IPDL: A Simple Framework for Formally Verifying Distributed Cryptographic Protocols

Greg Morrisett
Cornell Tech

Elaine Shi
Carnegie Mellon University

Kristina Sojakova
INRIA

Xiong Fan
University of Maryland

Joshua Gancher [*]
Cornell University

Abstract

Although there have been many successes in verifying proofs of non-interactive cryptographic primitives such as encryption and signatures, formal verification of interactive cryptographic protocols is still a nascent area. While in principle, it seems possible to extend general frameworks such as Easycrypt to encode proofs for more complex, interactive protocols, a big challenge is whether the human effort would be scalable enough for proof mechanization to eventually acquire mainstream usage among the cryptography community.

We work towards closing this gap by introducing a simple framework, Interactive Probabilistic Dependency Logic (IPDL), for reasoning about a certain well-behaved subset of cryptographic protocols. A primary design goal of IPDL is for formal cryptographic proofs to resemble their on-paper counterparts. To this end, IPDL includes an *equational logic* to reason about approximate observational equivalence (i.e., computational indistinguishability) properties between protocols. IPDL adopts a *channel*-centric core logic, which decomposes the behavior of the protocol into the behaviors along each communication channel. IPDL supports straight-line programs with statically bounded loops. This design allows us to capture a broad class of protocols encountered in the cryptography literature, including multi-party, reactive, and/or inductively-defined protocols; meanwhile, the logic can track the runtime of the computational reduction in security proofs, thus ensuring computational soundness.

We demonstrate the use of IPDL by a number of case studies, including a multi-use, secure message communication protocol, a multi-party coin toss with abort protocol, several oblivious transfer constructions, as well as the two-party GMW protocol for securely evaluating general circuits. We provide a mechanization of the IPDL proof system and our case studies in Coq, and our code is open sourced at https://github.com/ipdl/ipdl.

[*]Author order randomized.

1 Introduction

With new decentralized computing paradigms such as blockchains and cloud outsourcing, distributed cryptographic protocols involving rich building blocks such as commitment, zero-knowledge proof, oblivious transfer, and multi-party computation are being rolled out at an unprecedented pace. Compositional and scalable computer-aided proofs for such cryptographic protocols have also become increasingly more important, since they help us verify that new cryptographic primitives match their purported security goals.

In the cryptographic literature, the *de facto* framework for reasoning about distributed cryptography is Universal Composability (UC) [15]. UC adopts a so-called simulation-paradigm, where we want to prove the protocol in question satisfies *approximate observational equivalence* (i.e., computational indistiniguishability) to a particular idealization which relies on trusted functionalities rather than cryptographic mechanisms. Any cryptographic protocol proven secure in UC is known to be concurrently and modularly composable.

To enable scalable formal verification for complex cryptography, our goal is to provide an easy-to-use system for encoding and mechanically checking proofs for multi-party protocols. For the system to be usable by cryptographers, it is important that proofs in the system *approximately match how cryptographers write proofs on paper*.

State of affairs. While a line of work focused on formal verification of cryptography, most earlier works fall short of verifying general, multi-party cryptographic protocols. Earlier works either focused on verification of non-interactive primitives and do not provide a native protocol-calculus for encoding interactive multi-party protocols (e.g., Easycrypt [10] and FCF [37]), or focused on restricted classes of protocols using specific cryptographic primitives such as encryption and message authentication, symbolically modeling encryption and authentication as certain ideal abstractions to facilitate formal verification [13, 20, 21, 31]. For the latter line of works (often referred to as the *symbolic* approach), it is imperative that the "symbolic" ideal abstractions of cryptography exactly match what actual cryptography can provide, and this turns out to be subtle and non-trivial. It was observed that in some works, the symbolic abstractions are a mismatch of what actual cryptography can provide, and consequently, even a formally verified protocol can be broken when instantiated with actual cryptography [3, 32]. CryptoVerif is a related symbolic-style prover that reasons directly in the computational model [12], but is focused on automation over expressivity.

Formally verifying distributed cryptographic protocols is an exciting nascent area that has recently attracted increasing attention from the cryptography as well as the formal methods communities. A couple recent systems, EasyUC [18] and CryptHOL [28] made initial attempts at this goal. These systems adopt bisimulation (i.e., relational invariant) to reason about observational equivalence between protocols (i.e., the observable traces induced by protocols are identically distributed). Bisimulation-style proofs mix reasoning about low-level distributional equivalence with higher-level cryptographic reasoning, making it somewhat cumbersome and unnatural for cryptographers to use these systems — improving the usability of such systems was phrased as a major open question in earlier works [18]. A few recent works [5, 22, 26, 38] made one-off efforts to mechanize the proofs of a single, specific MPC protocol in Easycrypt; however, these works do not aim to provide a general logic for encoding cryptographic protocols in general, and their approach does not natively reason about concurrent composition of protocols. Finally, an elegant line of work [7–9] beginning with Bana and Comon [8] combines both symbolic-style and equational reasoning with unconditional computational soundness. While their framework natively supports automation, it has not been mechanized and it remains unclear how easy it is to encode larger-scale developments such as secure function evaluation or *multi*-party protocols. We give a more detailed comparison with Bana-Comon in Section 7.

1.1 Our Contributions

In this paper, we propose IPDL (short for Interactive Probabilistic Dependency Logic), a language and proof system for reasoning about multi-party cryptographic protocols. IPDL is designed with the following desiderata in mind:

- *Ease of use.* As mentioned, we would like the experience of using IPDL to resemble how cryptographers write proofs on paper. One novelty of IPDL is that its logic directly captures approximate observational equivalence reasoning, which is at the core of common simulation-style proofs for cryptographic protocols Unlike previous works [18, 28] which reason about entire actors (i.e. parties or functionalities), IPDL adopts a *channel-centric* logic, which decomposes the behavior of the protocol into the behaviors along each communication channel. This is the insight that enables us to have a simple equational logic.

- *Support for a broad class of protocols.* IPDL's logic supports straightline protocols with *statically bounded loops* (i.e., loops with a-priori known bounds) — we stress, however, that adversaries are treated as arbitrary probabilistic polynomial-time machines in IPDL. The statically bounded loops can be used to *parametrize* the number of parties, the size of the circuit (e.g., in a secure function evaluation example), the number of invocations in a reactive functionality, and so on. This allows us to capture a broad class of protocols, including most protocols studied in the cryptography literature (see our case studies for examples).

- *Computational soundness.* Since IPDL's logic is straightline supporting statically bounded loops, it allows the logic to symbolically track the runtime and error of the reduction. In this way, the core logic can reason about the security loss in IPDL proofs.

- *Compositional guarantees.* IPDL's approximate observational equivalence notion (also referred to as approximate equivalence for simplicity) follows the elegant Universal Composability (UC) paradigm [15]. In a typical simulation-style proof, we want to reason that some real-world protocol's security is as strong as some ideal-world specification. To do this, we can encode a simulator in IPDL that interacts with the real-world adversary and the ideal specification, and we prove in IPDL that the real-world and ideal-world executions are approximately equivalent in terms of the view from an external environment. In this way, IPDL supports the reasoning of concurrent composition of cryptographic protocols (either with itself or with other protocols) [15].

While IPDL follows the UC paradigm to provide concurrent composition, it is not our goal to capture the full extent of the expressiveness of UC. For example, currently we assume a static corruption model where the set of corrupt parties are determined a-prori. As mentioned, we also impose certain restrictions on the protocols (i.e., straightline with statically bounded loops) to allow explicit tracking of the runtime of the programs, and ensure computational soundness. This seems to be the right sweet-spot between ease-of-use and comprehensiveness: we chose the simplifications carefully such that IPDL can nonetheless capture a broad class of protocols studied in the cryptography literature; and these simplifications allow us to encode proofs in IPDL that are concise and resemble on-paper proofs.

Mechanization using Coq. We have implemented IPDL in Coq, and open sourced it at `https://github.com/ipdl/ipdl`. The main strength of our implementation is that we support *parameterized* protocols, or families of protocols definable by a function in Coq. By doing so, we are able to support protocols indexed over arbitrary Coq types – this includes protocols defined for q adversary queries, n parties, or inductive protocols such as MPC for general circuits.

Rich case studies. We have implemented several case studies which demonstrate how IPDL can help scale up formal verification to more complex protocols than before. The case studies and proofs are included in our open source too. Currently, we provide the following case studies:

1. a multi-use, secure communication network out of an authenticated one using a CPA-secure encryption scheme;
2. a maliciously secure n-party coin toss with abort protocol assuming idealized commitments;
3. several semi-honest oblivious transfer (OT) constructions, including OT from trapdoor permutations [24], 1-out-of-4 OT from 1-out-of-2 [2, 36], and a preprocessing protocol for OT [1, 11]; and
4. a semi-honest 2-party GMW protocol [24], defined over a general family of circuits.

Proof effort. The secure network example can be encoded in 195 lines of code, including description and proofs. In comparison, the recent EasyUC work [18] required 12203 lines of code to realize a *single-use* secure network [1]. Among these above case studies, the most sophisticated is GMW, which uses the OT as a building block. Encoding the description and proof of GMW+OT in IPDL is accomplished with less than 3201 lines of code. As a rough point of comparison, the related work by [5] took 11069 lines of code to encode the garbled circuit + OT [39] protocol (and moreover, their work is a one-off effort focused on mechanizing a single protocol rather than providing a general logic and framework). For some other one-off efforts at mechanizing MPC proofs [5, 22, 38], we were not able to find open source code online, so we cannot provide a direct comparison in terms of lines of code.

2 IPDL by Example: Multi-use Secure Network

In this section, we introduce IPDL and its equational style of reasoning through an example protocol for constructing a simple secure communication network out of an authenticated one using a CPA-secure encryption scheme. Before we introduce the protocol, we will first introduce the basic syntax and semantics of IPDL, along with some background information protocol security.

2.1 Terminology and Background

We define some basic terminology and review some background on UC [15]. A *protocol* is any (distributed) message-passing system, which may give outputs and react to inputs. For protocols to have a meaningful security definition, they may exhibit probabilistic behavior but *not* (possibilistic) nondeterminism. Protocols communicate over *channels*, which for us are unidirectional (i.e., either input or output). The channels of a given protocol can be split into *interfaces*, which are subsets of the channels of a given protocol.

We express the security properties of a protocol through two interfaces: the *external* interface, which is used for the high-level I/O behavior of the protocol; and the *attacker* interface, which is used to define the threat model. Given two protocols P and Q with the same external interface but (possibly) differing attacker interfaces, we say that P *realizes* Q when there exists a *simulator* S which converts the attacker interface of Q to that of P, such that P is observationally equivalent to Q composed with S. This type of simulator is also called a converter in earlier work, such as Constructive Cryptography [30].

We often call the attacker for P the *adversary*, and call the attacker for Q the simulator. Intuitively, when P realizes Q, any adversary's capability of influencing the external interface in P is *upper bounded* by the simulator, since any attack on P can be turned into an equivalent attack on Q.

We typically think of P as the implementation (or the *real* protocol) and Q as the specification (or the *ideal* protocol). While P is comprised of parties who interact with each other using cryptographic mechanisms and may be arbitrarily corrupted by the adversary, Q is comprised of idealized parties who instead interact with a trusted third party (the *functionality*) which performs the protocol's logic in a centralized and trustworthy way. The simulator's capabilities in the ideal protocol are constrained by the functionality and ideal parties to be simple and easy to understand.

Relation to UC. Here, we compare our terminology to UC [15]. In UC, a real world protocol is defined to be the composition of all honest party's code and any hybrid functionalities, while an ideal protocol is usually taken to specifically mean the ideal functionality and all ideal parties. Our notion of protocol is more generic, and refers to any message-passing system definable in IPDL. In particular, any individual party's code in IPDL is considered a protocol, as is any arbitrary composition of protocols.

[1] Among the 12203 lines for EasyUC's secure network example, roughly speaking, 260 lines describe the real-world protocol, 182 lines describe the ideal functionality, 190 lines describe the simulator, and the remaining lines are the proof. We did not count their key exchange since our implementation assumed a trusted key setup. As the EasyUC paper [18] itself acknowledges, part of the complexity arises from the fact that EasyCrypt is procedure-based and does not natively support a protocol calculus; consequently, there was a significant amount of tedious work in writing "boilerplate" code that route messages in between parties and functionalities.

2.2 IPDL in a Nutshell

IPDL is a calculus and equational theory for reasoning about probabilistic protocols. The main judgement of IPDL is that $P \overset{\delta}{=} Q$, where P and Q are interactive protocols, and δ is a *computational error*, which upper-bounds the advantage of a computational attacker from distinguishing P from Q. IPDL supports UC-style reasoning: given protocols P and Q, one can ask whether there exists an appropriately typed simulator S such that $P \overset{\delta}{=} Q||S$, for a reasonable choice of δ.

In contrast to the semantics of UC [15] which utilizes systems of Interactive Turing Machines that must be activated sequentially, IPDL allows for messages to happen in any order consistent with the protocol. This is possible since all messages in IPDL are assumed to be scheduled by the attacker.

The key insight of IPDL is to adopt a *channel-centric* rather than agent-centric viewpoint. All channels in IPDL are write-once, and have a unique behavior associated to them by a *reaction*, which is a program that may sample probabilistic values and read the value of other channels (once those channels have fired). To enable a simple equational theory, all dependencies between channels are required to be statically determined. Thus, control flow may only happen at the level of *data*, but not on the level of the protocol. We stress, however, that the equational logic of IPDL is proven sound relative to a general cryptographic adversary who may use more complex control flows.

2.3 Multi-Use Secure Communication in IPDL

As a first example, we show how to construct a *secure* network from an *authenticated* network[2]. This is accomplished through message encryption, such that an eavesdropper cannot learn any information about the messages sent between Alice, the sender, and Bob, the recipient. Our secure network abstraction is multi-use, i.e., it is parametrized by a parameter q that denotes the number of messages exchanged between Alice and Bob. In this example, we assume that both Alice and Bob are honest and the adversary is a passive eavesdropper (although later on, in our case studies, we will have cases where the parties can be semi-honest or maliciously corrupt).

We conduct the example using syntax from our Coq mechanization which encodes our IPDL core logic. In our Coq implementation, the counterpart of our main judgement in IPDL, $P \overset{\delta}{=} Q$, is written as `P ~= Q`. Our current Coq mechanization has not yet implemented the tracking of error bounds which is part of the IPDL core logic described in Section 3; nonetheless we prove on paper that our logic is computationally sound (see Section 3).

[2]The cryptographic literature uses the terms "secure channel" and "authenticated channel" [15], but we avoid overloading the term "channel" so "channel" always means the low-level, write-once IPDL channels.

More details for our Coq embedding is given in Section 5.

Definitions of authenticated and secure network. Figure 1 shows the IPDL encoding of the definitions of an authenticated network and a secure network, respectively. A network `Net` is parametrized with 1) q, which denotes the maximum number of messages exchanged, 2) m, which denotes the length of each message, and 3) `leakage`, which denotes a leakage function, and a parameter `l` which denotes the length of the leakage. In the definition of `Net`, `I` denotes a vector of q input channels between Alice and the ideal functionality `Net`, `O` denotes a vector q output channels between Bob and the ideal functionality `Net`; and `leak` and `ok` each denotes a vector of q channels between the ideal functionality and the adversary.

In Lines 8-9, for each $j < q$, `Net` reads from the jth channel in `I` to obtain a message `x`, applies the leakage function to `x`, and returns the result to the adversary through the jth channel in `leak`. To do so, we first we perform a parallel composition in IPDL over all $j < q$ (written `\||_(j < q)`). We then assign the jth channel of `leak` (written `leak ## j`) to the *reaction* through the syntax `::=`.

Similarly, in Lines 10-12, for each $j < q$, `Net` reads from the jth channel in `ok`, the jth channel in `I` to obtain a message, and assigns the message to the jth channel in `O`. Note here that we do not use the value along the jth `ok` channel, but only its *timing*: thus, the jth channel of `O` only fires once the jth channel of `ok` does. In other words, the receiver Bob receives the message only when the adversary okays it.

Given our definition of `Net`, we can now define both the authenticated and secure networks. On Line 15, we define an authenticated network `Auth q m` to be equal to `Net` instantiated with the leakage function `id` (i.e., the identity function). This means that the adversary can read the contents of all messages in an authenticated network. Similarly, on Line 16 we define a secure network `Sec q m` to to be equal to `Net` with the constant leakage function `fun _ => [tuple]`, which returns the empty bitstring. Thus in `Sec`, the adversary receives no information about message contents. However, the adversary still learns the timing information of each message; and moreover, since the messages length m is a public parameter, the adversary is assumed to know m, too.

Protocol realizing a secure network. Figure 2 shows the IPDL encoding of a protocol that realizes a secure network from an authenticated one, through the use of encryption. The construction takes the same parameters as `Sec`, i.e., q and m, for the number of messages and length of each message, respectively. Additionally, it takes the following parameters for c and k, for the ciphertext length and key length, and `genK`, `enc`, and `dec`, for generating keys, encryption, and decryption.

We first define the key generation functionality (in Line 12), which samples a key from the distribution `genK`, and assigns it to the channel `K` (which is taken as a parameter of

```
1  Definition Net (q m : nat) {l : nat} (leakage : m -> l)
2     (* External channels *)
3     (I O : q.-tuple (chan m))
4     (* Attacker channels *)
5     (leak : q.-tuple (chan l)) (ok : q.-tuple (chan TUnit))
6          :=
7    [||
8       \||_(j < q) (leak ## j) ::= (x <-- Read (I ## j);;
9                                    Ret (leakage x));
10      \||_(j < q) (O ## j) ::= (_ <-- Read (ok ## j);;
11                                   x <-- Read (I ## j);;
12                                   Ret x)
13   ].
14
15 Definition Auth q m := Net q m id.
16 Definition Sec q m := Net q m (fun _ => [tuple]).
```

Figure 1: Definitions of authenticated and secure networks in IPDL. Both networks are parameterized by the number of queries in question, *q*, and the length of messages, *m*.

the functionality). Then, we define the code for Alice in Line 14, who is parameterized by a vector of channels `I` for message inputs, a channel `K` for the encryption key, and a vector of channels `send` for communicating with the authenticated network. For each $j < q$, Alice: reads a message from the jth channel of `I`; a key from the channel `K`; generates a ciphertext by encrypting the message under the key; and assigns the ciphertext to the jth `send` channel. We define Bob similarly on Line 21: for each $j < q$, we read the jth ciphertext, read the key, and output the corresponding decryption to the jth channel of `O`.

We have the real protocol in total on Line 27. It is parameterized similarly to the `Sec` functionality, except it has leakage channels of length c instead of zero. We compose protocols together in IPDL through first *generating* local communication channels, and composing the subprotocols together using these local channels. On Line 30, we generate the channel `K` for key generation (using the Coq syntax `k <- new k`). Then, on Lines 31 and 32, we generate two fresh *vectors* of channels, `send` and `recv` for the underlying authenticated network. This is done with similar syntax `send <- newvec q @ c`: here, `q` is the length of the vector, and `c` is the length of messages on each channel. Finally, on Lines 34-37, we compose together the key generation functionality, Alice, Bob, and the underlying authenticated network. Thus, the authenticated network will deliver ciphertexts from `send` to `recv`, but only after leaking the ciphertexts along `leak` and receiving the `ok` message.

2.4 Simulator and Proof

To show that our real protocol is secure, we must show that the attacker's capabilities in the real world are *upper bounded* by those in the ideal world, wherein Alice and Bob rely on the functionality `Sec` to communicate. Recall that in the real

```
1  Section AuthToSec.
2  (* Same as in Net *)
3  Context (m q : nat).
4  (* Ciphertext and Key length *)
5  Context (c k : nat).
6
7  (* Algorithms for encryption *)
8  Context (genK : Dist k).
9  Context (enc : m -> k -> Dist c).
10 Context (dec : c -> k -> m).
11
12 Definition FKey (K : chan k) := (K ::= Samp genK).
13
14 Definition alice (I : q.-tuple (chan m)) (K : chan k)
15                  (send : q.-tuple (chan c)) :=
16   \||_(j < q) (send ## j) ::=
17                  (msg <-- Read (I ## j) ;;
18                   key <-- Read K ;;
19                   ctxt <-- Samp (enc msg key) ;;
20                   Ret ctxt)).
21
22 Definition bob (recv : q.-tuple (chan c)) (K : chan k)
23                (O : q.-tuple (chan m)) :=
24   \||_(j < q) (O ## j) ::=
25                  (ctxt <-- Read (recv ## j) ;;
26                   key <-- Read K ;;
27                   Ret (dec ctxt key)).
28
29 Definition Real (I O : q.-tuple (chan m))
30             (leak : q.-tuple (chan c))
31             (ok : q.-tuple (chan TUnit)) :=
32   K <- new k ;;
33   send <- newvec q @ c ;;
34   recv <- newvec q @ c ;;
35   [||
36       FKey K;
37       alice I k send;
38       bob recv K O;
39       Auth q c send recv leak ok
40   ].
```

Figure 2: Authenticated-to-secure network protocol in IPDL.

```
1  Definition Sim
2     (leakI : q.-tuple (chan 0)) (okI : q.-tuple (chan TUnit))
3     (leakR : q.-tuple (chan c)) (okR : q.-tuple (chan TUnit))
4  :=
5  K <- new key ;;
6  [||
7    K ::= (Samp genK) ;
8    \||_(j < q) (leakR ## j) ::=
9            (_ <-- Read (tnth leakI j);;
10            key <-- Read K;;
11            e <-- Samp (enc [tuple of nseq _ false] key);;
12            Ret e);
13   \||_(j < q) (okI ## j) ::= (x <-- Read (okR ## j) ;;
14                               Ret x)
15 ]
```

Figure 3: Simulator for the authenticated-to-secure network example in IPDL.

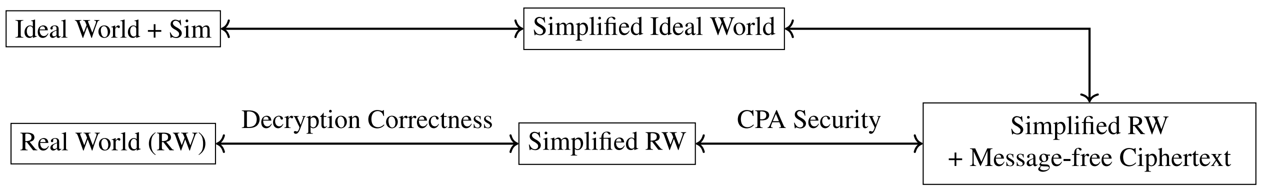

Figure 4: Outline of proof for our Secure Network example in IPDL.

world, the attacker learns any ciphertexts Alice sends through the `leak` channels, but in the ideal world, the attacker learns only the timing of each message. In both the real and ideal worlds, the attacker may use the `ok` channels to schedule the delivery of each message.

Definition of the simulator. The simulator is shown in Figure 3. We describe the simulator informally. It is parameterized by four vectors of channels: two, `leakI` and `okI`, communicate with the ideal world; and two, `leakR` and `okR` communicate with the real world. The simulator must do two things: it must receive timing information for the jth message through `leakI`, and produce a real-looking ciphertext along `leakR`; and it must receive scheduling information along `okR`, and produce scheduling information along `okI`.

To produce the real-looking ciphertexts for `leakR`, the simulator first generates a key channel `K`, similar to the real world, and samples `K` using `genK` (Line 6). Then, when it receives timing information for the jth message through `leakI` (Line 8), encrypts the all-zeroes message using `K` (Line 10), and outputs the ciphertext along the jth channel of `leakR`. (The all-zeroes message is encoded in Coq by `[tuple of nseq _ false]`.) This simulation is successful, since ciphertexts reveal no information about the message if the encryption scheme is CPA secure. Finally, the simulator may simply forward all scheduling decisions for from `okR` to `okI` (Line 12).

Proof of security. Once the simulator is constructed, we compose the ideal world with the simulator using locally generated channels and ask whether the result is approximately equivalent to the real world. This equivalence judgement proves that the adversary's view (`leakR` and `okR`), as well as Alice and Bob's view (`I` and `O`) cannot be distinguished between the two protocols. In the following theorem, `~=` is the Coq notation for the approximate equivalence judgement in IPDL (written on paper as $\overset{\delta}{=}$).

```
  Theorem AutSec_Security I O leakR okR :
    real I O leakR okR ~=
     (leakI <- newvec q @ 0 ;;
      okI <- newvec q @ TUnit ;;
      [||
          ideal I O leakI okI;
          Sim leakI okI leakR okR
      ].
```

We now outline the IPDL proof required to prove the above theorem. The proof is outlined in Figure 4, and contains a number of steps:

1. *Simplifying the ideal world with simulator:* We first apply a number of equational rewrites to the ideal world. In effect, these equational rewrites will inline the behavior of the simulator into the ideal functionality. Recall from Figure 1 in Line 10 that the jth channel of `O` reads from the jth channel of `ok`, which for our ideal functionality is named `okI`. However, the simulator from Figure 3 in Line 13 forwards the value along the jth channel of `okR` into `okI`. In this instance, we can *fold* the definition of `okI` into `O`, which rewrites `O` so that it reads from `okR` directly. Since the internal channel `okI` is now now longer used, we may eliminate it from the protocol. IPDL is specifically designed to perform these kinds of rewrites, and do so in a succinct manner.

   After doing the same inlining step for the internal channel `leakI`, we receive the following protocol:

```
K <- new k ;;
[||
  K ::= Samp genK;
  \||_(j < q) (leakR ## j) ::=
                  (key <-- Read K ;;
                   _ <-- Read (I ## j) ;;
                   c <-- enc [tuple of nseq _ key] key ;;
                   Ret c);
  \||_(j < q) (O ## j) ::=
                  (_ <-- Read (okR ## j) ;;
                   msg <-- Read (I ## j) ;;
                   Ret msg)
]
```

2. *Simplifying the real world:* After simplifying the ideal world, we perform similar inlinings in the real world. Specifically, we inline the definition of `send` (coming from Alice) into the authenticated network, and inline the definition of `recv` (coming from the authenticated network) into Bob. Once we do so, we get that the value of Bob's output is equal to the decryption of Alice's generated ciphertext. To simplify Bob's output we apply an *axiom* which models the correctness of decryption for the encryption scheme. The axiom allows us to perform an equational rewrite to each of Bob's output channels in `O`, and transform it to the reaction that simply reads the message from `I`.

3. *Applying CPA security in the real world:* When we apply the rewrites in Step 2, we receive the following protocol:

```
K <- new k ;;
[||
  K ::= Samp genK;
  \||_(j < q) (leakR ## j) ::=
                  (key <-- Read K ;;
                   msg <-- Read (I ## j) ;;
                   c <-- enc msg key ;;
                   Ret c);
  \||_(j < q) (O ## j) ::=
                  (_ <-- Read (okR ## j) ;;
                   msg <-- Read (I ## j) ;;
                   Ret msg)
]
```

Note that this protocol is almost the same as the simplified ideal world from Step 1: The behavior along the `O` channels are exactly the same, but the behavior of the `leakR` channels here encrypts the real message, while the protocol in Step 1 encrypts the filler message.

To prove these two protocols equivalent, we first apply a *congruence* rule, which allows us to factor out the common behaviors for `O`, and focus only on the equivalence of the `leakR` channels between the real and ideal worlds. At this point, we can directly apply our equational axiom for CPA security, which states that no adversary can tell the difference between encryptions of arbitrarily chosen messages from encryptions of filler messages (given that they key is secret). This axiom applies directly to our two worlds, which finishes the proof.

3 Core Logic

In this section, we describe the core calculus of IPDL in detail, along with its semantics. Sections 3.1, 3.2, and 3.3 describe the syntax, typing rules, and equational logic of IPDL. In Sections B and B.1 we describe the semantics of IPDL protocols and their interaction with adversaries.

3.1 Syntax

The syntax of IPDL is shown in Figure 5. All data types τ in IPDL are assumed to have a bitstring length $|\tau|$, along with an interpretation $[\![\tau]\!] : \{0,1\}^{|\tau|}$. For our examples, we assume a unit type, booleans, products, and bitstrings of a given length $n \in \mathbb{N}$, along with their standard bitstring interpretations.

Protocols in IPDL are composed of *expressions*, *distributions*, *reactions*, and *protocols*. Expressions are built out of collection of function symbols $\mathsf{f}(e_1,\ldots,e_n)$, with an assumed typing rule and interpretation mapping bitstrings to bitstrings (of the appropriate lengths, depending on the type of f). For clarity, we show the standard connectives for unit, bool, products, and bitstrings (not shown).

Distributions represent probabilistically determined messages. Along with distribution symbols $\mathsf{D}(e_1,\ldots,e_n)$ which,

Variables	x		
Channels	c		
Channel Sets	I,O,C	$::=$	$\{c_1,\ldots,c_n\}$
Data Types	τ	$::=$	$\mathsf{unit} \mid \mathsf{bool} \mid \tau_1 \times \tau_2$
		$\mid$	$\mathsf{bits}(n) \quad (\text{with } n \in \mathbb{N})$
		$\mid$	$\ldots$
Variable Contexts	Γ	$::=$	$x_1 : \tau_1,\ldots,x_n : \tau_n$
Channel Contexts	Δ	$::=$	$c_1 : \tau_1,\ldots,c_n : \tau_n$
Expressions	e	$::=$	$x \mid \mathsf{tt} \mid \mathsf{true} \mid \mathsf{false}$
		$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$
		$\mid$	$(e_1,e_2) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e)$
		$\mid$	$\mathsf{f}(e_1,\ldots,e_n)$
Distributions	D	$::=$	$1_e \mid x : \tau \leftarrow D_1; D_2$
		$\mid$	$\mathsf{Unif}(\tau) \mid \mathsf{D}(e_1,\ldots,e_n)$
Reactions	R	$::=$	$\mathsf{Ret}\ e$
		$\mid$	$\mathsf{Samp}\ D$
		$\mid$	$\mathsf{Read}\ c$
		$\mid$	$x : \tau \leftarrow R_1; R_2$
Protocols	P,Q	$::=$	$c := R \mid P_1 \mid\mid P_2$
		$\mid$	$\nu c : \tau.\ P \mid 0$

Figure 5: Syntax of IPDL Protocols.

similarly to function symbols, have a type and an interpretation, we assume the unit distribution 1_e, monadic bind, and the uniform distribution $\mathsf{Unif}(\tau)$ for any IPDL type τ. Distributions are assumed to always have unit mass.

Reactions can be seen as effectful programs which may sample from probability distributions and read from channels. Reactions themselves also carry a monadic structure. reaction with no variables is called *closed*; a reaction with no reads is necessarily equal to a sampling. Note that reactions may *not* contain any control flows themselves; thus, all effects which a reaction may perform are statically determined. Reactions intuitively have a semantics mapping valuations on channels to either distribution on return values or an error (if the required input channels do not have values set yet.)

Finally, a protocol is an interacting network of reactions. A protocol can either be defined by assigning a closed reaction to a channel, a parallel composition, the spawning of a new, fresh channel, or the zero protocol 0.

3.2 Typing

Typing $\Gamma \vdash e : \tau$ for expressions and $\Gamma \vdash D : \tau$ for distributions is standard. The typing $\Delta; \Gamma \vdash R : I \to \tau$ for reactions is shown in Figure 6; it says that R is a reaction reading from channels in I and returning a distribution of type τ, if successful. The channel context Δ declares the channels available for sending and receiving messages (we note that Δ stays unchanged throughout the typing judgement), and the variable context Γ is used for constructing messages.

$$\frac{\Gamma \vdash e : \tau}{\Delta;\ \Gamma \vdash \mathsf{Ret}\ e : \emptyset \to \tau}\ \textsc{Ret}$$

$$\frac{\Gamma \vdash D : \tau}{\Delta;\ \Gamma \vdash \mathsf{Samp}\ D : \emptyset \to \tau}\ \textsc{Samp}$$

$$\frac{\Delta \vdash c : \tau}{\Delta;\ \Gamma \vdash \mathsf{Read}\, c : \{c\} \to \tau}\ \textsc{Read}$$

$$\frac{\Delta;\ \Gamma \vdash R_1 : I_1 \to \tau_1 \qquad \Delta;\ \Gamma, x : \tau_1 \vdash R_2 : I_2 \to \tau_2}{\Delta;\ \Gamma \vdash x : \tau_1 \leftarrow R_1; R_2 : I_1 \cup I_2 \to \tau_2}\ \textsc{Bind}$$

Figure 6: Typing for Reactions.

Typing for programs has the form $\Delta \vdash P : I \to O$, where I and O are finite sets of channel names denoting inputs and outputs, respectively. The typing rules for IPDL are given in Figure 8. Rule RCT states that the inputs and outputs to a reaction $c := R$ are given by set I of the channels R reads from except c, and the single channel c, respectively. The most subtle rule is PAR, which states that $P \mid\mid Q$ is well-typed if P and Q have disjoint outputs; and if so, then the inputs of $P \mid\mid Q$ are inputs of either P or Q (or both) that do not appear as outputs in the other program, and the outputs of $P \mid\mid Q$ are the outputs of P or Q. This rule bears a close resemblance to typed approaches to module linking; *e.g.*, as in [23]. We note that $\Delta \vdash I \to O$ implies $I \cap O = \emptyset$.

Typing $\Gamma \vdash e : \tau$ for expressions and $\Gamma \vdash D : \tau$ for distributions is standard. The typing $\Delta;\ \Gamma \vdash R : I \to \tau$ for reactions is shown in Figure 6; it says that R is a reaction reading from channels in I and returning a distribution of type τ, if successful. The channel context Δ declares the channels available for sending and receiving messages (we note that Δ stays unchanged throughout the typing judgement), and the variable context Γ is used for constructing messages.

Typing for protocols has the form $\Delta \vdash P : I \to O$, where I and O are finite sets of channel names denoting inputs and outputs, respectively. The typing rules for IPDL are given in Figure 8. Rule RCT states that the inputs and outputs to a reaction $c := R$ are given by set I of the channels R reads from except c, and the single channel c, respectively. The most subtle rule is PAR, which states that $P \mid\mid Q$ is well-typed if P and Q have disjoint outputs; and if so, then the inputs of $P \mid\mid Q$ are inputs of either P or Q (or both) that do not appear as outputs in the other program, and the outputs of $P \mid\mid Q$ are the outputs of P or Q. This rule bears a close resemblance to typed approaches to module linking; *e.g.*, as in [23].

3.3 Equational Logic

The main feature of IPDL is that we are enabled to reason *equationally* about protocols using rewrite rules. To obtain computational soundness, our equational logic tracks the adversary's run time and computational error incurred during the proof.

At the level of expressions, we assume a user-defined equational theory supporting judgements of the form $\Gamma \vdash e_1 = e_2 : \tau$ for well-typed e_1 and e_2. We assume a similar judgement $\Gamma \vdash D_1 = D_2 : \tau$ for distributions. We assume that equality (both for expressions and distributions) is well-behaved with respect to substitution. For distributions, we additionally assume the equational theory for commutative monads as well as the weakening rule:

$$\frac{\Gamma \vdash D_1 : \tau \qquad x \notin D_2}{\Gamma \vdash x : \tau \leftarrow D_1; D_2 = D_2}$$

We now describe the equational theory for reactions, similarly written $\Gamma \vdash R_1 = R_2$. Most rules are standard, and encode the equational theory of commutative monads. We highlight the most interesting rules here, and leave the rest for the appendix in Figure 10. We first have two rules for relating the monadic structure of reactions and distributions:

$$\frac{}{\begin{array}{l}\Delta;\ \Gamma \vdash \mathsf{Samp}\ \big(x : \tau_1 \leftarrow D_1; D_2\big) \\ = \big(x : \tau_1 \leftarrow \mathsf{Samp}\ D_1; \mathsf{Samp}\ D_2\big)\end{array}}\ [\textsc{SampBind}]$$

$$\frac{}{\Delta;\ \Gamma \vdash \mathsf{Samp}\ 1_e = \mathsf{Ret}\ e}\ [\textsc{SampRet}]$$

Next, we have the *contraction* rule, stating that reading from the same channel twice is equivalent to reading it once:

$$\frac{}{\begin{array}{l}\Delta;\ \Gamma \vdash \big(x : \tau_1 \leftarrow \mathsf{Read}\ c; y : \tau_1 \leftarrow \mathsf{Read}\ c; R\big) \\ \qquad = \big(x : \tau_1 \leftarrow \mathsf{Read}\, c; R[y/x]\big)\end{array}}\ [\textsc{Contr}]$$

For protocols, we have the judgement $\Delta \vdash P \overset{\delta}{=} Q$, which states (informally) that any computational adversary (called the "environment" in UC [15]) with running time at most k cannot distinguish interaction with P from Q with advantage greater than $\delta(k)$. Here, $\delta : \mathbb{N} \to \mathbb{R}$ is an *error*, which maps adversary running times to an upper bound on distinguishing advantage. Since greater computation power allows the adversary to gain distinguishing advantage, we assume throughout that δ is an increasing function. We allow user defined axioms for (approximate) program equivalences, which are used to define assumptions on the security of a cryptosystem or hardness assumption. The equational theory of programs is given in Figure 7. Our judgement is directly inspired from the work on Task-PIOA [17]. We will write $\Delta \vdash P = Q$ for the special case of exact equality when δ is the constant zero function.

We now discuss a selection of the rules from Figure 7. The most important rule is [CompCong], which states that

$$\frac{}{\Delta \vdash P = P}\ [\textsc{Refl}] \qquad \frac{\Delta \vdash P_1 \overset{\delta}{=} P_2}{\Delta \vdash P_2 \overset{\delta}{=} P_1}\ [\textsc{Sym}] \qquad \frac{\Delta \vdash P_1 \overset{\delta_1}{=} P_2 \qquad \Delta \vdash P_2 \overset{\delta_2}{=} P_3 \qquad \delta_3 = \delta_1 + \delta_2}{\Delta \vdash P_1 \overset{\delta_3}{=} P_3}\ [\textsc{Trans}]$$

$$\frac{\Delta_1, x : \tau_1, y : \tau_2, \Delta_2 \vdash P_1 \overset{\delta}{=} P_2}{\Delta_1, y : \tau_2, x : \tau_1, \Delta_2 \vdash P_1 \overset{\delta}{=} P_2}\ [\textsc{Exchange}] \qquad \frac{\Delta \vdash P_1 \overset{\delta}{=} P_2 \qquad x \notin \Delta}{\Delta, x : \tau \vdash P_1 \overset{\delta}{=} P_2}\ [\textsc{Weakening}]$$

$$\frac{\Delta;\ \cdot \vdash R_1 = R_2}{\Delta \vdash (c \coloneqq R_1) = (c \coloneqq R_2)}\ [\textsc{ReactCong}] \qquad \frac{\Delta \vdash P \overset{\delta}{=} Q : I \to O \ \ \mathsf{axiom}}{\Delta \vdash P \overset{\delta}{=} Q}\ [\textsc{Axiom}]$$

$$\frac{\Delta \vdash P_1 \overset{\delta}{=} P_2 \qquad Q \ b\text{-bounded} \qquad \delta'(k) = \delta(c_{\mathsf{comp}} * |\Delta| * \mathsf{max}(k,b))}{\Delta \vdash P_1 \mid\mid Q \overset{\delta'}{=} P_2 \mid\mid Q}\ [\textsc{CompCong}] \qquad \frac{\Delta, c : \tau \vdash P \overset{\delta}{=} Q}{\Delta \vdash \nu c : \tau.\ P \overset{\delta}{=} \nu c : \tau.\ Q}\ [\textsc{NewCong}]$$

$$\frac{}{\Delta \vdash P_1 \mid\mid P_2 = P_2 \mid\mid P_1}\ [\textsc{CompSym}] \qquad \frac{}{\Delta \vdash (P_1 \mid\mid P_2) \mid\mid P_3 = P_1 \mid\mid (P_2 \mid\mid P_3)}\ [\textsc{CompAssoc}]$$

$$\frac{\Delta \vdash P : C \to \emptyset \qquad C \subseteq I \cup O}{\Delta \vdash P \mid\mid Q = Q}\ [\textsc{AbsorbComp}] \qquad \frac{c \notin P}{\Delta \vdash P \mid\mid \nu c : \tau.\ Q = \nu c : \tau.\ P \mid\mid Q}\ [\textsc{CompNew}]$$

$$\frac{}{\Delta \vdash \nu c_1 : \tau_1.\ \nu c_2 : \tau_2.\ P = \nu c_2 : \tau_1.\ \nu c_1 : \tau_2.\ P}\ [\textsc{NewExchange}]$$

$$\frac{x \notin R_2}{\begin{array}{l}\Delta \vdash \Big(c_1 \coloneqq (x : \tau_0 \leftarrow \mathsf{Read}\ c_0;;R_1) \mid\mid (c_2 \coloneqq (y : \tau_2 \leftarrow \mathsf{Read}\ c_1;;R_2))\Big)\\ = \Big(c_1 \coloneqq (x : \tau_1 \leftarrow \mathsf{Read}\ c_0;;R_1) \mid\mid (c_2 \coloneqq (x : \tau_0 \leftarrow \mathsf{Read}\ c_0;;\ y : \tau_1 \leftarrow \mathsf{Read}\ c_1;;R_2))\Big)\end{array}}\ [\textsc{ResourceTrans}]$$

$$\frac{c_1 \notin \Delta \qquad c_1 \notin R_1 \qquad c_1 \notin R_2}{\Delta \vdash \Big(\nu c_1 : \tau_1.\ c_1 \coloneqq R_1 \mid\mid c_2 \coloneqq (x : \tau_1 \leftarrow \mathsf{Read}\ c_1;;R_2)\Big) = \Big(c_2 \coloneqq (x : \tau_1 \leftarrow R_1;;R_2)\Big)}\ [\textsc{UnFold}]$$

$$\frac{\Delta \vdash \big(x : \tau_1 \leftarrow R_1;;\ y : \tau_1 \leftarrow R_1;;\ \mathsf{Ret}(x,y)\big) = \big(x : \tau_1 \leftarrow R_1;;\ \mathsf{Ret}(x,x)\big)}{\Delta \vdash \Big(c_1 \coloneqq R_1 \mid\mid c_2 \coloneqq (x : \tau_1 \leftarrow \mathsf{Read}\ c_1;;R_2)\Big) = \Big(c_1 \coloneqq R_1 \mid\mid c_2 \coloneqq (x : \tau_1 \leftarrow R_1;;R_2)\Big)}\ [\textsc{Substitution}]$$

$$\frac{x \notin R_2}{\Delta \vdash \Big(c_1 \coloneqq R_1 \mid\mid c_2 \coloneqq (x : \tau_1 \leftarrow \mathsf{Read}\ c_1;;R_2)\Big) = \Big(c_1 \coloneqq R_1 \mid\mid c_2 \coloneqq (x : \tau_1 \leftarrow R_1;;R_2)\Big)}\ [\textsc{UnusedResource}]$$

Figure 7: The IPDL proof system for protocol equivalence.

$$\frac{}{\Delta \vdash 0 : \emptyset \to \emptyset}\ \textsc{zero} \qquad \frac{\Delta;\ \cdot \vdash R : I \to \tau \qquad \Delta \vdash c : \tau}{\Delta \vdash (c \coloneqq R) : I \setminus \{c\} \to \{c\}}\ \textsc{rct}$$

$$\frac{\Delta, c : \tau \vdash P : I \to O \cup \{c\} \qquad c \notin I \qquad c \notin O}{\Delta \vdash \nu c : \tau.\ P : I \to O}\ \textsc{hide}$$

$$\frac{\begin{array}{c}\Delta \vdash P : I_1 \to O_1 \qquad \Delta \vdash Q : I_2 \to O_2 \qquad O_1 \cap O_2 = \emptyset\\ I = (I_1 \cup I_2) \setminus (O_1 \cup O_2) \qquad O = O_1 \cup O_2\end{array}}{\Delta \vdash P \mid\mid Q : I \to O}\ \textsc{par}$$

Figure 8: Typing Rules for Protocols.

if P_1 is approximately equivalent to P_2, then for any Q (of the appropriate type), $P_1||Q$ is approximately equivalent to $P_2||Q$. This is the rule that enables modular reasoning in IPDL. To reason about the error incurred by using this rule, we define the notion of *b-boundedness*: an IPDL program Q is b-bounded if, intuitively, its behavior can be simulated with a probabilistic algorithm with at most b time steps (defined formally in B). Given this notion, the rule [CompCong] changes the attacker's running time to $O(|\Delta| * \mathsf{max}(k,b))$; this is because the attacker for P_1 (and P_2) must simulate the behavior of Q, which increases its running time.

Similarly, [HideCong] states that $\overset{\delta}{=}$ forms a congruence under the spawning of a new channel. Rule [HideComp] states that our hiding operator commutes with parallel composition, under the assumption that no extra channels are affected. We note that this rule is closely related to the concept of *scope extrusion* in the π-calculus (i.e., as in [34]).

Rule [RESOURCE TRANS] states that if c_0 is an input to the reaction defining c_1, and c_1 is an input to the reaction defining c_2, then we may freely add c_0 to the inputs of c_2.

The last three rules specify under what conditions we may replace a read from a channel c by the reaction R defining it. The first scenario in which this is sound is when the reaction R is non-probabilistic – this is rule [SUBST]. The second case when we may replace a read from c by the reaction R is when the value read from c is in fact never used – this is rule [UNUSEDRESOURCE]. Lastly, we may perform this replacement if the channel c is read from in precisely one place – this is rule [UNFOLD]. Reading from right to left, this rule also serves to relate the monadic bind at the level of reactions to the parallel composition of programs.

3.4 Semantics

We now informally describe our semantic model for IPDL, as well as our proof of soundness. Technical details can be found in Section B. We interpret each IPDL program P as an *I/O automaton*, which is a probabilistic transition system that can deliver outputs and react to inputs.

IPDL equivalence judgements are proven sound relative to a semantic adversary, who is formulated as a *dual* automaton (along with some extra data). The adversary is responsible for interacting with the protocol, choosing the order in which outputs occur, and eventually outputting a decision bit after some k number of rounds. We define k-bounded adversaries to be those which run for k rounds, and each round may only take k time steps in its internal transition functions. [3] Given a k-bounded adversary A and IPDL program P, we write $A(P)$ to mean the distribution on booleans defined by letting A and P interact for k rounds, and observing the decision bit of A. We stress that our automata model, and thus our adversarial model, is *not* limited to the syntax of IPDL, but instead can describe arbitrary behaviors, including conditional branching and other forms of control flow.

Soundness. Our soundness theorem states that whenever $\Delta \vdash P \overset{\delta}{=} Q$, any k-bounded adversary has an advantage at most $\delta(k)$ in distinguishing P from Q. Note here that δ is derived from a proof in our logic, and will consist of the sum of a number of errors incurred by applying IPDL axioms.

Theorem 1. *Suppose $\Delta \vdash P : I \to O$ and $\Delta \vdash Q : I \to O$ are two IPDL programs such that $\Delta \vdash P \overset{\delta}{=} Q$. Then for all k-bounded adversaries A, $|\Pr[A(P) = 1] - \Pr[A(Q) = 1]| \leq \delta(k)$.*

The proof of Theorem 1 is given in Section B. We now give some detail about the proof. For the rules with error zero, we employ *bisimulation* arguments, to directly show the two protocols have the same behaviors. For the [COMPCONG] rule, we must transform an arbitrary adversary A for the composition $P_1||Q$ to an adversary $A||Q$ for the protocol P_1. The bound $c_{\mathsf{comp}} * |\Delta| * \max(k,b)$ comes directly from the proof.

[3]Without loss of generality we take k be the upper bound on the adversary's running time per round and the number of rounds.

4 Parameterized Programs and Computational Soundness

In this section, we consider *parameterized protocols*: families of IPDL protocols $\{P_j\}$, ranging over some index set j. Parametrization in IPDL can be used to encode the number of parties (e.g., our n-party coin flip with abort example), number of reactive sessions (e.g., our secure network example), as well as for ranging over more complicated index sets (e.g., for expressing arbitrary circuits in our GMW example). In Section 4.1, we describe how Theorem 1 applies to PPT adversaries and computational indistinguishability. In Section 4.2, we describe some derived equational rules for reasoning about parameterized programs.

4.1 Soundness for PPT Adversaries

While our core logic in Section 3 does not reason about parameterization, we show here that we can use the logic to reason about protocols which depend on a security parameter. In this section, we consider parameterized IPDL protocols of the form $\{P^\lambda\}$, parameterized by a security parameter $\lambda \in \mathbb{N}$. Similarly, we consider families of channel contexts $\{\Delta^\lambda\}$, and families of errors $\{\delta^\lambda\}$.

We lift computational indistinguishability to parameterized IPDL protocols in a straightforward manner. First, note that the family of errors $\{\delta^\lambda\}$ can be seen as a two-place function: the first argument is the security parameter, while the second is the adversary's running time. Correspondingly, we say that the family $\{\delta^\lambda\}$ is *negligible* if for all polynomials p, $\delta^\lambda(p(\lambda))$ is a negligible function of λ. We define *PPT adversaries* to be families of adversaries $\{A^\lambda\}$ such that there exists a polynomial p where A^λ is $p(\lambda)$-bounded. Then, we have the following corollary immediate from Theorem 1:

Corollary 1. *Suppose that $\Delta^\lambda \vdash P^\lambda \overset{\delta^\lambda}{=} Q^\lambda$, and the family $\{\delta^\lambda\}$ is negligible. Then, for any PPT adversary $\{A^\lambda\}$, the quantity*

$$|\Pr[A^\lambda(P^\lambda) = 1] - \Pr[A^\lambda(Q^\lambda) = 1]|$$

is a negligible function of λ.

The (parameterized) error parameter $\{\delta^\lambda\}$ may grow in IPDL for two reasons: either by applying an axiom, or by applying the [COMPCONG] rule, which grows the adversary's runtime by the runtime of the common context. As long as the proof has polynomially many rewrites, the error family for each axiom is negligible, and the runtime of each context for the [CompCong] rule is polynomial, we are guaranteed that $\{\delta^\lambda\}$ is a negligible family of errors.

4.2 Derived IPDL Constructs and Equations

We now turn to reasoning principles in IPDL for parameterized programs. To build parameterized programs systematically, we introduce two pieces of syntactic sugar on top of the core IPDL syntax. Let $n \in \mathbb{N}$ be a variable in the ambient meta-logic. First, *vectorized channel generation*, $\nu\ \overrightarrow{v}^n : \tau.\ P$, generates a fresh vector of channels $\{v_i\}_{i\in\{1\ldots n\}}$ for use in protocol P. Second is the notation $||_{j\in J} P_j$ for composing a family of protocols P_j together, for all j in some finite index set J. Both pieces of syntactic sugar are reflected in our Coq formalization, as seen e.g. in Section 2. While each P_j must be an IPDL program, we emphasize that the mapping $j \mapsto P_j$ and the index set J are all defined in the ambient logic and may make use of arbitrary set theoretic reasoning. This reflects our Coq formalization, which uses the `bigop` and `fintype` libraries from ssreflect [25] to manage parameterized composition and index sets of bounded natural numbers.

Derived IPDL rules. We additionally introduce a number of derived rules for IPDL for reasoning about parameterized programs. We describe the most important rules here, and leave the rest for Figure 11 in the Appendix.

One of our most widely used rules is [EqBig], which states that parameterized composition is a congruence, which states that in order to prove that $||_{j\in J}\ P_j$ is equivalent to $||_{j\in J}\ Q_j$, it suffices to show that P_i is equivalent to Q_i for each $i \in J$.

Next, we have a number of rules involving manipulating the index sets for parameterized composition, directly inspired from the `bigop` library. Most importantly, we have that we can arbitrarily split up compositions: any composition $||_{j\in J}\ P_j$ can be split into the composition of $||_{j\in J\cap K} P_j$ and $||_{j\in J\cap \widetilde{K}} P_j$, where $\widetilde{K}$ is the complement of K. We additionally have that composition is compatible with parameterized composition: that is, $||_{j\in J} P_j$ composed with $||_{j\in J} Q_j$ is equivalent to $||_{j\in J}(P_j\ ||\ Q_j)$.

Finally, we describe our most powerful rule, [Hybrid]:

$$\frac{\begin{array}{l}\forall k < n, \Gamma \vdash (\underset{j<k}{||}\ P_j)||R = (\underset{j<k}{||}\ Q_j)||R \\ \Rightarrow \Gamma \vdash (\underset{j<k}{||}\ P_j)||P_k||R = (\underset{j<k}{||}\ P_j)||Q_k||R\end{array}}{\Gamma \vdash (\underset{j<n}{||}\ P_j)||R = (\underset{j<n}{||}\ Q_j)||R}\ [\text{Hybrid}]$$

This rule states that to transform one composition of a protocol family $\{P_j\}$ into another one $\{Q_j\}$ (say, for the index set $\{0\ldots n\}$) in the presence of a common context R, we may instead prove that for any $k < n$, if we have the composition of $\{P_j\}_{j\leq k}$ along with R, we may rewrite the last P_k to Q_k.

5 Encoding in Coq

In this section, we describe our encoding of IPDL in Coq.

Basic syntax. First, we describe how we embed types, expressions, and distributions. Our encoding is *shallow*, meaning that expressions and functions in IPDL are represented using their native Coq analogues. IPDL types are given by an inductive Coq type `type := TBool | TUnit | TBits (n : nat) | TPair (t1 t2 : type)`. As is standard, IPDL types in Coq come equipped with a function `interpType : type -> Type`, which maps each IPDL type into its interpretation as a Coq type. This mapping is standard; we use the `tuple` library of ssreflect [25] to model bitstrings. We model distributions syntactically, as finite boolean decision trees.

We now turn to channels, reactions, and IPDL protocols:

```
1  Axiom chan : type -> Type.
2
3  Definition Chan := {t : type & chan t}.
4
5  Inductive rxn : type -> Type :=
6  | Samp {t : type} : Dist t -> rxn t
7  | Ret {t : type} : t -> rxn t
8  | Read {t : type} (c : chan t) : rxn t
9  | Bind {t1 t2 : type} : rxn t1
10                    -> (t1 -> rxn t2)
11                    -> rxn t2.
12
13 Inductive WfRxn : list Chan -> rxn t -> Prop := ...
14
15 Inductive ipdl : Type :=
16 | prot0 : ipdl
17 | Out {t} (c : chan t) : rxn t -> ipdl
18 | Par : ipdl -> ipdl -> ipdl
19 | New t : (chan t -> ipdl) -> ipdl.
```

To model channel binding in Coq, we opt for the *weak HOAS* approach [19], which models channels through a `type`-indexed abstract Coq type, given by an axiom. Since channels have type tags, we use a dependent sum to speak about the collection of all channels, `Chan`. Reactions are encoded monadically, as in Section 3. For ease of use, we adopt the usual monadic syntax `x <-- r ;; k` to represent monadic binds. For convenience, we do *not* enforce that reactions are well-typed through the Coq type system, but instead embed the typing judgement in the proposition `WfRxn`: if `WfRxn G r` holds, then `r` performs exactly the reads as specified through the sequence of channels `G`. This encoding is faithful to the syntax in Section 3, which does not allow pattern matching or branching at the level of reactions: since `WfRxn` enforces that all `Read` effects must be identical in all branches, the reaction is equivalent to one without reaction-level branching.

Finally, in Line 17, we encode IPDL programs through the datatype `ipdl`. In the datatype we use syntax `Out` and `Par`, but these are also captured by the Coq notations `::=` and `||` respectively. Since we use weak HOAS, we are enabled to encode channel binding in `New` through an ordinary Coq function. We allow use of the more natural syntax `x <- new t ;; P`. As is standard [19], this encoding requires us to additionally encode the predicate `chansOf : ipdl -> Chan -> Prop` to model the free variables of IPDL programs, since we cannot soundly assume decidable equality of channels. We provide tactics for (mostly) automatically discharging goals involving `chansOf` and related constructions.

Typing judgements. We similarly encode the typing judgement of IPDL programs through an inductive datatype. It follows the same rules as in Section 3, except for the ν operator for local channel generation. Since we cannot directly assume a specific channel is globally fresh in Coq (e.g., as in nominal calculi [4]), we parameterize the typing judgement by a finite collection of channels `X` and assume that the new channel `c` is only fresh *against* the channels in `X`.

Equational theory. We encode equivalences of reactions and IPDL programs through the inductive datatypes `EqRxn` and `EqProt`, respectively. Our libraries for IPDL make heavy use of Coq's support for setoid rewriting to enable easy proofs. Their definitions closely follow the rules in Section 3, except for the following differences: 1) we do not reason about a separate monadic bind operator for distributions and reactions, but only the one for reactions; 2) we give ourselves the liberty to include a few derived rules for managing channel dependencies and reasoning about probability distributions; 3) our Coq implementation currently does not reason about computational error (i.e., the δ parameter). We plan on introducing reasoning about computational error and protocol run time to a future iteration of our implementation.

Encoding of parameterized protocols. One of the major strenghts of our Coq encoding is that we are able to write arbitrary Coq programs to generate IPDL protocols, effectively using Coq as a meta-programming environment for IPDL. Following Section 4, we use the `bigop` library from ssreflect [25]: we model parameterized composition $||_{j \in J} P_j$ using the syntax `\||_(j < n | p j) f j` where `f` is a function of type `'I_n -> ipdl`, and `'I_n` is the type of natural numbers less than `n`. While we do have support for more general index sets – as in the bigop library, we support using sequences for index sets, as well as general finite types – we only use bounded natural numbers for our proof developments. We model parameterized channel generation $\nu\ \vec{v}^n : \tau.\ P$ through the notation `x <- newvec n @ t ;; P`, which is defined by induction on `n`. Here, the type of `x` is `n.-tuple (chan t)`, or the type of lists of length exactly `n`. This type is borrowed from the `tuple` library of `ssreflect`. All the derived rules in Section 4 are implemented as lemmas in Coq, proven from the basic equational rules of IPDL.

6 Case Studies

In this section, we briefly describe all case studies we have mechanized in IPDL. We defer more detailed description to Appendix C, including Coq sources for selected protocols.

6.1 Case Studies

Multi-use secure network. Our first case study is a multi-use secure network, and we refer the reader to the earlier Section 2 for more details. [4]

Semi-honest OT constructions. In (1-out-of-2) OT, there is a *sender* who has a pair of messages m_0 and m_1, and a *receiver* who has an index bit i. The ideal functionality for OT receives these three protocol inputs, and returns to the receiver m_i. The sender receives no protocol output. All OT protocols we consider are in the *semi-honest* setting, in which the adversary observes corrupted parties' private data, but cannot harm integrity. We encode semi-honest security in IPDL by *annotating* each corrupted party with explicit leakage channels for the adversary, and extending their protocol code appropriately.

We verify three OT protocols: OT from trapdoor permutations, the OT construction by Goldreich et al. [24], a simple preprocessing scheme for OT [1, 11], and construction of 1-out-of-4 OT from 1-out-of-2 OT [2, 36]. All OT constructions are roughly of the same complexity, and emphasize different parts of the system; in particular, the proofs for OT often require complex *rerandomization* steps, in which we transform uniform randomness to eliminate channel dependencies. More details about all OT protocols is given in Section C.1.

Semi-honest, two-party GMW protocol. Our second major case study for IPDL is the GMW protocol [24], a semi-honest secure multiparty computation protocol over bits based on secret sharing. First, we model boolean circuits in Coq as follows:

```
Inductive Op (A B k : nat) :=
  | InA : 'I_A -> Op A B k
  | InB : 'I_B -> Op A B k
  | And : 'I_k -> 'I_k -> Op A B k
  | Xor : 'I_k -> 'I_k -> Op A B k
  | Not : 'I_k -> Op A B k.

Definition Circ A B n := forall (k : 'I_n), Op A B k.

Definition CircOutputs n o := o.-tuple ('I_n).
```

Above, we first introduce the type `Op A B k` of *operations* which may make use of all of Alice's inputs (numbered $0 \dots A-1$), Bob's inputs (numbered $0 \dots B-1$), and all wire IDs from 0 to $k-1$. We then define a circuit to be a mapping

[4] While our example reasons only about a fixed size of message, it is straightforward to adapt our example to the variable length case by considering a type of messages *up to* a given length.

from all wire IDs $j < n$ to an operation which may make use of all wires up to $j-1$. This definition of boolean circuits is equivalent to a more ordinary, inductively defined variant, but is nicer to work with in proofs. Our circuits support multiple outputs, which are modeled through a finite mapping from wire IDs to output IDs, which we define using a `o.-tuple`, or fixed-size list of length `o`. (We assume the same outputs for each party.)

We describe how we encode the ideal/real protocol of GMW in Appendix C.3.

Coin flip with abort. This protocol allows n mutually distrusting parties to collaboratively generate fair randomness [14]. To do so, each party locally generates a bitstring uniformly from $\{0,1\}^k$ and sends a cryptographic commitment of the bitstring to all other parties. We assume a broadcast channel for the commitments to prevent equivocation. Once all other commitments have been collected, each party opens their respective bit, and all parties output the collective XOR of all opened bitstrings. We model the commitment and broadcast channels using a standard UC commitment functionality, which prevents equivocation by construction. Our proof is secure in the malicious model. Modeling details about the protocol are given in depth in Section C.4.

6.2 Proof Effort

In Figure 9, we outline the lines of code needed for each case studies considered. Our simplest example is our secure network example from Section 2, which consists of a number of simple rewriting steps along with the application of two IPDL axioms. Our OT examples, while simple to define, take a modest effort to prove, with the largest proof being the 1-4 OT at 749 lines of code. While the number of lines is moderate, the complexity of the proof script is low: most of the lines consist of repetitive tactic invocations as well as intermediate rewriting steps being explicitly defined as hybrids. It is likely that proofs like these can be further automated with additional engineering effort. Our most complex examples are the GMW protocol and the n-Coin Flip, both of which have proofs of less than 2000 lines of code. Out of the 1995 lines of code for the n-Coin Flip, 345 of them were definitions of intermediate hybrids while the rest were either proof scripts or auxiliary lemmas.

We compare our proof effort with related mechanization efforts in Section 1.

Case study	LoC (Definitions)	LoC (Proof)
Secure Network	73	122
Trapdoor OT	75	568
Preprocessing OT	40	249
1-4 OT	88	749
n-Coin Flip	100	1995
GMW	324	1397

Figure 9: Case studies considered and lines of code.

7 Additional Related Work

More detailed comparison with Bana-Comon. A promising direction ([8], [9], [7]) for protocol verification is initiated by Bana and Comon, where the attacker is not limited by interacting with idealized cryptography, but instead constrained by a number of logical axioms which state what the attacker is not able to do. While this framework has made advances compared to symbolic systems, there is to date no publicly available mechanization of their framework. While some IPDL proofs can likely be automated using these techniques, we anticipate that our more complicated parameterized proofs (e.g., inducting over circuits, handling n parties) would require significant engineering effort similar to ours to mechanize using their framework. Indeed, the strength of our parameterized approach is derived from the usage of a general-purpose theorem prover for defining parameterized protocols; this has no counterpart yet in the Bana-Comon framework.

Frameworks for cryptographic protocols. In the cryptography literature, Universal Composability [15] and Constructive Cryptography [30] are the two dominant definitional frameworks for simulation-based security. Several automata-based frameworks also exist, such as [6] and [16], which, while similar in spirit, aim for a more formal treatment. Additionally, some works use process calculi to model computational cryptographic protocols, such as [35]. A recent effort to formalize the semantics of UC is ILC [27]. While a useful step towards giving formal reasoning support for UC, it does not yet provide support for verification. Additionally, a number of works formalize standalone (non-UC) proofs of interactive protocol security using special-purpose embeddings of protocols into Easycrypt. For example, [26] gives an on-paper reduction of the security of Maurer's MPC protocol [29] to a certain trace property which is directly verified in Easycrypt

An interesting alternative framework is given in Micciancio and Tessaro [33] (hereafter M&T), where they use *complete partial orders* to represent cryptographic protocols as the least fixed point of a recursive set of equations. There is some amount of conceptual overlap between M&T and IPDL: their monotonicity requirement (that further inputs can only create more outputs) is similar to our encoding of protocols, which cannot make use of non-determinism through observing scheduling decisions. However, the framework is not mechanized, and cannot reason about computational error.

Acknowledgements

This research was partially supported by NSF grants 1617676 and 1704788, the IARPA HECTOR grant under a subcontract from IBM, and a DARPA SIEVE grant under a subcontract from SRI. Elaine Shi is partially supported by an ONR YIP award and a Packard Fellowship. Xiong Fan is partially supported by Xiaodi Wu's startup grant from University of Maryland.

References

[1] Introduction to secure computation, lecture 5. http://www.cs.umd.edu/~jkatz/gradcrypto2/f13/lecture5.pdf. Accessed: 2020-11-30.

[2] Secure multi-party computation (gmw protocol + malicious model). https://people.eecs.berkeley.edu/~sanjamg/classes/cs276-fall14/scribe/lec16.pdf. Accessed: 2020-11-30.

[3] Martın Abadi and Phillip Rogaway. Reconciling two views of cryptography. In *Proceedings of the IFIP International Conference on Theoretical Computer Science*, pages 3–22. Springer, 2000.

[4] Samson Abramsky, Dan R Ghica, Andrzej S Murawski, C-HL Ong, and Ian David Bede Stark. Nominal games and full abstraction for the nu-calculus. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 150–159. IEEE, 2004.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. A fast and verified software stack for secure function evaluation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1989–2006, 2017.

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

[7] 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.

[8] 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.

[9] 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, 2014.

[10] 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.

[11] Donald Beaver. Precomputing oblivious transfer. In *Annual International Cryptology Conference*, pages 97–109. Springer, 1995.

[12] 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.

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

[14] Manuel Blum. Coin flipping by telephone a protocol for solving impossible problems. *ACM SIGACT News*, 15(1):23–27, 1983.

[15] 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.

[16] 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.

[17] Ran Canetti, Ling Cheung, Dilsun Kaynar, Nancy Lynch, and Olivier Pereira. Compositional security for task-pioas. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 125–139. IEEE, 2007.

[18] 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.

[19] Alberto Ciaffaglione and Ivan Scagnetto. A weak hoas approach to the poplmark challenge. *arXiv preprint arXiv:1303.7332*, 2013.

[20] 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.

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

[22] Karim Eldefrawy and Vitor Pereira. A high-assurance evaluator for machine-checked secure multiparty computation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 851–868, 2019.

[23] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 250–261, 1999.

[24] 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.

[25] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A small scale reflection extension for the Coq system*. PhD thesis, Inria Saclay Ile de France, 2016.

[26] Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. Computer-aided proofs for multiparty computation with active security. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 119–131. IEEE, 2018.

[27] Kevin Liao, Matthew A Hammer, and Andrew Miller. Ilc: a calculus for composable, computational cryptography. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 640–654, 2019.

[28] 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.

[29] Ueli Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370–381, 2006.

[30] 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.

[31] 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.

[32] Daniele Micciancio. Symbolic encryption with pseudorandom keys. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 64–93. Springer, 2019.

[33] 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.

[34] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and computation*, 100(1):1–40, 1992.

[35] 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.

[36] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 245–254, 1999.

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

[38] Alley Stoughton and Mayank Varia. Mechanizing the proof of adaptive, information-theoretic security of cryptographic protocols in the random oracle model. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 83–99. IEEE, 2017.

[39] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

A Future Work

The first direction of future work for IPDL is to increase its expressivity while still retaining the equational flavor of its logic. For example, support for adaptive corruption and more flexible control flows would be interesting.

An exciting future direction is to integrate IPDL with an underlying battle-hardened cryptographic proof system (such as EasyCrypt [10]) 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. We anticipate that IPDL could also be seen as an equational interface for more expressive tools such as EasyUC [18] or CryptHOL [28]. Additionally, it would be interesting to combine IPDL with ILC [27], n programming language for UC semantics.

B Semantics

In this section, we give semantics to well-typed IPDL programs. Every type can be straightforwardly interpreted as the set of bitstrings of a certain length; if c is a channel declared in a typing context Δ, we will write $|c|$ for the length of bitstrings assigned to c and by abuse of notation we will use natural numbers n to stand for the set of bitstrings of length n. Analogously to types, we interpret variable typing contexts Γ as natural numbers, again corresponding to a set of bitstrings of the specified length. We interpret channel typing contexts Δ as mappings of channel names c to natural numbers $|c|$, specifying that a given channel will carry bitstrings of the given length.

We first describe our semantic model of I/O automata. Let Δ be a channel context, as above. Then, an I/O automaton consists of the following data, for I and O disjoint sets of channels which form Δ:

- a finite set St of *states*
- a start state $\mathsf{s}_\star : \mathsf{St}$
- a *valuation function* $\mathsf{St} \times (o : O) \to 1 + |o|$
- an *input transition function* $\mathsf{St} \times (\Sigma_{i:I}|i|) \to \mathcal{D}(\mathsf{St})$
- an *output transition function* $\mathsf{St} \times (o : O) \to \mathcal{D}(\mathsf{St})$

The valuation function tells us the value of the output o, if any, in a given state. The input transition function takes a state s and an assignment $i := v$, where v is a value of the correct type, and returns a distribution on states. The output transition function takes a state s and an output o, and returns a distribution on states.

We write $s|_o$ for the value of the output o in state s. Given a state s, we write $\langle i := v \rangle\, s$ and $\langle o \rangle\, s$ for the distribution resulting from performing the specified input or output. Using the monadic bind, we can generalize this to distributions σ as $\langle i := v \rangle\, \sigma$ and $\langle o \rangle\, \sigma$.

There are several canonical ways to produce new protocols from old ones. For our purposes, the following three are important:

- Given a protocol P in the typing context $\Delta, c \mapsto |c|$ with an output c, we can restrict P in the obvious way to obtain a new protocol $\nu c : |c|.\ P$ in the reduced typing context Δ. The new protocol has the same states as P but its valuation and output transition functions are now restricted to channels from Δ.
- Given a protocol P with an output o, we define a new protocol $P|_o$ as follows: we have the same states as in P but both before and after performing any input assignment or output computation, we perform o.
- Given two protocols P and Q in the same typing context with inputs I_1 and I_2 and outputs O_1 and O_2 such that $O_1 \cap O_2 = \emptyset$, we can define a new protocol $P \mid\mid Q$ as follows:
  - the states are pairs (s,t), where s and t are states of P and Q, respectively
  - the start state is $(\mathsf{s}_\star, \mathsf{t}_\star)$, where $\mathsf{s}_\star$ and $\mathsf{t}_\star$ are the start states of P and Q, respectively
  - the valuation is defined as
    - $(s,t)|_o := s|_o$ if $o \in O_1$
    - $(s,t)|_o := t|_o$ if $o \in O_2$
  - to perform an input assignment $i := v$ in (s,t), we perform $i := v$ in s and/or t as applicable:
    - If $i \in I_1$ and $i \notin I_2$, the resulting distribution is $\langle i := v \rangle\, s \times 1_t$.
    - If $i \notin I_1$ and $i \in I_2$, the resulting distribution is $1_s \times \langle i := v \rangle\, t$.
    - If $i \in I_1$ and $i \in I_2$, the resulting distribution is $\langle i := v \rangle\, s \times \langle i := v \rangle\, t$.
  - to compute an output o in (s,t), we compute o in s or t, accordingly as to whether o is an output of P or Q. If applicable, we forward the result to the other protocol:
    - If $o \in O_1$ and $o \notin I_2$, the resulting distribution is $\langle o \rangle\, s \times 1_t$.
    - If $o \in O_2$ and $o \notin I_1$, the resulting distribution is $1_s \times \langle o \rangle\, t$.
    - If $o \in O_1$ and $o \in I_2$, we draw a new state r from $\langle o \rangle\, s$. If $r|_o = u$ for some $u \in |o|$, the resulting distribution is $1_r \times \langle o := u \rangle\, t$, otherwise $1_r \times 1_t$.
    - If $o \in O_2$ and $o \in I_1$, we draw a new state r from $\langle o \rangle\, t$. If $r|_o = u$ for some $u \in |o|$, the resulting distribution is $\langle o := u \rangle\, s \times 1_r$, otherwise $1_s \times 1_r$.

For our soundness result, we also need to introduce the concept of a *channel embedding*. Given two contexts Δ and Δ', a channel embedding $\theta : \Delta \to \Delta'$ is an injective function from the indices in Δ to Δ' which preverse channel typing.

We are now ready to describe the interpretation of an IPDL program $\Delta \vdash P : I \to O$. We will proceed in two steps: in the first step we define a *one-step* interpretation $[[-]]_1$ using the above constructs, and in the second step we define the final interpretation $[[-]]$ in terms of the one-step interpretation. When asked to perform an output o, the one-step interpretation attempts to first compute all the hidden channels that o may directly or indirectly depend on; however, it does not yet attempt to compute any output channels, even those that o directly depends on. This is the job for the final interpretation.

Formally, we define $[[P]]_1$ by induction on the structure of P as follows:

- $[[0]]_1$ has a unique state and no output actions
- $[[o := R : \tau]]_1$ has mappings of channel names c to bit-strings of length $1 + |c|$ as states, where c is either an input to the reaction R or the output o. The start state maps every channel name to $\perp$. Performing an input assignment $i := v$ in a state s amounts to setting the value of i in s to v, if not already defined. To compute an output o in a state s, we check if c is already defined in s; if so, we do nothing. Otherwise we execute the reaction R in s (yielding $\perp$ if any of the required input channels are not defined in s).
- $[[P \mid\mid Q]]_1 := [[P]]_1 \mid\mid [[Q]]_1$
- $[[\nu c : \tau.\ P]]_1 := \nu c : [[\tau]].\ [[P]]_1|_c$

It is now possible to prove that for any output o, we have $[[P]]_1|_o|_o = [[P]]_1|_o$ and for any two outputs o_1, o_2 we have $[[P]]_1|_{o_1}|_{o_2} = [[P]]_1|_{o_2}|_{o_1}$. If $\{o_0, \ldots, o_n\}$ are the outputs of P, we define the final interpretation of P to be $[[P]] := [[P]]_1|_{o_0} \cdots |_{o_n}$. Thus, if an output o_2 depends on an output o_1, in the final interpretation the computation of o_2 will take into account the result of the computation of o_2, if any.

Another important property of our semantics of IPDL is that the $|_o$ operator is compatible with composition, in the following sense:

Lemma 1. *For any IPDL programs P and Q with disjoint outputs, and any output o of P,* $([[P]]_1 || [[Q]]_1)|_o = ([[P]]_1|_o || [[Q]]_1)|_o$.

The above lemma can be verified manually by enumerating the cases in which o may fire in each state of the composition, and whether o is an input of Q. A similar result holds for the symmetric case where we add $|_o$ to Q, instead of P.

By applying the above lemma many times, we have that $[[P||Q]] = ([[P]] || [[Q]])|_{o_1,\ldots,o_k}$, where $o_1, \ldots, o_k$ is an arbitrary enumeration of the outputs of P and Q.

Boundedness for IPDL programs In order to reason in a computationally sound manner, we need to estimate the running times of IPDL protocols. We say that an IPDL protocol Q is b-bounded when the size of the state of $[[Q]]$ (in bits) is bounded by b, and for each transition function of the final interpretation $[[Q]]$, there exists a probabilistic Turing machine that runs for at most b time steps which computes this function.

B.1 Adversaries

An *environment* or *adversary* for a protocol P with inputs I and outputs O is specified by:

- a dual adversary protocol A with states St, inputs $I' \subseteq O$, and outputs $I \subseteq O'$
- a *stepping function* $\mathsf{St} \to \mathcal{D}(\mathsf{St})$
- a *decision function* $\mathsf{St} \to \mathsf{bool}$
- an *accept function* $(O \cup O') \to \mathsf{St} \to \mathsf{bool}$
- a *schedule* $\{0, \ldots, k-1\} \to (O \cup O')$

In particular, the adversary does not have access to the states of the protocol. At each step, the schedule decides on performing one of the outputs (of either the protocol or the adversary). In each case, the adversary probabilistically steps to a new state as given by the stepping function. The adversary has the ability to refuse the execution of any scheduled channel.

We now describe how the adversary interacts with a semantic protocol P. Given a state s of the adversary, a state t of the protocol, and an output $o : O \cup O'$ to be performed, we probabilistically determine a new adversary state and a new protocol state as follows:

- We call the stepping function in state s and draw a new adversary state r from the resulting distribution.
- If the accept function for o at s is false, the resulting distribution is $1_r \times 1_t$.
- Otherwise we ask the composed protocol $A \mid\mid P$ to perform o in the state (r,t), to obtain the resulting distribution $\langle o \rangle\, (r,t)$.

We can lift this single execution step to act on distributions of pairs (s,t) of adversary and protocol states. We inductively perform this lifted execution step on each scheduled channel to obtain a final distribution on pairs of adversary and protocol states. At this point we call the decision function to turn the resulting distribution on adversary states to a distribution on booleans. This distribution, denoted $A(P)$, will be the result of the interaction between the adversary and the protocol.

We call an adversary *k-bounded* if:

- the states have length at most k
- the schedule has length at most k
- for each i, the corresponding input transition function is k-bounded

$$\frac{}{\Delta;\, \Gamma \vdash R = R}\ [\textsc{Refl}] \qquad \frac{\Delta;\, \Gamma \vdash R_1 = R_2}{\Delta;\, \Gamma \vdash R_2 = R_1}\ [\textsc{Sym}] \qquad \frac{\Delta;\, \Gamma \vdash R_1 = R_2 \qquad \Delta;\, \Gamma \vdash R_2 = R_3}{\Delta;\, \Gamma \vdash R_1 = R_3}\ [\textsc{Trans}]$$

$$\frac{\Gamma \vdash e_1 = e_2}{\Delta;\, \Gamma \vdash \mathsf{Ret} e_1 = \mathsf{Ret} e_2}\ [\textsc{RetCong}] \qquad \frac{\Gamma \vdash D_1 = D_2}{\Delta;\, \Gamma \vdash \mathsf{sample}\ D_1 = \mathsf{sample}\ D_2}\ [\textsc{SampleCong}]$$

$$\frac{\Delta;\, \Gamma \vdash R_1 = R_3 \qquad \Delta;\, \Gamma, x : \tau_1 \vdash R_2 = R_4}{\Delta;\, \Gamma \vdash \big(x : \tau_1 \leftarrow R_1;; R_2\big) = \big(x : \tau_1 \leftarrow R_3;;\, R_4\big)}\ [\textsc{BindCong}] \qquad \frac{}{\Delta;\, \Gamma \vdash \mathsf{sample}\ 1_e = \mathsf{Ret} e}\ [\textsc{SampleRet}]$$

$$\frac{}{\Delta;\, \Gamma \vdash \mathsf{sample}\ \big(x : \tau_1 \leftarrow D_1; D_2\big) = \big(x : \tau_1 \leftarrow \mathsf{sample}\ D_1;;\, \mathsf{sample}\ D_2\big)}\ [\textsc{SampleBind}]$$

$$\frac{}{\Delta;\, \Gamma \vdash \big(y : \tau_2 \leftarrow (x : \tau_1 \leftarrow R_1;; R_2);; R_3\big) = \big(x : \tau_1 \leftarrow R_1;;\, y : \tau_2 \leftarrow R_2;; R_3\big)}\ [\textsc{BindBind}]$$

$$\frac{}{\Delta;\, \Gamma \vdash \big(x : \tau_1 \leftarrow \mathsf{Ret} e;;\, R\big) = [e/x]R}\ [\textsc{RetBind}] \qquad \frac{}{\Delta;\, \Gamma \vdash \big(x : \tau \leftarrow R;;\, \mathsf{Ret} x\big) = R}\ [\textsc{BindRet}]$$

$$\frac{x \notin R_2 \qquad y \notin R_1}{\Delta;\, \Gamma \vdash \big(x : \tau_1 \leftarrow R_1;;\, y : \tau_2 \leftarrow R_2;; R_3\big) = \big(y : \tau_2 \leftarrow R_2;;\, x : \tau_1 \leftarrow R_1;; R_3\big)}\ [\textsc{Exchange}]$$

$$\frac{}{\Delta;\, \Gamma \vdash \big(x : \tau_1 \leftarrow \mathsf{read}\ c;;\, y : \tau_1 \leftarrow \mathsf{read}\ c;;\, R\big) = \big(x : \tau_1 \leftarrow \mathsf{read} c;;\, [y/x]R\big)}\ [\textsc{Contr}]$$

Figure 10: Equivalence of Reactions in IPDL.

- for each o, the corresponding output transition function is k-bounded
- for each i or o, the corresponding accept function is k-bounded
- the stepping function is k-bounded
- the decision function is k-bounded

We define a *bisimulation* between two comparable protocols P and Q as a binary relation $\sim$ on distributions on the states of P and Q respectively, satisfying the following conditions:

- Initial: the unit distributions on the respective initial states of P and Q are related by $\sim$
- Inputs: if $\mu \sim \eta$, then for any input assignment $i := v$ there exist (convex) coefficients $c_1, \ldots, c_n$ and distributions $\mu_1, \ldots, \mu_n$, $\eta_1, \ldots, \eta_n$ such that $\mu_k \sim \eta_k$ for each $k = 1, \ldots, n$ and

  $$\langle i := v \rangle \mu = \Sigma_{k:=1 \ldots n} c_k \mu_k = \Sigma_{k:=1 \ldots n} c_k \eta_k = \langle i := v \rangle \eta$$

- Outputs: if $\mu \sim \eta$, then for any output o there exist (convex) coefficients $c_1, \ldots, c_n$ and distributions $\mu_1, \ldots, \mu_n$, $\eta_1, \ldots, \eta_n$ such that $\mu_k \sim \eta_k$ for each $k = 1, \ldots, n$ and

  $$\langle o \rangle \mu = \Sigma_{k:=1 \ldots n} c_k \mu_k = \Sigma_{k:=1 \ldots n} c_k \eta_k = \langle o \rangle \eta$$

Any bisimulation between P and Q is also a bisimulation between $P|_o$ and $Q|_o$, and likewise between $\nu c : \tau.\ P$ and $\nu c : \tau.\ Q$. Of special interest are bisimulations where $\mu \sim \eta$ implies $\mu = 1_x$ and $\eta = 1_y$ for some states x and y (denoted $x \sim y$) such that $x|_o = y|_o$ for any output o. It is easy to see that the existence of such a bisimulation between protocols P and Q implies indistinguishablility of P and Q by any adversary of any bound.

Validity and Proof of Soundness We say the judgement $\Delta \vdash P \overset{\delta}{=} Q : I \to O$ is *valid* if for any channel embedding $\theta : \Delta' \to \Delta$ between channel contexts, and any k-bounded adversary,

$$|\Pr[A(\theta\ [[P]]) = 1] - \Pr[A(\theta\ [[Q]])]| \leq \delta(k).$$

Note here that the bound we prove is invariant up to channel embedding. This immediately implies Theorem 1, by applying the identity embedding.

We now sketch the proof of soundness for the equational rules in our logic:

- The [Refl], [Sym], and [Trans] rules are clear.
- The [Exchange] and [Weakening] rules follow at once from the invariance under protocol embeddings.
- The [ReactCong] rule is also clear and [axiom] holds by assumption.
- To prove [CompSym] and [CompAssoc], we define bisimulations by $(s,t) \sim (t,s)$ and $((s,t),u) \sim (s,(t,u))$ respectively.

$$\frac{}{\Gamma \vdash \nu\ \overrightarrow{v}^{n+1} : \tau.\ P = \nu\ \overrightarrow{u}^{n} : \tau.\ \nu x : \tau.\ P[\overrightarrow{u}\ x / \overrightarrow{v}]}\ [\textsc{NuVec-R}]$$

$$\frac{}{\Gamma \vdash \nu\ \overrightarrow{v}^{n+1} : \tau.\ P = \nu x : \tau.\ \nu\ \overrightarrow{u}^{n} : \tau.\ P[x\ \overrightarrow{u} / \overrightarrow{v}]}\ [\textsc{NuVec-L}] \qquad \frac{}{\Gamma \vdash \nu\ \overrightarrow{v}^{0} : \tau.\ P = P}\ [\textsc{NuVec-0}]$$

$$\frac{}{\Gamma \vdash \nu\ \overrightarrow{v}^{n} : \tau.\ \nu\ \overrightarrow{w}^{m} : \sigma.\ P = \Gamma \vdash \nu\ \overrightarrow{w}^{m} : \tau.\ \nu\ \overrightarrow{v}^{n} : \sigma.\ P}\ [\textsc{NuVec-Comm}] \qquad \frac{\Gamma, \overrightarrow{v}^{n} : \tau \vdash P = Q}{\Gamma \vdash \nu\ \overrightarrow{v}^{n} : \tau.\ P = \overrightarrow{v}^{n} : \tau.\ Q}\ [\textsc{Eq-NuVec}]$$

$$\frac{\forall j \in J, \Gamma \vdash P_j = Q_j}{\Gamma \vdash \underset{j\in J}{||} P_j = \underset{j\in J}{||} Q_j}\ [\textsc{EqBig}] \qquad \frac{}{\Gamma \vdash \underset{j\in J}{||}\ P_j = \underset{j\in J\cap K}{||}\ P_j \ ||\ \underset{j\in J\cap \widetilde{K}}{||}\ P_j}\ [\textsc{BigPar-Decomp}] \qquad \frac{}{\Gamma \vdash \underset{j\in \{k\}}{||}\ P_j \ = \ P_k}\ [\textsc{BigPar-1}]$$

$$\frac{}{\Gamma \vdash \underset{j\in\emptyset}{||}\ P_j \ = \ 0}\ [\textsc{BigPar-0}] \qquad \frac{}{\Gamma \vdash (\underset{j\in J}{||}\ P_j)\ ||\ (\underset{j\in J}{||}\ Q_j) = \underset{j\in J}{||}\ (P_j\ ||\ Q_j)}\ [\textsc{BigPar-Par}]$$

$$\frac{\forall i j, i \neq j \Rightarrow v.i \notin P_j}{\Gamma \vdash \nu\ \overrightarrow{v}^{n} : \tau.\ \underset{j<n}{||}\ P_j = \underset{j<n}{||}\ (\nu\ v : \tau.\ P_j[v/v.j])}\ [\textsc{BigPar-Nu}]$$

$$\frac{\forall k < n, \Gamma \vdash (\underset{j<k}{||}\ P_j)||R = (\underset{j<k}{||}\ Q_j)||R \Rightarrow \Gamma \vdash (\underset{j<k}{||}\ P_j)||P_k||R = (\underset{j<k}{||}\ P_j)||Q_k||R}{\Gamma \vdash (\underset{j<n}{||}\ P_j)||R = (\underset{j<n}{||}\ Q_j)||R}\ [\textsc{Hybrid}]$$

Figure 11: Derived rules for parameterized IPDL protocols.

- To prove [AbsorbComp], we define a bisimulation by $(\_, t) \sim t$.
- The rules [CompNew] and [NewExchange] are clear since both sides are interpreted as identical protocols.
- The rules [ResourceTrans], [Subst], and [UnusedResource] follow from the fact that we can choose our final interpretation of both sides to be $[[\cdot]]_1|_{c_1}|_{c_2}$, *i.e.*, prior to any query we attempt to fire c_1 before c_2.
- In the rule [UnFold], we can similarly choose our final interpretation of the body inside the program-level bind on the left-hand side to be $[[\cdot]]_1|_{c_1}|_{c_2}$. This again attempts to fire c_1 before c_2, and this amounts precisely to performing the reaction R_1 inside the reaction-level bind on the right-hand side.
- The rule [NewCong] follows from the fact that any adversary for the protocols on the bottom is also an adversary for the protocols on top.

It remains to prove the rule [CompCong]. We first give the following two constructions on adversaries:

Composition Given an adversary A for a semantic composition of protocols P and Q (not necessarily coming from IPDL), we can compose A with Q to form an adversary for P whose interaction with P yields precisely the same final distribution on booleans as the interaction of the original adversary A with $P\ ||\ Q$. Let $A : I' \to O'$ and $Q : I_2 \to O_2$. Let $\mathsf{d}, \mathsf{a}, \mathsf{s}$ be the decision, accept, and stepping functions of the adversary. The protocol for the new adversary is $A\ ||\ Q$; the schedule is the same as the one for A; the decision function maps a state $(s, \_)$ to $\mathsf{d}(s)$; the accept function for a channel $c \in O_2 \cup O'$ maps a state $(s, \_)$ to $\mathsf{a}_c(s)$; the accept function for a channel $c \in (I_2 \cup O_2)\ (I' \cup O')$ maps any state to $\mathtt{true}$; the step function maps a state (s,t) to $\mathsf{s}_c(s) \times 1_t$.

Restriction Given an adversary A for a protocol $P|_o$ (not necessarily coming from IPDL), we can turn A into an adversary for P whose interaction with P yields precisely the same final distribution on booleans as the interaction of the original adversary A with $P|_o$. Let S be the set of states of A. The new schedule is obtained by scheduling o before and after every channel in the schedule for A. The set of states for the new adversary is $\mathsf{S} + \mathsf{S} + \mathsf{S}$. We now define the rest of the structure:

- The states in the left branch encode the original states of A. All inputs and outputs leave a left-branch state unchanged (and will never be called on a left-branch state). Their decision function is the original decision function for A. They accept all channels for scheduling, even though the structure of the new schedule guarantees that only o is ever scheduled in a left-branch state. The step function for any channel turns a left-branch state into the corresponding middle-branch state.

- The states in the middle branch encode the states of A after performing o on the left. All inputs and outputs leave a middle-branch state unchanged (and will never be called on a middle-branch state). Their decision function maps every state to false (and will never be called on a middle-branch state). Their accept function is the original accept function for A. The step function for any channel is the original step function for A with the proviso that it furthermore turns a middle-branch state into a right-branch state.

- The states in the right branch encode the states of A before performing o on the right. The input and output transition functions are the original ones for A. Their decision function maps every state to false (and will never be called on a right-branch state). They accept all channels for scheduling, even though only o will ever be scheduled in a right-branch state. The step function for any channel turns a right-branch state into the corresponding left-branch state.

Now, we may prove the rule sound. Given a k-bounded adversary A for the protocols $[[P_1||Q]]$ and $[[P_2||Q]]$, we will turn it into an appropriate adversary for $[[P_1]]$ and $[[P_2]]$. First, by Lemma 1, we see that $[[P_1||Q]] = ([[P_1]] \mid\mid [[Q]])|_{o_1,\dots,o_\ell}$ (and similarly for P_2), where ℓ is the number of outputs of P_1 and Q. We then apply the second construction for restriction above ℓ times to receive an equivalent adversary for $[[P_1]] \mid\mid [[Q]]$. Finally, we apply the first construction for composition to receive an adversary A' for $[[P_1]]$. By construction, the behavior of A' interacting with $[[P_1]]$ produces the same final output distribution on booleans as the behavior of A interacting with $[[P_1||Q]]$, and similarly for P_2.

It now remains to estimate the bound for A', as a function of k, the bound for A. Suppose Q is b-bounded. Then, the first construction has a bound of $O(|\Delta| * \mathsf{max}(k,b))$, by estimating the runtime of each transition in the protocol $A \mid\mid [[Q]]$. The second construction has a bound of $O(k)$, since the schedule for the adversary grows by a constant amount, and each transition of the semantic protocol has a runtime of at most $O(k)$. Since we run the second construction at most $|\Delta|$ times (bounding the number of outputs), we have that the adversary A' is bounded by $O(|\Delta| * \mathsf{max}(k,b))$.

C More Details on Case Studies

C.1 OT from Trapdoor Permutations

The ideal functionality for (1-2) OT is given in Figure 12. It is given by a single reaction which simply selects a boolean from the receiver, a pair of messages from the sender, and outputs the appropriate component of the pair. Our definition of ideal OT is parameterized by the type of messages, `L`. (Recall that all IPDL types are in bijection with bitstrings of an appropriate length.) For this simple definition, we eschew the use of ideal parties; instead, if the receiver is corrupted, we simply spawn another copy of the OT functionality with the same inputs, but an output for the adversary. The adversary learns nothing if the sender is corrupted.

The trapdoor OT protocol depends on the security of a *hardcore predicate*, which consists of a family of trapdoor permutations f along with a predicate B such that it is difficult to distinguish the value $B(x)$ from uniform, given only f and $f(x)$ for a uniformly chosen x in the domain of f. While the type system of IPDL does not include general functions (since they take exponential space to describe), we can still model trapdoor permutations by representing f with the following data: an *evaluation key*, a *trapdoor key*, an distribution for generating trapdoor keys, a *derivation function* from trapdoor keys to evaluation keys, and *evaluation functions*, both forwards using the evaluation key, and backwards using the trapdoor key. Only the evaluation and trapdoor keys need to represented as IPDL values: the generation algorithm, derivation function, and evaluation functions can instead be represented as distributions and function symbols in IPDL, respectively. Given this data, we can easily model the hard-core predicate's security as an approximate equivalence between IPDL programs.

In the trapdoor OT protocol, the sender (Alice) sends a randomly chosen trapdoor permutation f to the receiver (Bob), but keeps the inverse of f secret. In return, Bob sends a pair of values, appropriately constructed using uniform randomness and f. Finally, Alice sends her pair of messages back to Bob, appropriately blinded by Bob's message. Assuming Bob constructed his message correctly, and that B is a hard-core predicate for f, this is a secure construction.

In this protocol, and as is common to all of our OT constructions, the adversary learns nothing in the case when Alice is corrupted; thus, we only focus on the case when Bob is corrupted. In this case, the simulator is able to read Bob's index bit and the output of the OT, and must reconstruct Bob's view of the protocol for the adversary. The most subtle part of the proof is that for Bob's view to be simulatable, we cannot reason only about the uniformity of a single bit $B(x)$, but instead of a *pair* of bits $B(x)$ and $B(y)$ (given only f, $f(x)$, and $f(y)$). We thus prove as a lemma that this generalized notion of security for the hard-core predicate follows from the usual one.

```
Definition OTIdeal (L : type) (i : chan TBool)
(m : chan (L ** L)) (o : chan L) :=
 o ::= (
    x_i <-- Read i ;;
    x_m <-- Read m ;;
    Ret (if x_i then x_m.`2 else x_m.`1)).
```

Figure 12: Specification of OT functionality in IPDL.

C.2 1-4 OT from 1-2 OT

While the above two OT protocols only operate over pairs of messages, the GMW protocol in Section 6.1 instead requires an OT protocol which operates over *four* messages, instead of two. This case study mechanizes a construction for 1-4 OT from three instances of 1-2 OTs. In the protocol, the sender blinds their four messages by a combination of *six* random strings, and sends these blinded messages to the receiver. These random strings are additionally given as input to the underlying OTs as messages. The receiver uses their two index bits as index bits for the underlying OTs. The randomness is carefully chosen so that the appropriate randomness only cancels out for the intended message, and all other messages appear uniformly random.

The IPDL proof of the above construction requires a subtle analysis which uses *rerandomization*, or mapping uniform randomness through a bijection. Specifically, we show the following two protocols are (exactly) equivalent in IPDL: the first takes as input a boolean on a channel i, and returns uniformly random values an channels c and d; instead, the second uniformly samples two values x and y, and sets c to be the value if i then x else y, and similarly sets d to be the value if i then y else x. Once the above lemma is established, the proof follows from a number of straightforward channel inlinings.

C.3 Two-Party GMW Protocol

Given the definitions in Section 6.1, the ideal protocol for GMW is straightforward. The ideal parties for Alice and Bob forward their inputs to the functionality, and eventually receive outputs from the functionality. We focus on the case when Alice is corrupt, so she also forwards her inputs and outputs to the simulator. We additionally assume that the simulator learns the *timing* of Bob's inputs (but not their content); this is important for a technical reason, which we will explain below. Given inputs from Alice and Bob along channel vectors $\overrightarrow{u}^A$ and $\overrightarrow{v}^B$, the functionality generates a fresh set of vectors $\overrightarrow{w}^n$ for the circuit wires, runs $\mathsf{evalCirc}(c, \overrightarrow{u}, \overrightarrow{v}, \overrightarrow{w})$, and delivers the circuit outputs to the ideal parties accordingly.

Notably, this definition of the functionality – and thus, also our GMW formalization – encodes *reactive* MPC, in which Alice and Bob can give inputs to the protocol depending on prior outputs. This is possible since our encoding has the feature that the only causal relationships between wires are those imposed by dataflow; thus, if an output wire w_k does not depend on Alice's jth output, Alice is enabled to give the jth input to the protocol after she receives the value for w_k.

The implementation of the GMW protocol is also straightforward, and follows the standard construction: Alice and Bob secret share their inputs, collaboratively compute the circuit over their secret shares, and open their shares for the output wires. To compute the nonlinear AND operation, Alice must use a 1-4 OT protocol to obliviously send Bob a single bit which encodes the XOR of the cross-terms of the two secret shared variables. As described in C.1, we model semi-honest corruption by instrumenting the corrupted party (here, Alice) to leak to the adversary any inputs she receives from Bob, and any randomness she generates during the protocol. Thus, the adversary receives five types of messages from Alice: Alice's randomness generated during the OTs, Alice's protocol input, Alice's secret share for Bob of her input, Alice's share of Bob's input, and Bob's opening of the output wires.

The simulator follows a standard construction in which it evaluates a "blinded" copy of the real protocol in its head, having access to only Alice's private data, but not Bob's. The central step in the proof of security is the construction of an *invariant* between the real world and ideal world with simulator, such that Bob's share of wire w in the real world is equal to the XOR of the true value of wire w in the ideal world with Alice's simulated share, coming from the simulator. By constructing this invariant, we use the [HYBRID] rule to easily reason about the GMW protocol without needing to perform an explicit induction on the circuit.

C.4 Coin Flip

Security for the coin flip protocol is defined as an ideal functionality that: generates a uniform bitstring; leaks it to the simulator; and once the simulator returns with an ok message, broadcasts the bitstring to all ideal parties. All non-corrupted ideal parties then output the same randomness from the functionality.

This functionality is intended to model three main properties: *fairness* (if one honest party receives output, they all do); *agreement* (all honest parties receive the same output); and *uniformity* of the agreed-upon output. However, we do *not* prove privacy of the output bit, or guaranteed delivery.

Unlike the other case studies, we prove this example secure in the *malicious* setting, where the adversary is able to take over the behavior of all corrupted parties. In order to do so in a structured way, we do not allow the adversary to directly control internal protocol channels, but instead give it access to distinguished adversarial channels as proxies. We then, for each corrupted party, write a *shim* which simply forwards messages between the internal protocol channels and those for the adversary (and vice versa).

Our protocol is defined over an arbitrary number of parties and an arbitrary corruption scenario, modeled as a func-

```
1  Definition CoinFunc (K : nat)
2     (* Channels for simulator *)
3     (leak : chan K) (ok : chan TUnit)
4     (* Broadcast channel for ideal party *)
5     (send : chan K) :=
6      b <- new K ;;
7      [||
8          b ::= (Unif K);
9          leak ::= (x <-- Read b ;; Ret x);
10         send ::= (_ <-- Read ok ;;
11                   x <-- Read b ;; Ret x)
12     ].
13
14 Definition CoinIParty (K : nat) {n : nat}
15    (honest : 'I_n -> bool)
16    (i : 'I_n)
17    (send out : chan K) :=
18        if honest i then
19            (out ::= (x <-- Read send ;; Ret x))
20                    else
21            prot0.
22
23 Definition CoinIdeal (K : nat) {n : nat}
24           (honest : 'I_n -> bool)
25           (out : n.-tuple (chan K))
26           (ok : chan TUnit)
27           (leak : chan K) :=
28     send <- new K ;;
29     [||
30         CoinFunc leak ok send;
31         \||_(i < n) CoinIParty honest i send (out ## i)
32     ].
```

Figure 13: Specification of ideal protocol for n-party coin flip in IPDL.

tion `honest : 'I_n -> bool`. However, for simplicity our proof assumes that there are at least two parties such that the first one is corrupted and the last is honest. This is without loss of generality, since the protocol is clearly symmetric, and the security goal is degenerate if all parties are corrupt and immediate if no parties are corrupt.

Since IPDL is channel-centric rather than process-centric, modeling and reasoning about a protocol with n parties and a fixed number of messages is no harder than reasoning about a protocol with a fixed number of parties, and n messages (such as the GMW protocol). Indeed, one of the first simplification steps we take in the proof is to isolate the behaviors among all channels. For a simple example, suppose that we have a protocol where n parties each first send a message x, and then a second message y. Instead of reasoning about the protocol $||_j P_j$, where P_j is the code of the jth party, we instead use the [BIGPAR-PAR] rule from Section 4 to rewrite the protocol as $(||_j x.j ::= r_j) \,||\, (||_j y.j ::= r'.j)$. While a simple observation, this form of rewrite enables a much smoother verification than without.

Encoding of the Ideal Protocol in IPDL The functionality and corresponding ideal protocol is given in IPDL in Figure 13. The functionality is parameterized over three channels: `leak` and `ok`, which are thought of as meant for the simulator, and `send`, which will be used to broadcast a value to all ideal parties. First, on Line 6, it generates a fresh channel `b` carrying a boolean for internal use. It then spawns off three subcomputations: first, on Line 8 we set `b` to be a uniformly random boolean; second, on Line 9, we leak `b` to the simulator, by copying its value to the `leak` channel; finally, on Line 10, we wait for the `ok` message from the simulator, then copy the value of `b` to the `send` channel.

On Lines 14 – 19, we have the ith ideal party, `CoinIParty`. The ideal party is parameterized by the total number of parties `n`, a predicate `honest : 'I_n -> bool` where `'I_n` is the type of natural numbers less than `n` (from ssreflect [25]), the index of the current party, `i : 'I_n`, and two channels, `send` and `out`. If the ith party is honest, then we simply copy the value from `send` to `out` (Line 17); otherwise, we do nothing (Line 19), given by the empty protocol `prot0`.

Finally, on Lines 21-29, we define the ideal protocol, which is composed of the functionality and all n ideal parties. In addition to the `ok` and `leak` channels for the simulator, the protocol is also parameterized by a *n-length vector* of output channels `out : n.-tuple (chan K)`. The protocol generates the internal `send` channel, and first invokes the functionality on Line 27. It then on Line 28 spawns, for each $i < n$, a copy of the ith ideal party, taking input along the `send` channel, and producing output on the ith output channel (written here as `out ## i`.) We make heavy use of the `bigop` library from `ssreflect` to handle n-ary compositions over an index set, as in Line 28. Also, note that while the `send` channel is defined once inside of the functionality, it is able to be read by all n parties; thus, all channels in IPDL naturally support broadcast.

Encoding of the Real Protocol In the real protocol, each party broadcasts a commitment of their randomly chosen bit, receives everyone else's commitments, and then broadcasts an opening of their commitment. We model the commitment by operating in a *hybrid* setting, wherein each party has access to an ideal functionality for performing commitments. This functionality is given below:

```
Definition FComm (K : nat)
     (* inputs from party *)
     (commit : chan K) (open : chan TUnit)
     (* outputs to broadcast *)
     (committed : chan TUnit) (opened : chan K) :=
[||
    committed ::= (_ <-- Read commit ;; Ret tt);
    opened ::= (x <-- Read commit ;;
                _ <-- Read open ;; Ret x)
].
```

The commitment functionality is parameterized by input channels `commit` and `open`, which are to be used by the party the functionality is meant for, and output channels `committed` and `opened`, which will be broadcast to all. On Line 7, the channel `committed` is set to wait for `commit` before firing. On Line 8, the channel `opened` is set to the value of `commit`, but only after the channel `opened` has fired.

We now turn to the actual protocol, which is given in Figure 14. Similar to the ideal protocol, we model malicious corruption by splitting the party's code into two parts: one for if the party is honest, and one if the party is corrupted.

We first describe the honest party, given on Lines 25-36. We note that the party is parameterized by all channels appearing at the top of the Section, on Lines 15-21. These include the inputs from all broadcast commitments and openings, and the outputs from the party itself, both for its own commitment as well as its protocol output. First on Lines 25-26 we generate two fresh *vectors* of channels, `committed_sum` and `opened_sum`, which will be used for aggregation of multiple values. The first parameter to `newvec`, `n`, is the length of the channel vector, while the second parameter is the type of the channels. The party first commits to a uniformly chosen input, as given on Line 28. On Lines 29-32, the party then computes a *fold* over the signals coming from the channels in `committed`: since each channel in this vector carries a unit value, we are merely accumulating *timing dependencies* into the channels in `committed_sum`. On Line 33, the party then opens their commitment, based on the timing of the last channel in `committed_sum`. In effect, this causes the party to wait for all commitments to happen before the party opens theirs. Line 34 similarly folds the channels in `opened` together into `opened_sum`, so that the last channel in `opened_sum` carries the collective XOR of all opened commitments. The party outputs this value on Line 35.

To encode the corrupted party on Lines 39-46, for convenience we define a *shim* for the corrupted party, which acts to separate the adversary's channels from the internal protocol channels. The adversary's channels, defined on Lines 4-9, are divided into inputs and outputs. The inputs from the adversary are `advCommit` and `advOpen`, which allow the adversary to control the *i*th party's `commit` and `open` messages (if the *i*th party is corrupt.) This is reflected in Lines 40 and 41 in the corrupted party, which copy the *i*th channel of `advCommit` to the corrupted party's `commit` channel, and similarly for `open`. The outputs to the adversary are `advCommitted` and `advOpened`, which are both *tuples of tuples* of channels. On Line 42, the *i*th component of `advCommitted` is set equal (pointwise) to the *i*th party's view of the `committed` tuple of input message. Similarly, on Line 44 the *i*th component of `advOpened` is set to the *i*th party's view of `opened`. Finally, to define the party we again case split on whether party *i* is honest, and choose the corresponding implementation.

Finally, we now define the real protocol in total in Lines 55-68. We first generate all internal channel vectors for the commitment functionalities, and then spawn all *n* commitment functionalities and *n* parties.

```
 1  Context (K : nat) {n} (honest : 'I_n -> bool)
 2          (* inputs from adversary *)
 3          (advCommit : n.-tuple (chan K))
 4          (advOpen : n.-tuple (chan TUnit))
 5          (* outputs to adversary *)
 6          (advCommitted : n.-tuple (n.-tuple (chan K)))
 7          (advOpened : n.-tuple (n.-tuple (chan K)))
 8          (* output channels of protocol *)
 9          (out : n.-tuple (chan K)).
10
11  Section CoinRealParty.
12    Context {n} (i : 'I_n)
13            (* inputs to party *)
14            (committed : n.-tuple (chan TUnit))
15            (opened : n.-tuple (chan K))
16            (* outputs from party *)
17            (commit : chan K)
18            (open : chan TUnit)
19            (partyOut : chan K).
20
21    Definition CoinRealParty_honest
22      :=
23        committed_sum <- newvec n @ TUnit ;;
24        opened_sum <- newvec n @ K ;;
25        [||
26              commit ::= (Unif K);
27              cfold committed
28                  (fun _ _ => tt)
29                  (fun _ => tt)
30                  committed_sum;
31              open ::= (_ <-- Read (committed_sum ## ord_max);;
32                        Ret tt);
33              cfold opened xort id opened_sum;
34              partyOut ::= (Read (opened_sum ## ord_max))
35        ].
36
37    Definition CoinRealParty_corr
38      [||
39              commit ::= (Read (advCommit ## i));
40              open ::= ((advOpen ## i));
41              \||_(j < n) (advCommitted ## i ## j) ::=
42                            (Read (committed ## j))
43              \||_(j < n) (advOpened ## i ## j) ::=
44                            (Read (opened ## j))
45      ].
46
47    Definition CoinParty
48      if honest i then CoinRealParty_honest
49                  else CoinRealParty_corr.
50  End CoinRealParty.
51
52  Definition CoinReal :=
53      commit <- newvec n @ K ;;
54      committed <- newvec n @ TUnit ;;
55      open <- newvec n @ TUnit ;;
56      opened <- newvec n @ K ;;
57      [||
58          \||_(i < n)
59              FComm (commit ## i)
60                    (committed ## i)
61                    (open ## i)
62                    (opened ## i);
63          \||_(i < n) CoinParty
64                        i committed opened
65                        (commit ## i) (open ## i) (out ## i)
66      ].
```

Figure 14: Real protocol for *n*-party coin flip in IPDL.

Simulator and Proof Sketch To show that `CoinReal` realizes `CoinIdeal`, we must show the existence of a simulator `CoinSim` which transforms the adversarial channels of `CoinReal` into those of `CoinIdeal`.

Since the security condition is degenerate in the case when all parties are corrupted or all are honest, we focus without loss of generality on the case where the first party is corrupted, and the last party is honest. Intuitively, the simulator runs a copy of the real world protocol "in its head", but modified in the following way: the last party, instead of generating a commitment uniformly, generates its commitment by reading the commitments of all other parties (honest or not), and XORing all other commitments together, along with the value along the channel `leak` from the ideal world. This ensures that the bit that all the parties inside the simulated real world all agree to the same value as is chosen by the functionality. In turn, when all simulated parties open their commitments, the simulator then outputs `ok` to the functionality. Since all commitments by honest players appear uniform, and the simulator only submits `ok` after all corrupted players open their commitments, it follows that the adversary's view in the real and ideal worlds are identical, and all honest party's behavior is identical as well.