \mathrm{M}_1 : \mathrm{E}$. We apply $\pi = \mathrm{fresh}(\mathrm{M}_1, \Sigma_2)$ to obtain $\pi \cdot \Sigma_2; \mathrm{I}_2 \vdash \pi \cdot \mathrm{M}_2 : \mathrm{I}_1$ and weaken the heap to get $\Sigma_3; \mathrm{I}_2 \vdash \pi \cdot \mathrm{M}_2 : \mathrm{I}_1$. Using Lemma 4 we get $\Sigma_3; \mathrm{I}_2 \vdash \mathrm{M}_1 \circ \pi \cdot \mathrm{M}_2 : E$, and since $\mathrm{supp}(\mathrm{M}_2) \subseteq \mathrm{supp}(\Sigma_2)$, then $\pi \cdot \mathrm{M}_2 = \mathrm{fresh}(\mathrm{M}_1, \mathrm{M}_2) \cdot \mathrm{M}_2$, so we have derived $\Sigma_3; \mathrm{I}_2 \vdash \mathrm{M}_1 \mathbin{⊚} \mathrm{M}_2 : E$. □

Proof (Lemma 29). Similar to proof of Lemma 29. □

Proof (Theorem 30). Similar to proof of Lemma 19. □

Proof (Lemma 31). Note that

$$\mathrm{fresh}(\mathrm{ID(E)}, \mathrm{M}) = \mathrm{fresh}(\emptyset, \mathrm{M}) = \mathrm{id},$$

so

$$\begin{aligned}\mathrm{ID(E)} \mathbin{⊚} \mathrm{M} &= \mathrm{ID(E)} \circ \mathrm{M} = M \\ \mathrm{ID}(\textbf{interface}) \parallel \mathrm{M} &= \mathrm{ID}(\textbf{interface}) \mid \mathrm{M} = M\end{aligned}$$

Note that for any interface I, $\pi \cdot \mathrm{ID(I')} = \mathrm{ID(I')}$, so

$$\begin{aligned}\mathrm{M} \mathbin{⊚} \mathrm{ID(I)} &= \mathrm{M} \circ \mathrm{ID(I)} = \mathrm{M} \\ \mathrm{M} \parallel \mathrm{ID}(\textbf{interface}) &= \mathrm{M} \mid \mathrm{ID}(\textbf{interface}) = \mathrm{M}\end{aligned}$$

□

Proof (Lemma 32). Define

$$\begin{aligned}\pi_1 &= \mathrm{fresh}(\mathrm{M}_1, \mathrm{M}_2), \\ \pi_{12} &= \mathrm{fresh}(\mathrm{M}_1 \mathbin{⊚} \mathrm{M}_2, \mathrm{M}_3), \\ \pi_{12}' &= \mathrm{fresh}(\mathrm{fresh}(\mathrm{M}_1, \mathrm{M}_2 \cdot \mathrm{M}_2), \mathrm{M}_3), \\ \pi_1' &= \mathrm{fresh}(\mathrm{M}_1, \mathrm{M}_2 \mathbin{⊚} \mathrm{M}_3), \\ \pi_2' &= \mathrm{fresh}(\mathrm{M}_2, \mathrm{M}_3).\end{aligned}$$

We have

$$\begin{aligned}(\mathrm{M}_1 \mathbin{⊚} \mathrm{M}_2) \mathbin{⊚} \mathrm{M}_3 &= (\mathrm{M}_1 \circ \pi_1 \cdot \mathrm{M}_2) \circ \pi_{12} \cdot \mathrm{M}_3 \\ &= \mathrm{M}_1 \circ (\pi_1 \cdot \mathrm{M}_2 \circ \pi_{12} \cdot \mathrm{M}_3) \\ &\equiv \mathrm{M}_1 \circ (\pi_1 \cdot \mathrm{M}_2 \circ \pi_{12}' \cdot \mathrm{M}_3) \\ &\equiv \mathrm{M}_1 \circ (\pi_1' \cdot \mathrm{M}_2 \circ \pi_1' \pi_2' \cdot \mathrm{M}_3) \\ &= \mathrm{M}_1 \circ \pi_1' \cdot (\mathrm{M}_2 \circ \pi_2' \cdot \mathrm{M}_3) \\ &= \mathrm{M}_1 \mathbin{⊚} (\mathrm{M}_2 \mathbin{⊚} \mathrm{M}_3)\end{aligned}$$

The proof of associativity for parallel composition is similar. □

Proof (Lemma 33). The proof of interchange is similar to that of Lemma 32, but in this case there are four separated modules. In the mechanization, these proofs are completed using light proof automation. □

Proof (Theorem 34). Let E-games $\mathrm{G}_1, \mathrm{G}_2$ and an E-adversary $\mathcal{A}$ be given. Assume that $\pi \cdot \mathrm{G}_1 = \mathrm{G}_2$, then

$$\begin{aligned}&\Pr[\textbf{true} \leftarrow \mathrm{RUN}() \propto (\mathcal{A} \mathbin{⊚} \mathrm{G}_2)] \\ &= \Pr[\textbf{true} \leftarrow \mathrm{RUN}() \propto (\mathcal{A} \mathbin{⊚} \pi \cdot \mathrm{G}_1)] \\ &= \Pr[\textbf{true} \leftarrow \mathrm{RUN}() \propto \pi' \cdot (\mathcal{A} \mathbin{⊚} \mathrm{G}_1)] \\ &= \Pr[\textbf{true} \leftarrow \mathrm{RUN}() \propto (\mathcal{A} \mathbin{⊚} \mathrm{G}_1)]\end{aligned}$$

due to α-congruence for separated composition followed by equivariance, thus $\mathrm{Adv}_{\mathrm{G}_1, \mathrm{G}_2}(\mathcal{A}) = 0$. □

Proof (Theorem 35). We have

$$\begin{aligned}&\mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A})\\&\leq \mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_1{}'}(\mathcal{A}) + \mathrm{Adv}_{\mathrm{G}_1{}',\mathrm{G}_2{}'}(\mathcal{A}) + \mathrm{Adv}_{\mathrm{G}_2{}',\mathrm{G}_2}(\mathcal{A})\\&\leq \mathrm{Adv}_{\mathrm{G}_1{}',\mathrm{G}_2{}'}(\mathcal{A}),\end{aligned}$$

and likewise $\mathrm{Adv}_{\mathrm{G}_1{}',\mathrm{G}_2{}'}(\mathcal{A}) \leq \mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_2}(\mathcal{A})$. □

Proof (Theorem 36). Follows from associativity of separated sequential composition. □

Proof (Corollary 37). Let $\mathcal{A}$ be given. By Theorem 35 we have

$$\mathrm{Adv}_{\mathrm{G}_1,\mathrm{G}_3}(\mathcal{A}) = \mathrm{Adv}_{\mathrm{G}_2,\mathrm{G}_2}(\mathcal{A}) = 0.$$

□

Here are the omitted proofs.