OWL: Compositional Verification of Security Protocols via an Information-Flow Type System

Joshua Gancher Sydney Gibson Pratap Singh Samvid Dharanikota Bryan Parno
Carnegie Mellon University

Abstract

Computationally sound protocol verification tools promise to deliver full-strength cryptographic proofs for security protocols. Unfortunately, current tools lack either modularity or automation. We propose a new approach based on a novel use of information flow and refinement types for sound cryptographic proofs. Our framework, OWL, allows type-based *modular* descriptions of security protocols, wherein disjoint subprotocols can be programmed and automatically proved secure separately.

We give a formal security proof for OWL via a core language which supports standard symmetric and asymmetric primitives, Diffie-Hellman operations, and hashing via random oracles. We also implement a type checker for OWL along with a prototype extraction mechanism to Rust, and evaluate it on 14 case studies, including (simplified forms of) SSH key exchange and Kerberos.

1 Introduction

Cryptographic protocols (e.g., Kerberos, TLS, QUIC, or Signal) are widely deployed and yet frighteningly brittle. Their ubiquity means that when a core component like TLS breaks, the negative effects are pervasive, as illustrated by headline-grabbing attacks like FREAK [17] and Logjam [2].

Formal protocol verification can, in theory, rule out such attacks, and indeed, the academic community has developed a host of tools for this purpose [10, 24, 32]. However, existing methods for computer-aided security proofs struggle to handle the complexity of (and the threats faced by) modern protocols like TLS or QUIC. We posit (§2) that this is because no tool has simultaneously achieved *modularity*, *automation*, and *computational security*.

Modular Verification. Traditional on-paper cryptographic proofs that use game-hopping or simulator-based techniques typically consider an entire protocol at once. Possibly as a consequence, many tools for automating cryptographic proofs also employ monolithic reasoning [7, 23, 25, 57], meaning that security proofs for protocols cannot be constructed out of smaller proofs for subprotocols. Unfortunately, this approach does not scale with the complexity of modern protocols, as the effort required to use such tools to verify P composed with Q is generally much larger than the effort to verify P and Q separately. Furthermore, even when successful, monolithic verification efforts harm proof reuse and understandability, preventing subsequent verification efforts from benefiting from previous ones.

Proof Automation. On the other hand, tools that do support modular verification of protocols [11, 30, 55] provide so much generality that they can be difficult to automate, requiring an expert in both cryptography and verification to manually write the various cryptographic simulators needed to perform cryptographic reduction steps for the proof. While potentially feasible for simplified on-paper models, realistic protocols feature many moving parts – not all of them cryptographic in nature – and thus require automation to make verification feasible.

Indeed, security protocols such as TLS are *not* designed with cryptographic reduction steps in mind; instead, they are designed to protect confidentiality and integrity *assuming* that the underlying cryptographic mechanisms are secure. Verification tools should match this intuition and hide low-level cryptographic arguments from the user when possible.

Computational Security. Ideally, cryptographic protocols should be verified in the "gold standard" computational model (which is used ubiquitously in on-paper cryptography), since such a model makes the weakest assumptions about possible adversaries, and hence provides the strongest guarantees. Unfortunately, this focus on expressiveness and fidelity typically hinders automation and scalability [10].

Hence, to increase automation, considerable work verifies protocols in the presence of *symbolic* attackers [15, 18, 19, 34, 51]. These results give some assurance, but pen-and-paper proofs are prone to subtle errors [49], and simplified attacker models may abstract away key facets of a protocol's design that leave it vulnerable to attack. Indeed, "verified" versions of TLS [51] and SSH [15] have been successfully attacked [3, 61], due to limitations in the models the proofs employed.

Our Approach. We introduce OWL, the first language and formal tool to achieve automated, modular proofs of protocols in the computational model. OWL's modularity means that higher-level protocols (e.g., for creating and using a secure channel) may safely *assume* secure implementations of lower-level specifications (e.g., for key exchange), which can later be instantiated in different ways (e.g., via pre-shared keys or a PKI). Protocols written in OWL are verified in the computational model, which treats all values as bit strings, all cryptographic primitives as algorithms over them, and attackers as *arbitrary* polynomial-time probabilistic algorithms.

Automating computational reasoning is challenging, since computational hardness assumptions – such as IND-CPA and INT-CTXT for authenticated encryption, and EUF-CMA for digital signatures – are usually expressed as pairs of security games. This representation requires a *reduction* from the protocol at hand to (an instantiation of) the security game. Since

security games are typically specified as pairs of general probabilistic programs, reductions cannot easily be automated. The result is laborious security proofs consisting of multiple hand-written cryptographic games, as seen in computational verification frameworks such as CertiCrypt [12] and EasyCrypt [13].

The core insight underlying OWL is that for a wide class of hardness assumptions (including those mentioned above), we can automate reductions to security games by restricting the protocol's allowed *dataflows*. Once these dataflow restrictions are met, OWL's type system guarantees the existence of cryptographic reductions via a *once and for all* proof effort which, crucially, users of OWL do *not* need to understand to conduct proofs of protocol security.

OWL enforces these dataflow restrictions via a novel information-flow control (IFC) [65] type system augmented with refinement types [46]. Specifically, in OWL, security goals are expressed as either *secrecy* or *authenticity* properties. Secrecy properties are proven through information-flow labels. For example, if a message `c` has label `adv`, then the soundness of OWL's type system guarantees that any probabilistic polynomial-time adversary's view of `c` can be *simulated* using information the adversary already knows, even if `c` is derived from secret data using cryptographic operations (e.g., even if `c` is a ciphertext computed from a secret key).

Integrity properties are stated through *safety properties*, or refinements attached to data in the protocol. For example, authenticity for an ideal secure channel guarantees that if an honest party receives a message from the channel, then an honest sender must have previously sent that message.

OWL's typing rules are proven sound (on paper) once-and-for-all based on standard cryptographic assumptions. They can then be applied to automatically type-check arbitrary protocols. As our OWL prototype shows[1], we can perform this type checking in seconds, enabling the user to iteratively fix their protocol, guided by typing errors from the tool.

OWL currently supports a variety of cryptographic primitives, including MACs, public-key signatures, hash functions, (authenticated) symmetric and public key encryption, and Diffie-Hellman key exchange [38]. This collection has sufficed to verify the rich collection of case studies described below. OWL can be modularly extended with additional primitives by adding appropriate typing rules and proving them sound against the corresponding cryptographic security definitions.

To evaluate the expressiveness and usability of OWL, we implement, verify, and extract 14 case studies (§6) from a variety of domains, including classics like Needham-Schroeder [59], various RFID protocols, and (simplified forms of) SSH key exchange and Kerberos. This collection covers all of the case studies from two state-of-the-art tools [7, 23], allowing us to analyze how verification using OWL differs.

OWL is useful not only for abstract security proofs, but is designed with realistic implementations of protocols in mind. The protocol language in OWL introduces a number of features which both aid in security analysis and guide the automatic extraction of executable implementations. We demonstrate this by developing an automatic extraction mechanism to Rust, generating prototype implementations of protocols by a relatively direct translation of the OWL source code.

However, like all verification, OWL's guarantees rely on the correctness of our tools (OWL and its SMT solver, Z3 [35]), and on the developer to correctly write down the security specification they desire. Further, the guarantees OWL provides are asymptotic in the security parameter, rather than concrete bounds. OWL currently only supports static corruptions, so the adversary cannot spontaneously corrupt a party in mid-protocol exchange. Finally, to provide strong automation, OWL's type system deliberately overapproximates possible information flows, so it provides less generality than tools that support manual proofs of arbitrary probabilistic programs. Fortunately, our case studies suggest that OWL is still expressive enough to capture a wide variety of protocols and properties.

Contributions. In summary, this paper contributes:

1. The first protocol verification tool, OWL, for delivering *modular*, *automated* proofs of computational security;
2. A novel use of information flow and refinement types to enforce cryptographic protocol security; and
3. A number of case studies in OWL, exercising a wide range of language features and cryptographic primitives, and which extract to executable code.

2 Motivation and Related Work

The rich history of work on verifying cryptographic protocols has prompted multiple surveys, both of early foundational results [24, 32] and of quite recent work [10]. However, this community effort has not yet produced a tool that simultaneously supports automation, modularity, and computational security guarantees. Below, we restrict our discussion to recent work that targets a subset of these goals, deferring to the surveys for a broader perspective.

Symbolic Models. Most mechanized analyses of large-scale security protocols [18, 19, 31, 33, 34, 45, 48] are done in the *symbolic* model [39], which typically aids automation but overly constrains adversaries, relies on monolithic reasoning, and sometimes still requires manual intervention.

In the symbolic model, all values (e.g., keys or nonces) are represented as symbolic *terms*: these terms are considered atomic, meaning that they cannot be split into smaller pieces (the way real-world bit strings could be). Cryptographic primitives are modeled via equational theories that describe how an adversary can manipulate the relevant terms; e.g., symmetric encryption might have a rule that says $\texttt{Dec}(\texttt{Enc}(m,k),k) = m$, which says that decrypting the ciphertext produced by encryption results in the original message. Automated tools (e.g., Tamarin [57] or ProVerif [25]) can then apply these rules to exhaustively determine what an adversary might learn from the protocol. However, such automation comes at the cost of requiring the security researcher to enumerate all computations the adversary may perform. Failing to provide sufficiently rich rules can unrealistically constrain the modeled adversary, leading to "verified" protocols [15, 51] succumbing to attacks that exploit adversary capabilities not captured by the model. Unfor-

[1] https://github.com/secure-foundations/owl

Tool		RF	Auto	Modular	CB	Link	TCB
CertiCrypt	[12]	◑	○	○	●	●	Coq
CryptHOL	[55]	◑	○	●	●	○	Isabelle
EasyCrypt	[13]	◑	○	●	●	●	self, SMT
FCF	[62]	◑	○	●	●	●	Coq
F*	[66]	◐	○	●	○	●	self, SMT
CryptoVerif	[23]	◐	●	○	●	●	self
Squirrel	[7]	◑	○	○	○	○	self
OWL		◐	●	●	○	●	self, SMT

Reasoning Focus (RF)
◐ – automation focus
◑ – expressiveness focus

Concrete Bounds (CB)
● – Yes
○ – No

Figure 1: **Comparison With Other Computational Tools**. *Unlike prior computational tools,* OWL *supports* both *automation* and *modular reasoning. It provides a Link to executable code, but it only proves asymptotic, rather than concrete, security bounds.*

tunately, adding rules that enhance the adversary's capabilities (sometimes even something as simple as specifying that XOR is associative and commutative) can overwhelm the automation.

Most symbolic tools also employ monolithic reasoning, meaning that they must consider the entire protocol as a whole, rather than reasoning about the protocol's components in isolation and then composing the results. This hinders proof reuse, and it leads to challenges with scale.

Although largely automated, some symbolic tools (e.g., Tamarin [57]) still rely on developers to manually add lemmas to help partition the search space.

Computational Models. Unlike the symbolic world, computational models (which are used ubiquitously in on-paper cryptography) treat all values as bit strings, and cryptographic primitives are algorithms over them. Computational security properties are probabilistic and reason about the behavior of adversaries represented as Turing machines. Because keys are bit strings, an adversary can potentially learn some (but not all) of a key's bits and hence reduce the resources needed to recover an encrypted ciphertext. All of this additional detail and complexity provides greater confidence that positive verification results [19, 48, 54] imply real-world security, but they typically hinder automation and scalability [10].

Since OWL falls into this category, we compare it to other such tools in Figure 1, using the comparison criteria from a recent SoK [10], and in more detail below.

Traditional verifiers for computationally secure cryptography include FCF [62], CryptHOL [55], and EasyCrypt [13]. These tools work in roughly the same way, analyzing probabilistic programs interactively using higher-order logic. Such expressivity comes at the cost of a higher proof burden, requiring verification experts to manually construct cryptographic reductions to prove protocols secure. It would be an interesting challenge to use one of these expressive tools to formally prove OWL sound.

In contrast, CryptoVerif [23] performs computational proofs by (semi-)automatically finding reductions from the protocol to security games. Implemented as a stand-alone language, CryptoVerif does not support modular protocol analyses, forcing the entire protocol to be analyzed in one monolithic proof effort.

A long line of work, beginning with Abadi and Rogaway [1], attempts to combine the automation of the symbolic model with the guarantees of the computational model. A promising proposal in this space is the *computationally complete symbolic attacker* (CCSA [8, 9]) framework, which blends symbolic and computational techniques via proof steps inspired by on-paper cryptographic security definitions. The most advanced instantiation of this framework is the recent Squirrel prover [7], which can express many proofs in the CCSA model. However, CCSA (and hence Squirrel) makes extensive use of non-composable *syntactic side-conditions*, which require whole-program analysis. Squirrel also requires non-trivial manual proof help from the developer, e.g., to write out both real and ideal versions of a protocol and to interactively guide the proof via tactics.

Additionally, Squirrel and CryptoVerif do not directly support corruption models, instead requiring the developer to encode corruption indirectly through process functions.

Cryptographic Information Flow. There is some work on giving information-flow types to cryptographic mechanisms in a computationally sound way [6, 43, 52]. However, these works generally do not support the wide array of mechanisms found in security protocols (e.g., digital signatures, key derivation functions, and Diffie-Hellman-based key exchange). In contrast, we use information-flow types to capture *cryptographic dependencies*, instead of only program dependencies, for a wide variety of cryptographic mechanisms. Our alternative view on information flow allows a more straightforward, extensible proof strategy for cryptographic security.

Abstraction-Based Analyses. Another line of work [20, 21, 36, 42] proposes to verify protocols through iterative instantiation of modules with abstract interfaces in a dependently typed language, and has been applied to parts of TLS [20, 21] and QUIC [36]. While promising, this work hinges on complex metatheoretical properties of the proof assistant [42], such as parametricity properties of abstract types, and it requires subtle on-paper analysis to relate trusted models of cryptography in the proof assistant to on-paper cryptographic security guarantees.

3 OWL Overview

OWL enables a developer to write readable, high-level protocols whose security guarantees are given by the types used to define the protocol. As in other tools [7, 23, 25], we analyze security in the setting where the adversary completely controls the network, and is thus able to arbitrarily reroute and modify in-flight messages. OWL is *computationally sound*, meaning that protocol parties operate directly on bitstrings, and the adversary is an *arbitrary* polynomial time algorithm. As is common for security protocols [22], we analyze security in the static security setting: before protocol execution, the adversary chooses a set of information which it corrupts.

3.1 Running Example: Secure Transport

We demonstrate the use of OWL through this example of securely sending a secret message msg_i from Client_i to a Server.

$$\text{Server} \xrightarrow{\{\text{kdata}_i\}_{\text{psk}_i}} \text{Client}_i$$
$$\text{Server} \xleftarrow{\{\text{msg}_i\}_{\text{kdata}_i}} \text{Client}_i$$

The Server starts with an existing pre-shared key (psk_i) for each Client_i. When the Server wishes to receive the message from

```
1 locality Server
2 locality Client⟨i⟩
3 name msg⟨i⟩ : nonce @ Client⟨i⟩
4 name kdata⟨i⟩ : enckey (Name(msg⟨i⟩)) @ Server
5 name psk⟨i⟩ : enckey (Name(kdata⟨i⟩)) @ Server, Client⟨i⟩
```

Figure 2: **Name declarations for Secure Transport in OWL.**

Client_i, it generates an ephemeral key kdata_i, encrypts it using symmetric encryption under psk_i, and sends the ciphertext over the network. In return, Client_i encrypts msg_i under kdata_i, and sends the ciphertext back to the Server. Even for this simple protocol, a developer would reasonably want a variety of security properties:

- **Secrecy**: Any attacker who listens in on the network cannot learn the value of any secrets unless the corresponding party has been compromised. Additionally, if Client_i is compromised, then the security of Client_j is unharmed.
- **Correctness**: Assuming the encryption scheme is authenticated, unless the Server is corrupt, Client_i obtains the correct key kdata_i; similarly, unless Client_i is corrupt, the Server should obtain the correct secret message msg_i.
- **Authentication**: If the Server obtains msg_i from Client_i, then Client_i must have sent the corresponding ciphertext.

3.2 Secure Transport in OWL

Unlike existing tools (which try to verify arbitrary, possibly insecure protocols via model checking [25], direct security game transformations [23], or user-guided proof rules [7]), OWL follows the *intrinsic verification* paradigm: protocols are strongly typed to prove security properties.

Protocol Invariants through Names. In our example, to prove that ciphertexts successfully decrypted under psk_i always return kdata_i, we do *not* exhaustively check where psk_i may be used [7, 23, 25, 57], but instead attach an invariant to psk_i which guarantees this property by construction.

Cryptographic keys (along with other pieces of random data, such as msg_i) are defined by *names*, or abstract handles to randomness which may be used by the parties. Names may not be used arbitrarily, but are attached to invariants, or *name types*, which specify how they may be used.

The name declarations for our running example are given in Figure 2. Lines 1-2 define the *localities* for the protocol. Localities represent the set of parties, and are used to guide extraction of protocols to implementations (§5.2). Localities may be *indexed*, as in Client⟨i⟩, to define a large, symmetric set of parties.

Lines 3-5 declare the names for the protocol. Since our protocol has three logical pieces of randomness—msg, kdata, and psk, for each *i*—we have three corresponding name declarations, each of which is also indexed. All names in OWL are annotated with name types, specifying invariants which must hold throughout the protocol, and associated localities, specifying where the name is initially stored or generated.

We treat the message msg⟨i⟩ from Client⟨i⟩ as opaque data, so we give it the name type nonce on Line 3. On Line 4, we have the key kdata⟨i⟩ generated by the Server. The protocol specifies that kdata⟨i⟩ encrypts exactly msg⟨i⟩, so we give it the name type

```
1  enum Result⟨i⟩ {
2      | Ok Name(msg⟨i⟩)
3      | Err
4  }
5
6  def tr_server⟨i⟩() @ Server :
7      if sec(kdata⟨i⟩) then Result⟨i⟩ else Data⟨adv⟩ =
8    let c = samp enc(get(psk⟨i⟩), get(kdata⟨i⟩)) in
9    output c to endpoint(Client⟨i⟩);
10   input inp,_ in
11   corr_case kdata⟨i⟩ in
12   case dec(get(kdata⟨i⟩), inp)
13   | Some o ⟹ Ok⟨i⟩(o)
14   | None ⟹ Err⟨i⟩()
```

Figure 3: **Definition of the Server in OWL.**

enckey (Name(msg⟨i⟩)). This name type enforces that, unless the adversary has corrupted the Server or Client⟨i⟩, any honest encryption using the key kdata⟨i⟩ must contain exactly msg⟨i⟩. We enforce this using a *singleton type* Name(n) which only contains the value of name n. The declaration of psk⟨i⟩ on Line 5 is similar, but is annotated with the two localities Server and Client⟨i⟩, reflecting that it is assumed to be pre-shared.

Declaring cryptographic nonces and keys through names in OWL has a number of advantages. First, it requires the protocol designer to explicitly establish the invariants which keys must protect, instead of leaving these invariants to ad-hoc analyses. It also enforces a *hierarchy* among keys, which we will exploit in our type system for providing information-flow guarantees. Additionally, our formal tool OWL *typechecks* these invariants, to ensure that they will guarantee secure protocol instantiations. For example, name types for encryption keys rule out key cycles by fiat, since encryption keys may only encrypt data that has been previously declared.

Party Code in OWL. For this section, we focus on the code of the Server in Figure 3. Lines 1-4 specify a *datatype*, which OWL uses to build data structures. OWL natively supports enums and structs. The datatype Result⟨i⟩ is itself indexed, as it either contains the value of msg⟨i⟩ (if Ok) or nothing (if Err).

Lines 6-14 declare the code for the Server itself, through a *definition*. The definition tr_server is parameterized by i, as it specifies the code to interact with Client⟨i⟩. Ignoring the type annotations in Lines 7 and 11, the code for tr_server is straightforward. In Line 8, we obtain the values of psk⟨i⟩ and kdata⟨i⟩, encrypt the latter under the former, and output the ciphertext in Line 9. In Line 10, we obtain the input inp, which we decrypt using kdata⟨i⟩ in Line 12. Network outputs and inputs happen through the adversary. Decryption returns an option type, so we pattern match on the result, and either return Ok with the plaintext if decryption succeeds, or Err otherwise.

Name-Based Corruption. A key insight of OWL is that the traditional paradigm of specifying adversarial corruptions by party is too coarse for a formal tool. Instead, OWL specifies corruptions by *name*. We express this corruption model through *information flow labels* [65].

In OWL, labels are either atomic labels [n] where n is a name (e.g., msg⟨i⟩), or the conjunction of two labels $\ell_1 \wedge \ell_2$. Labels

support a *flows-to* predicate $\ell_1 \leq \ell_2$, which specifies that the names captured by ℓ_2 are a superset of those captured by ℓ_1.

The adversary is specified by a label adv, which for our example, is some conjunction of the atomic labels [msg⟨i⟩], [kdata⟨i⟩], or [psk⟨i⟩] for any i. A name n is considered *corrupt* when n flows to the adversary label adv.

Corruption in OWL is *hierarchical*: if an encryption key k associated to the nametype enckey t is corrupt, then all information present in t is corrupt. For example, psk⟨i⟩ being corrupt implies kdata⟨i⟩ is corrupt, since the adversary can use psk⟨i⟩ to decrypt in-flight ciphertexts to obtain kdata⟨i⟩. Transitively, corrupting psk⟨i⟩ also implies corrupting msg⟨i⟩.

Casing on the Adversary. Recall from §3.1 that integrity for our protocol means that the Server obtains the correct value msg⟨i⟩ unless the other party is corrupt. We refine this to say that the Server gets the desired data unless *the name* kdata⟨i⟩ is corrupt. This security goal is reflected in the type annotation for the Server given in Line 7. If kdata⟨i⟩ is secret, then the Server obtains a value of type Result⟨i⟩, which is guaranteed to hold msg⟨i⟩ if it does not return Err. On the other hand, if kdata⟨i⟩ is corrupt, then all guarantees are lost for correctness; this is reflected in the Data⟨adv⟩ type, which represents arbitrary adversary-controlled data.

The type checker for OWL needs to consider both cases of whether kdata⟨i⟩ is corrupt separately. To do so, we insert a corr_case command in Line 11, which splits the type checking into the two corresponding cases (see §5 for details).

Secrecy through Label Checking. Secrecy for our protocol means, among other things, that no data is leaked when ciphertexts are output on the network. OWL guarantees secrecy by using information flow types to ensure that all flows to/from the adversary are valid and typed with label adv. Intuitively, data with label adv depends on information deducible from the adversary before protocol execution, using only the names that it has already statically corrupted. Thus, by ensuring that all outputs have label adv, we guarantee that all outputs to the network do not contain any more information than the adversary already knows. In particular, if we assume the adversary begins with the trivial label $\bot$ (meaning no corruptions), then we are guaranteed by construction that the adversary learns *no* computational information about any protocol secrets.

All types in OWL carry secrecy information via labels. For example, the type Name(n) has exactly the label [n], while the type Result⟨i⟩ has the label adv∧[msg⟨i⟩], since the choice of whether the value is Ok or Err must have label adv, while the data present in the Ok case has label [msg⟨i⟩].

Temporal Properties. In addition to secrecy and integrity, the running example carries an *authentication* property: if the Server receives the message msg⟨i⟩, Client⟨i⟩ must have sent it. Intuitively, this holds because only Client⟨i⟩ encrypts messages using the key kdata⟨i⟩. In OWL, we may encode this authentication property by *refining* the type associated to the name type for kdata⟨i⟩ from simply Name(msg⟨i⟩) to the *refinement type* (x:Name(msg⟨i⟩){happened(tr_client⟨i⟩())}).

To construct a value of this refinement type, the OWL type checker needs to check the refinement that the tr_client⟨i⟩ definition has been called. In turn, any code which inspects a value of this refinement type (e.g., when the Server decrypts data under kdata⟨i⟩) learns that this definition has been called. Our encoding of temporal properties using definition calls is similar in spirit to event predicates in CryptoVerif [23], but is tied closer to the protocol, since the user does not need to add extra events to the protocol to encode temporal predicates.

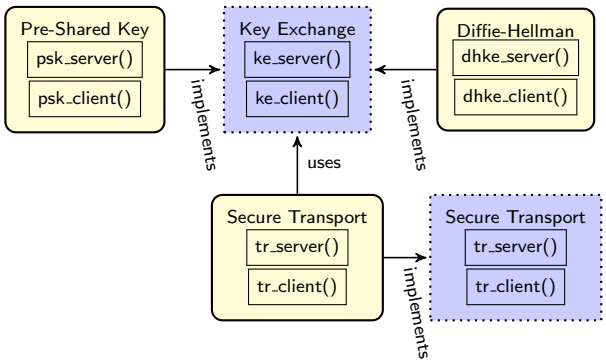

Figure 4: **Modular Verification of Secure Transport**. *Yellow, solid-border boxes denote module implementations; blue, dotted-border boxes denote module types. The inner boxes denote each locality's routines in the corresponding module. All modules—and all routines within each module—are type checked independently.*

Modular Specifications. A unique feature of OWL, not shared by whole-protocol analysis tools [7, 23, 25, 57], is that the code of individual parties may be type checked and proved secure *separately*. The security of tr_server does *not* depend on the security of the implementation of tr_client⟨i⟩, but instead *only on tr_client⟨i⟩ being well-typed*. Modular type checking of protocols has a number of benefits, including reduced proof effort, an interactive verification experience via type errors, and the ability to reason abstractly about the code.

Crucially, OWL also supports modular specifications of protocols themselves. Figure 4 shows a decomposition of the Secure Transport protocol into several components, each of which can be verified separately in OWL. We provide an abstract specification for Secure Transport by giving a *module type*, which specifies the high-level security properties expected of the protocol without any implementation details. The implementations of these routines are provided by the implementation of the Secure Transport module, which is typechecked against the module type.

Secure Transport itself relies on a Key Exchange subprotocol, consisting of routines ke_server() and ke_client() that securely send the key kdata to the client. As shown in Figure 4, Key Exchange can be implemented via a pre-shared key psk (as in §3.1), or it can be implemented via a Diffie-Hellman exchange that use a public-key infrastructure and digital signatures for integrity verification. Notably, verification of the Secure Transport module *does not* depend on which Key Exchange scheme is used, relying instead only on that scheme being well-typed against the Key Exchange module type. In turn, OWL developers could *assume* a Secure Transport protocol using the Secure Transport module type, and construct new protocols on top of

```
1 locality Server
2 locality Client⟨i⟩
3 name msg⟨i⟩ : nonce @ Client⟨i⟩
4
5 def tr_server⟨i⟩() @ Server :
6       if sec(msg⟨i⟩) then Result⟨i⟩ else Data⟨adv⟩
7
8 def tr_client⟨i⟩() @ Client⟨i⟩ : Unit
```

Figure 5: **Module Type for Abstract Secure Transport**.

it.

Figure 5 shows the details of the module type for Secure Transport in OWL, consisting of the type signatures of the definitions tr_server and tr_client. The locality definitions are the same, while the only name definition is the one for msg⟨i⟩; indeed, the keys kdata⟨i⟩ and psk⟨i⟩ only serve to *implement* secure transport of msg⟨i⟩, and should not be part of the specification.

The specification of tr_server on Line 5 is similar to the declaration in Figure 3, but is instead typed with the *supertype* if sec(msg⟨i⟩) then Result⟨i⟩else Data⟨adv⟩. Indeed, through hierarchical name corruption, if msg⟨i⟩ is secret, then kdata⟨i⟩ must also be. This specification matches our intuitive one in §3.1 for secrecy and integrity: for secrecy, if nobody is corrupted, then type soundness of the language guarantees that the adversary does not learn any secrets; for integrity, we have that the Server obtains the type Result⟨i⟩ whenever Client⟨i⟩ is uncorrupt (i.e., all names at that locality are secret).

In the secure transport example, the role of Client⟨i⟩ is to respond to the Server, and encrypt its secret message under the decrypted key. Thus, the type of tr_client⟨i⟩ in Figure 5 simply returns Unit. This type gives no nontrivial guarantees about the output, but the internal code of tr_client⟨i⟩ still guarantees security and integrity throughout since it is well-typed.

3.3 Implementation-Oriented Design

Unlike pure protocol verification tools such as Squirrel [7], OWL is designed to enable *both* verification of protocol designs *and* extraction of protocol implementations. As such, we designed OWL's surface language to support fully automated extraction of executable code, as described in §5.2. While our extraction pipeline currently produces un-optimized code, we believe it should be possible to extend this pipeline to produce performant, interoperable protocol implementations without significant changes to OWL's type system or core language.

Session and Party IDs. Unlike other tools [7], indices in OWL serve a purpose beyond considering protocols with large number of names. Indices in OWL come in two varieties: *session IDs*, used for multiple invocations of the same logical party, and *party IDs*, used for naming many symmetric parties (e.g., Client⟨i⟩). The running example in §3.2 uses party IDs throughout, but it can be augmented to use session IDs as well; our case studies in §6 make pervasive use of both in protocols. The distinction between session IDs and party IDs is crucial for extraction, which we describe in §5.2.

Endpoints. While simple input i and output o commands suffice for security analysis, extracted code needs to reason about concrete *endpoints*, or destination addresses for concrete network I/O. To do so, we support endpoints in OWL; input i, e binds an endpoint e coming from listening on a network port, while output o to e submits an output to endpoint e. As discussed in §5.2, endpoints are similarly crucial for realistic extractions.

3.4 Limitations

While our work is the first to use a type system to deliver modular computational soundness for security protocols, we currently have a small number of limitations to address in future work. First, OWL only supports static corruptions via the adversary label; this is inherited from our use of simulation-based security in Section 4. Indeed, commitments and encryptions are known to interact with simulation-based security in subtle ways [29]. However, OWL *can* encode forward secrecy properties through its hierarchical name model: if k_i encrypts k_{i+1}, and n is the lowest index such that k_n is corrupt, then we guarantee secrecy for all k_j for $j < n$. Essentially, we model forward secrecy by having the adversary commit to the point in time where compromise happens.

Second, while OWL supports secrecy and integrity properties, OWL currently does not support unlinkability properties [7] or injective correspondences [23]. However, we believe that OWL can, in principle, be extended to support both classes of properties: unlinkability can be encoded via a more refined model of control flow, while injective correspondences likely can be encoded via a linear typing discipline on subroutine calls.

Finally, our extraction mechanism in Section 5 is exploratory in nature, and not yet verified for functional correctness or security, and is not meant to be competitive with hand-written optimized implementations.

4 OWL Core Theory

To give OWL formal security guarantees, we present OwlLang, a core calculus for computationally sound reductions using information flow labels.

OwlLang guarantees that well-typed protocols satisfy *simulatability* and *correctness*. Simulatability states that, for any adversary $\mathcal{A}$ corrupting a chosen set of names, running the protocol cannot leak any more information to $\mathcal{A}$ than it had before the protocol's execution. Dually, correctness states that all refinements on data in the protocol hold with high probability.

Comparison with Surface Language. OwlLang has a few differences from the surface language in §3. Aside from modeling atomic sum and product types rather than general enums and structs, the major difference is our treatment of the adversary label. In OwlLang, the adversary label is considered fixed in the typing judgment, and the protocol is well-typed against this label. Thus, types such as if sec(k) then t1 else t2 do not appear in OwlLang, since the adversary is fixed. This does not harm generality, however, since our type checker (§5.1) universally quantifies over adversary labels using symbolic techniques.

Additionally, OwlLang does not model indices or the happened predicate. We believe indices can be faithfully modeled by metaprogramming techniques (e.g., as in Squirrel [7]), while the happened predicate can be modeled through extending our formal model with a notion of global trace.

Tables	T		
Atomic Exprs	a	::=	$x \mid v \in \{0,1\}^* \mid \mathsf{f}(a_1,\ldots,a_k)$
		|	$\mathsf{inl}(a) \mid \mathsf{inr}(a)$
		|	$\mathsf{Z}(a) \mid \mathsf{pair}_\tau(a_1,a_2)$
		|	$\mathsf{fst}_\tau(a) \mid \mathsf{snd}_\tau(a)$
		|	$\mathsf{get}(\mathsf{n}) \mid \ldots$
Crypto Ops	op	::=	$\mathsf{senc} \mid \mathsf{sdec} \mid \mathsf{dhpk} \mid \ldots$
Expressions	e	::=	$\mathsf{ret}(a) \mid \mathsf{input} \mid \mathsf{output}(a)$
		|	$\mathsf{case}\ a\ (x.\ e_1)\ (y.\ e_2)$
		|	$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid T[a]$
		|	$T[a] := a' \mid \mathsf{op}(a_1,\ldots,a_k) \mid H(a)$
Configurations	$\mathcal{K}$	::=	$[0\ldots m] \to e$
Labels	ℓ	::=	$\bot \mid [\mathsf{n}] \mid \ell \wedge \ell'$
Hash Labels	L		
Types	τ	::=	$\mathsf{Name}(\mathsf{n}) \mid \mathsf{Data}(\ell,\ell) \mid \mathsf{Unit}$
		|	$\tau+\tau \mid \tau\times\tau \mid x:\tau\{\phi\} \mid \ldots$
Predicates	ϕ	::=	$a = a \mid \top \mid \neg\phi \mid \phi\wedge\phi$
Table Contexts	$\mathcal{T}$	::=	$\cdot \mid \mathcal{T}, T:\tau$
Type Contexts	Γ	::=	$\cdot \mid \Gamma, x:\tau$
Idealization	I	::=	$\mathsf{ideal} \mid \mathsf{real}$
Name Types	nt	::=	$\mathsf{nonce} \mid \mathsf{enckey}^I(\tau)$
		|	$\mathsf{sigkey}^I(\tau) \mid \mathsf{DH}^I \mid \ldots$
Name Kinds	nk	::=	$\mathsf{nonce} \mid \mathsf{enckey} \mid \ldots$
Problems	P	::=	$\mathsf{sec}(\mathsf{n})$
Hash Patterns	pat	::=	$\mathsf{n}_{\mathsf{hash}} \mid a \mid (\mathsf{pat},\mathsf{pat})$
Name Contexts	Σ	::=	$\cdot \mid \Sigma, \mathsf{n} : \mathsf{nt}$ (Base Names)
		|	$\Sigma, \mathsf{n} : \mathsf{pat} \mapsto_P \mathsf{nt}$ (RO Names)

Figure 6: **Syntax of** OwlLang.

4.1 OwlLang Syntax

The syntax of OwlLang is given in Figure 6. As in the surface syntax, each party's code is given by a monadic language of expressions, with primitive effects for interacting with the network, mutable state, and probabilistic sampling. All effects take *atomic expressions* a as input, which specify pure (non-probabilistic) computations, including constructors for sum types, constructors and destructors for product types, and arbitrary (pure) user-defined functions. The operators for pairing and unpairing are parameterized by a type τ, which we discuss below. For our security proof, we add the expression $\mathsf{Z}(a)$ for computing the value $0^{|a|}$. Additionally, atomic expressions include a $\mathsf{get}(\mathsf{n})$ command, which retrives the value of name n from the current execution context.

The commands input and $\mathsf{output}(a)$ are used for communicating through the network, which as in prior work [7, 23, 25], we assume is controlled by the adversary. Parties have access to mutable maps through global *tables*: the command $T[a]$ retrieves the value of a in table T (if one exists), while the command $T[a] := a'$ sets the value of a to a' in T. All cryptographic operations are performed through expressions, such as encryption, $\mathsf{senc}(a,a')$, and decryption, $\mathsf{sdec}(a,a')$. The command $H(a)$ is used to interact with an idealized *random oracle*, which we use for key derivation operations.

Finally, *configurations* $\mathcal{K}$ map party identifiers $\mathsf{id} \in [0\ldots m]$ (for some m) to closed expressions. Configurations form the interface between the adversary and the protocol.

4.1.1 Type Syntax

Before presenting the semantics of OWL, we describe OWL's types, which we use to specify security policies.

Types in OWL are built on top of *labels*, which track dataflows throughout the program. Intuitively, a label indicates a set of dependencies on cryptographic names. Atomic dependencies are of the form $[\mathsf{n}]$, where names n are either *base names* or *random oracle names*. Base names are assumed to be sampled ahead of time (e.g., Alice's generated key k), while random oracle names correspond to results of interacting with the random oracle $H(\cdot)$. Crucially, our core calculus does *not* use a symbolic adversary label; instead, the adversary label is a parameter of the type system.

Our core calculus has five main type formers: $\mathsf{Name}(\mathsf{n})$, the singleton type corresponding to the value of name n; $\mathsf{Data}(\ell,\ell')$, arbitrary data whose value has label ℓ and length has label ℓ'; and Unit, $\tau+\sigma$, $\tau\times\sigma$, and $x:\tau\{\phi\}$, standing for the unit type, sum and product types, and refinement types, respectively. Refinements are boolean formulae constructed out of base equalities $a = a'$ between atomic expressions. We additionally support cryptosystem-specific singleton types; e.g., $\mathsf{VK}(\mathsf{n})$ for verification keys, described in Appendix B.

The rest of Figure 6 describes the various contexts used for our typing judgements. Type contexts Γ and table contexts $\mathcal{T}$ assign types to local variables and tables, respectively. Name contexts assign *name types* to base names $(\mathsf{n} : \mathsf{nt})$ and random oracle names $(\mathsf{n} : \mathsf{pat} \to_P \mathsf{nt})$. Name types specify how the name may be used in the protocol: the nonce name type is used for opaque, random data, while the $\mathsf{enckey}^I\ \tau$ name type is used for symmetric (authenticated) encryption keys, encrypting values of type τ. Here, $I \in \{\mathsf{Ideal},\mathsf{Real}\}$ is an *idealization*, which tracks whether or not this encryption key has been idealized. User-facing protocols only use names annotated with Real.

Finally, random oracle names are assigned a *hash pattern* pat, describing the value that should be hashed, and a *computational problem* P, which witnesses the unforgeability of the hash's preimage. We will discuss these more in Figure 4.5. To define the semantics of OwlLang, we also define *name kinds* nk to be name types with annotations removed, such as enckey.

4.2 Security Policies for OwlLang

We now outline the necessary definitions for OWL's security policies. Security policies in OWL are composed of secrecy and integrity policies: secrecy policies are defined by labels, while integrity policies are defined by induction on types.

Label System. Labels in OWL form a join-semilattice structure [5], in the style of information flow. The main difference from prior uses of information flow [65] is the level of granularity: we do not track principals (e.g., Alice/Bob), but instead track name dependencies of the form $[\mathsf{n}]$. Name dependencies primarily guarantee that cryptographic keys are used properly by the protocol. For example, symmetric encryption [63] gen-

$$\boxed{\Sigma \vdash \mathsf{n} : \mathsf{nt}} \qquad \frac{(\mathsf{n} : \mathsf{nt}) \in \Sigma}{\Sigma \vdash \mathsf{n} : \mathsf{nt}} \qquad \frac{(\mathsf{n} : \mathsf{pat} \mapsto_P \mathsf{nt}) \in \Sigma}{\Sigma \vdash \mathsf{n} : \mathsf{nt}}$$

$$\boxed{\Sigma \vdash \ell \leq \ell'} \qquad \frac{}{\Sigma \vdash \ell \leq \ell}\ \text{REFL}$$

$$\frac{\Sigma \vdash \ell_1 \leq \ell_2 \qquad \Sigma \vdash \ell_2 \leq \ell_3}{\Sigma \vdash \ell_1 \leq \ell_3}\ \text{TRANS} \qquad \frac{}{\Sigma \vdash \bot \leq \ell}\ \text{ZERO}$$

$$\frac{\Sigma \vdash \ell_1 \leq \ell_3 \qquad \Sigma \vdash \ell_2 \leq \ell_3}{\Sigma \vdash \ell_1 \wedge \ell_2 \leq \ell_3}\ \text{ANDL} \qquad \frac{\Sigma \vdash \ell \leq \ell_1}{\Sigma \vdash \ell \leq \ell_1 \wedge \ell_2}\ \text{ANDR1}$$

$$\frac{\Sigma \vdash \ell \leq \ell_2}{\Sigma \vdash \ell \leq \ell_1 \wedge \ell_2}\ \text{ANDR2} \qquad \frac{\Sigma \vdash \mathsf{n} : \mathsf{enckey}^I(\tau)}{\Sigma \vdash \lceil \tau \rceil \leq [\mathsf{n}]}\ \text{HIERARCHY}$$

Figure 7: **Label Checking Rules for** OwlLang. *Similar* HIERARCHY *rules hold for other cryptographic operations, including MACs, digital signatures, and public-key encryptions.*

$$\boxed{\lceil \tau \rceil} \qquad \boxed{|\tau|}$$

$$\begin{aligned}
\lceil \mathsf{Name}(\mathsf{n}) \rceil &:= [\mathsf{n}] & |\mathsf{Name}(\mathsf{n})| &:= \bot \\
\lceil \mathsf{Data}(\ell, \ell') \rceil &:= \ell \wedge \ell' & |\mathsf{Data}(\ell, \ell')| &:= \ell' \\
\lceil \mathsf{Unit} \rceil &:= \bot & |\mathsf{Unit}| &:= \bot \\
\lceil \tau + \sigma \rceil &:= \lceil \tau \rceil \wedge \lceil \sigma \rceil & |\tau + \sigma| &:= |\tau| \wedge |\sigma| \\
\lceil \tau \times \sigma \rceil &:= \lceil \tau \rceil \wedge \lceil \sigma \rceil & |\tau \times \sigma| &:= |\tau| \wedge |\sigma| \\
\lceil x : \tau\{\phi\} \rceil &:= \lceil \tau \rceil & |x : \tau\{\phi\}| &:= |\tau|
\end{aligned}$$

Figure 8: **Covering Label and Length Label for Types**.

erally only guarantees security if the key k is used as a key, and not otherwise used in the protocol. In particular, key cycles (e.g., k encrypts k', which encrypts k) break security, and thus are excluded by our type system.

Our label checking rules are given in Figure 7. All rules except the last one, HIERARCHY, are standard for join-semilattices [5]. The new rule, HIERARCHY, reflects OWL's *hierarchical* label model: if k encrypts τ, then the label consisting of all names in τ flows to the label $[k]$. We capture this flow using the *covering label* $\lceil \tau \rceil$ for τ, given in Figure 8, which joins together all labels present in the type. Similar HIERARCHY rules hold for other cryptographic operations, including MACs, digital signatures, and public-key encryption. We also have the *length label*, $|\tau|$, which bounds the amount of information present in the *lengths* of values of type τ. Length labels are important for guaranteeing that information does not leak through lengths (e.g., lengths of ciphertexts).

Integrity Policies. Integrity in OWL (e.g., both parties return the same key of type $\mathsf{Name}(\mathsf{n})$) is defined via our integrity policy, given in Figure 9, which defines for each type τ the set of values that satisfy τ. Our integrity policies are of the form $[\![\tau]\!]^{\Sigma,N,W}$, where N is a *name environment* mapping in-scope base names $(\mathsf{n} : \mathsf{nt}) \in \Sigma$ to values, while W is a *world*, which contains the mutable state that evolves throughout the protocol. Worlds map table variables (or the random oracle) to partial maps on values.

$$\boxed{N : \mathsf{Name} \to \{0,1\}^*}$$

$$\boxed{W : \mathsf{TVar} \cup \{\mathsf{RO}\} \to \{0,1\}^* \to \{0,1\}^* \cup \{\bot\}}$$

$$\begin{aligned}
[\![a]\!]^{\Sigma,N,W}_{\mathsf{hash}} &:= [\![a]\!]^N_I \\
[\![\mathsf{n}_{\mathsf{hash}}]\!]^{\Sigma,N,W}_{\mathsf{hash}} &:= W[\mathsf{RO}, v'] \text{ if } (\mathsf{n} : \mathsf{pat} \to_P \mathsf{nt}) \in \Sigma \wedge [\![\mathsf{pat}]\!]^{\Sigma,N,W}_{\mathsf{hash}} = v' \\
[\![(\mathsf{pat}, \mathsf{pat}')]\!]^{\Sigma,N,W}_{\mathsf{hash}} &:= [\![\mathsf{pat}]\!]^{\Sigma,N,W}_{\mathsf{hash}} + [\![\mathsf{pat}']\!]^{\Sigma,N,W}_{\mathsf{hash}} \text{ if both defined} \\
[\![\mathsf{Name}(\mathsf{n})]\!]^{\Sigma,N,W}(v) &:= \begin{cases} v = N(\mathsf{n}) & \text{if } (\mathsf{n} : \mathsf{nt}) \in \Sigma \\ v = [\![\mathsf{n}_{\mathsf{hash}}]\!]^{\Sigma,N,W}_{\mathsf{hash}} & \text{otherwise} \end{cases} \\
[\![\mathsf{Data}(\ell, \ell')]\!]^{\Sigma,N,W}(v) &:= \mathsf{True} \\
[\![\mathsf{Unit}]\!]^{\Sigma,N,W}(v) &:= v = 0 \\
[\![\tau + \sigma]\!]^{\Sigma,N,W}(v) &:= \begin{cases} [\![\tau]\!]^{\Sigma,N,W}(v') & \text{if } v = 0v', \text{ or} \\ [\![\sigma]\!]^{\Sigma,N,W}(v') & \text{if } v = 1v'. \end{cases} \\
[\![\tau \times \sigma]\!]^{\Sigma,N,W}(v) &:= \mathsf{bdry}_\tau(v) \neq \bot \wedge [\![\tau]\!]^{\Sigma,N,W}(v[\ldots \mathsf{bdry}_\tau(v)]) \wedge [\![\sigma]\!]^{\Sigma,N,W}(v[\mathsf{bdry}_\tau(v) \ldots]) \\
[\![x : \tau\{\phi\}]\!]^{\Sigma,N,W}(v) &:= [\![\tau]\!]^{\Sigma,N,W}(v) \wedge [\![\phi]\!]^N(v)
\end{aligned}$$

Figure 9: **Name Environments, Worlds, and Integrity Policies for Data**. *The interpretation I (§4.3) is implicit for atomic expressions.*

The integrity policy for $\mathsf{Name}(\mathsf{n})$ has two cases, depending on whether n is a base name of the form $(\mathsf{n} : \mathsf{nt}) \in \Sigma$, or a random oracle name of the form $(\mathsf{n} : \mathsf{pat} \to_P \mathsf{nt}) \in \Sigma$. In the former case, $[\![\mathsf{Name}(\mathsf{n})]\!]^{\Sigma,N,W}$ requires that the given value is equal to exactly the value of n in the name environment. In the latter case, we say that the given value is equal to the value of $[\![\mathsf{n}_{\mathsf{hash}}]\!]^{\Sigma,N,W}_{\mathsf{hash}}$, where $[\![\mathsf{pat}]\!]^{\Sigma,N,W}_{\mathsf{hash}}$ evaluates the value of the hash pattern, pat. The value of $[\![\mathsf{pat}]\!]^{\Sigma,N,W}_{\mathsf{hash}}$: maps $\mathsf{n}_{\mathsf{hash}}$ to the corresponding value of the random oracle (if it exists); maps atomic expressions a to the semantics $[\![a]\!]^N_I$, described in §4.3; and maps pairs $(\mathsf{pat}, \mathsf{pat}')$ to their concatenations.

The integrity policy for $\mathsf{Data}(\ell, \ell')$ is trivial, as this type contains arbitrary bitstrings, while the integrity policy for Unit is the singleton set $\{0\}$. The integrity policy for sum types $\tau + \sigma$ reflects that they are semantically tagged unions, while the policy for $\tau \times \sigma$ reflects that product types are semantically concatenations. For concatenations to be unambiguous, we say that $[\![\tau \times \sigma]\!]^{\Sigma,N,W}(v)$ only if v *parses* under τ, meaning that the boundary between the τ-half and the σ-half of v is well-defined. Parsing is discussed in more detail in Appendix A. Finally, the integrity policy for $x : \tau\{\phi\}$ is derived from that of τ, but requires that the refinement ϕ holds as well.

4.3 OwlLang Semantics

To define concrete semantics for OwlLang, we first need a global *interpretation* I, which defines the semantics for deter-

ministic functions, cryptographic operations, and name kinds. Looking ahead, we additionally define *PPT* interpretations to be families of interpretations I_λ with polynomial assignments and lengths:

Definition 1 (Interpretation)**.** *An interpretation I:*

- *assigns each function symbol* f *a mapping* $[\![\mathsf{f}]\!]_I$ *from lists of values in* $\{0,1\}^*$ *to* $\{0,1\}^*$*;*
- *assigns each cryptographic operation* op *a mapping* $[\![\mathcal{D}]\!]_I$ *from lists of values in* $\{0,1\}^*$ *to finitely supported probability distributions over* $\{0,1\}^*$*;*
- *assigns each name kind* nk_I *a fixed length of bits* L_{nk}*, along with a probability distribution* $[\![\mathsf{nk}]\!]$ *over* $\{0,1\}^{\mathsf{L}_{\mathsf{nk}}}$*;*
- *a length* $\mathsf{L}_{\mathsf{hash}}$ *for the random oracle.*

The family I_λ of interpretations is PPT *when all lengths* L_{nk}, $\mathsf{L}_{\mathsf{hash}}$ *are polynomial in* λ*, and all assigned functions and probability distributions have runtimes polynomial in* λ*.*

We assume that all interpretations are *standard*, which fixes the semantics of certain operations, and requires that the relevant cryptosystems are secure. Standard interpretations are defined in Appendix A.

Semantics for Expressions. We define the semantics of Owl protocols by first defining the semantics of individual expressions, then lifting this semantics to an *interaction* between configurations of expressions and a computational adversary.

We define the semantics for atomic and non-atomic expressions separately. Both are assumed to be closed throughout. Semantics for atomic expressions have the form $[\![a]\!]^N_I \in \{0,1\}^*$, where N is a *name environment* mapping in-scope base names n to bitstrings. This semantics is largely standard: atomic functions (e.g., inl/inr, pairing, and user-defined functions) are assumed to come from the interpretation I, while $[\![\mathsf{get}(\mathsf{n})]\!]^N_I = N(\mathsf{n})$. We will write $[\![a]\!]^N$ when the interpretation is implicit.

Because non-atomic expressions e are meant to be run in parallel and interactively queried by the adversary, we allow the adversary to guide the execution of e through a small-step semantics. On input i, e executes a single computational step, such as reducing a computation $f(a_i)$, performing an output, or receiving the input i. Along the way, e will modify the current *world*, or values of the current random oracle and in-scope tables. Formally, non-atomic expressions have a semantics

$$[\![e]\!]^N_I : \mathsf{World} \to \{0,1\}^* \to \mathsf{Dist}(\mathsf{Expr} \times \mathsf{World} \times \{0,1\}^*),$$

mapping worlds $W \in \mathsf{World}$ and inputs i to distributions over next expressions e, worlds W', and outputs o.

The semantics for OwlLang expressions is given in Figure 18. We use monadic syntax for probability distributions; i.e., $\mathsf{Ret}(x)$ returns the unit mass probability distribution, while $x \xleftarrow{\$} D_1; D_2$ samples from D_1, obtains a value x, and continues as D_2. While the semantics returns an output for every input, we return the empty bitstring ε if there is no next output, and discard the input if it is not used.

We now discuss selected semantic rules: the semantics for $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ first tests if e_1 is of the form $\mathsf{ret}(a)$; if it is, we proceed by reducing to $e_2[v/x]$, where v is the value of a. Otherwise, we reduce e_1, and propagate the results accordingly.

The semantics for $T[a]$ returns the bitstring 00 if the value of a is not found in the map, and returns a bitstring of the form $1v$ otherwise; this parallels our encoding of option types $\mathsf{Unit} + \tau$. Finally, the semantics for $H(a)$ returns the value of a in the random oracle, if it exists; otherwise, it samples a new value and returns it, along with updating the random oracle.

4.3.1 Adversarial Semantics

We give semantics to protocols via a security game, which allows an *arbitrary* computational adversary to interactively query the protocol over a number of rounds. Queries may provide inputs to parties, access the random oracle, and obtain certain values of base names (e.g., public keys and values of corrupted names). At the end of the interaction, we output a decision bit b from the adversary, along with a value $\mathsf{ok}(N, \mathcal{K})$, which specifies whether the integrity policy induced by our types (described below) is satisfied.

Security games are defined relative to *adversaries*.

Definition 2 (Adversary)**.** *An adversary $\mathcal{A}$ is given by a positive integer k and three probabilistic algorithms:*

- $\mathcal{A}_{\mathsf{query}}(s)$*, which takes as input a bitstring state s , and returns a pair (s', q) of a new state and a query $q \in \{0,1\}^*$;*
- $\mathcal{A}_{\mathsf{out}}(s, o)$*, which takes as input a bitstring state s and an output o from the protocol, and returns a new state s'.*
- $\mathcal{A}_{\mathsf{decide}}(s)$*, which takes as input a bitstring state s and returns a bit.*

The family $\mathcal{A}_\lambda$ is PPT *when k_λ and the runtime of all three algorithms is $O(\mathsf{poly}(\lambda))$.*

Given a name context Σ, configuration $\mathcal{K}_0$, types τ_i for each i in the domain of $\mathcal{K}_0$, an adversary $\mathcal{A}$ with associated label $\ell_{\mathcal{A}}$, and a name environment N, we define the security game $\mathcal{G}_I^{\Sigma,\{\tau_i\}}(N, \mathcal{K}_0, \ell_{\mathcal{A}}, \mathcal{A})$ as in Figure 10.

We initialize the interaction by setting the current configuration $\mathcal{K}$ to the initial one ($\mathcal{K}_0$), and setting the adversary's state s to the empty bitstring, ε. Then, for k rounds we query the adversary and react appropriately. We assume a canonical injective embedding of bitstrings into queries. If the adversary outputs $\mathsf{Input}(j, i)$, we run the jth party in W on input i according to the semantics in §4.3, obtaining a new expression e', world W', and output o. We then update the interaction state appropriately, delivering output o to the adversary using $\mathcal{A}_{\mathsf{out}}$ and updating $\mathcal{K}[j]$ to be e'. Random oracle queries are answered using $W[\mathsf{RO}, \cdot]$, as in the expression semantics.

Additionally, we allow the adversary to access *oracle queries* of the form $\mathsf{q} \in \mathsf{Orcl}(\Sigma, \ell_{\mathcal{A}}, N)$, where $\mathsf{Orcl}(\Sigma, \ell_{\mathcal{A}}, N)$ is the set of available oracle queries, defined on the bottom of Figure 10. We assume that $\mathsf{Orcl}(\Sigma, \ell_{\mathcal{A}}, N)$ contains at least the query $\mathsf{get}(\mathsf{n})$, which allows the adversary to obtain the value of corrupted (non-hash-derived) names. We additionally use $\mathsf{Orcl}(\Sigma, \ell_{\mathcal{A}}, N)$ to allow the adversary to obtain public keys for asymmetric cryptosystems (e.g., Diffie-Hellman and digital signatures).

At the end of the interaction, we query $\mathcal{A}_{\mathsf{decide}}$ to turn the adversary state into a bit b. We return b along with the value $\mathsf{ok}^{\Sigma,\{\tau_i\}}(N, W, \mathcal{K})$, which maps party indices j to a boolean or $\perp$, indicating whether the party has terminated, and whether the party's return value (if it exists) satisfies the refinement for τ_j.

Security game $\mathcal{G}_I^{\Sigma,\{\tau_i\}}(N,\mathcal{K}_0,\ell_{\mathcal{A}},\mathcal{A})$

$s, W := \varepsilon, \{\}$
$\mathcal{K} := \mathcal{K}_0$
for $\mathcal{A}.k$ rounds **do**
  $(s', q) \leftarrow \mathcal{A}_{\mathsf{query}}(s)$
  if $q = \mathsf{Input}(j, i)$ **then**
    $(e', W', o) \xleftarrow{\$} [\![W[j]]\!]^N(W, i)$
    $\mathcal{K} := \mathcal{K}[j := e']$
    $W := W'$
    $s \xleftarrow{\$} \mathcal{A}_{\mathsf{out}}(s', o)$
  else if $q = \mathsf{Hash}(i)$ **then**
    if $W[\mathsf{RO}, i] = \bot$ **then**
      $v \xleftarrow{\$} \{0,1\}^{[\![\mathsf{L_{hash}}]\!]}$
      $W := W[\mathsf{RO}, i := v]$
    $s \xleftarrow{\$} \mathcal{A}_{\mathsf{out}}(s, W[\mathsf{RO}, i])$
  else if $q = \mathsf{q}, \mathsf{q} \in \mathsf{Orcl}(\Sigma, \ell_{\mathcal{A}}, N)$: **then**
    $v \leftarrow \mathsf{Orcl}(\Sigma, \ell_{\mathcal{A}}, N, \mathsf{q})$
    $s \xleftarrow{\$} \mathcal{A}_{\mathsf{out}}(s, v)$
$b \xleftarrow{\$} \mathcal{A}_{\mathsf{decide}}(s)$
return $(b, \mathsf{ok}^{\Sigma,\{\tau_j\}}(N, W, \mathcal{K}))$

$$\mathsf{ok}^{\Sigma,\{\tau_i\}}(N,W,\mathcal{K}) := \left[j \mapsto \begin{cases} [\![\tau_j]\!]^{\Sigma,N,W}(v) & \text{if } \mathcal{K}[j] = \mathsf{ret}(v) \\ \bot & \text{otherwise} \end{cases} \right]$$

$$\mathsf{Orcl}(\Sigma, \ell_{\mathcal{A}}, N) := \begin{bmatrix} \mathsf{get(n)} \mapsto N(\mathsf{n}) \text{ if } \Sigma \vdash [\mathsf{n}] \leq \ell_{\mathcal{A}}, (\mathsf{n} : \mathsf{nt}) \in \Sigma \\ \mathsf{vk(n)} \mapsto [\![\mathsf{vk}]\!](N(\mathsf{n})) \text{ if } (\mathsf{n} : \mathsf{sigkey}^I) \in \Sigma \\ \mathsf{dhpk(n)} \mapsto [\![\mathsf{dhpk}]\!](N(\mathsf{n})) \text{ if } (\mathsf{n} : \mathsf{DH}) \in \Sigma \\ \cdots \end{bmatrix}$$

Figure 10: **Interaction of OWL Protocols with Adversary**. *The interpretation I is implicit.*

4.4 Security Goals

Security for OwlLang is split into two statements: *correctness* for integrity, and *simulatability* for secrecy. To define security, we first define the probability distribution $\mathsf{Gen}_I(\Sigma)$ for generating name environments:

Definition 3 (Name Generator)**.** *Given an interpretation I, $\mathsf{Gen}_I(\Sigma)$ is the probability distribution over name environments for Σ, where each name kind nk is sampled according to $[\![\mathsf{nk}]\!]_I$.*

Correctness. Integrity guarantees in OwlLang are encoded through the *correctness* of parties' return values, specified by their types. Correctness in OwlLang states that, whenever party i has final return type τ_i in the configuration $\mathcal{K}$, all final return values of party i must satisfy $[\![\tau_i]\!]$ (Figure 9) with all but negligible probability. For example, correctness for the return type $\mathsf{Unit} + (\mathsf{Name(n)} \times \mathsf{Name(n')})$, states that we return a tagged union that either contains a unit value to indicate failure, or contains a pair (k_1, k_2) of two keys corresponding to n and n', respectively.

We formally encode correctness through the ok predicate in Figure 10, which returns a partial map from party IDs to booleans, stating whether party j satisfied the refinement for their return value (if it exists).

Definition 4 (Correctness)**.** *Let I_λ be a family of interpretations, indexed by λ. We say that $\mathcal{K}$ is* correct *under Σ and $\{\tau_i\}$, written $I_\lambda; \Sigma; \ell_{\mathcal{A}}; \{\tau_i\} \vDash_{\mathsf{integ}} \mathcal{K}$ if, for all PPT adversaries $\mathcal{A}$, we have that*

$$\Pr[V(j) \neq \bot \implies V(j) = 1 \mid N \xleftarrow{\$} \mathsf{Gen}_{I_\lambda}(\Sigma), \\ (\_, V) \xleftarrow{\$} \mathcal{G}_{I_\lambda}^{\Sigma,\{\tau_i\}}(N, \mathcal{K}, \ell_{\mathcal{A}}, \mathcal{A}_\lambda)]$$

is overwhelming in λ.

Simulatability. Simulatability in OwlLang states that any computational information returned by the adversary in Figure 10 can be efficiently extracted by a *simulator*, with oracle access to $\mathsf{Orcl}(\Sigma, \ell_{\mathcal{A}}, N)$. Intuitively, this means that all computational information about names in the name environment N can be reconstructed using only public information.

Definition 5 (Simulatability)**.** *We say that $\mathcal{K}$ is* simulatable *under Σ and $\{\tau_i\}$, written $I_\lambda; \Sigma; \ell_{\mathcal{A}}; \{\tau_i\} \vDash_{\mathsf{sim}} \mathcal{K}$, if for all PPT $\mathcal{A}$, there exists PPT $\mathcal{S}_\lambda$ such that*

$$\Pr_{N \xleftarrow{\$} \mathsf{Gen}_{I_\lambda}(\Sigma)} [b = b' \mid (b, \_) \xleftarrow{\$} \mathcal{G}_{I_\lambda}^{\Sigma,\{\tau_i\}}(N, \mathcal{K}, \ell_{\mathcal{A}}, \mathcal{A}_\lambda), \\ b' \xleftarrow{\$} \mathcal{S}_\lambda^{\mathsf{Orcl}_{I_\lambda}(\Sigma, \ell_{\mathcal{A}}, \cdot, N)}]$$

is overwhelming in λ.

We then define *security* to be the conjunction of correctness and simulatability:

Definition 6 (Security)**.** *We say that $\mathcal{K}$ is* secure*, written $I_\lambda; \Sigma; \ell_{\mathcal{A}}; \{\tau_i\} \vDash \mathcal{K}$, whenever $I_\lambda; \Sigma; \ell_{\mathcal{A}}; \{\tau_i\} \vDash_{\mathsf{integ}} \mathcal{K}$ and $I_\lambda; \Sigma; \ell_{\mathcal{A}}; \{\tau_i\} \vDash_{\mathsf{sim}} \mathcal{K}$.*

4.5 Type System

We now present the typing rules for OwlLang. The typing rules are split into three parts: well-definedness of name contexts in Figure 11, the core, non-cryptographic typing rules in Figure 19, and the typing rules for cryptographic operations in Figure 12.

4.5.1 Core Rules

The typing rules for atomic expressions, given in Figure 19, are largely standard. We allow the pairing/unpairing operations $\mathsf{pair}/\mathsf{fst}/\mathsf{snd}$ to *fail*, returning $\mathsf{inl}(0)$, whenever the left-hand side of the pair fails to parse. We additionally require the left-hand side to be *parsable*, so that the result of parsing is well-defined; parsable types are defined in Figure 14. We assign $\mathsf{Z}(a)$ the type $\mathsf{Data}(|\tau|, |\tau|)$ whenever a has type τ, since $\mathsf{Z}(a)$ is semantically a function of only the length of a. The typing rule for refinement types states that a has type $x : \tau\{\phi\}$ whenever a has type τ, and we can prove semantically that ϕ holds of a.

Our typing rules for expressions are of the form $\Sigma; \mathcal{T}; \Gamma; \ell_{\mathcal{A}} \vdash e : \tau$. Here, $\ell_{\mathcal{A}}$ is the label for the adversary, which remains constant throughout the typing derivation. This reflects that OWL guarantees *static security*.

The rule for input returns data labelled with $\ell_{\mathcal{A}}$, while output requires data labelled with $\ell_{\mathcal{A}}$. The rule OP-TRIV allows us to

$$\boxed{\Sigma \vdash \mathsf{pat}\ \mathsf{context}\ P} \qquad \frac{\mathsf{pat} = \mathsf{get}(\mathsf{n}) \vee \mathsf{pat} = \mathsf{n}_{\mathsf{hash}}}{\Sigma \vdash \mathsf{pat}\ \mathsf{context}\ \mathsf{sec}(\mathsf{n})}$$

$$\frac{}{\Sigma \vdash \mathsf{dhcombine}(\mathsf{dhpk}(\mathsf{get}(\mathsf{n})), \mathsf{get}(\mathsf{n}'))\ \mathsf{context}\ \mathsf{DH}(\mathsf{n},\mathsf{n}')}$$

$$\frac{\Sigma \vdash \mathsf{pat}_i\ \mathsf{context}\ P \qquad i \in \{1,2\}}{\Sigma \vdash (\mathsf{pat}_1, \mathsf{pat}_2)\ \mathsf{context}\ P}$$

$$\frac{\begin{array}{c}\forall (\mathsf{n}' : \mathsf{pat}' \to_P \mathsf{nt}') \in \Sigma, \forall N\ W, \\ [\![\mathsf{pat}']\!]^{\Sigma,N,W}_{\mathsf{hash}}\ \text{defined} \implies [\![\mathsf{pat}]\!]^{\Sigma,N,W}_{\mathsf{hash}} \neq [\![\mathsf{pat}']\!]^{\Sigma,N,W}_{\mathsf{hash}}\end{array}}{\Sigma \vDash H(\mathsf{pat})\ \mathsf{undefined}}$$

$$\boxed{\vdash \Sigma} \qquad \frac{}{\vdash \cdot} \qquad \frac{\vdash \Sigma \qquad \Sigma \vdash \mathsf{nt} \qquad \mathsf{n} \notin \Sigma}{\vdash \Sigma, \mathsf{n} : \mathsf{nt}}$$

$$\frac{\begin{array}{c}\vdash \Sigma \\ \Sigma \vDash H(\mathsf{pat})\ \mathsf{undefined} \qquad \Sigma \vdash \mathsf{pat} \qquad \Sigma \vdash \mathsf{pat}\ \mathsf{context}\ P \\ \Sigma \vdash \mathsf{nt} \qquad \mathsf{nt} = \mathsf{enckey}^{\mathsf{Real}}\ \tau \vee \mathsf{nt} = \mathsf{nonce}\end{array}}{\vdash \Sigma, \mathsf{n} : \mathsf{pat} \mapsto_P \mathsf{nt}}$$

Figure 11: **Selected Rules for Well-Defined Contexts and Hash Patterns**.

over-approximate cryptographic operations by treating them as ordinary functions.

For control flow, we have *two* typing rules for the case expression: CASE, which consumes sum types; and CASE-CORR, which consumes arbitrary, possibly corrupted data labelled with $\ell_{\mathcal{A}}$.

Finally, we have rules for accessing global tables in $\mathcal{T}$, which reflect each table $(T : \tau) \in \mathcal{T}$ being a partial map from $\ell_{\mathcal{A}}$-labelled data to τ.

Configurations $\mathcal{K}$ are typed with a set of types $\{\tau_i\}$ whenever each party i in $\mathcal{K}$ is typed with τ_i.

4.5.2 Cryptographic Typing Rules

The main typing rules for cryptography are given in Figure 12.

Symmetric Encryption. First, we have symmetric (authenticated) encryption and decryption. Rule ENC-n states that if a_1 is a key of type $\mathsf{Name}(\mathsf{n})$, n is a non-idealized encryption key for τ, and a_2 is of the corresponding plaintext type, then $\mathsf{senc}(a_1, a_2)$ is data labelled with $\ell_{\mathcal{A}}$. We have the side condition $\Sigma \vdash |\tau| \leq \ell_{\mathcal{A}}$, which enforces that we only encrypt plaintexts with public lengths. Rule DEC-n operates in reverse, returning an option type if decryption fails. Both rules require that the label $[\mathsf{n}]$ does *not* flow to $\ell_{\mathcal{A}}$, which ensures that the adversary only views the key through well-formed encryptions and decryptions.

Hashing. Next, we have rules for hashing via the random oracle. Many constraints necesary for hashing to be secure in OwlLang are encoded in Figure 11, the well-formedness constraints on name contexts. Intuitively, $\mathsf{n} : \mathsf{pat} \to_P \mathsf{nt}$ is valid in Σ when:

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash a_1 : \mathsf{Name}(\mathsf{n}) \qquad \Sigma \vdash \mathsf{n} : \mathsf{enckey}^{\mathsf{real}}\tau \\ \Sigma;\Gamma \vdash a_2 : \tau \qquad \Sigma \not\vdash [\mathsf{n}] \leq \ell_{\mathcal{A}} \qquad \Sigma \vdash |\tau| \leq \ell_{\mathcal{A}}\end{array}}{\Sigma;\mathcal{T};\Gamma;\ell_{\mathcal{A}} \vdash \mathsf{senc}(a_1,a_2) : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}})}\ \text{ENC-n}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash a_1 : \mathsf{Name}(\mathsf{n}) \qquad \Sigma \vdash \mathsf{n} : \mathsf{enckey}^{\mathsf{real}}\tau \\ \Sigma;\Gamma \vdash a_2 : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}}) \qquad \Sigma \not\vdash [\mathsf{n}] \leq \ell_{\mathcal{A}}\end{array}}{\Sigma;\mathcal{T};\Gamma;\ell_{\mathcal{A}} \vdash \mathsf{sdec}(a_1,a_2) : \mathsf{Unit} + \tau}\ \text{DEC-n}$$

$$\frac{(\mathsf{n} : \mathsf{DH}^{\mathsf{Real}}) \in \Sigma \qquad \Sigma;\Gamma \vdash a : \mathsf{Name}(\mathsf{n})}{\Sigma;\mathcal{T};\Gamma;\ell_{\mathcal{A}} \vdash \mathsf{dhpk}(a) : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}})}\ \text{DHPK}$$

$$\frac{\begin{array}{c}(\mathsf{n} : \mathsf{pat} \mapsto_P \mathsf{nt}) \in \Sigma \qquad \Sigma;\Gamma \vdash a : \tau \\ \forall N\ W\ v.\ [\![\tau]\!]^{\Sigma,N,W}(v) \implies [\![\mathsf{pat}]\!]^{\Sigma,N,W}_{\mathsf{hash}} = v \\ \Sigma;\ell_{\mathcal{A}} \vdash \mathsf{unsolvable}\ P\end{array}}{\Sigma;\mathcal{T};\Gamma;\ell_{\mathcal{A}} \vdash H(a) : \mathsf{Name}(\mathsf{n})}\ \text{HASH-pat}$$

$$\frac{\Sigma;\Gamma \vdash a : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}})}{\Sigma;\mathcal{T};\Gamma;\ell_{\mathcal{A}} \vdash H(a) : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}})}\ \text{HASH-CORR}$$

$$\boxed{\Sigma;\ell_{\mathcal{A}} \vdash \mathsf{unsolvable}\ P} \qquad \frac{\Sigma \vdash \mathsf{n} : \mathsf{nonce} \qquad \Sigma \not\vdash [\mathsf{n}] \leq \ell_{\mathcal{A}}}{\Sigma;\ell_{\mathcal{A}} \vdash \mathsf{unsolvable}\ \mathsf{sec}(\mathsf{n})}$$

$$\frac{\begin{array}{c}\Sigma \vdash \mathsf{n} : \mathsf{DH} \\ \Sigma \vdash \mathsf{n}' : \mathsf{DH} \qquad \Sigma \not\vdash [\mathsf{n}] \leq \ell_{\mathcal{A}} \qquad \Sigma \not\vdash [\mathsf{n}'] \leq \ell_{\mathcal{A}} \qquad \mathsf{n} \neq \mathsf{n}'\end{array}}{\Sigma;\ell_{\mathcal{A}} \vdash \mathsf{unsolvable}\ \mathsf{DH}(\mathsf{n},\mathsf{n}')}$$

Figure 12: **Selected Rules for Cryptographic Operations in** OwlLang.

no other hash pattern pat' collides with pat, specified by the judgement $\Sigma \vDash H(\mathsf{pat})\ \mathsf{undefined}$; producing a value that satisfies pat requires solving the hash problem P, specified by the judgement $\Sigma \vdash \mathsf{pat}\ \mathsf{context}\ P$; and the resulting name type nt is either nonce or $\mathsf{enckey}^R\ \tau$, which guarantees that base names of type nt can be generated by hash values.

Now, we turn to the typing rules for computing hashes in Figure 12. Rule HASH-pat states that $a : \tau$ hashes to the name n whenever: the assignment $(\mathsf{n} : \mathsf{pat} \to_P \mathsf{nt})$ is in the name context; we have that τ semantically satisfies the hash pattern pat; and P is *unsolvable*. The second condition is satisfied whenever, for all worlds W and name environments N, the integrity predicate for τ under N and W is semantically the singleton set consisting of the semantic value of pat under N and W. The third condition, $\Sigma;\ell_{\mathcal{A}} \vdash \mathsf{unsolvable}\ P$, corresponds to the computational infeasibility of the adversary solving the *hash problem* P. We support two hash problems: $\mathsf{sec}(\mathsf{n})$, for difficulty of the adversary computing n; and $\mathsf{DH}(\mathsf{n},\mathsf{n}')$, for difficulty of the adversary computing the Diffie-Hellman shared secret for n and n'. Finally, we have the rule HASH-CORR, which states that $\ell_{\mathcal{A}}$-labelled inputs hash to $\ell_{\mathcal{A}}$-labelled outputs.

Diffie-Hellman Operations. We model Diffie-Hellman operations and their security not through extra typing rules, but via the hash problem $\mathsf{DH}(\mathsf{n},\mathsf{n}')$. We assume function symbols

for the atomic expressions $\mathsf{dhpk}(\cdot)$ and $\mathsf{dhcombine}(\cdot,\cdot)$ for obtaining Diffie-Hellman public keys and computing shared secrets from public and secret keys. Rule DHPK in Figure 12 states that $\mathsf{dhpk}(a)$ has type $\mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}})$ whenever a is a non-idealized Diffie-Hellman private key; this models that we consider public keys considered public. Figure 11 states that pat is a context for $\mathsf{DH}(\mathsf{n},\mathsf{n}')$ when pat contains the corresponding shared secret for n and n′, $\mathsf{dhcombine}(\mathsf{dhpk}(\mathsf{get}(\mathsf{n})),\mathsf{get}(\mathsf{n}'))$.

Signatures. Rules for digital signatures are given in Appendix B, along with other cryptographic operations such as MACs and public-key encryption. We model digital signatures via the verification operation $\mathsf{vrfy}(\mathsf{vk},x,t)$ returning the option type $\mathsf{Unit}+\tau$, similar to the return type of sdec. However, as digital signatures do not guarantee message privacy, we require that $\Sigma \vdash |\tau| \leq \ell_{\mathcal{A}}$ so that the message contents may safely be leaked.

4.6 Soundness

To guarantee security, we need to ensure that the family of interpretations I_λ is *secure*, meaning that the semantics of all cryptosystems satisfy relevant notions of security:

Definition 7 (Secure Interpretation)**.** *The family of interpretations I_λ is secure when:*

- *The triple $(\llbracket\mathsf{enckey}\rrbracket_{I_\lambda},\llbracket\mathsf{senc}\rrbracket_{I_\lambda},\llbracket\mathsf{sdec}\rrbracket_{I_\lambda})$ satisfies IND-CPA, INT-CTXT, and key privacy [14, 16] (e.g., as provided by AES-GCM [56]);*
- *The Gap Diffie-Hellman assumption [26] holds for the group induced by exponent generation $\llbracket\mathsf{DH}\rrbracket_{I_\lambda}$, the public-key operation $\llbracket\mathsf{dhpk}\rrbracket_{I_\lambda}$, and the shared-secret computation $\llbracket\mathsf{dhcombine}\rrbracket_{I_\lambda}$;*
- *The algorithms $\llbracket\mathsf{sigkey}\rrbracket_{I_\lambda},\llbracket\mathsf{vk}\rrbracket_{I_\lambda},\llbracket\mathsf{sign}\rrbracket_{I_\lambda}$, and $\llbracket\mathsf{vrfy}\rrbracket_{I_\lambda}$ induce an unforgeable signature scheme [63];*
- *... [similar conditions for other cryptographic primitives]*

Now, we have that well-typed configurations are secure, as defined in Definition 6:

Theorem 1. *Suppose that $\Sigma;\mathcal{T};\ell_{\mathcal{A}} \vdash \mathcal{K} : \{\tau_i\}$. Then, we have that, for any PPT secure interpretation I_λ, $I_\lambda;\Sigma;\ell_{\mathcal{A}};\{\tau_i\} \vDash \mathcal{K}$.*

Proof Overview. The intuition behind our proof is that if we have a typing derivation $\Sigma;\mathcal{T};\ell_{\mathcal{A}} \vdash \mathcal{K} : \{\tau_i\}$, then a sequence of suitable cryptographic reductions is guaranteed to exist for $\mathcal{K}$. Since each cryptographic reduction preserves security, we are guaranteed that $\mathcal{K}$ is secure only if the final, idealized protocol is secure. Standard programming language techniques (e.g., information-flow security [65]) then guarantee that this final protocol is secure. Formal proof details are given in Appendix C.

Indeed, our proof can be thought of as a "meta-programmed" version of CryptoVerif's [23] proof technique. For example, to idealize an encryption key name k, we first prove that k is only used as an encryption key (e.g., no other key encrypts k). Then, we *refactor* the configuration $\mathcal{K}$ to a reduction of the form $\mathcal{R}^{\mathsf{senc}(\mathsf{get}(k),\cdot),\mathsf{sdec}(\mathsf{get}(k),\cdot)}$, where k is not accessed by $\mathcal{R}$. At this point, we apply the security of the encryption scheme to replace the encryption and decryption oracles with idealized versions, which encrypt fake messages under fresh keys, and use an ideal decryption log to recover plaintexts from ciphertexts. As a result, the transformed protocol is well-typed under the simplified name context $\Sigma[\mathsf{n} \mapsto (\mathsf{n} : \mathsf{enckey}^{\mathsf{Ideal}}\ \tau)]$; thus, we may continue the "main loop" of the proof and apply more cryptographic reductions (or prove security directly if the protocol is fully ideal).

Aside from cryptographic reduction steps for encryptions and signatures, we use a *random-oracle idealization* step to replace random oracle calls $H(a)$ with reads from fresh, unique names via $\mathsf{get}(\mathsf{n})$ expressions. The soundness of this transformation relies on the corresponding hash pattern being unsolvable (Figure 12), so that the adversary cannot compute $H(a)$; additionally, we ensure from well-formedness of the name context (Figure 11) that n is sampled from the same probability distribution as hash values from the random oracle.

Our proof of security has a number of attractive features. Foremost, it is highly *extensible*, as each cryptographic reduction step may be performed separately, using domain-specific proof techniques. The only interface between differing reduction steps is the output typing judgement $\Sigma';\mathcal{T}';\ell_{\mathcal{A}} \vdash \mathcal{K} : \{\tau'_i\}$ after idealization, which guarantees that further cryptographic primitives may be idealized.

5 Implementation

To evaluate Owl's expressiveness, we have implemented a validating compiler in ∼7,300 lines of Haskell (plus ∼580 lines of shared Rust code used by extracted implementations) and applied it to a large collection of case studies (§6). The compiler includes both a type checker that ensures the security of Owl protocols, and a (trusted) mechanism for automatically producing executable code for a protocol's parties.

5.1 Proof Checking via Typing

Unlike most computational verification tools (§2), Owl is fully-automated and requires no manual proof effort, except for writing the protocol itself in Owl.

Refinement Type Checking. All types in Owl carry an integrity policy, corresponding to the predicate $\llbracket\tau\rrbracket$ in Figure 9. Integrity policies in the type checker correspond to refinement type checking routines, and as in other systems [53, 66] are dispatched using the Z3 [35] SMT solver.

Symbolic Label Checking. In §4.6, we prove that if $\Sigma;\mathcal{T};\ell_{\mathcal{A}} \vdash \mathcal{K}$, then configuration $\mathcal{K}$ is secure against adversaries with label $\ell_{\mathcal{A}}$. Crucially, typing judgements depend on $\ell_{\mathcal{A}}$; for example, the rule for decryption in §4.5.2 only guarantees high-integrity results if the corresponding key name does *not* flow to $\ell_{\mathcal{A}}$.

However, we wish to prove security against *all* adversaries. To do so, Owl reasons about a *symbolic* adversary, effectively proving the family of judgements $\Sigma;\mathcal{T};\ell_{\mathcal{A}} \vdash \mathcal{K}$ for all $\ell_{\mathcal{A}}$. To support symbolic adversaries, we additionally encode our label checking rules into SMT, defining the adversary label to be an uninterpreted, universally quantified constant.

In addition to the core typing rules, certain primitives in Owl are used to reason about the adversary label. The command corr_case n in .. performs a case split on whether n is corrupt, splitting the type checking procedure into two cases for $\lceil\mathsf{n}\rceil \leq \ell_{\mathcal{A}}$ and $\lceil\mathsf{n}\rceil \not\leq \ell_{\mathcal{A}}$. Since the cases may yield different return types, the Owl type if sec(n) then t1 else t2 is used to combine them.

Proof Performance. Formal tools built atop an SMT solver often inherit the solver's incompleteness or timeouts. However,

in our case studies, we have not faced any issues regarding SMT performance. We attribute this to the following techniques.

- Limited theories: SMT solvers generally perform worse when faced with more difficult theories, such as nonlinear arithmetic, bitvector computations, or associative operations. We have not needed such reasoning in our case studies, since the verification conditions our type system exports are generally simple boolean refinement formulas and label checking queries, which we axiomatize using uninterpreted functions.
- Separate queries: SMT solvers such as Z3 often perform worse when attempting to prove $p \wedge q$ in one query, rather than proving p and q separately. Our system is designed to output one SMT query per verification condition, which we have found yields predictable proof performance.
- Hybrid reasoning: unlike tools such as Dafny [53], we do *not* export all nontrivial reasoning to SMT, instead preferring to solve subtyping queries inside our Haskell implementation whenever possible. This is enabled by our liberal use of special-purpose singleton types (e.g., $\mathsf{VK}(\mathsf{n})$ and $\mathsf{DHPK}(\mathsf{n})$, for verification keys and Diffie-Hellman public keys, respectively), which often encapsulates complex reasoning that would otherwise be performed in SMT.

5.2 Producing Executable Protocol Code

To show that protocols modeled in OWL are realizable, we developed an *extraction* pipeline that automatically generates executable implementations from OWL protocols. We briefly discuss the pipeline's design below.

Protocol verification tools run the risk of operating at too high a level of abstraction, such that implementations of the verified protocols must fill in unspecified, security-relevant details. In the worst case, such protocols could be so abstract that they are unrealizable in executable code. OWL's extraction pipeline ensures that OWL protocols are concrete enough to be directly implementable, and that those concrete details are checked for security by the type system. In §6, we discuss an example where extraction allowed us to catch a low-level bug in a protocol that was not detected by prior work.

The OWL extraction pipeline emits safe Rust [64] code. We chose Rust since it lets us control the memory layout of the emitted code while providing static safety guarantees. Notably, the semantics of OWL specify a concrete representation of all OWL data types as byte-strings; we mirror these semantics by extracting all OWL data types to `Vec<u8>` in Rust. For `struct`s and `enum`s, we also cache the offsets of the member fields. We rely on the standard `RustCrypto` crates to implement cryptographic primitives. Inputs are received and outputs are sent via TCP using pre-configured socket addresses. The entire pipeline is automatic—the OWL programmer need not write any Rust code to produce an implementation of their protocol.

Developing the OWL extraction pipeline forced certain choices in the design of the source language. For instance, endpoints are required to specify message routing. Extraction also necessitated a more complex design for indices than Squirrel [7] uses. We distinguish three kinds of indices: ghost indices (i.e., that exist only for verification purposes), indices representing sessions of a particular protocol, and indices representing a family of localities with the same functionality but different names. We extract names parameterized by a session ID to a map from indices to names, where names are generated ephemerally as needed. Names parameterized by a locality ID are extracted to a single executable implementation accompanied by multiple statically generated configurations of names, each corresponding to a different locality index. Additionally, extraction requires that all `struct` fields are annotated with a static length, to allow automatic generation of parsing code.

Limitations. Our current extraction pipeline serves to demonstrate that OWL protocols can be compiled automatically into working executable implementations. As such, it does not currently aim to provide state-of-the-art performance, nor does it aim to generate code that can interoperate with existing implementations of protocols. In particular, OWL does not specify packet formats, magic numbers, or other features of real-world protocol communication. We believe that it should be possible to extend the OWL source language and extraction pipeline to allow automatic extraction of performant and compliant implementations without significant changes to the core type system, and we hope to investigate this in future work.

6 Case Studies

We compare OWL against two state-of-the-art tools that, like OWL, provide computational security guarantees. CryptoVerif [23] focuses on automation, whereas Squirrel [7] focuses on expressivity; neither, however, supports modularity. OWL, however, aims to offer all three.

To evaluate OWL's success, we implement, verify, and extract 14 case studies covering all those presented by CryptoVerif [23] and Squirrel [7]. Figure 13 has high-level quantitative comparisons (note that all case studies verify in seconds), while we provide more details, along with qualitative differences, below.

	LoC			Time	
Name	**OWL**	**Other**	**Rust**	**(s)**	**Source**
Basic-Hash [27]	47	58	543	0.86	SQ
Hash-Lock [47]	66	123	686	1.18	SQ
LAK [44]	73	92	998	1.25	SQ
MW [58]	78	359	949	1.24	SQ
Feldhofer [41]	35	215	421	0.63	SQ
Private Auth [9]	73	74	809	0.87	SQ
Needham-Schroeder (sym) [59]	121	126	1228	3.49	CV
Needham-Schroeder (pub) [59]	82	107	1025	9.51	CV
Otway-Rees [60]	253	108	2659	7.63	CV
Yahalom (sym) [28]	191	83	2050	4.76	CV
Denning-Sacco (pub) [37]	107	119	1382	1.79	CV
Kerberos [50]	228	271	2148	8.75	CV
Diffie-Hellman Key Ex [38]	67	152	719	1.84	SQ
SSH Forwarding Agent [67]	164	304	1449	6.57	SQ

Figure 13: **Case Studies**. *Groupings indicate RFID, Authentication, and DH Protocols. CV indicates CryptoVerif; SQ indicates Squirrel.*

RFID Protocols. RFID systems typically contain several low-power tags that communicate with a reader. From Squirrel, we adopt a variety of tag-to-reader protocols.

We verify the RFID protocols, proving standard secrecy and

authentication guarantees. As an example, in the Basic Hash protocol [27], a tag proves possession of its key to the reader by transmitting a MAC of a public nonce. OWL's type system allows for an expressive specification that states that a valid tag indeed sent the MAC and has authenticated itself successfully to the reader if the MAC verifies:

```
1 enum reader_response {
2   | Ok (∃ i. (x:Data⟨adv⟩{sec(K⟨@i⟩) ⟹
3       ((x = get(NT⟨@i⟩)) ∧ happened(tag_main⟨@i⟩()))}))
4   | No
5 }
6 def reader_main () @ reader : reader_response = ...
```

Above, each tag corresponds to a different party index i and executes the tag_main⟨@i⟩() function. The reader returns an Ok with data x only if it has successfully authenticated the tag (i.e., if the MAC verification is successful). The refinement on x's type means that, if the MAC key was not corrupted, then tag_i ran (encoded using OWL's happened predicate) and was the one trying to authenticate itself. Note that the MAC verification may still succeed even if the key has been corrupted. However, in this case, an adversary-controlled tag may have sent the MAC, and hence the refinement does not (and cannot!) guarantee happened(tag⟨i⟩()).

Notably, all of the OWL protocols verify automatically, whereas in Squirrel, considerable manual effort is required from the developer (as hinted at by the differences in line counts between the two tools). However, unlike Squirrel, we do not yet prove unlinkability [4] between sessions.

Authentication Protocols. We also verify a variety of traditional authentication protocols, primarily based on CryptoVerif's case studies. These protocols are typically hierarchically designed, using pre-shared keys to guarantee the integrity of fresh session keys, which are then used for secure communication between authenticated parties.

A particularly complex example is Kerberos [50]. A client first authenticates to the Authentication Service (AS) to obtain a Ticket Granting Ticket (TGT). This authentication process requires a secure channel between the client and AS, which can be established via a pre-shared symmetric key (generated from client passwords), or via a PKI (using Kerberos' PKINIT extension). The client then uses the TGT to interact with a Ticket Granting Server (TGS) and obtain a Service Ticket. Finally, it uses the Service Ticket to securely interact with a Service.

Compared with CryptoVerif and Squirrel, OWL verifies Kerberos in a modular fashion. Specifically, we first write a module interface for the client's interaction with the AS. We then verify that both an implementation based on a pre-shared symmetric key and one based on a PKI verify successfully against this interface. Finally, we verify the rest of Kerberos using the interface; i.e., the rest of the protocol is agnostic as to whether the AS exchange used symmetric or public keys. OWL proves end-to-end authentication by guaranteeing that the client and Service obtain the same session key at the end of the protocol. This is succinctly represented via the return type of each party's procedure. Unlike CryptoVerif, however, OWL does not yet guarantee the freshness of the shared key. This requires proving a bijection from the session index of the authentication server to the session indices of the authenticating parties.

Squirrel provides another interesting case study, the Private Authentication protocol [9], which illustrates how OWL's support for extraction catches protocol descriptions that elide important implementation details. We first verified this protocol as specified by Squirrel: the parties each start by encrypting a message using public-key encryption. However, OWL's extraction complained because the messages being encrypted were inherently too long for the public-key cipher we use (2048-bit RSA with OAEP). We then extended our version with the usual hybrid encryption technique; i.e., each party encrypts an ephemeral symmetric key with the other party's public key and then encrypts the message with the symmetric key. OWL successfully extracted and ran this new version. As discussed in §5.2, this illustrates the risks of verifying protocols that are so abstract they elide important details (e.g., length restrictions on public-key cipher messages), and it underscores the usefulness of extraction as a prototyping tool for protocol design.

When comparing the protocols in CryptoVerif vs. OWL, we find both tools achieve comparable automation. Some OWL implementations are larger, due in part to implementation details (like struct definitions) to facilitate extraction. However, we have found automation in OWL to be particularly robust compared to CryptoVerif, as typing errors in one party's code will not cause the other party to fail to typecheck. This modularity of verification effort is not available in CryptoVerif. An additional qualitative difference from CryptoVerif is that secrecy properties in OWL are guaranteed for free via its information flow type system, while the secrecy of each piece of data must be queried for individually in CryptoVerif.

DH Protocols. The Diffie-Hellman protocol [38] allows two parties to securely establish a shared key over an adversary-controlled channel. Using OWL's support for DH primitives (e.g., modular exponentiation/elliptic-curve multiplication) and random oracles, we specify this key-exchange protocol as well the SSH [67] key exchange protocol with a forwarding agent. The latter is essentially two rounds of the DH key-exchange performed in sequence. The first key-exchange sets up a secure channel between the client and the forwarding agent, and the second sets up a secure channel between the forwarding agent and server using the key generated from the first-exchange.

Unlike Squirrel, OWL's support for modularity allows us to verify these key-exchanges independently of each other. For each exchange, we prove that both parties successfully receive the same key and that the keys are secure, subject to whether any signing keys (used by the parties to authenticate their DH public key during the exchange) are corrupt.

7 Conclusions

We present OWL, the first formal tool for analyzing security protocols that simultaneously achieves automation, modularity, and computational security. Rather than whole-protocol techniques, OWL relies on information flow types. We evaluate OWL on a number of case studies and show that it is competitive with related tools [7, 23]. Finally, OWL's prototype extraction pipeline to Rust produces executable implementations.

Acknowledgements

This work was funded in part by National Science Foundation (NSF) Grant No. 2224279, a fellowship from the Alfred P. Sloan Foundation, and grants from the Intel Corporation and Rolls-Royce. Sydney Gibson was also funded by the NSF Graduate Research Fellowship Program under Grant No. DGE1745016.

References

[1] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2), 2002.

[2] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *ACM CCS*, 2015.

[3] M. Albrecht, K. Paterson, and G. Watson. Plaintext recovery attacks against SSH. In *Proc. IEEE S&P*, May 2009.

[4] M. Arapinis, T. Chothia, E. Ritter, and M. Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *2010 23rd IEEE CSF*, 2010.

[5] O. Arden, J. Liu, and A. C. Myers. Flow-limited authorization. In *2015 IEEE 28th CSF*, pages 569–583, 2015.

[6] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. *Theoretical Computer Science*, 402(2), 2008. Trustworthy Global Computing.

[7] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and S. Moreau. An interactive prover for protocol verification in the computational model. In *Proceedings of the IEEE S&P*, May 2021.

[8] G. Bana and H. Comon-Lundh. Towards unconditional soundness: Computationally complete symbolic attacker. In *Proceedings of the Conference on Principles of Security and Trust (POST)*, 2012.

[9] G. Bana and H. Comon-Lundh. A computationally complete symbolic attacker for equivalence properties. In *Proceedings of the ACM CCS*, 2014.

[10] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. SoK: Computer-aided cryptography. In *Proc. IEEE S&P*, May 2021.

[11] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub. Easycrypt: A tutorial. *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*, pages 146–166, 2014.

[12] G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the ACM POPL*. ACM, 2009.

[13] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of IACR CRYPTO*, 2011.

[14] M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval. Key-privacy in public-key encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2001.

[15] M. Bellare, T. Kohno, and C. Namprempre. Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the encode-then-encrypt-and-MAC paradigm. *ACM Transactions on Information and System Security*, 1, 2004.

[16] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Proceedings of AsiaCrypt 2000*, 2000.

[17] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *IEEE S&P*, 2015.

[18] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele. DY* : A modular symbolic verification framework for executable cryptographic protocol code. In *IEEE EuroS&P*, Sept. 2021.

[19] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE S&P*, May 2017.

[20] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, and J. K. Zinzindohoué. Implementing and proving the TLS 1.3 record layer. In *Proceedings of the IEEE S&P*, 2017.

[21] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *Proceedings of the IEEE S&P*, 2013.

[22] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin. Proving the TLS handshake secure (as it is). In *Annual Cryptology Conference*, pages 235–255. Springer, 2014.

[23] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, 2008.

[24] B. Blanchet. Security protocol verification: Symbolic and computational models. In *Proceedings of the Conference on Principles of Security and Trust (POST)*, 2012.

[25] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1, Oct. 2016.

[26] J. Brendel, M. Fischlin, F. Günther, and C. Janson. Prf-odh: Relations, instantiations, and impossibility results. Cryptology ePrint Archive, Paper 2017/517, 2017. https://eprint.iacr.org/2017/517.

[27] M. Bruso, K. Chatzikokolakis, and J. Den Hartog. Formal verification of privacy for RFID systems. In *2010 23rd IEEE CSF*. IEEE, 2010.

[28] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report 39, DEC Systems Research Center, Feb. 1989.

[29] R. Canetti and M. Fischlin. Universally composable commitments. In *Advances in Cryptology—CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceed-*

ings 21, pages 19–40. Springer, 2001.

[30] R. Canetti, A. Stoughton, and M. Varia. EasyUC: Using EasyCrypt to mechanize proofs of universally composable security. In *Proceedings of the IEEE CSF*, 2019.

[31] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila. A formal security analysis of the Signal messaging protocol. In *Proc. IEEE EuroS&P*, 2017.

[32] V. Cortier, S. Kremer, and B. Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reasoning*, 46(3-4), 2011.

[33] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the ACM CCS*, 2017.

[34] C. Cremers, M. Horvat, S. Scott, and T. v. d. Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *Proceedings of the IEEE S&P*, May 2016.

[35] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.

[36] A. Delignat-Lavaud, C. Fournet, B. Parno, J. Protzenko, T. Ramananandro, J. Bosamiya, J. Lallemand, I. Rakotonirina, and Y. Zhou. A security model and fully verified implementation for the IETF QUIC record layer. In *Proceedings of the IEEE S&P*, May 2021.

[37] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8):533–536, aug 1981.

[38] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22, Nov. 1976.

[39] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29, 1983.

[40] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 71–80. IEEE, 2009.

[41] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong authentication for RFID systems using the AES algorithm. In *International workshop on cryptographic hardware and embedded systems*. Springer, 2004.

[42] C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In *Proc. ACM CCS*, 2011.

[43] C. Fournet, J. Planul, and T. Rezk. Information-flow types for homomorphic encryptions. In *Proc. ACM CCS*, 2011.

[44] L. Hirschi, D. Baelde, and S. Delaune. A method for unbounded verification of privacy-type properties. *Journal of Computer Security*, 27(3), 2019.

[45] S. Ho, J. Protzenko, A. Bichhawat, and K. Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. In *Proceedings of the IEEE S&P*, May 2022.

[46] R. Jhala, N. Vazou, et al. Refinement types: A tutorial. *Foundations and Trends® in Programming Languages*, 6(3–4):159–317, 2021.

[47] A. Juels and S. A. Weis. Defining strong privacy for rfid. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–23, 2009.

[48] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *Proceedings of the IEEE EuroS&P*, 2017.

[49] N. Koblitz and A. J. Menezes. Another look at "provable security". *Journal of Cryptology*, 20(1), 2007.

[50] J. Kohl and B. C. Neuman. The Kerberos Network Authentication Service (V5). RFC 1510, Sept. 1993.

[51] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *Proceedings of IACR CRYPTO*, 2001.

[52] P. Laud. On the computational soundness of cryptographically masked flows. In *Proceedings of the ACM POPL*, 2008.

[53] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[54] B. Lipp, B. Blanchet, and K. Bhargavan. A mechanised cryptographic proof of the WireGuard virtual private network protocol. In *Proc. IEEE EuroS&P*, 2019.

[55] A. Lochbihler and S. R. Sefidgar. A tutorial introduction to CryptHOL. Cryptology ePrint Archive, Report 2018/941.

[56] D. A. McGrew and J. Viega. The security and performance of the Galois/counter mode of operation. In *Proc. (INDOCRYPT)*, 2004.

[57] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Proc. (CAV)*, 2013.

[58] D. Molnar and D. Wagner. Privacy and security in library RFID: Issues, practices, and architectures. In *Proceedings of the 11th ACM CCS*, 2004.

[59] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), 1978.

[60] D. Otway and O. Rees. Efficient and timely mutual authentication. *ACM SIGOPS Operating Systems Review*, 21(1):8–10, 1987.

[61] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *Proc. IACR ASIACRYPT*, 2011.

[62] A. Petcher and G. Morrisett. The foundational cryptography framework. In *Proceedings of the Conference on Principles of Security and Trust (POST)*, 2015.

[63] M. Rosulek. The joy of cryptography. https://joyofcryptography.com.

[64] Rust Development Team. *The Rust programming language*, 2022.

[65] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 21, 2003.

$$\frac{}{\Sigma \vdash \mathsf{Name}(\mathsf{n})\ \text{parsable}} \qquad \frac{}{\Sigma \vdash \mathsf{Unit}\ \text{parsable}}$$

$$\frac{\Sigma \vdash \tau\ \text{parsable} \qquad \Sigma \vdash \sigma\ \text{parsable}}{\Sigma \vdash \tau \times \sigma\ \text{parsable}} \qquad \frac{\Sigma \vdash \tau\ \text{parsable}}{\Sigma \vdash x : \tau\{\phi\}\ \text{parsable}}$$

$$\frac{\Sigma \vdash \tau\ \text{parsable} \qquad \Sigma \vdash \sigma\ \text{parsable}}{\Sigma \vdash \tau + \sigma\ \text{parsable}}$$

$$\mathsf{bdry}_{\mathsf{Name}(\mathsf{n})}(x) = |\mathsf{n}| \qquad \mathsf{bdry}_{\mathsf{Unit}}(x) = 1$$

$$\mathsf{bdry}_{a:\tau\{\phi\}}(x) = \mathsf{bdry}_{\tau}(x)$$

$$\mathsf{bdry}_{\tau\times\sigma}(x) = \mathsf{bdry}_{\sigma}(x[\mathsf{bdry}_{\tau}(x)\ldots]) + \mathsf{bdry}_{\tau}(x)$$

$$\mathsf{bdry}_{\tau+\sigma}(x) = \begin{cases} 1 + \mathsf{bdry}_{\tau}(y) & \text{if } x = 0y \\ 1 + \mathsf{bdry}_{\tau}(y) & \text{if } x = 1y \end{cases}$$

$$\mathsf{fst}_{\tau}(x) = \begin{cases} 1\ x[\ldots \mathsf{bdry}_{\tau}(x)] & \text{if } \mathsf{bdry}_{\tau}(x) \text{ defined} \\ 00 & \text{otherwise} \end{cases}$$

$$\mathsf{snd}_{\tau}(x) = \begin{cases} 1\ x[\mathsf{bdry}_{\tau}(x)\ldots] & \text{if } \mathsf{bdry}_{\tau}(x) \text{ defined} \\ 00 & \text{otherwise} \end{cases}$$

$$\mathsf{pair}_{\tau}(x,y) = \begin{cases} 1\ x +\!\!+ y & \text{if } \mathsf{bdry}_{\tau}(x) = |x| \\ 00 & \text{otherwise} \end{cases}$$

Figure 14: Parsability of types, and specification of $\mathsf{fst}_\tau/\mathsf{snd}_\tau/\mathsf{pair}_\tau$ for parsable types.

[66] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the ACM POPL*, 2016.

[67] T. Ylonen. The secure shell (SSH) transport layer protocol. RFC 4253, 2006.

A Standard Interpretations

Here, we state the axioms which all interpretations for OwlLang must satisfy to guarantee security.

Parsing of Products. First, we require that fst_τ, snd_τ, and pair_τ satisfy the equations of Figure 14. To extract out both components v_1, v_2 of a pair $v_1 +\!\!+ v_2$ of type $\tau \times \sigma$, we need to compute the *boundary* for where v_1 resides in the pair. This is computed by $\mathsf{bdry}_\tau(v)$, which returns the index of the boundary in $v : \tau \times \sigma$, if it is well-defined. The calculation bdry_τ is undefined if $\tau = \mathsf{Data}(\ell, \ell')$, as this type carries no guarantee about its length. In the type system, we use the judgement $\Sigma \vdash \tau$ parsable to guarantee that $\mathsf{bdry}_\tau(v)$ is defined. In turn, we define all pairing/unpairing operations in terms of bdry_τ. For pairing, we check that the first argument x satisfies $\mathsf{bdry}_\tau(x) = |x|$, so that it can be unpaired later; for unpairing, we check that bdry_τ is defined on the input, so that we can divide the pair accordingly.

Additional Axioms. We assume the following for constructing sum types, the zeroes operation Z, and name types:

$$[\![\mathsf{inl}]\!](x) = 0x \qquad [\![\mathsf{inr}]\!](x) = 1x \qquad [\![\mathsf{Z}]\!](x) = 0^{|x|}$$

$$[\![\mathsf{nonce}]\!] = [\![\mathsf{enckey}]\!] = \text{uniform over } \mathsf{L}_{\mathsf{hash}}$$

The last constraint above guarantees that the output of the random oracle can be used as a valid nonce or encryption key.

B Typing Rules for Additional Primitives

The typing rules for MACs, digital signatures, and public-key encryption are given in Figure 15. To model signatures, we make use of an additional singleton type, $\mathsf{VK}(\mathsf{n})$, for the verification key corresponding to n, with label $\lceil \mathsf{VK}(\mathsf{n}) \rceil = |\mathsf{VK}(\mathsf{n})| = \bot$. Values of type $\mathsf{VK}(\mathsf{n})$ are only created by the computation $\mathsf{vk}(a)$, when $a : \mathsf{Name}(\mathsf{n})$ and $\Sigma \vdash \mathsf{n} : \mathsf{sigkey}^I\ \tau$. The rules for MACs are similar to those for digital signatures, and guarantee security if the underlying MAC is unforgeable [63].

The typing rule for public-key encryption is more interesting, as the cryptosystem is not authenticated. The rule for encryption is similar to the one for symmetric encryption, but using a singleton type $\mathsf{PK}(\mathsf{n})$ for the public encryption key. The rule for decryption is written in continuation passing style, as the continuation needs to be checked under *two* different return types for $\mathsf{pkdec}(k,c)$: either the result has type $\mathsf{Unit} + \tau$ or has type $\mathsf{Unit} + \mathsf{Data}(\ell_{\mathcal{A}}, \ell_{\mathcal{A}})$. The two cases correspond to the two possible results of decryption: the plaintext either came from the protocol itself, or it originated from the adversary. Our typing rules for public-key encryption imply security when the cryptosystem is CCA secure [63], and require the set of oracles $\mathsf{Orcl}(\Sigma, \ell_{\mathcal{A}}, N)$ to allow the simulator to obtain public encryption keys.

C Proof of Theorem 1

In this section, we give a full proof of soundness for the core language of Owl. To do so, first fix a secure family of interpretations I_λ. We leave the interpretation implicit throughout.

Scope of Proof. While the proof of security Owl is intended to be extensible to handle a variety of cryptographic operations, we focus in this section on proving security for hashing, symmetric (authenticated) encryption, and Diffie-Hellman operations. We discuss in §C.5 how our proof can be lifted to handle extra operations.

Relation Lifting. To reason about the semantics of expressions, we use *relation lifting* [11, 62], which extends non-probabilistic predicates to operate over probability distributions.

Definition 8 (Lifting)**.** *Suppose $R \subseteq A \times B$ is a relation, and D_1, D_2 are distributions over A and B, respectively. Then, $\vDash D_1 \sim_{\mathcal{L}}$*

$$\frac{\Sigma;\Gamma \vdash k : \mathsf{Name}(\mathsf{n}) \quad \Sigma \vdash \mathsf{n} : \mathsf{sigkey}^{\mathsf{Real}}\ \tau \quad \Sigma;\Gamma \vdash x : \tau \quad \Sigma \vdash \lceil \tau \rceil \leq \ell_{\mathcal{A}}}{\Sigma;\mathcal{T};\Gamma;\ell_{\mathcal{A}} \vdash \mathsf{sign}(k,x) : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}})}\ \textsc{sign}$$

$$\frac{\Sigma;\Gamma \vdash k : \mathsf{VK}(\mathsf{n}) \quad \Sigma \vdash \mathsf{n} : \mathsf{sigkey}^{\mathsf{Real}}\ \tau \quad \Sigma;\Gamma \vdash x : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}}) \quad \Sigma;\Gamma \vdash t : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}}) \quad \Sigma \not\vdash [\mathsf{n}] \leq \ell_{\mathcal{A}}}{\Sigma;\mathcal{T};\Gamma;\ell_{\mathcal{A}} \vdash \mathsf{vrfy}(k,x,t) : \mathsf{Unit}+\tau}\ \textsc{vrfy}$$

$$\frac{\Sigma;\Gamma \vdash k : \mathsf{Name}(\mathsf{n}) \quad \Sigma \vdash \mathsf{n} : \mathsf{mackey}^{\mathsf{real}}\tau \quad \Sigma;\Gamma \vdash x : \tau \quad \Sigma \vdash \lceil \tau \rceil \leq \ell_{\mathcal{A}}}{\Sigma;\mathcal{T};\Gamma;\ell_{\mathcal{A}} \vdash \mathsf{mac}(k,x) : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}})}\ \textsc{mac}$$

$$\frac{\Sigma;\Gamma \vdash k : \mathsf{Name}(\mathsf{n}) \quad \Sigma \vdash \mathsf{n} : \mathsf{mackey}^{\mathsf{real}}\tau \quad \Sigma;\Gamma \vdash x : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}}) \quad \Sigma;\Gamma \vdash t : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}}) \quad \Sigma \not\vdash [\mathsf{n}] \leq \ell_{\mathcal{A}}}{\Sigma;\mathcal{T};\Gamma;\ell_{\mathcal{A}} \vdash \mathsf{mvrfy}(k,x,t) : \mathsf{Unit}+\tau}\ \textsc{vrfy}$$

$$\frac{\Sigma;\Gamma \vdash k : \mathsf{PK}(\mathsf{n}) \quad \Sigma \vdash \mathsf{n} : \mathsf{pkekey}^{\mathsf{real}}\tau \quad \Sigma;\Gamma \vdash x : \tau \quad \Sigma \vdash |\tau| \leq \ell_{\mathcal{A}}}{\Sigma;\mathcal{T};\Gamma;\ell_{\mathcal{A}} \vdash \mathsf{pkenc}(k,x) : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}})}\ \textsc{pkenc}$$

$$\frac{\Sigma;\Gamma \vdash k : \mathsf{Name}(\mathsf{n}) \quad \Sigma \vdash \mathsf{n} : \mathsf{pkekey}^{\mathsf{real}}\tau \quad \Sigma;\Gamma \vdash c : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}}) \quad \Sigma \not\vdash [\mathsf{n}] \leq \ell_{\mathcal{A}} \quad \Sigma;\mathcal{T};\Gamma,x : \mathsf{Unit}+\tau;\ell_{\mathcal{A}} \vdash e : \sigma \quad \Sigma;\mathcal{T};\Gamma,x : \mathsf{Unit}+\mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}});\ell_{\mathcal{A}} \vdash e : \sigma}{\Sigma;\mathcal{T};\Gamma;\mathsf{pc};\ell_{\mathcal{A}} \vdash \mathsf{let}\ x = \mathsf{pkdec}(k,c)\ \mathsf{in}\ e : \sigma}\ \textsc{dec}$$

Figure 15: **Typing for MACs, Signatures, and PKE**.

$D_2 : R$ *only if*

$$\begin{aligned}&\exists C : \mathsf{Dist}(A \times B).\\ &\quad D_1 = (x \xleftarrow{\$} C;\ \mathsf{Ret}(\mathsf{fst}(x))) \wedge\\ &\quad D_2 = (x \xleftarrow{\$} C;\ \mathsf{Ret}(\mathsf{snd}(x))) \wedge\\ &\quad \forall x,y.\ C(x,y) > 0 \implies (x,y) \in R.\end{aligned}$$

General Logical Relation. To carry out each cryptographic reduction step and reason about ideal name contexts, we make use of a general form of *logical relation* to prove various integrity and noninterference properties by induction on the type structure.

We are now ready to define our general form of logical relation:

Definition 9 (General Logical Relation)**.** *Suppose $\mathcal{T}$ is a table context, N_1 and N_2 are name environments, and and $\mathcal{V}^\tau \subset (\{0,1\}^* \times \mathsf{World}) \times (\{0,1\}^* \times \mathsf{World})$ is a binary relation on pairs of values and worlds, indexed by types, τ. Then, for closed expressions e_1 and e_2 and worlds W_1 and W_2, we define $\mathcal{T};\mathcal{V};N_1,N_2 \vDash \langle W_1,e_1\rangle \sim \langle W_2,e_2\rangle : \tau$ coinductively as in Figure 16.*

Intuitively, if $\mathcal{T};\mathcal{V};N_1,N_2 \vDash \langle W_1,e_1\rangle \sim \langle W_2,e_2\rangle : \tau$, then we are guaranteed that: the input/output behaviors of the two programs are identical; whenever W_1 and W_2 are related, all future worlds generated by the two programs are still related; and the final outputs of the two programs are related by $\mathcal{V}^\tau$. To model the monotonicity of the random oracle, we consider W' a valid future of W, written $W \leq_{\mathsf{RO}} W'$, whenenver $W'[\mathsf{RO},\cdot]$ contains $W[\mathsf{RO},\cdot]$. We relate two worlds W_1 and W_2 under $\mathcal{V}$, written $\mathcal{V} \vDash W_1 \sim W_2$, if, for any $(T : \tau) \in \mathcal{T}$ and any x, $W_1[T,x]$ and $W_2[T,x]$ are either both undefined, or defined and related by $\mathcal{V}^\tau$.

This relation is specifically designed to be *robust*, in the sense that if the two programs are interrupted by other parts of their respective protocols (or the adversary), then the logical relation still holds as long as the two resulting worlds are monotonically increasing w.r.t $\leq_{\mathsf{RO}}$ and still related.

Reasoning about the Logical Relation. While programs in OwlLang are semantically simple, we opt to use a coinductive logical relation (rather than a step-indexed one [40]) for technical convenience and generality for future language extensions.

The interactive nature of protocols in OwlLang requires us to reason about arbitrary interleavings of parties. To support interleavings, we require our value relation to be *stable*. Stable value relations guarantee that, if an intermediate computation results in a well-formed value relative to the current world, it continues to be well-formed in all future worlds:

Definition 10 (Stable Value Relations)**.** *The relation $\mathcal{V}$ is $\mathcal{T}$-*stable *if, whenever $\mathcal{V}^\tau((W_1,v_1),(W_2,v_2))$ and $W_1 \leq_{\mathsf{RO}} W_1'$, $W_2 \leq_{\mathsf{RO}} W_2'$ and $\mathcal{T};\mathcal{V} \vDash W_1' \sim W_2'$, we have that $\mathcal{V}^\tau((W_1',v_1),(W_2',v_2))$.*

Crucially, our logical relation does *not* require complex coinductive proof techniques. Instead, we are able to localize all required coinductive reasoning to two proof rules: one for ret statements, and one for let statements.

Lemma 1 (Logical Relation for Return)**.** *Suppose that $\mathcal{V}$ is $\mathcal{T}$-stable, $\mathcal{T};\mathcal{V} \vDash W_1 \sim W_2$, and that $\mathcal{V}^\tau((\llbracket a_1 \rrbracket^{N_1},W_1),(\llbracket a_2 \rrbracket^{N_2},W_2))$. Then, $\mathcal{T};\mathcal{V};N_1,N_2 \vDash \langle\langle W_1,\mathsf{ret}(a_1)\rangle \sim \langle W_2,\mathsf{ret}(a_2)\rangle : \tau$.*

Proof. By coinduction, generalizing a_1 and a_2. We are able to continue the coinduction by making use of the fact that $\mathcal{V}$ is stable. ∎

Lemma 2 (Logical Relation for Let)**.** *Suppose that $\mathcal{V}$ is $\mathcal{T}$-stable, $\mathcal{T};\mathcal{V};N_1,N_2 \vDash \langle W_1,e_1\rangle : \tau$, and that for all v_1,v_2,W_1',W_2', we have that if $\mathcal{V}^\tau((v_1,W_1'),(v_2,W_2'))$, $\mathcal{T};\mathcal{V} \vDash W_1',W_2'$, $W_1 \leq_{\mathsf{RO}} W_1'$, and $W_2 \leq_{\mathsf{RO}} W_2'$, then $\mathcal{T};\mathcal{V};N_1,N_2 \vDash \langle W_1',k_1[v_1/x]\rangle \sim \langle W_2',k_2[v_2/x]\rangle : \sigma$, where each k_i has one free variable, x. Then, we have that $\mathcal{T};\mathcal{V};N_1,N_2 \vDash \langle W_1,\mathsf{let}\ x = e_1\ \mathsf{in}\ k_1\rangle \sim \langle W_2,\mathsf{let}\ x = e_2\ \mathsf{in}\ k_2\rangle : \sigma$.*

Proof. Also by coinduction, generalizing over the e_i and W_i. By hypothesis, we know that e_1,e_2 are both either of the form $\mathsf{ret}(a_i)$, where $\llbracket a_i \rrbracket^{N_i}$ are related by $\mathcal{V}^\tau$, or both can take a step and continue to preserve the logical relation. In the former case, we apply the hypothesis on the k_i; in the latter case, we continue the coinduction with a reduced e_1',e_2', and altered worlds W_1',W_2'. ∎

$$\mathcal{T};\mathcal{V};N_1,N_2 \vDash \langle W_1,e_1\rangle \sim \langle W_2,e_2\rangle : \tau := \bigwedge \begin{cases} (\exists a_1.\ e_1 = \mathsf{ret}(a_1)) \iff (\exists a_2.\ e_2 = \mathsf{ret}(a_2)) \\ \forall a_1,a_2.\ e_1 = \mathsf{ret}(a_1) \wedge e_2 = \mathsf{ret}(a_2) \implies \mathcal{V}^{\tau}((\llbracket a_1 \rrbracket^{N_1}, W_1),(\llbracket a_2 \rrbracket^{N_2}, W_2)) \\ \forall i\ W_1'\ W_2', \\ \quad W_1 \leq_{\mathsf{RO}} W_1' \wedge W_2 \leq_{\mathsf{RO}} W_2' \wedge \mathcal{T};\mathcal{V};N_1,N_2 \vDash W_1 \sim W_2 \\ \quad \vDash \llbracket e_1 \rrbracket^{N_1}(W_1',i) \sim_{\mathcal{L}} \llbracket e_2 \rrbracket^{N_2}(W_2',i) : \lambda(e_1',W_1'',o_1)\ (e_2',W_2'',o_2). \\ \quad \bigwedge \begin{cases} o_1 = o_2 \\ \mathcal{T};\mathcal{V};N_1,N_2 \vDash W_1'' \sim W_2'' \\ W_1' \leq_{\mathsf{RO}} W_1'' \\ W_2' \leq_{\mathsf{RO}} W_2'' \\ \mathcal{T};\mathcal{V};N_1,N_2 \vDash \langle W_1'',e_1'\rangle \sim \langle W_2'',e_2'\rangle : \tau \end{cases} \end{cases}$$

$$W \leq_{\mathsf{RO}} W' := \forall x,\ W[\mathsf{RO},x] \neq \bot \implies W'[\mathsf{RO},x] = W[\mathsf{RO},x]$$

$$\mathcal{T};\mathcal{V} \vDash W \sim W' := W[\mathsf{RO},\cdot] = W'[\mathsf{RO},\cdot] \wedge$$
$$\forall (T:\tau) \in \mathcal{T}, \forall x, (W[T,x] = W'[T,x] = \bot) \vee (W[T,x] \neq \bot \wedge W'[T,x] \neq \bot \wedge \mathcal{V}^{\tau}((W[T,x],W),(W'[T,x],W')))$$

Figure 16: Generalized Logical Relation for the core language.

All other language constructs (e.g., cryptographic operations, input/output, and case splits) do not require coinductive reasoning. Instead, it suffices to finitely unroll the logical relation, and use a suitable hypothesis to continue the proof. For example, we have that $\mathcal{T};\mathcal{V};N_1,N_2 \vDash \langle W_1,\mathsf{input}\rangle \sim \langle W_2,\mathsf{input}\rangle : \tau$ whenever we have that for all i and W, $\mathcal{V}^{\tau}((W,i),(W,i))$.

C.1 Non-Cryptographic Security

We begin with the final component of the proof, which establishes security after all possible cryptographic operations have been idealized under $\ell_{\mathcal{A}}$. This happens when the name context is *ideal*:

Definition 11 (Ideal Name Context)**.** *A name context $\vdash \Sigma$ is $\ell_{\mathcal{A}}$-ideal if:*

- *whenever $H(a) \mapsto_P \mathsf{nt}$, $\Sigma;\ell_{\mathcal{A}} \not\vdash \mathsf{unsolvable}\ P$;*
- *whenever $\mathsf{n} : \mathsf{enckey}^{\mathsf{Real}}\ \tau \in \Sigma$, $\Sigma \vdash [\mathsf{n}] \leq \ell_{\mathcal{A}}$.*

Ideal name contexts only contain non-idealized keys and hashes whenever their corresponding names and hash problems, respectively, are controlled by the adversary label $\ell_{\mathcal{A}}$. If Σ is $\ell_{\mathcal{A}}$-ideal, we see that no typing rules with non-trivial information flows (e.g., encryption, HASH − pat) are typable. Thus, security of configurations typed under Σ only depends on type soundness of the core rules in Figure 19.

We now show security for ideal name contexts.

Theorem 2 (Ideal Security)**.** *Suppose Σ is $\ell_{\mathcal{A}}$-ideal, $\Sigma;\ell_{\mathcal{A}} \vdash N_1 \sim N_2$, and $\Sigma;\mathcal{T};\ell_{\mathcal{A}};\cdot \vdash K : \{\tau_i\}$. Then, for any secure PPT interpretation I_λ, we have that $I_\lambda;\Sigma;\ell_{\mathcal{A}};\{\tau_i\} \vDash \mathcal{K}$.*

We do so by instantiating our general logical relation in Figure 16. To do so, we define a value relation, $\mathcal{V}_{\mathsf{Ideal}}$:

Definition 12 (Ideal Value Relation)**.** *Given name context Σ, adversary label $\ell_{\mathcal{A}}$, and name environments N_1,N_2, we define the value relation $\mathcal{V}^{\tau}_{\mathsf{Ideal};\Sigma,\ell_{\mathcal{A}},N_1,N_2}((W_1,v_1),(W_2,v_2))$ to be the judgement $\Sigma;\ell_{\mathcal{A}};N_1,N_2 \vdash v_1 \sim_{\mathsf{Ideal}} v_2 : \tau$, defined in Figure 17.*

For our logical relation to be sound, we need to consider *low-equivalent* name environments, which require public names to be equal, and Diffie-Hellman keys to only differ if the key is idealized:

Definition 13 (Low Equivalence for Name Environments)**.** *We say that $\Sigma;\ell_{\mathcal{A}} \vdash N_1 \sim N_2$ if, for all n:*

- *if $\Sigma \vdash [\mathsf{n}] \leq \ell_{\mathcal{A}}$, then $N_1(\mathsf{n}) = N_2(\mathsf{n})$;*
- *if $\mathsf{n} : \mathsf{DH}^I \in \Sigma$, then $N_1(\mathsf{n}) = N_2(\mathsf{n})$ or $I = \mathsf{Ideal}$.*

Additionally, we need to define when substitutions are related:

Definition 14 (Related Substitutions)**.** *Suppose that γ,γ' map variables in Γ to bitstrings. Then, we say that $\mathcal{V} \vDash \langle W,\gamma\rangle \sim \langle W',\gamma'\rangle : \Gamma$ if for all x in the domain of the Γ, $\mathcal{V}^{\Gamma(x)}((W,\gamma(x)),(W',\gamma'(x)))$.*

We show soundness of logical relations in two steps. First, we have soundness for atomic expressions:

Lemma 3 (Ideal Logical Relation for Atomic Expressions)**.** *Suppose Σ is $\ell_{\mathcal{A}}$-ideal, $\Sigma;\ell_{\mathcal{A}} \vdash N_1 \sim N_2$, and $\Sigma;\Gamma \vdash a : \tau$. Then, for all $\mathcal{T};\mathcal{V}_{\mathsf{Ideal};\Sigma,\ell_{\mathcal{A}},N_1,N_2} \vDash W_1 \sim W_2$, and $\mathcal{V}_{\mathsf{Ideal};\Sigma,\ell_{\mathcal{A}},N_1,N_2} \vDash \langle W_1,\gamma_1\rangle \sim \langle W_2,\gamma_2\rangle : \Gamma$, we have that $\mathcal{V}^{\tau}_{\mathsf{Ideal};\Sigma,\ell_{\mathcal{A}},N_1,N_2}((W_1,\llbracket \gamma_1(a_1) \rrbracket^{N_1}),(W_2,\llbracket \gamma_2(a_2) \rrbracket^{N_2}))$.*

Proof. By induction on the typing derivation. ■

Next, we have soundness for all expressions:

Lemma 4 (Ideal Logical Relation)**.** *Suppose Σ is $\ell_{\mathcal{A}}$-ideal, $\Sigma;\ell_{\mathcal{A}} \vdash N_1 \sim N_2$, and $\Sigma;\mathcal{T};\ell_{\mathcal{A}};\Gamma \vdash e : \tau$. Then, if $\mathcal{T};\mathcal{V}_{\mathsf{Ideal};\Sigma,\ell_{\mathcal{A}},N_1,N_2} \vDash W_1 \sim W_2$, and $\mathcal{V}_{\mathsf{Ideal};\Sigma,\ell_{\mathcal{A}},N_1,N_2} \vDash \langle W_1,\gamma_1\rangle \sim \langle W_2,\gamma_2\rangle : \Gamma$, we have that $\mathcal{T};\mathcal{V}_{\mathsf{Ideal};\Sigma,\ell_{\mathcal{A}},N_1,N_2};N_1,N_2 \vDash \langle W_1,\gamma_1(e)\rangle \sim \langle W_2,\gamma_2(e)\rangle$.*

Proof. Note that $\mathcal{V}_{\mathsf{Ideal};\Sigma,\ell_{\mathcal{A}},N_1,N_2}$ does not make use of its world arguments; this is because they are only used to validate well-typed random oracle results, which are excluded since Σ is ideal. This automatically guarantees that $\mathcal{V}_{\mathsf{Ideal};\Sigma,\ell_{\mathcal{A}},N_1,N_2}$ is $\mathcal{T}$-stable.

The proof is by induction on the typing derivation. Throughout, we use Lemma 3 to prove that all arithmetic expressions result in well-formed and related values. The ret and let cases are handled by Lemma 1 and Lemma 2, respectively. Note that since Σ is ideal, none of the typing rules in Figure 12 apply. ■

To prove Theorem 2, we additionally need a result which says that the use of public keys in ideal contexts does not harm simulatability, since these public keys are computable by the simulator:

Lemma 5 (Diffie-Hellman Public-Key Reduction)**.** *Suppose that* $\Sigma;\mathcal{T};\ell_{\mathcal{A}} \vdash \mathcal{K} : \{\tau_i\}$, $(\mathsf{n} : \mathsf{DH}^{\mathsf{Real}}) \in \Sigma$, *and* Σ *is an ideal name context (Definition 11). Then, there exists a configuration* $\mathcal{K}'$ *such that* $I_\lambda;\Sigma;\ell_{\mathcal{A}};\{\tau_i\} \vDash_{\mathsf{sim}} \mathcal{K}$ *if* $I_\lambda;\Sigma;\ell_{\mathcal{A}};\{\tau_i\} \vDash_{\mathsf{sim}} \mathcal{K}'$, *and* $\Sigma[\mathsf{n} \mapsto \mathsf{DH}^{\mathsf{Ideal}}];\mathcal{T};\ell_{\mathcal{A}} \vdash \mathcal{K} : \{\tau_i\}$.

Proof. We let $\mathcal{K}'$ be the same as $\mathcal{K}$, but all well-typed computations of $\mathsf{dhpk}(a)$ with $a : \mathsf{Name}(\mathsf{n})$ are replaced with $\mathsf{ret}(0)$. Now, suppose that $\mathcal{K}'$ satisfies simulatability. Then we have that $\mathcal{K}$ does as well: any adversary's view in $\mathcal{G}^{\Sigma,\{\tau_i\}}(N,\mathcal{K},\ell_{\mathcal{A}},\mathcal{A})$ can be simulated using the adversary's view in $\mathcal{G}^{\Sigma,\{\tau_i\}}(N,\mathcal{K}',\ell_{\mathcal{A}},\mathcal{A})$, along with the Diffie-Hellman public key $\mathsf{dhpk}(\mathsf{get}(\mathsf{n}))$, computable using $\mathsf{Orcl}(\Sigma,\ell_{\mathcal{A}})(N)$. ■

Proof of Theorem 2. From assumption, we obtain via Lemma 4 that if $\Sigma;\ell_{\mathcal{A}} \vdash N_1 \sim N_2$, then for all i, $\mathcal{T};\mathcal{V}^{\tau}_{\mathsf{Ideal};\Sigma,\ell_{\mathcal{A}},N_1,N_2};N_1,N_2 \vDash \langle\{\},\mathcal{K}_i\rangle \sim \langle\{\},\mathcal{K}_i\rangle : \tau_i$. We prove simulatability and correctness separately.

For correctness, we first show that, if $\mathcal{V}^{\tau}_{\mathsf{Ideal}}((W_1,v_1),(W_2,v_2))$, then $[\![\tau]\!]^{\Sigma,N,W_1}(v_1)$ and $[\![\tau]\!]^{\Sigma,N,W_2}(v_2)$. This is proven by induction on τ. The result then follows from our logical relation.

For simulatability, we must first reduce to the case where all names $\mathsf{n} : \mathsf{DH}^I$ are idealized. Since Σ is ideal, we are able to do this by repeatedly applying Lemma 5.

Now, assume that Σ is ideal, and for all $\mathsf{n} : \mathsf{DH}^I$ in Σ, $I = \mathsf{Ideal}$. For simulatability, we must obtain a simulator $\mathcal{S}^{\mathsf{Orcl}(\Sigma,\ell_{\mathcal{A}})(N)}$, which must output a decision bit b close to that of a given adversary $\mathcal{A}$. We define the simulator as follows: given oracle O, it first generates its own name environment, N', from $\mathsf{Gen}(\Sigma)$; then, it locally executes $\mathcal{G}^{\Sigma,\{\tau_i\}}_{I_\lambda}(N_{\mathcal{S}},\mathcal{K},\ell_{\mathcal{A}},\mathcal{A}_\lambda)$, where

$$N_{\mathcal{S}}(\mathsf{n}) = \begin{cases} O(\mathsf{get}(\mathsf{n})) & \text{if } \Sigma \vdash [\mathsf{n}] \leq \ell_{\mathcal{A}} \\ N'(\mathsf{n}) & \text{otherwise} \end{cases}$$

We have that, for any N and N', $\Sigma;\ell_{\mathcal{A}} \vdash N \sim N_{\mathcal{S}}$.

The simulator then outputs the decision bit given by the adversary in the security game with $N_{\mathcal{S}}$. We have that this decision bit is equal in distribution to the one obtained by $\mathcal{G}^{\Sigma,\{\tau_i\}}_{I_\lambda}(N,\mathcal{K},\ell_{\mathcal{A}},\mathcal{A}_\lambda)$.

C.2 Encryption Reduction

Now, we perform a generic cryptographic reduction in order to idealize a single encryption key in Σ. First, we define *ideal names*, which are names which may not participate in cryptographic operations.

$$\boxed{\Sigma;\ell_{\mathcal{A}};N_1,N_2 \vdash v_1 \sim_{\mathsf{Ideal}} v_2 : \tau}$$

$$\frac{\Sigma \vdash \ell \leq \ell_{\mathcal{A}} \implies v_1 = v_2 \qquad \Sigma \vdash \ell' \leq \ell_{\mathcal{A}} \implies |v_1| = |v_2|}{\Sigma;\ell_{\mathcal{A}};N_1,N_2 \vdash v_1 \sim_{\mathsf{Ideal}} v_2 : \mathsf{Data}(\ell,\ell')}$$

$$\frac{v_1 = N_1(\mathsf{n}) \qquad v_2 = N_2(\mathsf{n})}{\Sigma;\ell_{\mathcal{A}};N_1,N_2 \vdash v_1 \sim_{\mathsf{Ideal}} v_2 : \mathsf{Name}(\mathsf{n})}$$

$$\frac{v_1 = v_2 = 0}{\Sigma;\ell_{\mathcal{A}};N_1,N_2 \vdash v_1 \sim_{\mathsf{Ideal}} v_2 : \mathsf{Unit}}$$

$$\frac{\Sigma;\ell_{\mathcal{A}};N_1,N_2 \vdash v_1 \sim_{\mathsf{Ideal}} v_2 : \tau \qquad [\![\phi]\!]^{N_1}(v_1) \qquad [\![\phi]\!]^{N_2}(v_2)}{\Sigma;\ell_{\mathcal{A}};N_1,N_2 \vdash v_1 \sim_{\mathsf{Ideal}} v_2 : x : \phi\{\tau\}}$$

$$\frac{v_1 = bv_1' \qquad v_2 = bv_2' \qquad \Sigma;\ell_{\mathcal{A}};N_1,N_2 \vdash v_1 \sim_{\mathsf{Ideal}} v_2 : \tau_b}{\Sigma;\ell_{\mathcal{A}};N_1,N_2 \vdash v_1 \sim_{\mathsf{Ideal}} v_2 : \tau_0 + \tau_1}$$

$$\frac{\begin{array}{c}\mathsf{bdry}_\tau(v_1) \neq \bot \qquad \mathsf{bdry}_\tau(v_2) \neq \bot \\ \Sigma;\ell_{\mathcal{A}};N_1,N_2 \vdash v_1[\ldots \mathsf{bdry}_\tau(v_1)] \sim_{\mathsf{Ideal}} v_2[\ldots \mathsf{bdry}_\tau(v_2)] : \tau \\ \Sigma;\ell_{\mathcal{A}};N_1,N_2 \vdash v_1[\mathsf{bdry}_\tau(v_1)\ldots] \sim_{\mathsf{Ideal}} v_2[\mathsf{bdry}_\tau(v_2)\ldots] : \sigma\end{array}}{\Sigma;\ell_{\mathcal{A}};N_1,N_2 \vdash v_1 \sim_{\mathsf{Ideal}} v_2 : \tau \times \sigma}$$

Figure 17: Value Relations for Ideal Security

Definition 15 (Ideal Name)**.** *The name* $\Sigma \vdash \mathsf{n} : \mathsf{nt}$ *is* ideal *in* Σ *if* $\mathsf{nt} = \mathsf{nonce}$ *or of the form* $\mathsf{nk}^{\mathsf{Ideal}}\ \tau$, *for some cryptographic name kind* nk *(e.g.,* enckey*).*

Next, we define *high* names, which are names that cannot be *used* in a cryptographic operation.

Definition 16 (High Name)**.** *The name* n *is* high *in* Σ *if, for all* n', *if* $\Sigma \vdash [\mathsf{n}] \leq [\mathsf{n}']$, *then* $\mathsf{n} = \mathsf{n}'$ *or* n' *is ideal in* Σ.

To define security for the cryptographic reduction, we introduce *reducibility*:

Definition 17 (Reducibility)**.** *We say that* $\Sigma,\mathcal{K}$ *is* reducible *to* $\Sigma',\mathcal{K}'$ *under* I_λ, $\ell_{\mathcal{A}}$, *and* $\{\tau_i\}$ *if we have that* $I_\lambda;\Sigma;\ell_{\mathcal{A}};\{\tau_i\} \vDash \mathcal{K}$ *if* $I_\lambda;\Sigma';\ell_{\mathcal{A}};\{\tau_i\} \vDash \mathcal{K}'$. *We write reducibility as* $I_\lambda;\ell_{\mathcal{A}};\{\tau_i\} \vDash \langle\Sigma,\mathcal{K}\rangle \to \langle\Sigma',\mathcal{K}'\rangle$.

Finally, we prove the following result for encryptions.

Lemma 6 (Encryption Reduction)**.** *Suppose that* $\Sigma;\mathcal{T};\ell_{\mathcal{A}} \vdash \mathcal{K} : \{\tau_i\}$, $(\mathsf{n} : \mathsf{enckey}^{\mathsf{Real}}\ \tau) \in \Sigma$, $\Sigma \not\vdash [\mathsf{n}] \leq \ell_{\mathcal{A}}$, *and* n *is high in* Σ. *Then, there exists* $\mathcal{K}'$ *such that* $I_\lambda;\ell_{\mathcal{A}};\{\tau_i\} \vDash \langle\Sigma,\mathcal{K}\rangle \to \langle\Sigma[\mathsf{n} \mapsto (\mathsf{n} : \mathsf{enckey}^{\mathsf{Ideal}}\ \tau)],\mathcal{K}'\rangle$ *and* $\Sigma[\mathsf{n} \mapsto (\mathsf{n} : \mathsf{enckey}^{\mathsf{Ideal}}\ \tau)];\mathcal{T},T : \tau;\ell_{\mathcal{A}} \vdash \mathcal{K}' : \{\tau_i\}$, *where* $T \notin \mathcal{T}$.

To carry out the proof, we need an altered interaction, $\mathcal{G}^{\Sigma,\{\tau_i\},-\mathsf{n}}_I$.

Definition 18 (n-Game)**.** *We define the security game* $\mathcal{G}^{\Sigma,\{\tau_i\},-\mathsf{n}}$ *identically to* $\mathcal{G}^{\Sigma,\{\tau_i\}}$, *except that the final computation of* $\mathsf{ok}^{\Sigma,\{\tau_j\}}(N,W,\mathcal{K})$ *is altered so that* $[\![\mathsf{Name}(\mathsf{n})]\!]^{\Sigma,N,W}(v) = \mathsf{True}$.

Looking ahead, the altered security game $\mathcal{G}^{\Sigma,\{\tau_i\},-\mathsf{n}}$ is designed so that it can be evaluated without directly using the value of n.

C.2.1 Proof

We prove Lemma 6 via four hybrid arguments.

Hybrid 1. We first use the typing derivation to transform $\mathcal{K}$ into a configuration of the form $\mathcal{R}^{\mathsf{get(n)},\mathsf{senc(get(n),\cdot)},\mathsf{sdec(get(n),\cdot)}}$, where $\mathsf{get(n)}$ does not appear in $\mathcal{R}$. The two protocols behave identically, except that all variables $a : \mathsf{Name(n)}$ used for encryption and decryption are replaced with $\mathsf{get(n)}$. This transformation is sound due to type safety, and the fact that n is high.

First, we construct the cryptographic reduction, $\mathcal{R}$. Let Ψ be a typing derivation of $\Sigma;\mathcal{T};\ell_{\mathcal{A}} \vdash \mathcal{K} : \{\tau_i\}$, let k be an atomic expression, and let E_1, E_2 be closed expressions with a single hole for an atomic expression. We construct a configuration context $\mathcal{R}[k, E_1, E_2]$ as follows:

- Whenever we visit ENC-n, we replace $\mathsf{enc}(a_1, a_2)$ with $E_1[a_2]$;
- Whenever we visit DEC-n, replace $\mathsf{dec}(a_1, a_2)$ with $E_2[a_2]$;
- Whenever we otherwise visit $\mathsf{get(n)}$, we replace it by k.
- Otherwise, behave the same as Ψ dictates.

Formally, we have that for any $\mathcal{A}$ and N in the support of $\mathsf{Gen}(\Sigma)$,

$$\mathcal{G}_I^{\Sigma,\{\tau_i\}}(N, \mathcal{K}, \ell_{\mathcal{A}}, \mathcal{A}) = \mathcal{G}_I^{\Sigma,\{\tau_i\}}(N, \mathcal{R}^{\mathsf{get(n)},\mathsf{enc(get(n),\cdot)},\mathsf{dec(get(n),\cdot)}}, \ell_{\mathcal{A}}, \mathcal{A}).$$

The central complication of this result is that, since not all cryptographic keys have been idealized, we are *not* guaranteed that arbitrary atomic expressions $a : \mathsf{Name(n')}$ are equal to $N(\mathsf{n'})$, since a may (for example) have been computed from a non-idealized decryption operation. However, since n is high, we have that values of type n may only arise from $\mathsf{get(n)}$ expressions.

To prove the result, we instantiate the general logical relation. Define $\mathcal{V}^{\tau}_{1,N}((v_1, W_1), (v_2, W_2))$ to be $v_1 = v_2 \wedge \mathsf{safe}^{\tau}_{\mathsf{n},N}(v_1)$, where

$$\mathsf{safe}^{\mathsf{Name(n)}}_{\mathsf{n},N}(v) = (v = N(\mathsf{n}))$$
$$\mathsf{safe}^{\mathsf{Name(n')}}_{\mathsf{n},N}(v) = \mathsf{True}, \text{ for } \mathsf{n} \neq \mathsf{n'}$$
$$\mathsf{safe}^{\mathsf{Unit}}_{\mathsf{n},N}(v) = \mathsf{True}$$
$$\mathsf{safe}^{x:\tau\{\phi\}}_{\mathsf{n},N}(v) = \mathsf{safe}^{\tau}_{\mathsf{n},N}(v)$$
$$\mathsf{safe}^{\tau+\sigma}_{\mathsf{n},N}(v) = \begin{cases} \mathsf{safe}^{\tau}_{\mathsf{n},N}(v') & \text{if } v = 0v' \\ \mathsf{safe}^{\sigma}_{\mathsf{n},N}(v') & \text{if } v = 1v' \\ \mathsf{n} \notin \tau \wedge \mathsf{n} \notin \sigma & \text{if } v = \varepsilon \end{cases}$$
$$\mathsf{safe}^{\tau\times\sigma}_{\mathsf{n},N}(v) = \begin{cases} \mathsf{n} \notin \tau \wedge \mathsf{n} \notin \sigma & \text{if } \mathsf{bdry}_\tau(v) = \bot \\ \mathsf{safe}^{\tau}_{\mathsf{n},N}(v[\ldots L]) & \\ \quad \wedge\, \mathsf{safe}^{\sigma}_{\mathsf{n},N}(v[L\ldots]) & \text{if } \mathsf{bdry}_\tau(v) = L \neq \bot \end{cases}$$

Essentially, $\mathsf{safe}^{\tau}_{\mathsf{n},N}(v)$ demonstrates that any sub-value of v which should correspond to type $\mathsf{Name(n)}$ is equal to $N(\mathsf{n})$. A complication of this relation is that, since we are in the middle of the proof, not all encryption keys have been idealized. Since v may have been obtained from a non-ideal decryption, we are not yet guaranteed that all values of sums and product types will unparse correctly. However, we are guaranteed that if a value of $\tau \times \sigma$ does not parse, then we have that n cannot appear syntactically in τ or σ.

Now, the desired result follows from showing that, for any N generated from $\mathsf{Gen}(\Sigma)$ and any i,

$$\mathcal{T};\mathcal{V}_{1,N};N,N \vDash \langle\{\}, \mathcal{K}_i\rangle \sim \langle\{\}, \mathcal{R}[\mathsf{get(n)}, \mathsf{enc(get(n),\cdot)}, \mathsf{dec(get(n),\cdot)}]_i\rangle : \tau_i.$$

Similar to the ideal logical relation, we prove this one by showing soundness for arithmetic expressions, and then using this intermediate result for general expressions.

Hybrid 2. Next, we remove direct access from $\mathsf{get(n)}$ from $\mathcal{R}$, by showing the following:

$$\mathcal{G}_I^{\Sigma,\{\tau_i\}}(N, \mathcal{R}^{\mathsf{get(n)},\mathsf{enc(get(n),\cdot)},\mathsf{dec(get(n),\cdot)}}, \ell_{\mathcal{A}}, \mathcal{A}) = \mathcal{G}_I^{\Sigma,\{\tau_i\},-\mathsf{n}}(N, \mathcal{R}^{\mathsf{Z(get(n))},\mathsf{enc(get(n),\cdot)},\mathsf{dec(get(n),\cdot)}}, \ell_{\mathcal{A}}, \mathcal{A}).$$

In the right game, we zero out all uses of $\mathsf{get(n)}$ outside of well-formed encryptions and decryptions. To compare this censored version of the execution to the real one, we evaluate it under the altered security game $\mathcal{G}^{\Sigma,\{\tau_i\},-\mathsf{n}}$ (Definition 18), which makes the integrity predicate for $\mathsf{Name(n)}$ trivial. The equivalence now holds since since $\mathsf{get(n)}$ cannot flow to the adversary, and any use of n as a return value by a party is soundly ignored.

We show the above result by instantiating the logical relation with $\mathcal{V}^{\tau}_{2,N}((v_1, W_1), (v_2, W_2)) := \mathsf{safe}^{\tau}_{\mathsf{n},N}(v_1) \wedge v_1 \approx_\tau v_2$, where safe is the same as in Hybrid 1. We define $\approx_\tau$ below:

$$v_1 \approx_{\mathsf{Data}(\ell,\ell')} v_2 := (\Sigma \not\vdash [\mathsf{n}] \leq \ell \wedge \ell' \implies v_1 = v_2) \wedge (\Sigma \not\vdash [\mathsf{n}] \leq \ell' \implies |v_1| = |v_2|)$$
$$v_1 \approx_{\mathsf{Name(n')}} v_2 := |v_1| = |v_2| \wedge (\Sigma \not\vdash [\mathsf{n}] \leq [\mathsf{n'}] \implies v_1 = v_2)$$
$$v_1 \approx_{\mathsf{Unit}} v_2 := v_1 = v_2$$
$$v_1 \approx_{x:\tau\{\phi\}} v_2 := v_1 \approx_\tau v_2$$
$$v_1 \approx_{\tau+\sigma} v_2 := \begin{cases} v_1' \approx_\tau v_2' & \text{if } v_1 = 0v_1',\ v_2 = 0v_2' \\ v_1' \approx_\sigma v_2' & \text{if } v_1 = 1v_1',\ v_2 = 1v_2' \\ v_1 = v_2 & \text{otherwise} \end{cases}$$
$$v_1 \approx_{\tau\times\sigma} v_2 := \mathsf{bdry}_\tau(v_1) = \mathsf{bdry}_\tau(v_2) \wedge (\mathsf{bdry}_\tau(v_1) = L \neq \bot \implies v_1[\ldots L] \approx_\tau v_2[\ldots L] \wedge v_1[L\ldots] \approx_\sigma v_2[L\ldots]) (\mathsf{bdry}_\tau(v_1) = \bot \implies v_1 = v_2)$$

Essentially, $v_1 \approx_\tau v_2$ holds when all information in v_1 and v_2 are identical, *except* any information with label greater than or equal to $[\mathsf{n}]$. We replace $\mathsf{get(n)}$ with $\mathsf{Z(get(n))}$, since the latter is well-typed with label $\bot$.

Hybrid 3. At this point, we can apply security of the cryptosystem: Let T be a table variable not in $\mathcal{T}$ and $e_{\mathsf{senc}}[\cdot]$, $e_{\mathsf{sdec}}[\cdot]$ be the expression contexts

$$e_{\mathsf{senc}}[a] := \mathsf{let}\ k = \mathsf{freshkey}()\ \mathsf{in}\ \mathsf{let}\ c = \mathsf{senc}(k, \mathsf{Z}(a))\ \mathsf{in}\ T[c] := a;\ \mathsf{ret}(c)$$
$$e_{\mathsf{sdec}}[a] := T[a].$$

Then, we have the following:

$$\Pr_{N \xleftarrow{\$} \mathsf{Gen}(\Sigma)} [\mathcal{G}_I^{\Sigma,\{\tau_i\},-\mathsf{n}}(N, \mathcal{R}^{\mathsf{Z}(\mathsf{get}(\mathsf{n})),\mathsf{enc}(\mathsf{get}(\mathsf{n}),\cdot),\mathsf{dec}(\mathsf{get}(\mathsf{n}),\cdot)}, \ell_{\mathcal{A}}, \mathcal{A}) = \mathcal{G}_I^{\Sigma,\{\tau_i\},-\mathsf{n}}(N, \mathcal{R}^{\mathsf{Z}(\mathsf{get}(\mathsf{n})),e_{\mathsf{senc}}[\cdot],e_{\mathsf{sdec}}[\cdot]}, \ell_{\mathcal{A}}, \mathcal{A}) \geq 1 - \mathsf{negl}(\lambda).$$

This equivalence is implied by the encryption scheme satisfying INT-CTXT, IND-CPA, and key privacy [14, 16], due to the following properties: first, since $\mathcal{K}$ is initially well-typed, we have that $T \notin \mathcal{R}$; second, we have that $\mathsf{Z}(\mathsf{get}(\mathsf{n}))$ is a function only of I, so can be evaluatable without the value of n; finally, since we are using the altered security game $\mathcal{G}^{\Sigma,\{\tau_i\},-\mathsf{n}}$, we have that the first two properties imply that the output of the left security game can be evaluated using *only the encryption/decryption oracles* for n.

Hybrid 4. Finally, we have that, for all N in the support of $\mathsf{Gen}(\Sigma)$,

$$\mathcal{G}_I^{\Sigma,\{\tau_i\},-\mathsf{n}}(N, \mathcal{R}[\mathsf{Z}(\mathsf{get}(\mathsf{n})), e_{\mathsf{senc}}[\cdot], e_{\mathsf{sdec}}[\cdot]], \ell_{\mathcal{A}}, \mathcal{A}) = \mathcal{G}_I^{\Sigma,\{\tau_i\}}(N, \mathcal{R}[\mathsf{get}(\mathsf{n}), e_{\mathsf{senc}}[\cdot], e_{\mathsf{sdec}}[\cdot]], \ell_{\mathcal{A}}, \mathcal{A}).$$

This hybrid is proven in a similar way to Hybrid 2.

Putting the Hybrids Together. From Hybrids 1-4, we have that

$$\Pr_{N \xleftarrow{\$} \mathsf{Gen}(\Sigma)} [\mathcal{G}_I^{\Sigma,\{\tau_i\}}(N, \mathcal{K}, \ell_{\mathcal{A}}, \mathcal{A}) = \mathcal{G}_I^{\Sigma,\{\tau_i\}}(N, \mathcal{K}', \ell_{\mathcal{A}}, \mathcal{A})] \leq 1 - \mathsf{negl}(\lambda).$$

Preservation of security follows.

Finally, we have that $\Sigma; \mathcal{T}, T : \tau; \ell_{\mathcal{A}} \vdash \mathcal{K}' : \{\tau_i\}$, since $\Sigma; \mathcal{T}, T; \tau; \Gamma; \ell_{\mathcal{A}} \vdash e_{\mathsf{senc}}(a) : \mathsf{Data}(\ell_{\mathcal{A}}, \ell_{\mathcal{A}})$ whenever $\Sigma; \Gamma \vdash a : \tau$, and $\Sigma; \mathcal{T}, T; \tau; \Gamma; \ell_{\mathcal{A}} \vdash e_{\mathsf{sdec}}(a) : \mathsf{Unit} + \tau$ whenever $\Sigma; \Gamma \vdash a : \mathsf{Data}(\ell_{\mathcal{A}}, \ell_{\mathcal{A}})$. The former typing lemma is guaranteed from well-typed encryptions only arising whenever $\Sigma \vdash |\tau| \leq \ell_{\mathcal{A}}$.

C.3 Random Oracle Reduction

We now turn to idealizing random oracle calls. If we have $(\mathsf{n} : \mathsf{pat} \to_P \mathsf{nt})$ in Σ, and the pair (pat, P) is high (defined below), then we are able to idealize all calls to hashes returning $\mathsf{Name}(\mathsf{n})$ with $\mathsf{get}(\mathsf{n})$, replacing n with a local name in Σ.

Our reduction requires that the pair (pat, P) is *high*, defined below:

Definition 19 (Support of Hash Problem)**.** *The support* $\mathsf{supp}(P)$ *of hash problem P is the set of names used to define it:*

$$\mathsf{supp}(\mathsf{sec}(\mathsf{n})) := \{\mathsf{n}\}$$
$$\mathsf{supp}(\mathsf{DH}(\mathsf{n}_1, \mathsf{n}_2)) := \{\mathsf{n}_1, \mathsf{n}_2\}$$

Definition 20 (High Hash)**.** *The pair* (pat, P) *is* $\ell_{\mathcal{A}}$-high *in* Σ *if:*

- *we have that* $\Sigma; \ell_{\mathcal{A}} \vdash \mathsf{unsolvable}\ P$*;*
- *we have that, for all* $\mathsf{n}' \in \mathsf{supp}(P)$*, we have that* n' *is high in* Σ *as defined in Definition 16.*

Lemma 7 (Random Oracle Reduction)**.** *Suppose that* $\Sigma; \mathcal{T}; \ell_{\mathcal{A}} \vdash \mathcal{K} : \{\tau_i\}$, $(\mathsf{n} : \mathsf{pat} \to_P \mathsf{nt}) \in \Sigma$, *and* (pat, P) *is* $\ell_{\mathcal{A}}$*-high in* Σ*. Then, there exists* $\mathcal{K}'$ *such that* $I_\lambda; \ell_{\mathcal{A}}; \{\tau_i\} \vDash \langle \Sigma, \mathcal{K} \rangle \to \langle \Sigma[\mathsf{n} \mapsto (\mathsf{n} : \mathsf{nt})], \mathcal{K}' \rangle$*, and* $\Sigma[\mathsf{n} \mapsto (\mathsf{n} : \mathsf{nt})]; \mathcal{T}; \ell_{\mathcal{A}} \vdash \mathcal{K}' : \{\tau_i\}$.

Proof. First, we show that no PPT adversary is able to query the random oracle with the value of pat with non-negligible probability.

- If $P = \mathsf{sec}(\mathsf{n})$, then we have that the value of n does not flow to the adversary except through random oracle invocations, since (pat, P) is $\ell_{\mathcal{A}}$-high and $\Sigma \vdash \mathsf{pat}$ context P. Thus, the adversary cannot guess the value of n except with negligible probability.
- Similarly , if $P = \mathsf{DH}(\mathsf{n}, \mathsf{n}')$, then we have that the adversary's view of n and n' is only of the form $\mathsf{dhpk}(\mathsf{get}(\mathsf{n}))$, $\mathsf{dhpk}(\mathsf{get}(\mathsf{n}'))$, and hashes of the corresponding shared secret. Thus, by the Gap Diffie-Hellman assumption [26], we have that the probability of the adversary computing the shared secret is negligible.

Now, we have that the view of the adversary in $\mathcal{G}_I^{\Sigma,\{\tau_i\}}(N, \mathcal{K}, \ell_{\mathcal{A}}, \mathcal{A})$ is indistinguishable from that of $\mathcal{G}_I^{\Sigma[\mathsf{n} \mapsto (\mathsf{n}:\mathsf{nt})],\{\tau_i\}}(N, \mathcal{K}', \ell_{\mathcal{A}}, \mathcal{A})$, where we replace all hashes in $\mathcal{K}$ corresponding to pat with $\mathsf{get}(\mathsf{n})$. Additionally, both games have identical failure probabilities for integrity. ■

C.4 Overall Security

We now prove Theorem 1 using lemmas from the previous section. We drive the main proof via the following lemma:

Lemma 8 (Name Context Progress)**.** *Suppose that* $\vdash \Sigma$*, and* $\Sigma \vdash \ell_{\mathcal{A}}$*. Then, one of the three conditions hold:*

1. Σ *is* $\ell_{\mathcal{A}}$*-ideal;*
2. *There exists* $(\mathsf{n} : \mathsf{enckey}^{\mathsf{Real}}\tau)$ *in* Σ *such that* n *is high in* Σ *and* $\Sigma \not\vdash [\mathsf{n}] \leq \ell_{\mathcal{A}}$*;*
3. *There exists* $(\mathsf{n} : \mathsf{pat} \to_P \mathsf{nt})$ *in* Σ *such that* (pat, P) *is* $\ell_{\mathcal{A}}$*-high in* Σ*.*

Proof. We first look for the rightmost name n in Σ such that $\Sigma \not\vdash [\mathsf{n}] \leq \ell_{\mathcal{A}}$ and $(\mathsf{n} : \mathsf{nk}^{\mathsf{Real}}\tau) \in \Sigma$. If found, this name fits (2) above, since the rightmost name cannot be encrypted by another real key.

Next, we look for any hash $(\mathsf{n} : \mathsf{pat} \to_P \mathsf{nt})$ in Σ such that $\Sigma; \ell_{\mathcal{A}} \vdash \mathsf{unsolvable}\ P$. Since no real encryption keys exist, we have that all names in $\mathsf{supp}(P)$ are high.

If neither of the two cases above apply, Σ must be ideal. ■

Theorem 1 now follows by induction on $|\Sigma|$, where

$$|(\mathsf{n} : \mathsf{nt})| = \begin{cases} 0 & \text{if } \mathsf{nt} \text{ is an ideal name} \\ 1 & \text{if } \mathsf{nt} = \mathsf{nk}^{\mathsf{Real}}\ \tau, \text{ for some name kind } \mathsf{nk}. \end{cases}$$

$$|(\mathsf{n} : \mathsf{pat} \to_P \mathsf{nt})| = 2$$

$$|\Sigma| = \sum_{(\mathsf{n}:\mathsf{nt}) \in \Sigma} |(\mathsf{n} : \mathsf{nt})| + \sum_{(\mathsf{n}:\mathsf{pat} \to_P \mathsf{nt})} |(\mathsf{n} : \mathsf{pat} \to_P \mathsf{nt})|$$

If Σ is ideal, then we are done by Theorem 2. Otherwise, we have that $|\Sigma| > 0$, and by Lemma 8 there exists a real key of the form $\mathsf{enckey}^{\mathsf{Real}}\ \tau$ or a hash which may be idealized by Lemma 6, or Lemma 7, respectively. In either case,

there exists a configuration $\mathcal{K}'$ and name context Σ' such that $I_\lambda; \ell_{\mathcal{A}}; \{\tau_i\} \vDash \langle \Sigma, \mathcal{K} \rangle \to \langle \Sigma', \mathcal{K}' \rangle$ and $|\Sigma'| < |\Sigma|$. We then proceed the induction on Σ'.

C.5 Reductions for Related Cryptosystems

We focus on encryptions and Diffie-Hellman operations to illustrate our technique, but our proof can naturally be extended to handle MACs, digital signatures, and public key encryptions using a reduction technique similar to that in §C.2. For example, for signatures, we prove that the initial configuration $\mathcal{K}$ behaves similarly to one of the form $\mathcal{R}[\mathsf{vk}(\mathsf{get}(\mathsf{n})), \mathsf{sign}(\mathsf{get}(\mathsf{n}), \cdot)]$, where $(\mathsf{n} : \mathsf{sigkey}^{\mathsf{Real}}\ \tau) \in \Sigma$ and $\mathsf{n} \notin \mathcal{R}$. At this point, we are able to replace the signing oracle in $\mathcal{R}$ with an idealized one. After we idealize the cryptosystem, we need to use a technique similar to Lemma 5 to allow the simulator to handle public-key operations.

$$\boxed{[\![a]\!]^N} \qquad [\![v]\!]^N := v \qquad [\![\mathsf{f}(a_1,\ldots,a_k)]\!]^N := [\![\mathsf{f}]\!]([\![a_1]\!]^n,\ldots,[\![a_k]\!]^N) \qquad [\![\mathsf{get}(\mathsf{n})^n]\!] := N(n)$$

$$\boxed{[\![e]\!]^N : \mathsf{World} \to \{0,1\}^* \to \mathsf{Dist}(\mathsf{Expr} \times \mathsf{World} \times \{0,1\}^*)} \qquad [\![\mathsf{ret}(a)]\!]^N(W,i) := \mathsf{Ret}(\mathsf{ret}([\![a]\!]^N), W, \varepsilon)$$

$$[\![\mathsf{input}]\!]^N(W,i) := \mathsf{Ret}(\mathsf{ret}(i), W, \varepsilon) \qquad [\![\mathsf{output}(a)]\!]^N(W,i) := \mathsf{Ret}(\mathsf{ret}(0), W, [\![a]\!]^N)$$

$$[\![\mathsf{case}\ a\ (x.\ e_1)\ (y.\ e_2)]\!]^N(W,i) := \begin{cases} \mathsf{Ret}(e_1[\varepsilon/x], W, \varepsilon) & \text{if } [\![a]\!]^N = \varepsilon \\ \mathsf{Ret}(e_1[v/x], W, \varepsilon) & \text{if } [\![a]\!]^N = 0v \\ \mathsf{Ret}(e_2[v/x], W, \varepsilon) & \text{if } [\![a]\!]^N = 1v \end{cases}$$

$$[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]^N(W,i) := \begin{cases} \mathsf{Ret}(e_2[[\![a]\!]^N/x], W, \varepsilon) & \text{if } e_1 = \mathsf{ret}(a) \\ (e_1', W', o) \xleftarrow{\$} [\![e_1]\!]^N(W,i);\ \mathsf{Ret}(\mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2, W', o) & \text{otherwise} \end{cases}$$

$$[\![T[a]]\!]^N(W,i) := \begin{cases} \mathsf{Ret}(\mathsf{ret}(1W[T,[\![a]\!]^N]), W, \varepsilon) & \text{if } W[T,[\![a]\!]^N] \neq \bot \\ \mathsf{Ret}(\mathsf{ret}(00), W, \varepsilon) & \text{otherwise} \end{cases} \qquad [\![T[a] := a']\!]^N(W,i) := \mathsf{Ret}(\mathsf{ret}(0), W[T,[\![a]\!]^N := [\![a']\!]^N], \varepsilon)$$

$$[\![\mathsf{op}(a_1,\ldots,a_k)]\!]^N(W,i) := v \xleftarrow{\$} [\![\mathsf{op}]\!]([\![a_1]\!]^N,\ldots,[\![a_k]\!]^N);\ \mathsf{Ret}(\mathsf{ret}(v), W, \varepsilon)$$

$$[\![H(a)]\!]^N(W,i) := \begin{cases} \mathsf{Ret}(\mathsf{ret}(W[\mathsf{RO},[\![a]\!]^N]), W, \varepsilon) & \text{if } W[\mathsf{RO},[\![a]\!]^N] \neq \bot \\ v \xleftarrow{\$} \{0,1\}^{[\![\mathsf{L_{hash}}]\!]};\ \mathsf{Ret}(\mathsf{ret}(v), W[\mathsf{RO},[\![a]\!]^N := v], \varepsilon) & \text{otherwise} \end{cases}$$

Figure 18: Semantics for OWL. The interpretation I is implicit.

$$\boxed{\Sigma \vdash \mathsf{n} : \mathsf{nt}} \qquad \frac{(\mathsf{n} : \mathsf{nt}) \in \Sigma}{\Sigma \vdash \mathsf{n} : \mathsf{nt}} \qquad \frac{(\mathsf{n} : \mathsf{pat} \mapsto_P \mathsf{nt}) \in \Sigma}{\Sigma \vdash \mathsf{n} : \mathsf{nt}} \qquad \boxed{\Sigma;\Gamma \vdash a : \tau} \qquad \frac{(x : \tau) \in \Gamma}{\Sigma;\Gamma \vdash x : \tau} \qquad \frac{}{\Sigma;\Gamma \vdash v : \mathsf{Data}(\bot,\bot)}$$

$$\frac{(\mathsf{n} : \mathsf{nt}) \in \Sigma}{\Sigma;\Gamma \vdash \mathsf{get}(\mathsf{n}) : \mathsf{Name}(\mathsf{n})} \qquad \frac{\forall i, \Sigma;\Gamma \vdash a_i : \mathsf{Data}(\ell,\ell)}{\Sigma;\Gamma \vdash \mathsf{f}(a_1,\ldots,a_k) : \mathsf{Data}(\ell,\ell)}\ \text{APP} \qquad \frac{\Sigma;\Gamma \vdash a : \tau \qquad \Sigma \vdash \tau \leq \sigma}{\Sigma;\Gamma \vdash a : \sigma}$$

$$\frac{\Sigma;\Gamma \vdash a : \tau \qquad \Sigma \vdash \ell \qquad \Sigma \vdash \sigma}{\Sigma;\Gamma \vdash \mathsf{inl}(a) : \tau + \sigma} \qquad \frac{\Sigma;\Gamma \vdash a : \sigma \qquad \Sigma \vdash \ell \qquad \Sigma \vdash \tau}{\Sigma;\Gamma \vdash \mathsf{inr}(a) : \tau + \sigma} \qquad \frac{\Sigma;\Gamma \vdash a : \tau \qquad \Sigma \vdash \tau\ \mathsf{parsable} \qquad \Sigma;\Gamma \vdash b : \sigma}{\Sigma;\Gamma \vdash \mathsf{pair}_\tau(a,b) : \mathsf{Unit} + (\tau \times \sigma)}$$

$$\frac{\Sigma;\Gamma \vdash a : \tau \times \sigma \qquad \Sigma \vdash \tau\ \mathsf{parsable}}{\Sigma;\Gamma \vdash \mathsf{fst}_\tau(a) : \mathsf{Unit} + \tau} \qquad \frac{\Sigma;\Gamma \vdash a : \tau \times \sigma \qquad \Sigma \vdash \tau\ \mathsf{parsable}}{\Sigma;\Gamma \vdash \mathsf{snd}_\tau(a) : \mathsf{Unit} + \sigma} \qquad \frac{\Sigma;\Gamma \vdash a : \tau}{\Sigma;\Gamma \vdash \mathsf{Z}(a) : \mathsf{Data}(|\tau|,|\tau|)}$$

$$\frac{\Sigma;\Gamma \vdash a : \tau \qquad \forall N\ W\ \gamma.\ \Gamma, N, W \vDash \gamma \wedge [\![\tau]\!]^{\Sigma,N,W}([\![\gamma(a)]\!]^N) \implies [\![\phi]\!]^N([\![\gamma(a)]\!]^N)}{\Sigma;\Gamma \vdash a : (x : \tau\{\phi\})}$$

$$\boxed{\Sigma \vdash \tau \leq \sigma} \qquad \frac{\Sigma \vdash \tau \leq \sigma \qquad \Sigma \vdash \sigma \leq \rho}{\Sigma \vdash \tau \leq \rho} \qquad \frac{}{\Sigma \vdash \tau \leq \mathsf{Data}(\lceil\tau\rceil,|\tau|)} \qquad \frac{\Sigma \vdash \ell_1 \leq \ell_1' \qquad \Sigma \vdash \ell_2 \leq \ell_2'}{\Sigma \vdash \mathsf{Data}(\ell_1,\ell_2) \leq \mathsf{Data}(\ell_1',\ell_2')} \qquad \frac{}{\Sigma \vdash (x : \tau\{\phi\}) \leq \tau}$$

$$\boxed{\Sigma;\mathcal{T};\ell_{\mathcal{A}};\Gamma \vdash e : \tau} \qquad \frac{\Sigma;\Gamma \vdash a : \tau}{\Sigma;\mathcal{T};\ell_{\mathcal{A}};\Gamma;\vdash \mathsf{ret}(a) : \tau} \qquad \frac{}{\Sigma;\mathcal{T};\ell_{\mathcal{A}};\Gamma \vdash \mathsf{input} : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}})} \qquad \frac{\Sigma;\Gamma \vdash a : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}})}{\Sigma;\mathcal{T};\ell_{\mathcal{A}};\Gamma \vdash \mathsf{output}(a) : \mathsf{Unit}}$$

$$\frac{\Sigma;\mathcal{T};\ell_{\mathcal{A}};\Gamma \vdash e_1 : \sigma \qquad \Sigma;\mathcal{T};x : \sigma,\ell_{\mathcal{A}};\Gamma \vdash e_2 : \tau}{\Sigma;\mathcal{T};\ell_{\mathcal{A}};\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau} \qquad \frac{\forall i, \Sigma;\Gamma \vdash a_i : \mathsf{Data}(\ell,\ell)}{\Sigma;\Gamma \vdash \mathsf{op}(a_1,\ldots,a_k) : \mathsf{Data}(\ell,\ell)}\ \text{OP-TRIV}$$

$$\frac{\Sigma;\Gamma \vdash a : \sigma + \tau \qquad \Sigma;\mathcal{T};x : \sigma;\ell_{\mathcal{A}};\Gamma \vdash e_1 : \rho \qquad \Sigma;\mathcal{T};y : \tau;\ell_{\mathcal{A}};\Gamma \vdash e_2 : \rho}{\Sigma;\mathcal{T};\ell_{\mathcal{A}};\Gamma \vdash \mathsf{case}\ a\ (x.\ e_1)\ (y.\ e_2) : \rho}\ \text{CASE}$$

$$\frac{\Sigma;\Gamma \vdash a : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}}) \qquad \Sigma;\mathcal{T};x : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}});\ell_{\mathcal{A}};\Gamma \vdash e_1 : \rho \qquad \Sigma;\mathcal{T};y : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}});\ell_{\mathcal{A}};\Gamma \vdash e_2 : \rho}{\Sigma;\mathcal{T};\ell_{\mathcal{A}};\Gamma \vdash \mathsf{case}\ a\ (x.\ e_1)\ (y.\ e_2) : \rho}\ \text{CASE-CORR}$$

$$\frac{(T : \tau) \in \mathcal{T} \qquad \Sigma;\Gamma \vdash a : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}})}{\Sigma;\mathcal{T};\ell_{\mathcal{A}};\Gamma \vdash T[a] : \mathsf{Unit} + \tau} \qquad \frac{(T : \tau) \in \mathcal{T} \qquad \Sigma;\Gamma \vdash a : \mathsf{Data}(\ell_{\mathcal{A}},\ell_{\mathcal{A}}) \qquad \Sigma;\Gamma \vdash a' : \tau}{\Sigma;\mathcal{T};\ell_{\mathcal{A}};\Gamma \vdash T[a] := a' : \mathsf{Unit}}$$

$$\boxed{\Sigma;\mathcal{T};\ell_{\mathcal{A}} \vdash \mathcal{K} : \{\tau_i\}} \qquad \frac{\forall i, \Sigma;\mathcal{T};\ell_{\mathcal{A}};\cdot \vdash \mathcal{K}[i] : \tau_i}{\Sigma;\mathcal{T};\ell_{\mathcal{A}} \vdash \mathcal{K} : \{\tau_i\}}$$

Figure 19: **Selected Core Typing Rules for** OwlLang. *Rules for cryptographic operations are given in Figure 12.*