IPDL: A Probabilistic Dataflow Logic for Cryptography

Xiong Fan* Joshua Gancher† Greg Morrisett† Elaine Shi† Kristina Sojakova†

December 6, 2019

Abstract

While there have been many successes in verifying cryptographic security proofs of noninteractive primitives such as encryption and signatures, less attention has been paid to interactive cryptographic protocols. Interactive protocols introduce the additional verification challenge of *concurrency*, which is notoriously hard to reason about in a cryptographically sound manner.

When proving the (approximate) observational equivalance of protocols, as is required by simulation based security in the style of Universal Composability (UC), a *bisimulation* is typically performed in order to reason about the nontrivial control flows induced by concurrency. Unfortunately, bisimulations are typically very tedious to carry out manually and do not capture the high-level intuitions which guide informal proofs of UC security on paper. Because of this, there is currently a large gap of formality between proofs of cryptographic protocols on paper and in mechanized theorem provers.

We work towards closing this gap through a new methodology for iteratively constructing bisimulations in a manner close to on-paper intuition. We present this methodology through Interactive Probabilistic Dependency Logic (IPDL), a simple calculus and proof system for specifying and reasoning about (a certain subclass of) distributed probabilistic computations. The IPDL framework exposes an *equational* logic on protocols; proofs in our logic consist of a number of rewriting rules, each of which induce a single low-level bisimulation between protocols.

We show how to encode simulation-based security in the style of UC in our logic, and evaluate our logic on a number of case studies; most notably, a semi-honest secure Oblivious Transfer protocol, and a simple multiparty computation protocol robust to Byzantine faults. Due to the novel design of our logic, we are able to deliver mechanized proofs of protocols which we believe are comprehensible to cryptographers without verification expertise. We provide a mechanization in Coq of IPDL and all case studies presented in this work.

1 Introduction

With new decentralized computing paradigms such as blockchains and cloud outsourcing, the community's pace at designing and rolling out new, efficient cryptographic protocols has accelerated. Many of the protocols being deployed involve rich building blocks such as commitment schemes, zero-knowledge proofs, oblivious transfer, and multi-party computation. As a result, it has become increasingly important to scale up the effort of proving cryptographic protocols secure. An important and necessary step towards this goal is to enable *easy-to-use* computer-aided proof systems for general and possibly complex cryptographic protocols.

*University of Maryland. Email: xfan@cs.umd.edu. Part of this work was done while at Cornell.

†Cornell University. Email: jrg358@cornell.edu, elaine@cs.cornell.edu, greg.morrisett@cornell.edu, sojakova.kristina@gmail.com

Existing systems for mechanically verifying cryptography are either symbolic in nature [BSCS18, MSCB13, DY83, Cre08] and therefore do not exactly match the computational reduction-style of reasoning that underlies the mathematical foundation of cryptography, or do not provide much support for encoding general protocols (e.g., EasyCrypt [BGHZ11], CryptVerif [Bla06], and FCF [PM15]). Moreover, except for a few recent/concurrent endeavors [CSV19, LSBM19], most prior systems do not provide good support for reasoning about general multi-party cryptographic protocols and their composition.

In the cryptography literature, Universal Composition (UC) [Can01] is the *de facto* framework for modeling distributed cryptographic protocols and conduct compositional security reasoning. To prove a protocol secure in the UC framework, one must prove it *approximately equivalent* to a particular idealization. This is typically done through a number of low-level (approximate) equivalences to intermediate protocols, each of which either simplifies the protocol in some way or applies a cryptographic hardness assumption.

In order to formally prove one of these low-level protocol equivalences correct, the formal methods community typically employs a *bisimulation* argument (e.g., as seen in EasyUC [CSV19], FCF [PM15], and CryptHOL [LSBM19]), which requires one to construct a particular relational invariant between (distributions of) states of the two protocols which satisifes certain behavioral properties. Unfortunately, it is quite unsatisfactory to use bisimulations directly to reason about cryptographic protocols, since bisimulations induce overly technical proofs which do not match cryptographers' style of reasoning.

1.1 Our Contributions

Our goal is to design a new technique for proving approximate equivalences of cryptographic protocols that is *easy-to-use* and matches the style of on-paper cryptographic proofs. We do so by constructing a new methodology based on *channel dependency* which does not require the end user to manually construct bisimulations. Instead, the end user inteacts with an *equational logic* in order to conduct cryptographic proofs. In this work, we introduce Interactive Probabilistic Dependency Logic (IPDL), one such equational logic for cryptographic protocols.

For simplicity, we currently only consider a certain important subclass of protocols (those which do not make use of nontrivial control flow); however, we believe our proof technique can be readily extended to more complicated scenarios. More detail about this limitation (and others) can be found in Section 9.

A probabilistic logic capturing reactions and dependencies IPDL encodes a distributed computation using *channels* and *reactions*: when all inputs to a reaction have been collected, the reaction is triggered and a value is written to its output channel. This style of encoding features *dependency among channels* (both dependency in *timing* and *value*) as a first-class citizen, since each reaction effectively captures the dependency of the output events on the input events. Since many cryptographic proofs are fundamentally about removing dependencies (e.g., removing a message from a ciphertext), we believe our dependency-centric perspective enables proofs which are often preferable to manually constructing bisimulations. Our perspective can be seen as following in the spirit of equational security [MT13].

In IPDL, a protocol π is expressed as the collection of code for each protocol participant. Channels that are not internally consumed by π form the interface of π to the outside world, including interfaces to an external environment who provides inputs to the protocol and receives outputs and interfaces to the adversary. In this way, IPDL allows us to capture UC-style protocol proofs: to show that π_R implements π_I, we can define a simulator Sim in IPDL which converts attacks on

π_R to attacks in π_I. We can then ask in IPDL whether the execution of π_R is (approximately) equivalent to the execution of $\pi_I||\mathsf{Sim}$, where the notion of equivalence implicitly and universally quantifies over the environment that connects to the external interfaces of the protocol.

A compositional semantics for IPDL Proofs in IPDL can be obtained by applying a set of high-level rewrite rules defined over channels and dependencies. The main judgement in IPDL is that $\Sigma \leftrightsquigarrow_\varepsilon \Sigma'$, meaning that (informally) no environment interacting with Σ can distinguish it from Σ' with probability better than ε. We prove the equational logic of IPDL sound for approximate protocol equivalence through corresponding each rewrite rule to a low-level bisimulation.

The semantics for protocols in our setting is quite subtle, since concurrency is usually formalized using nondeterminism. In the presence of cryptography, however, reasoning about nondeterminism is undesirable, since it intuitively corresponds to *demonic choice*, which leads to an unrealistic attacker model. We take an approach similar to CryptHOL [LSBM19] and UC [Can01], in which all internal computations which usually are resolved through nondeterminism are instead resolved by the protocol agents themselves. Unlike previous approaches, however, the user of our system is not exposed to the technical formalities this solution induces in resolving this internal choice.

Implementation in Coq We have implemented the IPDL logic in Coq as well as mechanized proofs for all of the case studies considered in this paper. Our Coq implementation is publicly available at https://github.com/ipdl/ipdl. To enable succinct mechanized proofs, we provide a library of tactics which automatically apply our rewriting rules, hiding low-level details of the proof from the user.

Case studies To illustrate the usefulness of IPDL, we present case studies for several cryptographic protocols, including 1) realizing a secure channel from an authenticated channel under trusted setup and an encryption scheme with suitable security; 2) the ElGamal encryption system; 3) a UC-secure rock-paper-scissors protocol which employs UC-secure commitments; and 4) an oblivious transfer (OT) scheme realized from trapdoor permutations. All case studies have been mechanized in Coq and come with our open source repository. We demonstrate in Section 7 that IPDL delivers proofs which scale well with the complexity of the protocol in question.

In summary, we provide a mechanized proof system for cryptographic protocols expressed naturally as an equational logic on protocols. IPDL is distinctive in that channel dependency is captured as a first-class notion. Such a dependency-centric design allows every step of a cryptographic proof to be expressed using a simple and concise rewrite rule. We believe that our dependency-based perspective has the potential to deliver concise and easy-to-understand proofs. As an example, for the most sophisticated case study we consider, Oblivious Transfer, the full definitions and proofs contain 926 lines of Coq code; we argue this is a quite reasonable number, given the complexity of the protocol.

The full version, containing all IPDL examples and proofs, can be found at https://github.com/ipdl/ipdl.

2 IPDL by Example: Authenticated to Secure Channel

As an introductory example, we will consider a simple protocol that constructs a one-time use secure communication network between a sender S and a receiver R from an authenticated network and a trusted setup for a symmetric encryption key. (This example is similar to the main example in [Mau11].) For simplicity, we do not model message size, but instead only consider messages of

some arbitrary but fixed length. It is straightforward to extend this example to support a dynamic message length.

The construction is modeled as an interactive protocol π_R between participants S, R, the actual network $\mathcal{F}_{\mathsf{auth}}$, a key functionality $\mathcal{F}_{\mathsf{key}}$, and an implicit external context. The context is thought of as serving two roles: an *environment*, supplying and receiving external inputs and outputs from the parties, and an *attacker*, who interacts with the attack surface of the protocol. Channels meant for the attacker are written in red, while those for the external environment are written in blue. The sender receives an input message m from the external environment, a key k from the trusted setup, and sends the value $c \xleftarrow{\$} \mathsf{Enc}(k, m)$ to the network. The network leaks this ciphertext c to the attacker, and waits for a confirmation ans_R from the attacker before delivering the ciphertext to the receiver, who may decrypt the ciphertext using k, and output the decrypted message.

In Figure 1, we see the protocol π_R encoded in IPDL. (Strictly speaking, we see the *uncorrupted execution* of π_R. This terminology is described in Section 4. In this section, we use the term "protocol" loosely, but define it formally later.) Protocols in IPDL are collections of *inputs* and *reactions*, which interact through *channels*. Channels are used for external input, output, as well as internal hidden computations. The first line, in_S : msg inp, names a hidden channel in_S of type msg, and declares it an external input to the protocol. All channels may only be assigned to once. Lines 2-4 implement the trusted setup $\mathcal{F}_{\mathsf{key}}$ through three reactions: the first samples from the distribution $\mathsf{Rnd}(\mathcal{K})$ and assigns the result to a local channel keygen, while the next two simply copy this result to key_S and key_R. Reactions may fire only when all input to the reaction is available and the output channel of the reaction has not been assigned to yet. When this is the case, we sample from the corresponding distribution and assign the return value to the output channel.

Line 5 corresponds to the code of the sender: when the input message m and the key k are available, the local channel send_R^S is probabilistically assigned to from the distribution $\mathsf{Enc}(k, m)$. This ciphertext is then copied to the channel leak on the next line. This channel leak is tagged with a *visibility label* vis, meaning that the external adversary is able to read from this channel. Channels without a visibility label are implicitly tagged with hid, meaning that the channel is hidden from the context.

Next, we wait for an answer from the adversary on channel ans_R. (If no type is present, it is implicitly of unit type.) This channel models a network adversary that can control if and when the message gets delivered. Once the adversary has given ans_R, we deliver the ciphertext on send_R^S to the sender through the channel deliv. The sender may now decrypt the ciphertext, using the input from deliv and the key key_R.

Our main goal of IPDL is to keep the formal framework surrounding protocols as simple as possible, while still maintaining enough expressive power to do interesting cryptographic proofs. This is done by restricting our scope in two ways: first, as discussed above, channels may only be written once. This keeps the value of all channels unambiguous, and guarantees that our protocols will terminate. Second, all behaviors on channels are constrained to only be a function of the input values of the channel. Thus, we do not consider protocols in which a channel's value depends on the order in which input arrives.

Reactions are generally of the form $\Gamma \vdash \mathsf{c}\ \ell : \tau \leftarrow D$, read as "once all input channels in Γ have been set, if c has not been set, sample from D and write its value of type τ to c." As discussed above, $\ell \in \{\mathsf{hid}, \mathsf{vis}\}$ controls whether the channel is visible to the outside world. We model non-probabilistic assignments through the syntactic sugar $\Gamma \vdash \mathsf{c}\ \ell : \tau := e$, which is an abbreviation for $\Gamma \vdash \mathsf{c}\ \ell : \tau \leftarrow \delta(e)$, where $\delta(e)$ is the point mass distribution which assigns probability 1 to the value at e.

1. $\quad \mathsf{in}_S : \mathsf{msg}\ \mathsf{inp}$
2. $\quad \cdot \vdash \mathsf{keygen} : \mathsf{key} \leftarrow \mathsf{Rnd}(\mathcal{K})$
3. $\quad k : \mathsf{keygen} : \mathsf{key} \vdash \mathsf{key}_S : \mathsf{key} := k$
4. $\quad k : \mathsf{keygen} : \mathsf{key} \vdash \mathsf{key}_R : \mathsf{key} := k$
5. $\quad m : \mathsf{in}_S : \mathsf{msg}, k : \mathsf{key}_S : \mathsf{key}$
   $\vdash \mathsf{send}_R^S : \mathsf{ct} \leftarrow \mathsf{Enc}(m, k)$
6. $\quad c : \mathsf{send}_R^S : \mathsf{ct} \vdash \mathsf{leak}\ \mathsf{vis} := c$
7. $\quad \mathsf{ans}_R\ \mathsf{inp}$
8. $c : \mathsf{send}_R^S : \mathsf{ct}, \mathsf{ans}_R \vdash \mathsf{deliv} : \mathsf{ct} := c$
9. $\quad c : \mathsf{deliv} : \mathsf{ct}, k : \mathsf{key}_R : \mathsf{key}$
   $\vdash \mathsf{out}_R\ \mathsf{vis} : \mathsf{msg} := \mathsf{Dec}(c, k)$

Figure 1: The (uncorrupted) execution of the secure network protocol π_R expressed in IPDL. Each line is either a locally controlled reaction of the form $\Gamma \vdash \mathsf{c} : \tau \leftarrow D$, or an input of the form $\mathsf{c} : \tau\ \mathsf{inp}$. Locally controlled reactions are hidden by default; visible reactions are marked vis.

Reasoning in IPDL The main feature in IPDL is that protocol events are naturally structured according to their *dependency* on other events. This is in contrast to more conventional program logics, which impose a linear order on the statements in programs. Structuring our protocols in terms of their dependency allows us to reason easily about equivalence of protocols.

Since we know exactly what distribution the values of a channel may take given its input, we are able to manipulate channels not only by rewriting their output distributions to equivalent ones, but by rewriting their *dependency* to equivalent ones. Consider the channel out_R in Figure 1: it decrypts a ciphertext c coming from the channel deliv, where deliv is simply copied from the channel send_R^S. Additionally, we know from the protocol that key_S and key_R are equal, when both are defined. We may reflect this information in the protocol by performing a sequence of rewrites to obtain that out_R is equivalent to the reaction

$$c : \mathsf{send}_R^S : \mathsf{ct}, k : \mathsf{key}_S : \mathsf{key}, \mathsf{ans}_R \vdash \mathsf{out}_R\ \mathsf{vis} : \mathsf{msg} := \mathsf{Dec}(c, k),$$

thus essentially performing a partial evaluation of the protocol in order to propagate the value of send_R^S to out_R. We may then perform additional rewrites on the protocol to obtain that this reaction is equivalent to

$$m : \mathsf{in}_S : \mathsf{msg}, k : \mathsf{key}_S : \mathsf{key}, \mathsf{ans}_R \vdash \mathsf{out}_R\ \mathsf{vis} : \mathsf{msg} := m,$$

by using the correctness property of the encryption scheme. Now, we see that out_R no longer uses the value of k: in certain cases, we may then soundly *remove* dependencies from reactions, producing the following simplified reaction for out_R:

$$m : \mathsf{in}_S : \mathsf{msg}, \mathsf{ans}_R \vdash \mathsf{out}_R\ \mathsf{vis} : \mathsf{msg} := m.$$

Protocol Simulation We express protocol security in the style of UC [Can01] by using IPDL to express and prove equivalence of protocols. In Figure 2 we have an idealized version of a secure channel. The sender and receiver do not communicate using an encryption scheme, but instead in cleartext. However, the network functionality only reports the message query to the adversary, and does not leak any information about the ciphertext. Similarly to π_R, the network here waits for the adversary to send ans_I before delivering the message.

$$\mathsf{in}_S : \mathsf{msg}\ \mathsf{inp}$$
$$m : \mathsf{in}_S : \mathsf{msg} \vdash \mathsf{send}_I^S : \mathsf{msg} := m$$
$$m : \mathsf{send}_I^S : \mathsf{msg} \vdash \mathsf{query}\ \mathsf{vis}$$
$$\mathsf{ans}_I\ \mathsf{inp}$$
$$m : \mathsf{send}_I^S : \mathsf{msg}, \mathsf{ans}_I \vdash \mathsf{deliv} : \mathsf{msg} := m$$
$$m : \mathsf{deliv} : \mathsf{msg} \vdash \mathsf{out}_R\ \mathsf{vis} : \mathsf{msg} := m$$

Figure 2: The idealized secure network π_I expressed in IPDL.

We wish to argue that π_R somehow performs the same job as π_I. Note that the input and output channels for the two parties are the same across π_R and π_I; the only input and visible channels that differ are those which correspond to the respective adversaries. In π_R, the adversary learns the ciphertext from leak, and chooses to deliver that ciphertext using ans_R; in π_I, the adversary only learns the message has been sent using query, and chooses to deliver the message using ans_I.

In order for π_R to be as effective as π_I, we need a way of *converting* all adversarial attacks on π_R to attacks on π_I. If we can invent a converter Sim (also called the *simulator*) which converts attacks on π_R to attacks on π_I, then we can ask in IPDL whether π_R is equivalent to $\mathsf{Sim}||\pi_I$, where $||$ is the parallel composition operator in IPDL. Details on our composition operator are given in Section 3. If such a Sim exists, we say that π_R *realizes* π_I. Crucially, this simulator only has access to the channels designated for the adversary. Following UC, we may also *corrupt* certain parties in both protocols, and require the corrupted protocols also satisfy this realization property. More details on simulators, corruption, and protocol realization is in Section 4.

In the case when neither party is corrupted, the simulator is standard: for Sim to convert an attack on π_R to an attack on π_I, it will receive a query message from π_I and need to deliver a corresponding leak message to π_R. It does not have access to the key nor the message, so it will generate a fresh key k' from $\mathsf{Rnd}(\mathcal{K})$ and output a ciphertext from $\mathsf{Enc}(0, k')$, where 0 is any distinguished message from the message space. More details on this simulator are given in Section 5.

Approximate compositional reasoning In order to show that π_R is equivalent to $\mathsf{Sim}||\pi_I$, we must apply a cryptographic assumption about the encryption scheme; namely, that over the randomness of k, ciphertexts coming from $\mathsf{Enc}(m, k)$ look identical to ciphertexts coming from $\mathsf{Enc}(0, k)$ for all m. In Figure 3, we express this cryptographic assumption as an approximate equivalence of two reaction sets OSS_R and OSS_I (OSS standing for *one-shot security*). In order to apply this assumption to π_R and $\mathsf{Sim}||\pi_I$, we first must rewrite π_R to be equivalent to $\pi'_R||\mathsf{OSS}_R$. Thus, we "factor out" the encryption performed in π_R to OSS_R. This rewrite is made possible by our simplifications on channel out_R from above, wherein we use the correctness property of the encryption scheme to essentially give the plaintext value directly to the receiver.

Once this factoring is done, we make use of a general rule in IPDL which states that the equivalence notion $\leftrightsquigarrow$ is a congruence: since OSS_R is (approximately) equivalent to OSS_I by assumption, we know that $\pi_R \leftrightsquigarrow \pi'_R||\mathsf{OSS}_R$ is approximately equivalent to $\pi'_R||\mathsf{OSS}_I$, where the encryption is replaced with a dummy ciphertext. We may then use the logic to deduce that $\pi'_R||\mathsf{OSS}_I$ is equivalent to $\mathsf{Sim}||\pi_I$, which concludes our proof. More detail about this proof (as well as a security proof when the sender is malicious) is given in Section 5.

In Section 3, we describe IPDL and its equational logic in detail; in Section 4, we present a general framework for expressing cryptographic security in IPDL, including both semi-honest and malicious corruption models; in Sections 5 and 6, we present multiple case studies of cryptographic proofs in IPDL. Finally, in Section 7, we discuss our mechanization of IPDL in Coq.

$$\begin{array}{l} \mathsf{in}_{\mathsf{OSS}} : \mathsf{msg\ inp} \\ \cdot \vdash \mathsf{keygen} : \mathsf{key} \leftarrow \mathsf{Rnd}(\mathcal{K}) \\ m : \mathsf{in} : \mathsf{msg}, k : \mathsf{keygen} : \mathsf{key} \\ \quad \vdash \mathsf{out}_{\mathsf{OSS}}\ \mathsf{vis} : \mathsf{ct} \leftarrow \mathsf{Enc}(k, m) \end{array} \qquad \begin{array}{l} \mathsf{in}_{\mathsf{OSS}} : \mathsf{msg\ inp} \\ \cdot \vdash \mathsf{keygen} : \mathsf{key} \leftarrow \mathsf{Rnd}(\mathcal{K}) \\ m : \mathsf{in} : \mathsf{msg}, k : \mathsf{keygen} : \mathsf{key} \\ \quad \vdash \mathsf{out}_{\mathsf{OSS}}\ \mathsf{vis} : \mathsf{ct} \leftarrow \mathsf{Enc}(k, 0) \end{array}$$

Figure 3: Definition of one-shot security of an encryption scheme in IPDL. The top reaction set, OSS_R, is approximately equivalent to OSS_I. (Our definition of approximate equivalence is given in Section 3.)

3 Logic

Protocols in our logic are modeled as *reaction sets*, whose syntax is shown in Figure 4. Reaction sets operate on a set of *channels*, which may model I/O communication with the environment as well as internal computation through hidden channels. Input channels are defined through the syntax $c\ \mathsf{inp} : \tau$, which declares the channel c as an external input of type τ. Output and hidden channels are modeled through *reactions* of the form $x_1 : c_1 : \tau_1, \ldots, x_n : c_n : \tau_n \vdash c\ \ell : \tau \leftarrow D$, which says that if for all $i \in \{1, \ldots, n\}$, c_i is bound to a value x_i of type τ_i, and if c is not yet bound to a value, then sample from the distribution D, ranging over values of type τ, and bind the result to c. Thus, once a value is set to a channel, it remains set on the channel for the entire execution. The left side $x_1 : c_1 : \tau_1, \ldots, x_n : c_n : \tau_n$ is called the *context* of the channel c. The label ℓ may be either out or hid, denoting if the channel is a hidden internal computation, or an output to an external environment. Reactions written without a label are implicitly marked hidden.

Our logic is parameterized over a universe of types τ including products and probability distributions. All probabilistic reasoning in IPDL is factored out into an equational theory $\mathcal{E}$ supporting

$$\begin{array}{lcll} \tau & := & \cdots \mid \tau \times \tau \mid \mathcal{D}(\tau) & \text{types} \\ \ell & := & \mathsf{out} \mid \mathsf{hid} & \text{output/hidden channels} \\ \Gamma & := & \emptyset \mid x : c; \Gamma & \text{reaction context} \\ \Sigma & := & \emptyset \mid \Gamma \vdash c\ \ell \leftarrow D; \Sigma & \text{locally controlled reactions} \\ & & \mid c\ : \tau\ \mathsf{inp}; \Sigma & \text{inputs} \end{array}$$

Figure 4: Main Syntax of IPDL.

the judgement $\Gamma \vDash_{\mathcal{E}} D = D'$, meaning that for all valuations of the variables in Γ, the distributions D and D' are equivalent. A channel c is *defined* in reaction set Σ if it appears in Σ as an input, output, or hidden channel. Given a reaction set Σ, we define the sets $\mathsf{out}(\Sigma)$ to be the output channels set in Σ, and similarly for $\mathsf{in}(\Sigma)$ and $\mathsf{hid}(\Sigma)$. We assume the following well-formedness conditions on reaction sets Σ and contexts Γ:

1. All variable names x_i in Γ are distinct.
2. No duplicate channels exist in Σ.
3. All appearances of a channel in Σ have the same type.

In particular, the above conditions imply that the sets $\mathsf{out}(\Sigma)$, $\mathsf{in}(\Sigma)$, and $\mathsf{hid}(\Sigma)$ are pairwise disjoint.

We now define a parallel composition operator for reaction sets. Two reaction sets Σ and Σ' are *compatible* if their output channels are disjoint, and no hidden channel in one reaction set appears in the other. For compatible Σ and Σ', we define the union $\Sigma \cup \Sigma'$ to be the union of all the hidden and output channels in $\Sigma \cup \Sigma'$, plus all the input channels of Σ and/or Σ' that do not simultaneously appear as outputs of the other reaction set. If c is an output channel of a reaction set Σ, we denote by $\Sigma \,\mathsf{hide}\, c$ the reaction set that defines c to be a hidden, rather than an output channel. Finally, for compatible Σ and Σ' we denote by $\Sigma \parallel \Sigma'$ the union of Σ and Σ' followed by a successive hiding of all the output channels of Σ and/or Σ' that simultaneously appear as inputs to the other reaction set.

3.1 Semantics

We model IPDL protocols in a manner similar to CryptHOL [LSBM19] and UC [Can01]. Similarly to an IPDL protocol, a protocol agent has a set of typed *input* and *output* channels. Input channels are partitioned into two kinds: *token input channels* and *listener input channels*; the latter allow a certain form of broadcast in our semantics. A *listener input* I_L is a pair of a listener input channel I and a value v of the type τ associated to the channel I; similarly, we define token inputs I_T and outputs O to be pairs of channels and values on that channel.

Unlike an IPDL protocol, a general protocol agent does not have any explicit hidden channels. Instead, the behavior of a protocol agent is completely determined by a set S of states (with a distinguished start state s) and two transition functions μ and ν:

- The transition function μ for listener inputs associates to each state e and each listener input I_L a distribution $\mathcal{D}(S)$ on states. Intuitively, this models the fact that the protocol agent learns the value v sent on the channel I but does not take control of the computation.
- The transition function ν for token inputs associates to each state e and each token input I_T a distribution $\mathcal{D}\big(S \times (1 + O)\big)$ on states and, optionally, outputs. This models the fact that the protocol agent learns the value v sent on the channel I and takes control of the computation, optionally executing an output.

Similarly to IPDL programs, we say that two protocol agents are *compatible* if their respective token input channels and their respective output channels are disjoint, and all appearances of the same channel common to both have the same type. The same channel C of type τ can appear, *e.g.*, as a listener input to both protocol agents, as a listener input to one and a token input to the other, or as an input (of either kind) to one and an output of the other. A *protocol* σ is a finite family of pairwise compatible protocol agents. Every channel appearing in σ can be classified as one of three kinds:

- a *free input channel* is a channel that does not appear as an output of any protocol agent of σ
- a *free output channel* is a channel that does not appear as a token input of any protocol agent of σ
- an *internal channel* is a channel that appears as both an output and a token input channel of some protocol agent(s) of σ

Two protocols are *comparable* if their respective sets of free input channels, free output channels, and internal channels coincide. Two protocols σ and σ' are *compatible* if the protocol agents selected from each protocol are pairwise compatible, no internal channel of σ appears in σ' and vice versa, and no free output of σ appears as a free output of σ'. In this case the union $\sigma \parallel \sigma'$ of σ and σ' forms a new protocol.

In contrast with other formalisms such as Task PIOA [CCK+18], this form of modeling protocols does not involve any nondeterministic choice; instead, the control flow of the protocol is entirely determined by the protocol agents themselves who decide which message to send next. To deterministically execute a protocol σ consisting of n protocol agents $P_1, \ldots, P_n$, we assume a *context* $\mathcal{C}$, specified by the following data:

- A protocol agent A ("adversary") that has no listener inputs, whose token input channels are a subset of the free output channels of σ, and whose output channels are a subset of the free input channels of σ. This in particular means that A does not have access to any communication happening on the channels internal to σ.
- An *initial distribution* $\mathcal{D}\big(S \times (1 + O)\big)$ on the states and possible outputs of A.
- A *yield transition function* ξ associating to each state e and each (token) input I_T of A a distribution $\mathcal{D}\big(S \times (1 + O)\big)$ on the states and possible outputs of A.

The adversary A thus adaptively generates free inputs to σ while observing the communication on the free outputs of σ. The initial distribution specifies how to start the execution of σ, whereas the yield transition function ξ_i specifies how to continue the execution of σ in the case when some protocol agent (but the adversary does not learn which) received a token input but has not followed by an output (hence *yielded* to the adversary). The effects of the initial distribution and the yield transition functions do not factor into the trace of the execution since they do not exist as inputs to A.

If σ has a free input channel that does not appear as an output channel of A, then no communication on this channel is possible. On the other hand, if there is a free output channel of σ that does not appear as an input channel to σ, then any message sent on this channel will necessarily terminate the protocol. Likewise, if an agent P_i yields to A and A does not follow by an output, the protocol terminates. This is also the case if the initial distribution does not induce an output (this case is the least interesting but we allow it for consistency).

A context $\mathcal{C}$ for a protocol σ deterministically induces a distribution on executions of length $\leq k$, and thus on execution traces of length $\leq k$ (which we denote by $\mathsf{trDist}_k(\sigma)$), by executing σ for k steps for each $k \in \mathbb{N}$. Following [CCK+06a, CCK+06b, CCK+18], we compare protocols by comparing their trace distributions: let $\varepsilon(\tau, \mathcal{C}, n)$ be a mapping from protocols, contexts, and natural numbers to real numbers in $[0, 1]$. Then, given two comparable protocols σ and σ', we say that $\sigma \leftrightsquigarrow_\varepsilon \sigma'$ if for all compatible protocols τ and contexts $\mathcal{C}$ for $\sigma \parallel \tau$ (or $\sigma' \parallel \tau$), and all $k \in \mathbb{N}$, $d(\mathsf{trDist}_k(\sigma), \mathsf{trDist}_k(\sigma')) \leq \varepsilon(\tau, \mathcal{C}, k)$ where $d(D_1, D_2) = \sup_x |D_1(x) - D_2(x)|$ is the TV-distance on distributions.

We call this $\varepsilon(\tau, \mathcal{C}, k)$ an *error*, and will write 0 for the error that maps all protocols, contexts, and trace lengths to zero. It is easy to see that this notion of protocol equivalence is an equivalence relation and a congruence.

3.2 Reaction sets as protocol agents

We present a translation from reaction sets Σ to protocol agents $[\![\Sigma]\!]$, as described above. As an intermediate step, we first describe a natural semantics for reaction sets as a certain kind of probabilistic I/O automaton [CCK+06a, CCK+06b, CCK+18]. The state space is a function mapping each channel $c : \tau$ of Σ to a value of type $1+\tau$, *i.e.*, the channel c has either not been set yet or is set to a value of its assigned type. The start state is when each channel is undefined. Given a state s, a channel c defined by the reaction $x_1 : c_1 : \tau_1, \ldots, x_n : c_n : \tau_n \vdash c\ \ell : \tau \leftarrow D$ induces a distribution on states as follows. If c is already set in s, it leaves s unchanged; if any of the channels $c_1, \ldots, c_n$ are not yet set, it leaves s unchanged; otherwise we draw v from the distribution $D(v_1, \ldots, v_n)$, where v_i is the value of channel c_i, and set the value of c to v. A *schedule* is a sequence of hidden or output channels and any schedule deterministically induces a distribution on states.

To turn this automaton into a protocol agent, we proceed as follows. The listener input channels are the input channels of the reaction sets. The effect of an input v arriving on an input channel i of type τ has the effect of setting the value of i in the state s to v if it has not yet been set, and leaving s unchanged otherwise. For each output channel o, we have one token input get_o, and one output o. To describe the effect of get_o on a state s, we proceed as follows. Call a schedule *proper* for the pair (s, o) if it has the following properties:

- Each channel in the schedule is a hidden channel that o transitively depends on.
- Whenever o depends on a hidden channel h, then either h has already been set in s or h is in the schedule.
- Whenever h_2 transitively depends on a hidden channel h_1 and h_2 is in the schedule, then either h_1 is already set in o or h_1 precedes h_2 in the schedule.

Proper schedules have some important properties:

- If all the visible channels that o transitively depends on have been set in s, a proper schedule guarantees that o will be set after the schedule has been executed.
- Without loss of generality we can assume that every channel h appears at most once in a proper schedule.
- If h_1 immediately precedes h_2 in a proper schedule, and h_2 does not depend on h_1, then we can swap the order of h_1 and h_2 in the schedule without altering the resulting distribution on states.
- Any two proper schedules induce the same distribution on states.

To execute get_o, we first check whether o is already set in the state s; if so, we return the value of o. Otherwise we check whether every visible channel that o transitively depends on has been set in s; if not, we leave the state unchanged and yield as there is no way for us to compute o. Otherwise we select an arbitrary proper schedule for (s, o); if such a schedule does not exist (perhaps due to a circular dependency on channels such as $h \vdash h$) then we again leave the state unchanged and yield. Otherwise we execute this proper schedule followed by o to induce a distribution on states, and for each state s' in the support we output the value of the channel o in s'.

$$\frac{\Sigma, \Sigma' \text{ permutations}}{\Sigma \leftrightsquigarrow \Sigma'} \text{ [Perm-}\Sigma\text{]} \qquad \frac{\Gamma, \Gamma' \text{ permutations}}{\Sigma\{\Gamma \vdash c\ \ell \leftarrow D\} \leftrightsquigarrow \Sigma\{\Gamma' \vdash c\ \ell \leftarrow D\}} \text{ [Perm-}\Gamma\text{]}$$

$$\frac{\Sigma \leftrightsquigarrow_{\varepsilon} \Sigma' \text{ and } \Sigma' \leftrightsquigarrow_{\varepsilon'} \Sigma''}{\Sigma \leftrightsquigarrow_{\varepsilon+\varepsilon'} \Sigma''} \text{ [Trans]} \qquad \frac{\Gamma \vDash_{\mathcal{E}} D = D'}{\Sigma\{\Gamma \vdash c\ \ell \leftarrow D\} \leftrightsquigarrow \Sigma\{\Gamma \vdash c\ \ell \leftarrow D'\}} \text{ [Ext]}$$

$$\frac{h \notin \Sigma}{\Sigma\{\Gamma \vdash h\ \mathsf{hid} \leftarrow D;\ \Gamma', \Gamma, x : h \vdash c\ \ell \leftarrow D'\} \leftrightsquigarrow \Sigma\{\Gamma', \Gamma \vdash c\ \ell \leftarrow (x \leftarrow D; D')\}} \text{ [Fold]}$$

$$\frac{x \notin FV(D')}{\Sigma\{\Gamma, x : c \vdash c'\ \ell \leftarrow D; \Gamma', y : c' \vdash c''\ \ell' \leftarrow D'\} \leftrightsquigarrow \Sigma\{\Gamma, x : c \vdash c'\ \ell \leftarrow D; \Gamma', x : c, y : c' \vdash c''\ \ell' \leftarrow D'\}} \text{ [Dep-Trans]}$$

$$\frac{}{\Sigma\{\Gamma \vdash c\ \ell := e; \Gamma', \Gamma, x : c \vdash c'\ \ell' \leftarrow D\} \leftrightsquigarrow \Sigma\{\Gamma \vdash c\ \ell := e; \Gamma', \Gamma, x : c \vdash c'\ \ell' \leftarrow D[x := e]\}} \text{ [Subst]}$$

$$\frac{x \notin FV(D')}{\Sigma\{\Gamma \vdash h\ \mathsf{hid} \leftarrow D; \Gamma', \Gamma, x : h \vdash c\ \ell \leftarrow D'\} \leftrightsquigarrow \Sigma\{\Gamma \vdash h\ \mathsf{hid} \leftarrow D; \Gamma', \Gamma \vdash c\ \ell \leftarrow D'\}} \text{ [Hidden Resource]}$$

$$\frac{h \notin \Sigma}{\Sigma\{\Gamma \vdash h\ \mathsf{hid} \leftarrow D\} \leftrightsquigarrow \Sigma} \text{ [Add/Rem]}$$

$$\frac{h \notin \Sigma}{\Sigma\{\Gamma \vdash c\ \ell \leftarrow D\} \leftrightsquigarrow \Sigma\{\Gamma \vdash h\ \mathsf{hid} \leftarrow D; x : h \vdash c\ \ell := x\}} \text{ [Rename]}$$

$$\frac{\Sigma \leftrightsquigarrow_{\varepsilon} \Sigma' \qquad P \text{ compatible with } \Sigma \text{ and } \Sigma'}{\Sigma \cup P \leftrightsquigarrow_{\varepsilon(\llbracket P \rrbracket \mid\mid \cdot, \cdot, \cdot)} \Sigma' \cup P} \text{ [Congr]} \qquad \frac{\Sigma \leftrightsquigarrow_{\varepsilon} \Sigma}{\mathsf{hide}(\Sigma, c) \leftrightsquigarrow_{\varepsilon} \mathsf{hide}(\Sigma', c)} \text{ [Hiding]}$$

Figure 5: The IPDL Proof System.

3.3 Proof System

We present a proof system for reasoning about reaction sets. The main judgement of the logic is $\Sigma \leftrightsquigarrow_{\varepsilon} \Sigma'$, meaning that the protocol represented by Σ is indistinguishable from Σ', up to error ε. We read $\Sigma \leftrightsquigarrow_{\varepsilon} \Sigma'$ as Σ *rewrites to* Σ' (and vice versa) with error ε. Recall from Section 3.1 that we measure cryptographic error with a *function*, which takes as input an automata and natural number and outputs a real number in $[0, 1]$. The automata we give as an argument to the error function corresponds to the simulator of the approximate rewrite. We say the judgement $\Sigma \leftrightsquigarrow_{\varepsilon} \Sigma'$ is *valid* if $\llbracket \Sigma \rrbracket \leftrightsquigarrow_{\varepsilon} \llbracket \Sigma' \rrbracket$, as defined in Section 3.1. When the error is zero, we leave it out of the judgement; i.e., $\Sigma \leftrightsquigarrow \Sigma'$ is an abbreviation for $\Sigma \leftrightsquigarrow_0 \Sigma'$.

The proof system of IPDL is defined by the rules in Figure 5. The rules [Perm-Σ] and [Perm-Γ] state that reaction sets and contexts of individual reactions are equivalent up to reordering, and in particular imply that $\leftrightsquigarrow$ is reflexive, and that $||$ is symmetric and associative. Rule [Ext] states that we may replace one distribution by another if the underlying equational theory proves them to be the same. The [Trans] rule states that approximate equivalence is transitive, except that we accumulate error.

The [Fold] rule states that sampling a channel c from a monadic bind $(x \xleftarrow{\$} D; D')$ where x is free in D' is equivalent to generating a fresh hidden channel h for x, sampling D in this fresh channel, and passing the result to c. The context for h may be any subcontext of c that allows D to be defined. This rule connects the syntax of the underlying monadic language of distributions to IPDL. In particular, we may use the [Fold] rule to decompose a channel which carries a product

distribution into two different channels.

The [Dep-trans] rule states that dependence between channels is transitive. The rule [Subst] deals with non-probabilistic assignments: it says that if channel c is assigned to value e under context Γ, then any channel with a context that includes c and its context may propagate this value e in for c.

The rule [Hidden Resource] states that if a hidden channel h depends on the context Γ, and c is a channel with context Γ' such that all of Γ appears in Γ', then we may add or remove h from Γ' at will. Rule [Add/Rem] states that unused hidden channels may be removed from the reaction set. Rule [Rename] allows us rename internal computations using hidden channels. Rule [Congr] states that $\leftrightsquigarrow_\varepsilon$ is a congruence for $\cup$; we appropriately modify the error ε to take into account the common context P. Note that this implicitly allows us to reason about execution time, since the error ε may take into account the context P. Finally, rule [Hiding] allows us to hide output channels of the protocol.

We have proven our logic to be sound:

Theorem 1. *If the judgement* $\Sigma \leftrightsquigarrow_\varepsilon \Sigma'$ *is derivable in IPDL, then it is valid; i.e., that* $[\![\Sigma]\!] \leftrightsquigarrow_\varepsilon [\![\Sigma']\!]$.

The proof of Theorem 1 can be found in the appendix, in Section A. We stress that the soundness proof quantifies over adversaries which have the power of general probabilistic (polytime) algorithms, and are not limited to the IPDL language.

4 Encoding Protocol Security

The primary function of our logic is to show when two reaction sets, Σ and Σ', behave (nearly) identically on their external I/O channels. This problem is orthogonal to many of the tasks performed by protocol security frameworks, such as Universal Composability [Can01]. We now detail how protocol security may be encoded in our logic in a way compatible with UC.

An *interface* for a reaction set Σ is a set of I/O channels between Σ and the outside context. Each security protocol π is defined to be a set of *parties* $\{P_i \mid i \in 1 \ldots n\}$ and a set of *trusted functionalities* $\{F_i \mid i \in 1 \ldots k\}$, both of which are specified as reaction sets. Protocols may be used to express real, runnable code which performs cryptography, or they may be used to express idealized executions which replace cryptography with trusted third parties. Each protocol π has two kinds of I/O interfaces: the first is the *external interface* $\mathsf{Ext}(\pi)$, which is used for specifying high-level I/O behaviors of the protocol; the second is the *attacker interface* $\mathsf{Att}(\pi)$, used for specifying the *attacks* by which the outside world can influence the protocol. The external interface is supplied by the parties, while the attacker interface can be supplied by both parties and trusted functionalities.

We now specify how these interfaces are defined for protocols. First, we describe in more detail the role of parties and trusted functionalities:

Parties Each party P_i participates in a protocol as an independent agent with possibly distinct roles in the protocol. The I/O channels of each party P_i are partioned into the following disjoint sets:

- $\mathsf{Fun}(P_i)$, the interface between P_i and the set of trusted functionalities in the protocol;
- $\mathsf{Att}(P_i)$, the interface between P_i and the *attacker*, and
- $\mathsf{Ext}(P_i)$, the set of *external* I/O channels of P_i.

The *functionality interface* $\mathsf{Fun}(P_i)$ specifies the channels along which a party may invoke a trusted functionality. These may be used for a variety of tasks, such as sending a message over a network, sampling a key in a trusted manner, invoking an abstract commitment, etc. The *attacker interface* $\mathsf{Att}(P_i)$ specifies the ways in which an attacker may influence P_i. (Strictly speaking, the attacker is not separate from the context which influences $\mathsf{Ext}(P_i)$. The attacker interface is used to constrain how the simulator is constructed. This is discussed in more detail below.) The way in which the attacker interface is constructed depends on the *corruption model*, which we describe below in Section 4.2. The *external interface* is the intended external I/O behavior of the party in the protocol. (In UC-style protocol composition, the caller and callee of a subprotocol communicate along this external interface.) Note that we *do not* allow parties to communicate directly with one another; they may only indirectly communicate through functionalities which act as a network.

Functionalities Each functionality F_i specifies a trusted third party which is uncorruptable. Functionalities are used for specifying everything that is not a party, including the network by which parties communicate. The I/O channels of each functionality are partitioned into disjoint sets:

- $\mathsf{Att}(F_i)$, the interface between F_i and the attacker;
- $\mathsf{Fun}(F_i)$, the interface between F_i and other functionalities.
- $\mathsf{Par}(F_i)$, the interface between F_i and the parties in the protocol.

Channels in $\mathsf{Fun}(F_i)$ may only be matched with their dual channel in $\mathsf{Fun}(F_j)$ for some j. (Thus, functionalities may not directly communicate with the attacker interface of another functionality.) Channels in $\mathsf{Par}(F_i)$ may only be matched with channels in $\mathsf{Fun}(P)$ for some P. The channels in $\mathsf{Fun}(P)$, $\mathsf{Fun}(F)$ and $\mathsf{Par}(F)$ for all P and F are called *internal*. A protocol π consisting of parties P_i and functionality F_i is *fully specified* if all internal channels are matched with their dual. If this is the case, composing all parties and functionalities together using the hiding composition operator $||$ will eliminate all internal channels. This operation, called the *execution of* π, is denoted by $\Sigma(\pi)$. When we do so, we are left with the attacker interface $\mathsf{Att}(\pi)$ and the external interface $\mathsf{Ext}(\pi)$ consisiting of the union of all attacker and external interfaces in π respectively.

4.1 Comparing protocols

Given this setup, we now describe how we may compare two protocols through their external and attacker interfaces. There are two ways to do so, corresponding to *malicious* and *semi-honest* security. Common to both is the notion of a *simulation*: given two protocols π and π' with identical external interfaces, a simulator Sim is a reaction set which *converts* the attacker interface of π' to that of π by closing off all channels in $\mathsf{Att}(\pi')$, and exposing exactly those channels in $\mathsf{Att}(\pi)$. (Thus, Sim *supplies* inputs to $\mathsf{Att}(\pi')$, while it *consumes* inputs from $\mathsf{Att}(\pi)$; dually, Sim consumes outputs from $\mathsf{Att}(\pi')$, and supplies outputs to $\mathsf{Att}(\pi)$.)

Two protocols π and π' are *comparable* if they have the same number of parties and the ith party of each protocol has the same external interface. If this is the case, we say that π *reduces from* π' if there exists a simulator Sim such that in IPDL, $\Sigma(\pi) \rightsquigarrow \Sigma(\pi')||\mathsf{Sim}$, where $\Sigma(\pi)$, the execution of π, is equal to the composition of all parties and functionalities in π. (Note that both attacker and external interfaces of π and $\pi'||\mathsf{Sim}$ are the same.)

4.2 Corrupting protocols

When proving security of a protocol π relative to an idealized protocol π', we do so relative to a corruption model, which specifies the way in which parties can be corrupted by the adversary. We consider two corruption models here: *semi-honest* and *malicious* security. In both cases, given a corruption model $\mathcal{C} \in \{\mathsf{sh}, \mathsf{mal}\}$ and an index set $J \subseteq \{1, \ldots, n\}$ of parties to corrupt, we construct *derived protocols* $\pi^J_{\mathcal{C}}$ and $\pi'^J_{\mathcal{C}}$, and say that π *$\mathcal{C}$-realizes* π' if for all J, $\pi^J_{\mathcal{C}}$ reduces from $\pi'^J_{\mathcal{C}}$. (Note that there is a separate proof of reduction for each index set J, so there exists a simulator Sim^J for each J. Differing choices of J will give in general very different information to the simulator, and may require differing proof techniques to prove security.)

Below, we detail both corruption models, and show how to derive protocols in these two models.

4.2.1 Semi-honest security

In semi-honest security, the adversary has no control over the behavior of the corrupted party, but learns its internal state including any messages sent and received. This is modeled by appropriately instrumenting the code of each party $P_i, i \in J$ to leak all probabilistic channels as well as any input channels. (A channel is probabilistic when it is not of the form $\Gamma \vdash c\ \ell := e$; i.e., when it is associated to a nontrivial probability distribution.) Thus, the attacker interface of P_i grows to include these leaked channels.

4.2.2 Malicious Security

In malicious security, the adversary has full control over the party. This is modeled by performing the following modifications to π, for each $i \in I$:

- Remove the code for P_i from π.
- Move all channels in $\mathsf{Fun}(P_i)$ to the adversary interface on their respective functionality.
- Disregard all channels in $\mathsf{Ext}(P_i)$ and $\mathsf{Att}(P_i)$.

The most surprising modification is perhaps the third one, stating that we do not consider the channels in $\mathsf{Ext}(P_i)$ and $\mathsf{Att}(P_i)$ if P_i is corrupted. The intuition behind this is that we do not need to reason about the context's outputs to the corrupted party along either the external or attacker interface, since the context and corrupted party are one and the same; for a similar reason, we need not reason about the inputs from a corrupted party to the context.

4.3 Comparison with UC

Our formalism is restricted compared to full UC; in UC, corruption may happen dynamically, while in our setting we currently reason only about static corruption. The other major difference from UC is how we model the attacker. In UC, the adversary and environment play two differing but similar roles: the adversary provides an attack on a system, while the environment attempts to distinguish two protocols through their external channels.

We choose to model a simpler but equally expressive model, where the "adversary" and "environment" are both played by the context. The simulator, which in full UC is simply another adversary, is instead a "converter" of attacks between one protocol and another. This converter is constrained by the choice of channels which appear in the attacker interfaces of the two protocols. This viewpoint is supported by two theorems in the theory of UC: the first is black-box simulatability [Can01], which states that it suffices to consider UC-simulators which behave the same for all adversaries, and have

only black-box access to the adversary; the second is the dummy adversary theorem [Can01], which states that it suffices to consider a "dummy" UC-adversary which simply forwards all messages to and from the UC-environment. By combining these two theorems, we collapse the "adversary" and "environment" together, and only consider simulators which attach to the attacker interfaces of the two protocols (which formerly would have been meant for the UC-adversary.)

5 IPDL Proofs for Running Example

We continue our discussion of the running example from Section 2 in this section. We first study the case when neither party is corrupted. Remember that Figure 1 and 2 describe the real and ideal protocol executions in this case. In the no-corruption case, when the simulator Sim receives message query from the protocol π_I, it internally generates a symmetric key k and sends $\mathsf{Enc}(k, 0)$ along channel leak to π_R. The message ans_R is then forwarded to π_I as message ans_I. Thus, we get the simulator in Figure 6.

$$\begin{gathered}
\mathsf{query}\ \mathsf{inp} \\
q : \mathsf{query} \vdash \mathsf{keygen} : \mathsf{key} \\
k : \mathsf{keygen} : \mathsf{key} \vdash \mathsf{leak}\ \mathsf{vis} : \mathsf{ct} := \mathsf{Enc}(0, k) \\
\mathsf{ans}_R\ \mathsf{inp} \\
\mathsf{ans}_R \vdash \mathsf{ans}_I := \mathsf{ans}_R
\end{gathered}$$

Figure 6: Definition of the simulator for the secure channel example when neither party is corrupted.

The security argument is based on semantic security of the symmetric key encryption, as formalized in Figure 3, which says the encryption of a specified message is indistinguishable from encryption of message 0, which means the adversary cannot distinguish whether the ciphertext comes from simulator Sim or the real protocol π_R. The proof follows our description given in Section 2.

In order to get a feel for conducting a proof in IPDL, we will detail the most important step of this proof. Our goal is to obtain a factoring $\pi_R \leftrightsquigarrow \pi'_R|||\mathsf{OSS}_R$. We first simplify out the key_S and key_R channels of π_R. Note that both channels simply copy the value of keygen onwards, and are only used by the send^S_R and out_R channels, respectively. We thus use the [Dep-Trans] rules to move keygen into the contexts of both send^S_R and out_R. Now, we may use the [Subst] rule to substitute the value of key_S into send^S_R, and similarly for key_R and out_R. At this point, we have the following channels (suppressing unused variable names and types):

$$\begin{gathered}
k : \mathsf{keygen} \vdash \mathsf{key}_S := k \\
k : \mathsf{keygen} \vdash \mathsf{key}_R := k \\
m : \mathsf{in}_S, k : \mathsf{keygen}, \mathsf{key}_S \vdash \mathsf{send}^S_R \leftarrow \mathsf{Enc}(m, k) \\
c : \mathsf{deliv}, k : \mathsf{keygen}, \mathsf{key}_R \vdash \mathsf{out}_R\ \mathsf{vis} := \mathsf{Dec}(c, k).
\end{gathered}$$

Because the context of key_S is fully contained within send^S_R and send^S_R does not use the value of key_S, we now may remove key_S from the context of send^S_R. At this point, key_S is a hidden channel not used anywhere, so we use the [Add/Rem] rule to remove it from the reaction set. We do the same for key_S, removing it from the context of in_S and then from the protocol.

At this point, we need to reason about the value of $\mathsf{Dec}(c,k)$ in out_R. We will first rename the channel send_R^S to $\mathsf{out}_{\mathsf{OSS}}$ using [Rename], in anticipation for later in the proof. We first use the [Dep-Trans] rule to add keygen to the dependencies of leak, which we may do since leak depends on $\mathsf{out}_{\mathsf{OSS}}$ which depends on keygen. The relevant channels are now below:

$$\begin{aligned} \cdot &\vdash \mathsf{keygen} \leftarrow \mathsf{Rnd}(\mathcal{K}) \\ m : \mathsf{in}_S, k : \mathsf{keygen} &\vdash \mathsf{out}_{\mathsf{OSS}} \leftarrow \mathsf{Enc}(m,k) \\ c : \mathsf{out}_{\mathsf{OSS}}, k : \mathsf{keygen}, \mathsf{ans}_R &\vdash \mathsf{out}_R\ \mathsf{vis} := \mathsf{Dec}(c,k) \\ k : \mathsf{keygen}, c : \mathsf{out}_{\mathsf{OSS}} &\vdash \mathsf{leak}\ \mathsf{vis} := c \end{aligned}$$

We need to somehow connect the value of $\mathsf{Enc}(m,k)$ to the value of $\mathsf{Dec}(c,k)$. This is typically done using the [Fold] rule. However, we cannot fold the value of $\mathsf{out}_{\mathsf{OSS}}$ directly into out_R, since the channel leak also needs to use this ciphertext. This is complicated by the fact that out_R needs to wait on ans_R from the adversary, but leak needs to happen before. Thus, the first thing we do is separate out the computation of $\mathsf{Dec}(c,k)$ from the firing of out_R. This is done by rewriting $\mathsf{Dec}(c,k) \equiv (m \leftarrow \mathsf{Dec}(c,k); \mathsf{ret}\ m)$ (by the basic monadic identities), and using [Fold] to unfold the computation of $\mathsf{Dec}(c,k)$. This gives us a new channel, tmp, such that we get the two below reactions:

$$\begin{aligned} c : \mathsf{out}_{\mathsf{OSS}}, k : \mathsf{keygen} &\vdash \mathsf{tmp} := \mathsf{Dec}(c,k) \\ m : \mathsf{tmp}, \mathsf{ans}_R &\vdash \mathsf{out}_R\ \mathsf{vis} := m \end{aligned}$$

(We also used the [Dep-trans] rule in reverse to remove the dependencies of tmp from out_R). At this point, we can see that the dependencies of tmp and leak are the same, so we will use the [Fold] rule to combine them together. This will construct a new hidden channel, P, that outputs the pair $(c, \mathsf{Dec}(c,k))$ given the ciphertext c and key k. By properly removing channels from leak and tmp, we now get that the only channel which uses $\mathsf{out}_{\mathsf{OSS}}$ is this new channel P. Thus, we may fold the $\mathsf{Enc}(m,k)$ distribution into $\mathsf{out}_{\mathsf{OSS}}$. After removing some redundant dependencies using [Dep-Trans] ins reverse, the situation is shown below:

$$\begin{aligned} \cdot &\vdash \mathsf{keygen} \leftarrow \mathsf{Rnd}(\mathcal{K}) \\ m : \mathsf{in}_S, k : &\mathsf{keygen} \\ &\vdash \mathsf{P} \leftarrow (c \leftarrow \mathsf{Enc}(m,k); \mathsf{ret}\ (c, \mathsf{Dec}(c,k))) \\ p : \mathsf{P} &\vdash \mathsf{leak}\ \mathsf{vis} := p.1 \\ p : \mathsf{P}, \mathsf{ans}_R &\vdash \mathsf{out}_R\ \mathsf{vis} := p.2 \end{aligned}$$

At this point, we can see that by the correctness of decryption, $(c \leftarrow \mathsf{Enc}(m,k); \mathsf{ret}\ (c, \mathsf{Dec}(c,k))) \equiv (c \leftarrow \mathsf{Enc}(m,k); \mathsf{ret}\ (c,m))$; thus we use the [Ext] rule to perform this rewrite. By carrying out the above operations in reverse (i.e., unfolding the monadic bind in P to obtain $\mathsf{out}_{\mathsf{OSS}}$, and substituting in the value of P into leak and out_R), we finally get a simplified protocol π_R^{simp} containing the channel

$$m : \mathsf{in}_S, \mathsf{ans}_R \vdash \mathsf{out}_R := m$$

as shown in Section 2. The corresponding mechanized proof that $\pi_R \leftrightsquigarrow \pi_R^{\mathsf{simp}}$ is shown in the Appendix, in Figure 11.

What the above protocol simplification allows us to do is express the real protocol in a way such that *only the ciphertext* is needed to carry out the protocol, and not the decryption key. In this simplified protocol π_R^{simp}, the channel $\mathsf{out}_{\mathsf{OSS}}$ is used to generate the ciphertext. In order to

carry out the security argument, we may now *factor* π_R^{simp} into the reaction set $\pi'_R||\mathsf{OSS0}$. This π'_R is equal to π_R^{simp}, except that we remove the keygen channel and turn the channel $\mathsf{out}_{\mathsf{OSS}}$ into an input, which is output by $\mathsf{OSS0}$. This factored protocol π'_R can be thought of as the simulator for the encryption scheme.

At this point, we now use the [Congr] rule to rewrite to $\pi'_R||\mathsf{OSS1}$, where the ciphertext is now equal to $\mathsf{Enc}(0,k)$. It is now a straightforward series of substitutions to see that $\pi'_R||\mathsf{OSS1} \leftrightsquigarrow \mathsf{Sim}||\pi_I$. Altogether, by applying [Trans] we receive that $\pi_R \leftrightsquigarrow_{\varepsilon_{\mathsf{OSS}}([\![\pi'_R]\!]||\cdot,\cdot,\cdot)} \pi_I$, where $\varepsilon_{\mathsf{OSS}}$ is the error incurred from rewriting from $\mathsf{OSS0}$ to $\mathsf{OSS1}$. We are able to concretely track which simulator we used for the rewrite involving $\mathsf{OSS0}$ and $\mathsf{OSS1}$. By inspecting this simulator π'_R, we see that no inefficient computations are performed; thus, we remain computationally sound.

We also analyze the case when the sender is corrupted. We follow the framework in Section 4 to derive corrupted protocols $\pi_{R_{\mathsf{mal}}}^{\{S\}}$ and $\pi_{I_{\mathsf{mal}}}^{\{S\}}$. In the real protocol, when we corrupt the sender we expose the channels key_S and send_R^S to the sender as an input and output channel respectively. In the ideal protocol, the adversary obtains the channel send_I^S.

Thus when the sender is corrupted, the simulator needs to provide the real-world adversary with a key by running the key generation algorithm itself. When the real-world adversary outputs a ciphertext on send_R^S, the simulator decrypts it using this key and sends the decrypted message along send_I^S. (We may assume wlog that the adversary is constrained to send well-formed ciphertexts.) The simulator then handles query, leak, ans_R, and ans_I as before, except now the value of leak is not generated randomly but comes from send_R^S from the adversary. Since the adversary is the one providing the ciphertext and the simulator knows the key, we do not need to apply the security assumption about the encryption scheme here. The proof is a straightforward sequence of rewriting steps.

6 Case Studies

In this section, we describe more case studies.

6.1 The ElGamal Cryptosystem

We replicate in IPDL the standard security of the ElGamal cryptosystem [ElG85], which constructs a CPA-secure public key encryption scheme from the Decisional Diffie Hellman (DDH) assumption, which states that two certain distributions, $\mathsf{DDH0}$ and $\mathsf{DDH1}$ are computationally indistinguishable. This assumption is interpreted as an axiom $\Sigma_{\mathsf{DDH0}} \leftrightsquigarrow_{\varepsilon_{\mathsf{DDH}}} \Sigma_{\mathsf{DDH1}}$ in IPDL. We define the security of the ElGamal scheme similar to Figure 3, comparing a functionality which takes a message as input and encrypts it to one which disregards the message and instead encrypts a dummy message. Since we are proving semantic security, we additionally insert reactions which leak the public key of the encryption scheme to the adversary. Conducting this proof in IPDL requires us to "factor" the real ElGamal execution $\Sigma_{\mathsf{real}}^{\mathsf{EG}}$ into two reaction sets $\Sigma_{\mathsf{factor}}||\Sigma_{\mathsf{DDH0}}$. We then apply the [Congr] rule to rewrite Σ_{DDH0} into Σ_{DDH1}. The security of the cryptosystem follows, since doing this rewrite essentially rerandomizes the ciphertext so that it no longer depends on the message.

6.2 MPC in the $\mathcal{F}_{\mathsf{comm}}$-hybrid Model

We prove secure a two party secure multiparty computation protocol for Rock-Paper-Scissors (RPS) using an idealized commitment scheme. In the ideal protocol, each party sends its input to the ideal functionality who computes the result of the computation and sends it back to the two parties. The real protocol for RPS is implemented using two trusted functionalities $\mathcal{F}_{\mathsf{comm}}$ implementing

commitment schemes: (1) Both parties send (`commit`, input) to $\mathcal{F}_{\mathsf{comm}}$; (2) When $\mathcal{F}_{\mathsf{comm}}$ collects (`commit`, input) from both parties, it sends `committed` to both parties and `ready` to adversary; (3) Upon receiving `commit` from $\mathcal{F}_{\mathsf{comm}}$, both party send `open` to $\mathcal{F}_{\mathsf{comm}}$; (4) Upon receiving `open` from one party and `ok` from adversary, $\mathcal{F}_{\mathsf{comm}}$ send one party's input to the other party. The ideal functionality can be appropriately modified to allow adv to schedule message delivery

We prove security in the malicious corruption model. There are no adversarial channels exposed when neither party is corrupted; thus, in this case, to prove security we let the simulator be empty, and prove directly that the real protocol rewrites under $\leftrightsquigarrow$ to the ideal one. Since the only exposed channels are the parties' external channels, this amounts to proving functional correctness of the real protocol; this proof is a straightforward application of the rewriting rules.

When one of the parties (wlog, party B) is corrupted, the simulator sends a fake commitment message to the adversary, who then responds with their input. The simulator forwards this input to the trusted functionality in the ideal world, who responds back with the result of the computation. Now that the simulator has party B's input as well as the result of the computation, it can deduce party A's input and that input to the adversary. Once this simulator is specified in IPDL, it is a straightforward exercise to see that the real world simplifies to the same reaction set as the ideal world with this adversary. We describe more details on the RPS protocol in Appendix B.

6.3 Semi-honest Oblivious Transfer Protocol

Our final case study is the classic oblivious transfer (OT) protocol of [GMW87], which constructs an OT protocol from any hard-core predicate associated to a trapdoor permutation family. [1]The protocol consists of two parties, the transmitter and receiver, communicating over an authenticated network. The network is modeled similarly to Section 2, where we instrument the network with explicit leak channels. The transmitter takes as input two bits, m_0 and m_1, while the receiver takes an input an index bit i. The goal is for the receiver to learn m_i, while the receiver learns nothing.

We briefly describe the 1-out-of-2 OT protocol with message space $\{0,1\}$, using trapdoor permutation family $\mathcal{F}$ associated with hardcore predicate $b(\cdot)$. On input of two messaage $m_0, m_1 \in \{0,1\}$, the transmitter samples a random permutation (f, f^{-1}) from family $\mathcal{F}$, and sends the permutation description f to receiver. On input index i and permutation f from transmitter, the receiver computes $y_i = f(r_i)$ and $y_{1-i} = r_{1-i}$, where (r_i, r_{1-i}) are sampled randomly from the domain of permutation f. The receiver then sends (y_i, y_{1-1}) to the transmitter. Next, transmitter first uses trapdoor f^{-1} to invert (y_i, y_{1-1}) and xor the message (m_i, m_{1-i}) with hardcore predicate $b(\cdot)$, i.e. $c_i = m_i \oplus b(f^{-1}(y_i))$. Upon message (c_i, c_{1-i}) from transmitter, the receiver can recover message m_i by computing $m_i = c_i \oplus b(r_i)$.

The protocol is secure in the semi-honest corruption model, as described in Section 4. The protocol is modeled similarly to our previous example, where the two ideal parties send their inputs (m_0, m_1) and i to a trusted functionality who securely communicates m_i to the ideal receiver. We focus on the case where the receiver is corrupted. Following Section 4, this is modeled by instrumenting the code of both the real and ideal world receivers in order to leak to the adversary all secret state and input from other parties. The interesting aspect of this case study is its complexity; while this OT protocol has been studied before in formal cryptography frameworks [CCK+05], we have found that our abstraction of dependency leads to a very manageable proof. While expressible on paper, this proof is further enhanced by the proof automation capabilities of our mechanization, which we present in Section 7.

[1]An on-paper description of our formal proof can be seen at https://github.com/ipdl/ipdl.

7 Mechanization

In this section, we describe our mechanization of IPDL as a logic shallowly embedded in Coq, publicly available at https://github.com/ipdl/ipdl. All IPDL proofs in this paper have been mechanized. The soundness proof of IPDL using probabilistic automata is not yet mechanized.

Our embedding of IPDL in Coq is *shallow*, in the sense that individual reactions of IPDL are encoded using Coq functions. By doing so, we are able to turn our rewriting rules of Figure 5 into rewriting tactics in Coq. In order to assist proof automation, we also make heavy use of Ltac, Coq's native proof automation mechanism. We use a shallow embedding in order to encode IPDL reactions into Coq. By doing so, we are able to write reactions that look very close to their on-paper counterpart: for example, the reaction $c : \mathsf{send}_R^S : \mathsf{ct}, k : \mathsf{key}_S : \mathsf{key}, \mathsf{ans}_R \vdash \mathsf{out}_R\ \mathsf{vis} : \mathsf{msg} := \mathsf{Dec}(c, k)$ is encoded in Coq using the concrete syntax

```
[:: ("sendRS", tyCtx); ("keyS", tyKey);
    ("ansR", tyUnit)] ~>
    ("outR", tyMsg) dvis
    (fun ct k _ => dec (c, k)).
```

Each rule in our logic corresponds (roughly) to a single Coq tactic. The main proof goal in Coq is written `S1 <~~> S2`, corresponding to the $\leftrightsquigarrow$ judgment. [2]Most of our tactics work on either side of this proof goal. For example, the tactic `r_move_at pos c1 c2` takes a boolean flag `pos` indicating which reaction set to operate on (either `leftc` or `rightc`, borrowing this design from FCF [PM15]) and two channel names `c1` and `c2`, and swaps the position of `c1` with `c2`. Doing so automatically applies the [Perm-Σ] rule. We have a similar tactic for permuting the elements of a reaction's context.

While doing these proofs, we found the most often used tactic was `autosubst_at pos c1 c2`, which given a non-probabilistic channel `c1` automatically fills the context of `c2` with the the context of `c1`, substitutes the value of `c1` into `c2`, and removes `c1` from the context of `c2`. This rule is often paired with the tactic `remove_at pos c`, which removes unused hidden actions from the reaction set.

The tactic `r_ext_at pos c D` (corresponding to the rule [Ext]) replaces the value of `c` with the distribution `D`. When this tactic is called, a subgoal is generated that states for all possible interpretations of the context of `c`, the distribution currently in `c` is equal to the distribution `D`. We use a custom library for representing probability distributions. Our library has the feature that representations of distributions are naturally *extensional*, meaning that equality of the distribution coincides with Coq equality. This feature is not essential for the mechanization, but is very important for reducing the complexity of the proof engineering.

Our proof development is overviewed in Figure 7. Only the file `Logic.v` contains trusted definitions; all others are either derived tactics, auxiliary lemmas, or our case studies. The longest proof is of the Oblivious Transfer protocol, at 816 lines total; our other proofs are small in size, at about 300-400 lines each. An example excerpt from our mechanization of the running example from Section 2 is given in the Appendix, in Figure 11. Our mechanized proof sizes are either favorable or on par with relevant work. For protocols, we compare with the recent prior work EasyUC [CSV19] (which uses EasyCrypt as the proof assistant) and the independent work CryptHOL [LSBM19]. EasyUC encoded a secure channel example with more than 10,000 lines of code, while CryptHOL encoded one-time pad with 347 lines of code. For non-interactive primitives, we have 244 lines of

[2]We do not currently reason about approximate error in the mechanization; i.e., our mechanization can be seen as manipulating statements of the form $\exists \varepsilon, s \leftrightsquigarrow_\varepsilon s'$.

code for ElGamal encryption whereas FCF [PM15] is at 249 lines (although FCF does not provide an easy-to-use logic for expressing protocols). Our most complicated example, OT, was not implemented by these other systems — however, we can compare with the *on-paper* proof in the Task-PIOA paper [CCK+18] which proved OT in 24 pages (*c.f.* ours is 816 lines of actual code). We conclude that our framework delivers proofs of succinct size[3].

Section	LoC	Section	LoC
Core Logic	338	RPS	372
Tactics and Lemmas	802	ElGamal	244
Secure Channel	307	OT	816

Figure 7: Lines of code for our IPDL Coq development.

8 Related Work

Symbolic techniques for protocol verification. A long line of research in the verification of cryptographic protocols has been done in the so-called *Dolev-Yao model* [DY83], where all cryptography is assumed to be perfectly secure. While admitting simple and effective proof methods and supported by multiple mechanized provers ([BSCS18, MSCB13, Cre08]), this technique is limited by requiring complicated *computational soundness* arguments, which prove that a symbolic attacker is equivalent in power to a corresponding computational one which interacts with instantiations of the idealized cryptographic primitives.

A promising direction ([BCL12], [BCL14], [BCEO19]) in this space is initiated by Bana and Comon, where the attacker is not limited by interacting with idealized cryptography, but instead constrained by a number of axioms which state what the attacker is not able to do. By considering the most powerful attacker which does not violate the axioms, it is easy to see such attackers are equivalent to their computational counterparts, if the axioms are sound. The soundness of these axioms are subtle and require careful expert analysis.

In comparison, our work aims to match the *computational reduction* model of reasoning from the ground-up, and therefore match the mathematical foundations of cryptography. While systems based on symbolic systems like Bana and Comon would require an additional soundness proof outside the system, systems based on computational cryptography are intrinsically sound. This difference reflects a tradeoff between the two approaches: while symbolic systems generally support automatic reasoning, computational systems support more general forms of reasoning. Because of this tradeoff, it is likely most profitable to combine aspects of both systems into one, such as partially automating computational verification tools using symbolic techniques [BGJ+19]. We additionally point out that a design goal of IPDL is to enable *human-readable* formal proofs (either on paper on in a theroem prover), while symbolic systems generally favor computer-generated proofs.

Computational mechanized security proofs. To avoid some of the limitations of the symbolic approach, a separate line of work instead chooses to reason directly in the computational model; i.e., with explicit reasoning principles about computation time, computational error, and probability. CryptoVerif [Bla06] reasons in the computational model about security protocols in a manner

[3]We did not directly compare succinctness with Bana and Comon [BCL12, BCL14, BCEO19], Micciancio and Tessaro [MT13], and other related systems which either do not have an implementation or we did not find an open-source implementation.

similarly to symbolic systems such as ProVerif [BSCS18], thus mainly aims for automated proofs for a well-chosen set of cryptographic primitives.

There are a number of successful verification tools for computational cryptographic proofs, such as Easycrypt[BGHZ11] and FCF [PM15], which focus on noninteractive primitives and thus do not reason about concurrency explicitly. These tools are capable of nontrivial probabilistic reasoning which is currently not captured directly in IPDL, such as explicit reasoning about the security parameter, and up-to-bad reasoning for bounding the kinds of attacks the adversary may perform. It is interesting future work to combine these advanced forms of cryptographic reasoning with IPDL.

Frameworks for cryptographic protocols. In the cryptography literature, Universal Composability [Can01] and Constructive Cryptography [Mau11] are the two dominant definitional frameworks for simulation-based security. Several automata-based frameworks also exist, such as [BPW07] and [CCK+18], which, while similar in spirit, aim for a more formal treatment. Additionally, some works use process calculi to model computational cryptographic protocols, such as [MRST01]. Our basic semantic framework is closest to that of UC, since we semantically model a protocol as a configuration of multiple agents, each of which may send at most one message after becoming activated. This requirement that only one message is sent when each protocol agent is activated is naturally captured through a linear typing judgement in ILC [LHM19], a recent effort to formalize the semantics of UC in a process calculus. While ILC does not yet offer formal proofs, it would be a natural next step to formalize the semantics of IPDL using ILC.

An interesting alternative framework is given in Micciancio and Tessaro [MT13] (hereafter M&T), where they use *complete partial orders* to represent cryptographic protocols as the least fixed point of a recursive set of equations. Each communication channel induces a stream of values, which are recursively interrelated through the protocol's execution. We follow a number of intuitions made by M&T: both IPDL and M&T choose to view communication channels as pure values which may be related through equational rewrites. In both systems, these rewrites are sound under the assumption that the protocols in question do not make use of timing information, but only the values taken on the channels. However, we prove soundness in a more general semantic framework which allows for general adversaries which do make use of timing information. We point out that the monotonicity requirement in M&T (that further inputs can only create more outputs) is naturally captured through our semantic framework, since protocol agents in our framework only operate on single messages at a time.

Computational proofs of simulation-based security. Two recent concurrent works, EasyUC [CSV19] and CryptHOL [LSBM19], deliver mechanized computational proofs of simulation-based security of cryptographic protocols. EasyUC does so by directly encoding the coroutine-based semantics of UC in EasyCrypt. As the authors acknowledged in their paper, encoding proofs of cryptographic protocols in EasyUC is highly non-trivial, which is largely due to EasyCrypt not natively supporting reasoning about concurrency. The EasyUC work points out a need for a DSL for specifying and reasoning about cryptographic protocols in EasyCrypt; future work on IPDL could potentially allow it to fill this gap.

CryptHOL is a formalization of the Constructive Cryptography framework [Mau11] in Isabelle. While superficially similar to our work, CryptHOL (and EasyUC) serve a complementary purpose to IPDL: while the former works serve to formalize the semantics of cryptographic protocols in a mechanized theorem prover, IPDL explores the space of appropriate abstractions for conducting cryptographic proofs. We believe that the reaction-based equational logic of IPDL could potentially benefit proofs in EasyUC and CryptHOL, both of which employ handcrafted bisimulations to

conduct equivalence proofs.

9 Limitations and Future Work

Given our focus on simplicity and ease-of-use, IPDL currently imposes some restrictions on the cryptographic proofs it can model. We believe, however, that the scope of IPDL nonetheless covers a a wide and interesting class of commonly encountered cryptographic proofs.

First, IPDL currently does not explicitly encode and reason about the security parameter. This does not harm any computational reasoning, since proofs in IPDL consist of a finite number of rewrites. Thus, the error incurred in an IPDL proof will be upper bounded by $n\varepsilon$, where n is the size of the proof, and ε is the largest error incurred during the proof. By thinking of the security parameter as implicit in the IPDL protocol, we see that the error incurred by any proof is negligible, assuming that ε is. However, if extensions to IPDL enable a way to encode loops, then this reasoning breaks down and an explicit security parameter is required.

Second, IPDL currently does not capture protocols where events are triggered based on nontrivial properties about their dependencies. Thus, we do not yet consider protocols which make nontrivial use of control flow, such as certain consensus protocols. We note that this has previously been argued by Micciancio and Tessaro [MT13], who have a similar restriction in their work, to nevertheless represent an expressive class of cryptographic protocols. This restriction also means that for UC-style simulation proofs, we currently support only straightline, blackbox notions of simulation.

Third, it would not be too hard to extend IPDL to support a parametrized number of parties and a parameterized number of executions (as long as the number of parties or sessions are fixed a-priori). We plan to extend IPDL in this direction in the near future. We point out that since IPDL adopt UC-flavor security definitions, just like UC [Can01], the security notion allows us to reason about a single protocol instance while guaranteeing concurrent or sequential composition with polynomially unbounded instances/sessions.

We stress that we do not require that the adversary be an IPDL program — the adversary is assumed to be a probabilistic polynomial-time algorithm. It would be an interesting direction to extend IPDL and allow the syntax to describe a *family* of channels, with a uniform description of their values and dependencies. This would allow us to describe protocols and security games which operate over many queries, up to some (publicly known) parameter q. Security analysis in this setting would require explicit security parameter analysis, since handling an arbitrary number of queries is similar to handling loops.

We currently capture only static security, in which parties are corrupted a-priori, rather than becoming so in the middle of the protocol. We leave adaptive corruption to later work.

An exciting future direction is to integrate IPDL with an underlying battle-hardened cryptographic proof system (such as EasyCrypt) which may enable more expressiveness, thus achieving ease-of-use and generality simultaneously. Other exciting future directions include to provide a greater degree of proof automation, compiling IPDL programs to executable code (e.g., in C) and proving the correctness of the compilation. As discussed in Section 8, there are a number of directions in which our work can potentially intersect with other concurrent work in formalized cryptography proofs. Integrating our method for constructing bisimulations into EasyUC [CSV19] or CryptHOL [LSBM19] may allow for more accessible proofs for a certain class of protocols. Additionally, it would be interesting to prove our bisimulation methodology sound in ILC [LHM19], a formalization of UC semantics which is likely compatible which our own semantics.

References

[BCEO19] Gergei Bana, Rohit Chadha, Ajay Kumar Eeralla, and Mitsuhiro Okada. Verification methods for the computationally complete symbolic attacker based on indistinguishability. *ACM Transactions on Computational Logic (TOCL)*, 21(1):2, 2019.

[BCL12] Gergei Bana and Hubert Comon-Lundh. Towards unconditional soundness: Computationally complete symbolic attacker. In *International Conference on Principles of Security and Trust*, pages 189–208. Springer, 2012.

[BCL14] Gergei Bana and Hubert Comon-Lundh. A computationally complete symbolic attacker for equivalence properties. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 609–620. ACM, 2014.

[BGHZ11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Heidelberg, Germany.

[BGJ+19] Gilles Barthe, Benjamin Grégoire, Charlie Jacomme, Steve Kremer, and Pierre-Yves Strub. Symbolic methods in computational cryptography proofs. 2019.

[Bla06] Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *2006 IEEE Symposium on Security and Privacy*, pages 140–154, Berkeley, CA, USA, May 21–24, 2006. IEEE Computer Society Press.

[BPW07] Michael Backes, Birgit Pfitzmann, and Michael Waidner. The reactive simulatability (rsim) framework for asynchronous systems. *Information and Computation*, 205(12):1685–1720, 2007.

[BSCS18] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. Proverif 2.00: Automatic cryptographic protocol verifier, user manual and tutorial, 2018.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

[CCK+05] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. Using probabilistic i/o automata to analyze an oblivious transfer protocol. Cryptology ePrint Archive, Report 2005/452, 2005. http://eprint.iacr.org/2005/452.

[CCK+06a] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. Task-structured probabilistic i/o automata. In *2006 8th International Workshop on Discrete Event Systems*, pages 207–214. IEEE, 2006.

[CCK+06b] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. Time-bounded task-pioas: A framework for analyzing security protocols. In *International Symposium on Distributed Computing*, pages 238–253. Springer, 2006.

[CCK+18] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. Task-structured probabilistic i/o automata. *Journal of Computer and System Sciences*, 94:63–97, 2018.

[Cre08] Cas JF Cremers. The scyther tool: Verification, falsification, and analysis of security protocols. In *International Conference on Computer Aided Verification*, pages 414–418. Springer, 2008.

[CSV19] Ran Canetti, Alley Stoughton, and Mayank Varia. Easyuc: Using easycrypt to mechanize proofs of universally composable security. In *32nd IEEE Computer Security Foundations Symposium*, 2019. https://eprint.iacr.org/2019/582.

[DY83] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[LHM19] Kevin Liao, Matthew A. Hammer, and Andrew Miller. Ilc: A calculus for composable, computational cryptography. In *PLDI*, 2019. https://eprint.iacr.org/2019/402.

[LSBM19] Andreas Lochbihler, S. Reza Sefidgar, David Basin, and Ueli Maurer. Formalizing constructive cryptography using crypthol. In *32nd IEEE Computer Security Foundations Symposium*, 2019. http://www.andreas-lochbihler.de/pub/lochbihler2019csf.pdf.

[Mau11] Ueli Maurer. Constructive cryptography–a new paradigm for security definitions and proofs. In *Joint Workshop on Theory of Security and Applications*, pages 33–56. Springer, 2011.

[MRST01] John Mitchell, Ajith Ramanathan, Andre Scedrov, and Vanessa Teague. A probabilistic polynomial-time calculus for analysis of cryptographic protocols:(preliminary report). *Electronic Notes in Theoretical Computer Science*, 45:280–310, 2001.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 696–701, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[MT13] Daniele Micciancio and Stefano Tessaro. An equational approach to secure multi-party computation. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 355–372. ACM, 2013.

[PM15] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In *International Conference on Principles of Security and Trust*, pages 53–72. Springer, 2015.

A Soundness proof for IPDL

We proceed by case analysis on the proof rules. In each case, we need to provide a bisimulation that relates distributions of states in the first protocol to distribution of states in the second, such that related distributions of states exhibit the same behavior. (This distribution bisimulation is similar to that proposed in [LSBM19].)

We proceed by case analysis on the proof rules:

- Cases [Perm-Σ], [Perm-Γ], [Ext], [Trans], [Subst] are clear, since the two protocol agents considered are equal.
- Case [Add/Rem]: The corresponding bisimulation relates a Dirac distribution on state s_1 of the reaction set on the left to a Dirac distribution on state s_2 of the reaction set on the right iff the value of h in s_1 is undefined and the values of s_1 and s_2 for all remaining channels agree.
- Case [Rename]: The corresponding bisimulation relates a Dirac distribution on state s_1 of the reaction set on the left to a Dirac distribution on state s_2 of the reaction set on the right iff the value of h in s_2 is equal to the value of c in both s_1 and s_2, and the values of s_1 and s_2 for all remaining channels agree.
- Case [Hidden Resource]: The corresponding bisimulation relates a distribution η on states of the reaction set on the left to a Dirac distribution on state s of the reaction set on the right iff either
  - η is a Dirac distribution on the same state s, or
  - η is the result of applying h to the state s
- Case [Fold]: The corresponding bisimulation relates a distribution η_1 on states of the reaction set on the left to a distribution η_2 on states of the reaction set on the right iff either
  - η_1 is a Dirac distribution on state s_1, η_2 is a Dirac distribution on state s_2, the values of h and c in s_1 are undefined, the value of c in s_2 is undefined, and s_1 and s_2 agree on all remaining channels, or
  - there are Dirac distributions on states s_1 and s_2, respectively, that satisfy the conditions of the previous point, such that η_1 is the result of applying h and then c to s_1 and η_2 is the result of applying c to s_2.
- Case [Congr]: This follows from the observation that the protocol consisting of the single IPDL program $P \cup Q$ is equivalent to the protocol consisting of the two IPDL programs P and Q. The bisimulation for this observation relates a Dirac distribution on state s_1 to the Dirac distribution on the combined state (s_2, s_3) iff s_1 agrees with s_2 and also with s_3 on any common channels.
- Case [Hiding]: This follows from the observation that given a protocol σ compatible with the protocol agent $P \,\mathsf{hide}\, c$, we can extend σ by another protocol agent Q whose (token) inputs are requests to execute outputs o of P (*i.e.*, inputs of the form get_o). If o does not depend on c, Q forwards this request to P. Otherwise it first asks P to execute h and if it receives a value back from P, it forwards to P the request to execute o. If instead of a value for h the protocol Q receives another execution request ,*e.g.*, for a channel l, this means the previous attempt to execute h was unsuccessful and the effort to execute o is abandoned.

$\mathsf{comm}_P : \mathsf{play}\ \mathsf{inp}$

$c_P : \mathsf{comm}_P : \mathsf{play} \vdash \mathsf{committed}_P\ \mathsf{vis}$

$\mathsf{open}_P\ \mathsf{inp}$

$c_P : \mathsf{comm}_P : \mathsf{play}, \mathsf{open}_P \vdash \mathsf{val}_P\ \mathsf{vis} : \mathsf{play} := \mathsf{comm}_P$

$\mathsf{in}_P : \mathsf{play}\ \mathsf{inp}$

$i_P : \mathsf{in}_P : \mathsf{play} \vdash \mathsf{comm}_P\ \mathsf{vis} : \mathsf{play} := \mathsf{in}_P$

$\mathsf{committed}_{\bar{P}}\ \mathsf{inp}$

$c_P : \mathsf{comm}_P : \mathsf{play}, c'_{\bar{P}} : \mathsf{committed}_{\bar{P}} \vdash \mathsf{open}_P\ \mathsf{vis}$

$\mathsf{val}_{\bar{P}} : \mathsf{play}\ \mathsf{inp}$

$i_P : \mathsf{in}_P : \mathsf{play}, v_{\bar{P}} : \mathsf{val}_{\bar{P}} : \mathsf{play}$

$\quad \vdash \mathsf{out}_P\ \mathsf{vis} : \mathsf{ans} := f_{\mathsf{rps}}(\mathsf{in}_P, \mathsf{val}_{\bar{P}})$

Figure 8: The real protocol for RPS. Left: the commitment functionalities $\mathcal{F}_{\mathsf{comm}}$, for parties $P \in \{\text{A,B}\}$. Right: the code for party P. We use the notation $\bar{P}$ to denote B if $P = A$, and A if $P = B$.

B More on RPS Protocol

The ideal protocol is implemented in IPDL as having each party $P \in \{\text{A, B}\}$ listen on input channel in_P, forward this input to send_P, receive a reply on recv_P, and forward this value on out_P; correspondingly, the ideal functionality listens on both send_A and send_B, and sends the value $f_{\mathsf{rps}}(x, y)$ on both recv_A and recv_B, where x and y are the two parties' inputs. We say the real protocol is secure when it reduces from the ideal protocol, in the terminology of Section 4. In particular, this means input privacy (no party can learn more about another party's input than what the function outputs), and input independence (no party can correlate its input with the input of another party).

The real protocol for RPS is implemented using two trusted functionalities implementing commitment schemes. In UC terminology, this means we are operating in the $\mathcal{F}_{\mathsf{comm}}$-hybrid model. The commitment functionality operates by receiving a value $x \in \{\mathsf{rock}, \mathsf{paper}, \mathsf{scissors}\}$ on input comm_P, and sending an indication to the other party $\bar{P}$ that a value has been committed. Once P sends open_P to the functionality, it then reveals the commitment to the other party via the val_P channel.

Informally, the protocol is secure since the commitment functionalities enforce this input independence property described above. The input privacy requirement for this protocol is degenerate, since each party can deduce the other's private input, given its own private input and the output value of the protocol.

There are no adversarial channels exposed when neither party is corrupted; thus, in this case, to prove security we let the simulator be empty, and prove directly that the real protocol rewrites under $\rightsquigarrow$ to the ideal one. Since the only exposed channels are the parties' external channels, this amounts to proving functional correctness of the real protocol; this proof is a straightforward application of the rewriting rules.

When one of the parties (wlog, party B) is corrupted, the simulator gets access to the real world channels comm_B (coming from the adversary); $\mathsf{committed}_A$ (coming from the simulator); open_B (coming from the adversary); and val_A (coming from the simulator). In the ideal world, it gets access to the channels send_B (coming from the simulator) and recv_B (coming from the functionality.) This situation is shown in Figure 9.

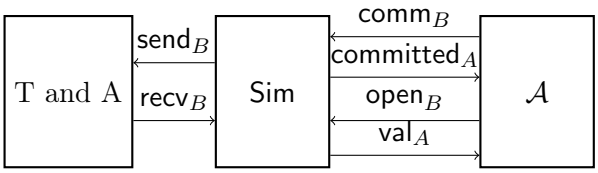

Figure 9: Illustration of Corrupted Bob

The simulator operates as follows: it may immediately fire $\mathsf{committed}_A$, indicating that the other party has committed its input. On input comm_B from the adversary, the simulator forwards this input to send_B in the ideal protocol. The simulator receives the value open_B from the adversary, but does nothing with it. After receiving recv_B from the ideal protocol, Sim can compute and send $\mathsf{val}_A := f_{\mathsf{rps}}^{-1}(\mathsf{recv}_B, \mathsf{send}_B)$, where the function f_{rps}^{-1} computes A's input based on the result recv_B and B's input send_B.

$$\mathsf{comm}_B : \mathsf{play}\ \mathsf{inp}$$
$$c_B : \mathsf{comm}_B : \mathsf{play} \vdash \mathsf{send}_B : \mathsf{play} := \mathsf{comm}_B$$
$$\cdot \vdash \mathsf{committed}_A$$
$$\mathsf{recv}_B : \mathsf{ans}\ \mathsf{inp}, \mathsf{open}_B\ \mathsf{inp}$$
$$s_B : \mathsf{send}_B : \mathsf{play}, r_B : \mathsf{recv}_B : \mathsf{ans} \vdash \mathsf{val}_A : \mathsf{play} := f_{\mathsf{rps}}^{-1}(s_B, r_B)$$

Figure 10: Formalization of Simulator Sim

Once we describe the simulator, this proof is similar to the honest case and only requires one to compute that the real protocol and the ideal protocol with the simulator simplify to the same reaction set.

```
Theorem noCorr_real_simp : realNoCorr < ~ > realSimp.
  rewrite /realNoCorr /realSimp /rlist_comp_hide; vm_compute RChans; simpl.
  autosubst_at leftc "keyR" "outR".
  remove_at leftc "keyR".
  autosubst_at leftc "keyS" "sendSR".
  remove_at leftc "keyS".

  autosubst_at leftc "deliv" "outR".
  remove_at leftc "deliv".

  rename_at leftc "sendSR" "outCPA".
  arg_move_at leftc "outCPA" "inS" 0.
  trans_at leftc "outCPA" "leak" "keygen" tyKey.

  arg_move_at leftc "outR" "keygen" 0.
  r_ext_at leftc "outR" (rbind [:: ("keygen", tyKey); ("outCPA", tyCtx)]
                               [:: ("ans", tyUnit)] ("tmp", tyMsg) ("outR", tyMsg)
                               (fun x y => ret (dec y x)) (fun (x : msg) _ _ _ => ret
      intros; unlock rbind; rewrite //=.
      msimp; done.
  unfold_at leftc "outR".

  trans_rev_at leftc "tmp" "outR" "keygen".
  trans_rev_at leftc "tmp" "outR" "outCPA".
  pair_at leftc "leak" "tmp" "X".
  trans_rev_at leftc "X" "leak" "outCPA".
  trans_rev_at leftc "X" "tmp" "outCPA".
  trans_at leftc "outCPA" "X" "inS" tyMsg.
  r_move_at leftc "outCPA" 0.
  r_move_at leftc "X" 1.
  arg_move_at leftc "X" "outCPA" 0.
  fold_at leftc.
  r_ext_at leftc "X" (fun m k => n2 <- enc m k; ret (n2, m)).
      intros.
      apply mbind_eqP.
      intros; msimp.
      erewrite dec_correct.
      apply: erefl.
      done.
 unfold_bind2_at leftc "X" "outCPA" tyCtx.
 unfold_at leftc "X".
 autosubst_at leftc "X" "leak".
 autosubst_at leftc "X" "tmp".
 remove_at leftc "X".
 autosubst_at leftc "tmp" "outR".
 remove_at leftc "tmp".
 trans_rev_at leftc "outCPA" "leak" "keygen".
 trans_rev_at leftc "outCPA" "leak" "inS".
 hid_str_at leftc "outCPA" "outR".
 hid_str_at leftc "keygen" "outR".
 align.
 reflexivity.
Qed.
```

Figure 11: Example proof from our mechanization, showing that $\pi_R \leftrightsquigarrow \pi_R^{\mathsf{simp}}$, from Section 2. Our library is generic over a type for names of channels (here, strings) and a universe of types (here, including a type for messages, ciphertexts, and keys). Each tactic invocation soundly edits the current goal state, thus enabling the user to reason about protocols through interactive transformations.