Adversary Safety by Construction in a Language of Cryptographic Protocols

Timothy M. Braje*, Alice R. Lee†, Andrew Wagner‡, Benjamin Kaiser§, Daniel Park¶,
Martine Kalke*, Robert K. Cunningham**, Adam Chlipala††

*MIT Lincoln Laboratory, Lexington, MA, USA, Email: {tbraje,kalke}@ll.mit.edu
†Technische Universität Wien, Vienna, Austria, Email: alee122@gmail.com
‡ Northeastern University, Boston, MA, USA, Email: ahwagner@ccs.neu.edu
§Princeton University, Princeton, NJ, USA, Email: bkaiser@princeton.edu
¶Rensselaer Polytechnic Institute, Troy, NY, USA, Email: dancwpark@gmail.com
**University of Pittsburgh, Pittsburgh, PA, USA, Email: robertkcunningham@pitt.edu
††MIT CSAIL, Cambridge, MA, USA, Email: adamc@csail.mit.edu

***Abstract*—Compared to ordinary concurrent and distributed systems, cryptographic protocols are distinguished by the need to reason about interference by adversaries. We suggest a new layered approach to tame that complexity, via an executable protocol language whose semantics does not reveal an adversary directly, instead enforcing a set of intuitive hygiene rules. By virtue of those rules, protocols written in this language provably behave identically with or without interference by active Dolev-Yao-style adversaries. As a result, formal reasoning about protocols can be simplified enough that even naïve model checking can establish correctness of a multiparty protocol, through analysis of a state space with no adversary.**

We present the design and implementation of *SPICY*, short for Secure Protocols Implemented CorrectlY, including the semantics of its input languages; the essential safety proofs, formalized in the Coq theorem prover; and the automation techniques. We provide a preliminary evaluation of the tool's performance and capabilities via a handful of case studies.

I. Introduction

It is distressingly common for a cryptographic mechanism to be carefully designed, implemented, and deployed, only for attackers to later discover critical flaws and vulnerabilities that undermine the system [1], [2], [3], [4], [5], [6], [7], [8]. Attackers have the advantages of time, resources, a complex design space to exploit, and the ability to employ attack methods and capabilities that the system designers did not (or could not) anticipate. With an ever-growing reliance on cryptography in critical systems, there is an urgent need for new approaches to evaluating the soundness of cryptographic mechanisms in the face of strong, unpredictable adversaries.

Could it be possible to establish protocol-development languages that prevent adversary interference in the same way that, e.g., Java prevents buffer overflows? Such a language would include at least two unusual constructs. First, as Java hides memory management and thus the chance to make mistakes in memory management, our new kind of protocol language would expose higher-level constructs that make certain kinds of bugs impossible to express. Second, as Java raises an exception on an out-of-bounds array access, our new kind of protocol language could similarly signal exceptions when it detects that certain conservative best practices are not being followed. Protocol designers would want to prove that, in fact, such exceptions are *never* signaled. With such reasoning in hand, it becomes possible to analyze protocol execution with no adversary, porting the results to strong classes of active adversaries. As Java takes opinionated stances, such as forcing programmers to manage memory in certain ways, our new language would do the same, making no claim to represent all or even most cryptographic protocols found in the world today. However, developers of a new protocol could adopt our language and find a drastically simplified setting for rigorous reasoning about security. Indeed, much as Java programmers need not be educated about buffer overflows, developers who understand distributed systems but not cryptography could conceivably use our new language to develop new secure protocols.

We report on the first steps in developing that kind of language and its associated formal-verification tooling, as a library within the Coq [9] theorem prover. Here, we will demonstrate that it is, in fact, possible to design languages that abstract away adversarial capability into a few "safety rules" akin to type-safety checks, so that protocol designers and developers can focus on the details of protocol development rather than reasoning explicitly about safety. Verifying the safety properties within Coq using simple model-checking techniques, which can be applied automatically, shows that our ideas are implementable at a cost of verification performance, which can be overcome with future engineering work.

This effort falls within the growing field of Computer-Aided Cryptography (CAC) [10]. By applying rigorous, machine-checked verification and analysis to every level of the cryptographic stack – from the design of cryptographic protocols to the implementation of the low-level primitives on which those protocols depend – CAC tools can provide strong assurances of soundness. Mainstream software is increasingly incorporating CAC techniques: for example, CAC tools verified the design

and many implementations of TLS 1.3 [11], [12], and the popular Chrome and Firefox browsers currently ship with verified cryptographic primitives [13], [14].

At the top of the CAC stack sit so-called "design-level" tools [10]. Though this is hardly a homogeneous space, tools of this variety all strive to catch vulnerabilities early in the life cycle of a protocol. Design-level CAC tools are used to detect flaws in higher-level reasoning, and a protocol blessed by such a tool may be considered sound given reasonable assumptions about lower-level primitives.

While these tools share a common goal, they vary in the interfaces they present to protocol designers. Whereas an ideal verifier might simply consume a protocol description and declare it sound or unsound, this is rarely, if ever, viable for practical protocols. Instead, some tools (e.g., EasyCrypt [15]) require that the protocol designer write a machine-checkable proof similar to the pen-and-paper proof that a cryptographer would write. Others trade some of that proof obligation for stronger assumptions about, e.g., the cryptographic model. Regardless of the strength of the assumptions, as the class of potential adversaries grows so too does the number of ways a protocol's execution might be perturbed, forcing the tool to consider more cases.

To see this problem in action, consider the CAC technique of model checking. By exhaustively traversing the state space of a system (usually after simplifying the space with clever choices of abstraction), a model checker evaluates whether the system is correct according to a formal specification. Even in a nonadversarial setting, a system's state space can be immense. For example, concurrent programs can force a model checker to explore exponentially many interleavings of threads. Introducing an adversary who executes *unknown* code compounds the challenge, leading to a state space that can grow arbitrarily large – if it can be enumerated at all.

Contributions. We demonstrate in this work a protocol-language design that supports coding and formal verification of new protocols *with no need to reason about adversary behavior*. We fix the adversary class and the language of protocols, which allows us to shift the burden of adversarial reasoning off of particular protocols and onto *the language as a whole*. Specifically, we prove a Strong Preservation Theorem (Thm. 1), which states that any protocol in our language that is safe in a world *without an adversary* is also safe in a world *with an active Dolev-Yao [16] adversary*.

We develop this idea in Secure Protocols Implemented CorrectlY (*SPICY*), a framework written and verified in Coq. Borrowing the terminology of the Real/Ideal paradigm from traditional cryptography proofs, but requiring no familiarity with that paradigm, *SPICY* protocols are developed in two domain-specific languages (DSLs). The *Ideal World* specification language presents a simpler syntax that abstracts away cryptographic primitives from the *Real World* implementation language. Typically, a protocol designer would develop her protocol specification first (using the *Ideal World* DSL) and then implement the specification (using the *Real World* DSL). *SPICY* would then formally check that the implementation matches the specification and that no safety violations were encountered. *Real World* protocols are executable, for example, via the standard Coq extraction pipeline to high-level functional code. One easy execution engine is a lightweight interpreter that takes protocol descriptions like ours as input, invoking crypto and messaging side effects as appropriate. The same technique has been used in other projects like the FSCQ verified file system [17], where experiments showed competitive performance versus conventional file-system implementations. We use a symbolic model of cryptography and fix the adversary to the Dolev-Yao malicious active attacker who is unable to take actions that require an unknown private key. This adversary also obtains any message passing through the network; is a legitimate user and thus can initiate conversations, as either herself or someone else; and can replay received messages. We prove, *as a property of the language*, that if the *Real World* protocol simulates the specification then it behaves the same in a world with or without an arbitrary (Dolev-Yao) adversary.

With the Strong Preservation Theorem in hand, we automate the simulation-argument construction in an adversary-free world using a symbolic model-checking technique. We demonstrate that this technique is viable for a collection of protocols inspired by the literature. Our goal in this paper is to highlight how symbolic-model verification tools can be constructed as little more than direct interpreters for carefully designed programming languages, allowing for much shorter tool implementations than for competitors (roughly a factor-of-10 savings in implementation length; see §IV-B), despite achieving significantly *higher* levels of rigor (via Coq proofs).

Organization. The remainder of the paper is structured as follows. In §II, we walk through the development of an example protocol, identifying and iteratively eliminating common errors along the way. We then apply *SPICY* to the example, showing how it (1) helps the protocol designer avoid these pitfalls and (2) fosters correct design of the protocol. In §III, we introduce the proposed framework, consisting of two DSLs used to model protocol behavior, the languages' operational semantics, and formalized correctness and safety guarantees (including the Strong Preservation Theorem). We evaluate our framework in §IV, discussing protocols we have proven correct. We consider related work in §V and conclude and discuss avenues for future work in §VI.

Availability The source code, including the library and example protocols can be found at https://github.com/MIT-LL/spicy.

II. Verifying a Sample Protocol

To demonstrate how protocol development and verification work in *SPICY*, consider the SECRET SHARING PROTOCOL: two parties with mutually authenticated channels establish a secure communication channel, then one party sends a secret to the other.[1] We begin with a high-level intent, attempt to

[1]This example is illustrative; details of the semantics of the language will be covered in later sections.

write a safe implementation in pseudocode, and then demonstrate writing a formal, *Ideal World* specification and valid *Real World* implementation. Finally, we demonstrate a further simplified *Ideal World* specification which more succinctly captures the essence of the protocol.

First pseudocode attempt. A naïve first program may look like the pseudocode shown in Fig. 1. In this program, Bob generates an asymmetric encryption key (line 6), shares it with Alice (7), and waits for Alice to respond with a secret (8). Alice listens for the message containing the key (1), generates a secret (2), encrypts it with the key she received from Bob (3), sends it to Bob (4), and exits (5). Finally, Bob decrypts (9) and exits (10).

```
1  pk_B^ε = recv()
2  SECRET = rng()
3  c = encrypt (SECRET, pk_B^ε)
4  send(Bob, c)
5  print(SECRET)
```

(a) Alice's protocol code.

```
6  (pk_B^ε, sk_B^ε) = asym_keygen()
7  send(Alice, pk_B^ε)
8  c = recv()
9  m = decrypt(c, sk_B^ε)
10 print(m)
```

(b) Bob's protocol code.

Fig. 1: Insecure implementation of the SECRET SHARING PROTOCOL. Since this implementation does not verify message authenticity (lines 3-4 and 8-9), an attacker could trick Alice into divulging the secret.

The program in Fig. 1 includes notation for several commonly used primitives. We adopt a convention for keys such that pk and sk represent asymmetric public and private keys, respectively; and κ represents a symmetric key. We use subscripts on the keys to indicate ownership and use superscripts to clarify the purpose of each key, ε for encryption keys and σ for signing keys. To send or receive a message, we use `send` and `recv`. The primitives `encrypt` and `decrypt` secure message confidentiality, while `sign` and `verify` handle message authenticity. As a proxy for the secret, we use `rng` to generate a number nondeterministically. To obtain an asymmetric public/private key pair, we use `asym_keygen` (`sym_keygen` will be used for symmetric keys).

There is an obvious security problem in this implementation: an attacker could send a message to Alice pretending to be Bob, tricking Alice into sending the secret meant for Bob to the adversary. We solve this problem by using digital signatures to protect message authenticity. Furthermore, while not a security issue, this implementation could be improved with a hybrid encryption scheme. Rather than encrypting the secret with the asymmetric key, a modern, practical public-key cryptosystem should encrypt the payload with a symmetric key (which is more efficient), then encrypt the symmetric key with an asymmetric key (which is more convenient) since a more general version of this protocol might send payloads much longer than a symmetric key.

Second attempt. The next version of the protocol (Fig. 2) addresses both issues. All messages are signed and verified, and Alice encrypts the message data with a symmetric encryption key (line 7) rather than with the asymmetric key.

```
1  (pk_A^ε, sk_A^ε) = asym_keygen()
2  c_1 = sign(pk_A^ε, sk_A^σ)
3  send(Bob, c_1)
4  c_2 = recv()
5  κ^ε = verify(c_2, pk_B^σ)
6  SECRET = rng()
7  c_3 = sign_encrypt(SECRET, sk_A^σ, κ^ε)
8  send(Bob, c_3)
9  print(SECRET)
```

(a) Alice's protocol code.

```
10 c_1 = recv()
11 pk_A^ε = verify(c_1, pk_A^σ)
12 κ^ε = sym_keygen()
13 c_2 = sign(κ^ε, sk_B^σ)
14 send(Alice, c_2)
15 c_3 = recv()
16 m = decrypt_verify(c_3, κ^ε, pk_A^σ)
17 print(m)
```

(b) Bob's protocol code.

Fig. 2: A second (also insecure) implementation of the SECRET SHARING PROTOCOL. This time, Bob forgets to encrypt the shared key before sending it over the wire (12-14). As a result, an adversary can intercept the message, learn the key, and read any messages that are encrypted using it.

This second version still overlooks a catastrophic mistake (which our framework will catch). Bob sends the symmetric key (lines 12-14) without first encrypting it, so an attacker could steal it and use it to decrypt any secrets encrypted with that key. We fix that error and produce our almost-final protocol, shown in Fig. 3.

Third attempt. The implementation in Fig. 3 protects the symmetric key (line 13) and digitally signs all ciphertexts (2,7,13). The protocol is bootstrapped with preshared public signing keys amongst the honest parties, which would be handled in practice by a public key infrastructure (PKI) or similar service.

Two main nuisances remain. First, the adversary may send garbage messages that fail signature checking, aborting the protocol. Second, the adversary may replay legitimate messages, similarly aborting the protocol or, worse, tricking an honest participant into proceeding. We wait for our final implementation, in our new protocol language, to deal with those concerns.

Typically, the issues we iterated through are manually discovered via code reviews or bug reports. In the next paragraphs, we show how *SPICY* can prevent these errors automatically, with a digression through a formal specification for the protocol, before we return to a secure implementation.

```
1  (pk^ε_A , sk^ε_A ) = asym_keygen()
2  c_1 = sign(pk^ε_A , sk^σ_A )
3  send(Bob , c_1 )
4  c_2 = recv()
5  κ^ε = decrypt_verify( c_2 , sk^ε_A , pk^σ_B )
6  SECRET = rng()
7  c_3 = sign_encrypt(SECRET , sk^σ_A , κ^ε )
8  send(Bob , c_3 )
9  print(SECRET)
```

(a) Alice's protocol code.

```
10  c_1 = recv()
11  pk^ε_A = verify( c_1 , pk^σ_A )
12  κ^ε = sym_keygen()
13  c_2 = sign_encrypt(κ^ε , sk^σ_B , pk^ε_A )
14  send(Alice , c_2 )
15  c_3 = recv()
16  m = decrypt_verify( c_3 , κ^ε , pk^σ_A )
17  print(m)
```

(b) Bob's protocol code.

Fig. 3: A third implementation of the SECRET SHARING PROTOCOL. Private keys and secrets are encrypted before being sent (7,13), and all ciphertexts are digitally signed (2,7,13). However, garbage messages or replays would still interrupt proper protocol execution.

***Ideal World* specification.** A developer using *SPICY* begins with a formal protocol specification (written in the *Ideal World* DSL) that precisely explains the protocol intent and is used to verify the correctness of the implementation. The specification language treats messages as members of permissioned channels upon which each user can be granted read (r-), write (-w), or read-write (rw) access. Users may share their own channel permissions with others by sending permission-granting messages. Messages are never removed from channels, so granting access to one gives the receiving user access to the full message history on that channel. Further, to model an active network adversary properly, we want to be prepared for message reorderings, so *Ideal World* programs may receive duplicate and out-of-order messages. One complication arises via the channel-intersection ($\cap$) operation, which allows a user to create a new channel implicitly with the permissions constructed from the least permissive bits of the user's access for each constituent channel. Fig. 4 shows a specification for the SECRET SHARING PROTOCOL.

We begin with "authenticated" communication channels between the two parties Alice and Bob: $\mathrm{ch}_{\mathrm{AB}}$ and $\mathrm{ch}_{\mathrm{BA}}$, meaning everyone can read on the channels, but only Alice can write to $\mathrm{ch}_{\mathrm{AB}}$, and only Bob can write to $\mathrm{ch}_{\mathrm{BA}}$. *SPICY* is currently specialized to the existence of that kind of bootstrapping key distribution, much like how Java is specialized to particular memory layouts in a garbage-collected heap. Alice starts the protocol by creating a new channel (line 1) and sending -w access on that channel to Bob (2). The new channel created here will be used to help establish the symmetric

```
1  ch_1 ← CreateChannel
2  _ ← Send (ch_1 ↦ -w) ch_AB
3  ch_s ← Recv (ch_1 ∩ ch_BA)
4  SECRET ← Gen
5  _ ← Send SECRET (ch_s ∩ ch_AB)
6  Return SECRET
```

(a) Alice's protocol specification.

```
7  ch_1 ← Recv ch_AB
8  ch_s ← CreateChannel
9  _ ← Send (ch_s ↦ rw) (ch_1 ∩ ch_BA)
10 SECRET ← Recv (ch_s ∩ ch_AB)
11 Return SECRET
```

(b) Bob's protocol specification.

Fig. 4: Specification for the SECRET SHARING PROTOCOL. The channels $\mathrm{ch}_{\mathrm{AB}}$ and $\mathrm{ch}_{\mathrm{BA}}$ are preprovisioned authentication channels whose names are in scope for both users' specifications.

```
1  (pk^ε_1, sk^ε_1) ← GenerateKey Asym Encryption
2  c_1 ← Sign sk^σ_A û_B pk^ε_1
3  _ ← Send û_B c_1
4  c_2 ← Recv (SignedEncrypted pk^σ_B pk^ε_1)
5  κ^ε ← Decrypt c_2
6  SECRET ← Gen
7  c_3 ← SignEncrypt sk^σ_A κ^ε û_B SECRET
8  _ ← Send û_B c_3
9  Return SECRET
```

(a) Alice's protocol implementation.

```
10  c_1 ← Recv (Signed pk^σ_A)
11  pk^ε_1 ← Verify pk^σ_A c_1
12  κ^ε_2 ← GenerateKey Sym Encryption
13  c_2 ← SignEncrypt sk^σ_B pk^ε_1 û_A κ^ε_2
14  _ ← Send û_A c_2
15  c_3 ← Recv (SignedEncrypted pk^σ_A κ^ε_2)
16  m ← Decrypt c_3
17  Return m
```

(b) Bob's protocol implementation.

Fig. 5: *Real World* description of the SECRET SHARING PROTOCOL. We bootstrap the protocol with globally accessible shared public signing keys for authentication ($\mathsf{pk}^{\sigma}_{\mathrm{A}}$, $\mathsf{pk}^{\sigma}_{\mathrm{B}}$) so that the honest parties can communicate securely.

communication channel (akin to sharing a public encryption key). Bob receives the permission (7), creates another channel that the pair can use for secure communication (8), and grants Alice full access (rw) to this channel (9). Note, Bob sends this full permission grant using the "intersect" operation where he combines the properties of his authentication channel with the private channel received from Alice in (7). At this point, both parties have a channel that is secure (known to both as ch_{s}) along with their authentication channels, so they can perform secure authenticated communication. Alice generates a random secret (4) and securely sends it to Bob, line (5).

Real World implementation. The final *Real World* protocol that can pass the requirements in our language (Fig. 5) looks very similar to the result of the iterative process, so we do not describe it in detail. One bit of undescribed notation that can be found in the protocol implementation are references to honest parties (e.g., $\hat{u}_A$) which act as addresses to the users participating in the protocol. This implementation shows off the primary distinctive high-level features of *SPICY*. Most clearly, inputs from the network are associated with *message patterns* (see lines 4, 10, and 15), which spell out cryptographic requirements on messages, with the semantics that *the first compliant message in the message queue is read, while noncompliant messages are ignored*. In that way, no matter how much of a nuisance an adversary creates by sending messages, the honest parties may experience some denial of service, but, so long as the protocol finishes, its result will be as if the adversary sat on the sidelines. This guarantee holds even when the adversary replays honest messages, since *SPICY* builds in runtime generation and checking of nonces (activated on every send or receive by honest parties), much as Java improves safety with runtime storage and checking of array lengths.

SPICY enforces several rules that are necessary to overcome attacks by an active adversary. Centrally, we track a pool of *honest keys*, initialized with the bootstrapping key distribution and charged never to be leaked to the adversary. When a violation of an associated conservative rule is detected at runtime, *SPICY* effectively raises an exception (though really this convention is a proof technique, as we always prove that protocols do not raise exceptions). For example, *SPICY* prevents man-in-the-middle attacks by requiring all honest parties to sign their messages with honest keys, protects honest parties from leaking honest keys by ensuring that all private keys are encrypted before being sent, and guarantees that honest parties will not fall prey to message-replay attacks by properly adding nonces to ciphertexts and checking them on received messages. We defer to §III-C both the formal description of our safety result and the full analysis of how these safety properties emerge from our efforts in producing that final safety proof, where we describe the Strong Preservation Theorem (Thm. 1).

```
1  SECRET ← Gen
2  _ ← Send SECRET ch_AB
3  Return SECRET
```

(a) Alice's simple protocol specification.

```
4  m ← Recv ch_AB
5  Return m
```

(b) Bob's simple protocol specification.

Fig. 6: Simple specification, distilling the essence of the SECRET SHARING PROTOCOL in which we simply share the (random) secret between two parties by sending it over a preprovisioned channel.

The reader may reasonably wonder what makes Fig. 4 the right, simplest specification for the SECRET SHARING PROTOCOL. In fact, the specification in Fig. 4 arguably contains a bit too many ancillary details to instill absolute confidence in its validity[2] In effect, we consider this protocol specification to be one step in the verification process – leaking perhaps a bit too many protocol details, so that we can make the connection to the protocol implementation. Indeed, we expect that a full protocol analysis would involve proof of these specifications against others that avoid leaking protocol details. For instance, a natural choice here is to convey the secret directly using channels that are already shared. Though deriving such "simplest" specifications has not been the focus of our project to date, we have constructed one for the SECRET SHARING PROTOCOL (Fig. 6), distilling the protocol to its essence: Alice generates a secret and shares it with Bob over a predefined permissioned channel. The correctness of Fig. 6 relies almost exclusively on the proper permissioning of $\mathrm{ch}_{\mathrm{AB}}$, leaving very little opportunity for the specifier to make a mistake.

The point is that our method shows the process of refining a specification in a language *that does not require any explicit adversary modeling*. As a result, the rest of the correctness proof can be "just" a traditional verification of a message-passing concurrent system.

III. Design and Implementation of *SPICY*

A. Ideal World *Semantics*

Protocols written in the *Ideal World* language capture high-level correctness properties like communication patterns and information flow. The correctness of these specifications should either be obvious or easily verifiable (e.g., by hand). Constraints that would be imposed on realistic communications (e.g., the use of encryption) are abstracted away, and users instead communicate via perfectly secure, permissioned channels. *Permissions* Φ are two-bit records defined as: $\Phi ::= \{\mathsf{read} : \mathbb{B}; \mathsf{write} : \mathbb{B}\}$. *Messages* $\mathbb{M}$ contain either arbitrary content (modeled here as naturals $\mathbb{N}$), permissions, or pairs of messages ($\mathbb{M} ::= \mathbb{N} \mid \Phi \mid \mathbb{M} \times \mathbb{M}$).

The *Ideal World* language is defined as a *mixed embedding* [18] in Gallina, Coq's functional specification language. This means that *Ideal World* programs may appear in Gallina programs, and Gallina programs ($\mathbb{G}$) may appear in *Ideal World* programs. The syntax for the *Ideal World* is as follows:

$$\begin{aligned}
\textit{Return} &: \mathbb{G} \to \mathsf{exp} \\
\textit{Bind} &: \mathsf{exp} \to (\mathbb{G} \to \mathsf{exp}) \to \mathsf{exp} \\
\textit{Gen} &: \mathsf{exp} \\
\textit{Send} &: \mathbb{M} \to \iota \to \mathsf{exp} \\
\textit{Recv} &: \iota \to \mathsf{exp} \\
\textit{CreateChannel} &: \mathsf{exp}
\end{aligned}$$

[2]In this case, since the protocol is relatively straightforward, maybe this specification is good enough, but it is not hard to imagine scenarios to the contrary.

$$\frac{\langle cv, \mu[\hat{u} \mapsto \langle ps, c_1\rangle]\rangle \xrightarrow{\ell} \langle cv', \mu[\hat{u} \mapsto \langle ps', c_1'\rangle]\rangle}{\langle cv, \mu[\hat{u} \mapsto \langle ps, x \leftarrow c_1; c_2\rangle]\rangle \xrightarrow{\ell} \langle cv', \mu[\hat{u} \mapsto \langle ps', x \leftarrow c_1'; c_2\rangle]\rangle}\ \textsc{BindRecurse}$$

$$\frac{}{\langle cv, \mu[\hat{u} \mapsto \langle ps, x \leftarrow \textbf{\textit{Return}}(v); c(x)\rangle]\rangle \to \langle cv, \mu[\hat{u} \mapsto \langle ps, c(v)\rangle]\rangle}\ \textsc{BindProceed}$$

$$\frac{ps[\mathrm{ch}] = p \qquad -w \le p \qquad \forall(\mathrm{ch}', \phi) \in m.\ \phi \le ps[\mathrm{ch}']}{\langle cv[\mathrm{ch} \mapsto ms], \mu[\hat{u} \mapsto \langle ps, \textbf{\textit{Send}}(m, \mathrm{ch})\rangle]\rangle \xrightarrow{m} \langle cv[\mathrm{ch} \mapsto \{m\} \cup\ ms], \mu[\hat{u} \mapsto \langle ps, \textbf{\textit{Return}}()\rangle]\rangle}\ \textsc{Send}$$

$$\frac{ps[\mathrm{ch}] = p \qquad r\text{-} \le p \qquad m \in cv[\mathrm{ch}]}{\langle cv, \mu[\hat{u} \mapsto \langle ps, \textbf{\textit{Recv}}(\mathrm{ch})\rangle]\rangle \xrightarrow{m} \langle cv, \mu[\hat{u} \mapsto \langle ps \cup \{p \mid p \in m\}, \textbf{\textit{Return}}(m)\rangle]\rangle}\ \textsc{Recv}$$

$$\frac{\mathsf{fresh}\ \mathrm{ch}}{\langle cv, \mu[\hat{u} \mapsto \langle ps, \textbf{\textit{CreateChannel}}\rangle]\rangle \xrightarrow{\epsilon} \langle cv[\mathrm{ch} \mapsto []], \mu[\hat{u} \mapsto \langle ps[\mathrm{ch} \mapsto rw], \textbf{\textit{Return}}(\mathrm{ch})\rangle]\rangle}\ \textsc{CreateChannel}$$

$$\frac{}{\langle cv, \mu[\hat{u} \mapsto \langle ps, \textbf{\textit{Gen}}\rangle]\rangle \xrightarrow{\epsilon} \langle cv, \mu[\hat{u} \mapsto \langle ps, \textbf{\textit{Return}}(v)\rangle]\rangle}\ \textsc{GenContent}$$

Fig. 7: Small-step semantics for the *Ideal World* specification language.

Return and ***Bind*** are the standard monadic metalanguage definitions adapted to the mixed-embedded setting. As usual, we adopt the notation $\mathsf{x} \leftarrow \mathsf{exp}_1; \mathsf{exp}_2$ for ***Bind*** $\mathsf{exp}_1\ (\lambda x.\ \mathsf{exp}_2)$. ***Gen*** is a primitive for nondeterministic number generation, used as a proxy for some secret value unknown at the time of specification or implementation. ***Send*** and ***Recv*** are primitives for communication over channels drawn from ι. Communication may take place over a "single" channel (created by ***CreateChannel***) or an "intersection" channel created by merging two single channels together (deriving permissions as $\Phi_1 \cap \Phi_2 = \{\mathrm{read} : \Phi_1.\mathrm{r} \wedge \Phi_2.\mathrm{r}; \mathrm{write} : \Phi_1.\mathrm{w} \wedge \Phi_2.\mathrm{w}\}$).

Fig. 7 presents the operational semantics for the *Ideal World* – a small-step semantics using labels to track message movements. We annotate the stepping relation, as $c \xrightarrow{\ell} c'$ which states that the program steps from c to c' while emitting label ℓ (silent steps have no annotation). Each rule chooses a single party to step, then executes the appropriate step for that party. Configurations are pairs $\langle cv, \mu\rangle$, where cv (channel vector) maps channel identifiers to message heaps and μ maps party identifiers to parties. From these maps we draw ps and c for the currently stepping party, where ps represents a mapping from channel identifiers to that party's permissions on that channel, and c is the remaining program for that party. Initial configurations contain any preprovisioned channels.[3]

The rules BindRecurse and BindProceed are defined as usual. Send adds a specified message to the heap associated with the specified channel (assuming the sending party has write permission to the channel and is not sharing a permission that it does not itself have) and emits a label with the message m. Recv picks a message from the specified channel, adds any permissions granted by the message to the receiving party's permission set, and emits a label m. Recv requires read permissions on the receiving channel and that a message (chosen nondeterministically) be present on that channel (i.e., ***Recv*** is blocking). CreateChannel creates a *fresh* channel with an empty message heap, read-write permissions for the creating party, and no permissions elsewhere. GenContent nondeterministically produces a natural number.

[3]Bootstrapping these initial configurations is out-of-scope for *SPICY*.

B. Real World *Semantics*

Protocols written in the *Real World* language reintroduce the usual primitives that secure realistic communications (e.g., signatures and encryption). Messages are no longer sent on perfectly secure, permissioned channels but over public networks. Permissions from the *Ideal World* correspond to cryptographic *keys* in the *Real World*. All keys are either symmetric or asymmetric, used either for signing (σ) or encrypting (ε), and each is uniquely identified by some $k_{\mathrm{ID}} \in \mathbb{K}$.

Messages $\mathbb{M}$ in the *Real World* are either *plaintexts* or *ciphertexts* $\mathbb{C}$. Plaintext messages take the same shape as *Ideal World* messages, but with $\Phi ::= \mathbb{K} \otimes \{pub, priv\}$, where the latter component indicates whether the permission has full access to the key (i.e., if the key is symmetric or the permission is the private part of an asymmetric key). A ciphertext message wraps a plaintext message, affixing both a signature (to prevent forgery) and a unique nonce (to prevent replay), and may optionally be encrypted.

As in the *Ideal World*, the *Real World* is defined as a mixed embedding in Gallina, with the following syntax:

$$\begin{aligned}
\textbf{Return} &: \mathbb{G} \to \mathsf{exp}\\
\textbf{Bind} &: \mathsf{exp} \to (\mathbb{G} \to \mathsf{exp}) \to \mathsf{exp}\\
\textbf{Gen} &: \mathsf{exp}\\
\textbf{Send} &: \mathbb{M} \to \hat{u} \to \mathsf{exp}\\
\textbf{Recv} &: \mathit{pat} \to \mathsf{exp}\\
\textbf{SignEncrypt} &: k^{\sigma} \to k^{\varepsilon} \to \mathbb{M} \to \mathsf{exp}\\
\textbf{Decrypt} &: k^{\sigma} \to k^{\varepsilon} \to \mathbb{C} \to \mathsf{exp}\\
\textbf{Sign} &: k^{\sigma} \to \mathbb{M} \to \mathsf{exp}\\
\textbf{Verify} &: k^{\sigma} \to \mathbb{C} \to \mathsf{exp}\\
\textbf{GenerateKey} &: \mathit{keytype} \to \mathit{usage} \to \mathsf{exp}
\end{aligned}$$

Return, **Bind**, and **Gen** are just as in the *Ideal World*. **Send** and **Recv** are the standard communication primitives, where $\hat{u}$ is a user, and *pat* is a receive pattern that acts as a filter for the next message to draw from the queue. A receive pattern can either allow all messages or check for ciphertexts according to the following grammar:

$$pat ::= \mathsf{Accept} \mid \mathsf{Signed}\ k^{\sigma} \mid \mathsf{SignedEncrypted}\ k^{\sigma}\ k^{\varepsilon}$$

Recall that this pattern mechanism is the most visible novelty of *SPICY* over past languages for protocol design. It is important that protocol execution *ignores unmatching messages*, thus completely filtering the adversary's futile attempts to interfere, so long as the patterns are chosen wisely, i.e. to check signatures using keys the adversary does not have. Also recall that sending and receiving implicitly generate and check nonces, protecting against replay attacks. For resource-constrained environments, where minimizing message size is important, alternative replay-prevention strategies could be employed (like the implicit message-sequence numbering of TLS), though we have not yet implemented any.

Another important point to make is that any user may send a message to any other user, and the recipient does not automatically receive any kind of proof of who sent it. Thus, it is easy to model a strong, active adversary as a designated user in the same language, with the extra benefit of being delivered a copy of each message sent by others. That adversary may then flood the honest parties with as many messages as he likes, computed in any way consistent with the Dolev-Yao rules for symbolic cryptography. In other words, we do not require the separate *adversary message deduction rules* that related tools use to model adversary capabilities; instead, the adversary is modeled as another (unknown) program in the same DSL, constrained by the same operational semantics, with a few advantages versus honest parties that we summarize in this section.

SignEncrypt, **Decrypt**, **Sign**, and **Verify** are the usual cryptographic primitives, defined over encryption and signature keys where appropriate. Keys may not change usage after initialization. Finally, **GenerateKey**, analogous to ***CreateChannel***, creates a fresh key given a *keytype* (symmetric (**Sym**) or asymmetric (**Asym**)) and a *usage* (either **Encryption** [ε] or **Signing** [σ]).

The operational semantics for the *Real World* is given in Fig. 8. As in the *Ideal World*, it is a labeled small-step semantics that traces message movement, where each rule chooses a user to step and then executes the step. *Global configurations* are triples $\langle \mathsf{C}, \kappa, \mu \rangle$, where C maps cipher identifiers to their ciphertexts, κ maps key identifiers to their keys, and μ maps user identifiers to *local user configurations*. Local user configurations, which track the computation from the perspectives of individual parties, are five-tuples $\langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, c \rangle$, where ς is a local ciphertext heap, ks is a local key heap, $\boxtimes$ is a message queue, N is a three-tuple $\langle$ next nonce value, nonces sent, nonces received $\rangle$, and c is the remaining program.

The rules BINDRECURSE, BINDPROCEED, and GENCONTENT work as they did in the *Ideal World* (modulo the change in configurations). GENKEY takes information about key *type* and *usage*, returning (k_{ID}, *priv*) where k_{ID} is a fresh key identifier and adding the generated key to the global key heap κ. SIGN and ENCRYPT similarly create ciphertexts of the appropriate kinds with fresh identifiers, checking that the user has the right level of access to the keys used to sign and/or encrypt (*priv* access for signing and at least *pub* access for encrypting). SIGN and ENCRYPT also increment the user's current nonce value and affix it to the ciphertext[4], checking that the message only contains keys possessed by the user. Any of these hypotheses that we describe informally as "checks" can also be seen as analogous to an array-out-of-bounds exception in Java, a kind of signal that a conservative safety rule has been violated. The checks in *SPICY* differ from array-out-of-bounds checks in Java in a very important way – the checks in *SPICY* are performed completely statically, without the need for a runtime monitor. That is, we formulate this operational semantics as a proof tool, then prove that specific protocols never raise exceptions.

VERIFY and DECRYPT are the inverses of the previous operations, again checking that the necessary keys are owned by the stepping user (*priv* access for decrypting and at least *pub* access for verifying). There is an additional check that the ciphertext is in the user's local cipher heap. SEND adds a message to the receiving user's message queue after checking that all keys within the message are in the sending user's key heap, that all ciphers within the message are in the sending user's local cipher heap, and that the receiving and sending users are not the same. SEND also records the nonce of the sent ciphertext, and although it is not depicted in the rules, delivers the message to the adversary. Finally, RECV reads the next message in the user's message queue that matches the receive pattern, adds all unencrypted keys contained in the message to the user's local key heap, and records the nonce of the received ciphertext.

C. Protocol Hygiene and Adversary Safety

The top-level correctness property of *SPICY* protocols demonstrates that *any* possible program trace (generated via labels of the operational semantics) that can be produced by the *Real World* can be matched by one produced in the *Ideal World*. The ultimate safety property requires that these traces must be executed in an environment in which an adversary is actively attempting to disrupt the behavior of the protocol. A key benefit of *SPICY* is that it allows a protocol developer to design her protocol in a gentler, adversary-free context. As long as the developer follows a few rules, *SPICY* can appeal to the Strong Preservation Theorem to ensure that the final protocol is safe in the presence of an adversary.

To demonstrate program-trace inclusion, we generate a stronger criterion in the form of a simulation argument between the *Real* and *Ideal* protocol descriptions, i.e., we construct a binary relation R between the *Real* ($\mathfrak{R}$) and *Ideal*

[4]Adversary protocol rules differ here: they may use any nonce-generation strategies they choose.

$$\frac{\langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, c_1\rangle]\rangle \to \langle \mathsf{C}', \kappa', \mu[\hat{u} \mapsto \langle \varsigma', \mathsf{ks}', \boxtimes', \mathrm{N}', c_1'\rangle]\rangle}{\langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, x \leftarrow c_1; c_2(x)\rangle]\rangle \to \langle \mathsf{C}', \kappa', \mu[\hat{u} \mapsto \langle \varsigma', \mathsf{ks}', \boxtimes', \mathrm{N}', x \leftarrow c_1'; c_2(x)\rangle]\rangle}\ \textsc{BindRecurse}$$

$$\frac{}{\langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, x \leftarrow \mathbf{Return}(v); c_2(x)\rangle]\rangle \to \langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, c_2(v)\rangle]\rangle}\ \textsc{BindProceed}$$

$$\frac{\mathsf{fresh}\ k_{\mathrm{ID}} \qquad \kappa' = \kappa[k_{\mathrm{ID}} \mapsto (\mathit{keytype}, \mathit{usage})] \qquad \mathsf{ks}' = \mathsf{ks}[k_{\mathrm{ID}} \mapsto \mathit{priv}]}{\langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, \mathbf{GenerateKey}(\mathit{keytype}, \mathit{usage})\rangle]\rangle \to \langle \mathsf{C}, \kappa', \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}', \boxtimes, \mathrm{N}, \mathbf{Return}(k_{\mathrm{ID}})\rangle]\rangle}\ \textsc{GenKey}$$

$$\frac{\mathsf{fresh}\ c_{\mathrm{ID}} \qquad \mathsf{ks}[k^{\sigma}] = \mathit{priv} \qquad \forall (k'_{\mathrm{ID}}, \phi) \in m.\ \phi \leq \mathsf{ks}[k'_{\mathrm{ID}}] \qquad \mathsf{C}' = \mathsf{C}[c_{\mathrm{ID}} \mapsto (m, k^{\sigma}, \hat{u}_{rec})]}{\langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, \mathbf{Sign}(k^{\sigma}, \hat{u}_{rec}, m)\rangle]\rangle \to \langle \mathsf{C}', \kappa, \mu[\hat{u} \mapsto \langle \varsigma \cup \{c_{\mathrm{ID}}\}, \mathsf{ks}, \boxtimes, \mathrm{incNon}(\mathrm{N}), \mathbf{Return}(c_{\mathrm{ID}})\rangle]\rangle}\ \textsc{Sign}$$

$$\frac{\mathsf{fresh}\ c_{\mathrm{ID}} \qquad \mathsf{ks}[k^{\sigma}] = \mathit{priv} \qquad \mathsf{ks}[k^{\varepsilon}] \geq \mathit{pub} \qquad \forall (k'_{\mathrm{ID}}, \phi) \in m.\ \phi \leq \mathsf{ks}[k'_{\mathrm{ID}}] \qquad \mathsf{C}' = \mathsf{C}[c_{\mathrm{ID}} \mapsto (m, k^{\sigma}, k^{\varepsilon}, \hat{u}_{rec})]}{\langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, \mathbf{SignEncrypt}(k^{\sigma}, k^{\varepsilon}, \hat{u}_{rec}, m)\rangle]\rangle \to \langle \mathsf{C}', \kappa, \mu[\hat{u} \mapsto \langle \varsigma \cup \{c_{\mathrm{ID}}\}, \mathsf{ks}, \boxtimes, \mathrm{incNon}(\mathrm{N}), \mathbf{Return}(c_{\mathrm{ID}})\rangle]\rangle}\ \textsc{Encrypt}$$

$$\frac{c_{\mathrm{ID}} \in \varsigma \qquad \mathsf{ks}[k^{\sigma}] \geq \mathit{pub} \qquad \mathsf{C}[c_{\mathrm{ID}}] = (m, k^{\sigma}, \hat{u}')}{\langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, \mathbf{Verify}(k^{\sigma}, c_{\mathrm{ID}})\rangle]\rangle \to \langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, \mathbf{Return}(m)\rangle]\rangle}\ \textsc{Verify}$$

$$\frac{c_{\mathrm{ID}} \in \varsigma \qquad \mathsf{ks}[k^{\varepsilon}] = \mathit{priv} \qquad \mathsf{ks}[k^{\sigma}] \geq \mathit{pub} \qquad \mathsf{C}[c_{\mathrm{ID}}] = (m, k^{\sigma}, k^{\varepsilon}, \hat{u}') \qquad \mathsf{ks}' = \mathsf{ks} \cup \{k \mid k \in m\}}{\langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, \mathbf{Decrypt}(k^{\sigma}, k^{\varepsilon}, c_{\mathrm{ID}})\rangle]\rangle \to \langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}', \boxtimes, \mathrm{N}, \mathbf{Return}(m)\rangle]\rangle}\ \textsc{Decrypt}$$

$$\frac{\forall k \in m.\ k \in \mathsf{ks}_1 \quad \forall c \in m.\ c \in \varsigma_1 \quad \hat{u} \neq \hat{u}_{rec} \quad \mu[\hat{u}_{rec}] = \langle \varsigma_2, \mathsf{ks}_2, \boxtimes_2, \mathrm{N}_2, c_2\rangle \quad \mu' = \mu[\hat{u} \mapsto \langle \varsigma_1, \mathsf{ks}_1, \boxtimes_1, \mathrm{updSents}(\mathrm{N}_1, m), \mathbf{Return}()\rangle}{\langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma_1, \mathsf{ks}_1, \boxtimes_1, \mathrm{N}_1, \mathbf{Send}(m, \hat{u}_{rec})\rangle]\rangle \xrightarrow{(m, \hat{u})} \langle \mathsf{C}, \kappa, \mu'[\hat{u}_{rec} \mapsto \langle \varsigma_2, \mathsf{ks}_2, \boxtimes_2 +\!\!+ [m], \mathrm{N}_2, c_2\rangle]\rangle}\ \textsc{Send}$$

$$\frac{\boxtimes = ms_1 +\!\!+ m :: ms_2 \qquad m \in p \qquad \forall m \in ms_1.\ m \notin p \qquad \mathsf{ks}' = \mathsf{ks} \cup \{k \mid k \in m\}}{\langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, \mathbf{Recv}(p)\rangle]\rangle \xrightarrow{(m, \hat{u})} \langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma \cup \{c \mid c \in m\}, \mathsf{ks}', ms_1 +\!\!+ ms_2, \mathrm{updRecvs}(\mathrm{N}, m), \mathbf{Return}(m)\rangle]\rangle}\ \textsc{Recv}$$

$$\frac{}{\langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, \mathbf{Gen}\rangle]\rangle \to \langle \mathsf{C}, \kappa, \mu[\hat{u} \mapsto \langle \varsigma, \mathsf{ks}, \boxtimes, \mathrm{N}, \mathbf{Return}(v)\rangle]\rangle}\ \textsc{GenContent}$$

Fig. 8: Small-step semantics for the *Real World* language.

($\mathfrak{I}$) universe states. The simulation statement for *SPICY* is comprised of three clauses, explaining the allowed transitions between elements of R. A *Real World* silent step ($\to$) is matched by any number of *Ideal World* silent steps ($\to^*$). A *Real World* labeled step ($\xrightarrow{\ell}$) requires any number of *Ideal World* silent steps followed by a *matching* labeled step for the *same user*. Finally, we ensure that when the protocol is finished running (i.e., cannot take any more steps), any users whose protocols are terminal (**Return**) have matching ***Return*** values in the specification.[5] We illustrate these simulation-stepping rules in the commutative diagrams shown in Fig. 9.

Remember, the labels in *SPICY*'s operational semantics correspond to *observable* (send/receive) actions. We mark secrecy-enforcing operations (as well as adversary steps) as *silent*, which allows us to use a simpler semantics in the *Ideal World*. In general, the correspondence of cryptographic operations with actions on channels is very loose indeed, and there will be many ways of correctly matching the specification. To match labels (comprised essentially of messages) between *Real* and *Ideal* representations, we define an equivalence between messages. Simply, the underlying messages must be the same (after unwrapping *Real World* ciphers);

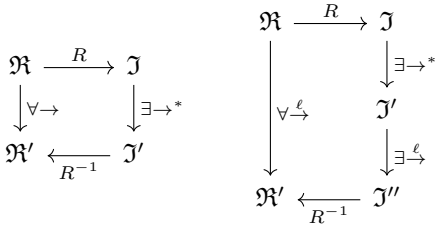

Fig. 9: Commutative diagrams depicting the simulation-statement stepping rules. *Real World* silent steps ($\mathfrak{R} \to \mathfrak{R}'$) allow the *Ideal World* to step silently. *Real World* labeled steps ($\mathfrak{R} \xrightarrow{\ell} \mathfrak{R}'$) allow the *Ideal World* to take any number of silent steps followed with a *matching* labeled step.

i.e., bare payloads must match, corresponding elements of paired messages must be the same, and permissions in the *Ideal World* must properly correspond to keys in the *Real World*. The permission-key correspondence is a little subtle. Access to a symmetric key or a private asymmetric key confers complete control over how that key is used, meaning the user must also have full control over the corresponding channel (rw). A signing key corresponds to an authenticated channel from one user to any number of others, representing r- access. Similarly, a public encryption key corresponds to a secure communication channel from any number of users to the key

[5] Incidentally, to prove refinement of the *Ideal World* to simplified *Ideal World* specification for the SECRET SHARING PROTOCOL, we used a simulation statement that only checks these final return labels, allowing intermediate labels to vary.

owner, representing -w access.

To ensure that protocols are implemented safely, we break from standard practice and add an additional clause to the simulation statement which enforces the hygiene rules. Ultimately, these rules ensure that honest parties cannot be tricked into leaking honest keys to adversaries. They refer to a set of *honest keys*, initialized to the bootstrapping keys of the protocol and growing to include all keys generated by any honest participants. The hygiene rules are defined step-by-step as a predicate (*safety*) over the next (honest) command to be executed, with interesting behavior for only the *Real World* DSL commands:

- **Sign:** all keys in a signed message are honest and public.
- **Encrypt:** messages are only encrypted with honest keys, and all keys within the payload must be honest.
- **Receive:** all receive patterns check for honest signing keys and enforce replay protection.
- **Send:** every sent message is encrypted or encrypted and signed by the sender, sent to the user it is addressed to, and has not been sent before.

These conditions are further analogues of e.g. Java array-bounds checks, a sort of dynamic opportunity to note that conservative safety rules have been violated. Actually, a better analogy would be (since the *SPICY* checks do not require dynamic runtime monitoring) trying to call a nonexistent method on a Java object, which should be caught statically. Interestingly, as for many aspects of Java type safety, we can check our safety rules statically, via a simple type checker we wrote to apply to honest-party protocol code (and proved to imply the declarative conditions). Provided that the developer adheres to these rules, *SPICY* provides a final safety proof in the presence of an adversary running arbitrary code.

We are ready now to state the Strong Preservation Theorem, beginning by defining "refinement." Given *Real* ($\mathfrak{R}$) and *Ideal* ($\mathfrak{I}$) protocol descriptions, the *Real World refines* the *Ideal World* ($\mathfrak{R} \trianglelefteq \mathfrak{I}$) if there exists a simulation relation (R) between them. The refinement condition is used to prove *trace inclusion* between the *Real* and *Ideal* Worlds. Additionally, since we have augmented the simulation statement with cryptographic-best-practice rules, *SPICY* can lift the refinement condition that the protocol developer proves in a *Real World* universe without adversaries into one with an active adversary. More precisely:

Theorem 1 (**Strong Preservation Theorem**). *Given a* Real World *(*$\mathfrak{R}$*)*[6] *and an* Ideal World *(*$\mathfrak{I}$*), if* $\mathfrak{R} \trianglelefteq \mathfrak{I}$ *in an adversary-free* Real World*, and we inject arbitrary adversary code (*$\mathcal{A}_{\text{code}}$*) into* $\mathfrak{R}$*, then any trace of the augmented* Real World $\mathfrak{R} \otimes \mathcal{A}_{\text{code}}$ *can also be produced by* $\mathfrak{I}$*.*

The key supporting lemma for Theorem 1 lifts refinement proofs from an adversary-free *Real World* to the *Real World* with an adversary executing arbitrary protocol code. Consider refinements predicated over the strength of the adversary $\mathfrak{R} \trianglelefteq \mathfrak{I} \;/\; P$, where $\mathfrak{R}$ and $\mathfrak{I}$ are the *Real* and *Ideal* universe states respectively, and P is a predicate over the adversary: $P : \mathcal{A}_{\text{code}} \to \mathbb{B}$. Then we state the following lemma:

Lemma 1.1. *Given* $P_R(A) = (\exists\ r.\ A = \textbf{\textit{Return}}\ r)$ *and* $\mathfrak{R} \trianglelefteq \mathfrak{I} \;/\; P_R$*, we can derive a new refinement condition for an adversary with arbitrary code:* $\mathfrak{R} \trianglelefteq \mathfrak{I} \;/\; \text{True}$.

The big insight in completing the proof of Lemma 1.1 is that we can derive a simulation relation (R') for the adversary-augmented *Real World* ($\mathfrak{R}'$) by simply stripping out all vestiges of the adversary from $\mathfrak{R}'$ and appealing to the original simulation relation R. Specifically, the stripping operation:

1) Removes dishonest keys from the global key heap and all users' permission heaps.
2) Removes dishonest ciphers from the global cipher heap.
3) Removes non-honestly-signed or replayed messages from all message queues.
4) Sets the adversary's code to a no-op (**Return**).

We can use the new derived simulation relation R' within the following lemma to prove Lemma 1.1 directly:

Lemma 1.2. *Given a simulation relation* R *for a protocol within a* Real World *without adversaries, if we augment the* Real World *with arbitrary adversary code (*$\mathfrak{R}' = \mathfrak{R} \otimes \mathcal{A}_{\text{code}}$*) with* $(\mathfrak{R}', \mathfrak{I}) \in R'$*, then for any* Real World *step (*$\mathfrak{R}' \to \mathfrak{R}''$*), we can find appropriate* Ideal World *step(s) (*$\mathfrak{I} \to^* \mathfrak{I}'$*) such that* $(\mathfrak{R}'', \mathfrak{I}') \in R'$.

There is an important point here that justifies *SPICY*'s codified hygiene rules. The *only* reason we are able to derive R' after performing the stripping operation is because of hygiene rule enforcement. Indeed, the rules emerged during the development of *SPICY* as necessary conditions for being able to lift the refinement property from an adversary-free *Real World* into one with an active adversary. Essentially, the hygiene rules place the necessary restrictions on honest-party protocols, allowing us to strengthen the induction hypothesis via automatically maintained predicates over the *Real World* universe state. These universe predicates are observations that the *Real World* universe state has not gone "wrong" and are only implementable because of the hygiene rules. The good news is that these emergent rules are simply part of the lore of building secure cryptographic protocols and do not represent anything that experts in the field do not already know.

We also want to return here to the subject of adversary message deduction rules. Other protocol-analysis approaches must mention those rules explicitly in the primary security condition for a protocol, to force consideration of every way an adversary could interfere. Thanks to our hygiene conditions, we avoid any explicit consideration of adversary actions. There truly is no analogue to those deduction rules in our framework; while our baseline *Real World* operational semantics are also applied to adversary processes, we do not need to reason explicitly about any such process, in verifying a protocol.

[6]Honest users' mailboxes must start with only honest messages, and all preprovisioned keys and ciphertexts must be honest.

D. Automated Protocol Model Checking

For a wide range of *Real* and *Ideal* World protocol descriptions, we can automatically generate the simulation relation R via symbolic protocol execution while simultaneously performing the necessary model-checking exploration. A predicate, *align*, checks that each *Real World* user who can make a labeled step has a corresponding, matching labeled *Ideal World* step. The step-relation rules found in Fig. 10 show how we track a tuple of *Real World*/*Ideal World* universe states $(\mathfrak{R} \otimes \mathfrak{I})$ through allowed transitions.

$$\frac{\mathfrak{R} \to \mathfrak{R}'}{(\mathfrak{R}, \mathfrak{I}) \to (\mathfrak{R}', \mathfrak{I})}\ \epsilon \qquad \frac{\neg\ align(\mathfrak{R}, \mathfrak{I})}{(\mathfrak{R}, \mathfrak{I}) \to \bot}\ \bot$$

$$\frac{\mathfrak{R} \xrightarrow{\ell_r} \mathfrak{R}' \quad \mathfrak{I} \to^* \mathfrak{I}' \quad \mathfrak{I}' \xrightarrow{\ell_i} \mathfrak{I}'' \quad \ell_r \cong \ell_i \quad align(\mathfrak{R}, \mathfrak{I})}{(\mathfrak{R}, \mathfrak{I}) \to (\mathfrak{R}', \mathfrak{I}'')}\ \ell$$

Fig. 10: Rules for model-checking state-space exploration. The rules are driven by *Real World* steps describing how we transition for silent steps (ϵ), misaligned steps ($\bot$), and aligned labeled steps (ℓ).

Using the state-transition rules in Fig. 10, we begin from an initial state with the full protocol description, along with any necessary predistributed channels and keys. From there, we continuously apply the stepping rules, which accommodate nondeterminism in the protocol execution by following all possible protocol paths, until the protocol gets stuck or there is a label misalignment (upon which we know the protocol is invalid and can terminate model checking). We check whether each intermediate state satisfies the necessary safety properties: *align* and *safety*. A lemma verifies the soundness of our model-checking procedure.

Lemma 1.3. *Given an initial protocol state $(\mathfrak{R}, \mathfrak{I})$, whose transition system is defined by Fig. 10: if $\mathfrak{R}$ only contains* honest *preprovisioned keys and empty message queues and* align *and* safety *are invariants of the transition system, then* $\mathfrak{R} \trianglelefteq \mathfrak{I}$.

As stated, the model-checking state space is essentially infinite in at least two different ways. We use the Strong Preservation Theorem to "quotient out" the adversary, focusing the search on *honest-party* execution paths while ignoring the *arbitrarily large* adversary executions. We further reduce the possible honest-party execution paths by performing silent steps greedily, since one party's silent steps have no observable consequences for others. One other source of state-space explosion is, e.g., in the space of possible plaintexts that are potentially allowed to pass among honest parties. Our model checker, implemented as a tactic in Coq, detects that kind of free variable of states and automatically quantifies them existentially in the simulations it computes. All of these state-space simplifications are justified via framework theorems which automatically tie per-protocol safety results to the top-level refinement condition.

In summary, using nearly the most naïve technique possible, we have demonstrated in *SPICY* that we can verify protocols with essentially infinitely sized state spaces using two tricks. First, we eliminate the infinitude of possible adversary execution interleavings by appealing to the Strong Preservation Theorem. Second, we allow for arbitrarily large message and ciphertext spaces by representing them symbolically during state exploration. In the next section, we evaluate how well this technique works on a collection of nontrivial protocols.

IV. Evaluation

As we have discussed in prior sections, *SPICY* is an experiment set up to study whether it is possible to design and implement a custom language for cryptographic protocols that forces security of implementations without the protocol author having to reason about adversaries explicitly. We implemented *SPICY* in the Coq proof assistant, providing us with strong security guarantees, at a cost in verification performance. In this first iteration, we chose to use naïve model-checking techniques to check safety of our case-study protocols and ensured that we could automate the proof scripts, giving the user a mostly push-button verification experience for correct protocols. Our high-level safety proofs (in particular, the Strong Preservation Theorem) establish the soundness of this approach.

In this evaluation of *SPICY*, we choose protocols that, together, a) exercise the main features of the languages, b) are close to real-world protocols, and c) exercise the limitations of our model-checking approach. We want to show that *SPICY* can, in fact, verify realistic protocols and that it can do so using a reasonable amount of resources on commodity hardware. To that end, for each protocol we measure proving time, proving memory, number of parties involved, and lines of code (LoC) and specification (LoS). The performance numbers were gathered on a Dell XPS 15 (2019) laptop with an Intel i9-9980HK processor and 64GB of RAM and should be considered approximate. Note that in all cases, the correctness proofs have been automated, and all that the *SPICY* user must provide are valid, corresponding *Real* and *Ideal* protocol descriptions.

A. The Evaluation Protocols

We have implemented and proven correct four protocols to demonstrate the capabilities of *SPICY*. Many of the protocols highlight the benefits that the mixed-embedding style provides *SPICY* – that one can run arbitrary Gallina computations as a part of a protocol without having to encode them directly as features of the DSL. The specification code, implementation code, and bootstrapping configurations for all of the following protocols can be found in the Appendix.

PGP variant. In §II, we presented SECRET SHARING PROTOCOL, which demonstrated a number of language features including key generation and cryptographic and messaging operations. A few tweaks to that protocol bring us one quite similar to the well-known PGP [19] protocol, with one key difference: a single PGP message is split into two consecutive

messages. In PGP, the sender transmits both the symmetrically encrypted content and the asymmetrically encrypted content key to the recipient in a single, signed message. *SPICY* currently only allows the creation of ciphertexts via a single operation, so ciphertexts cannot include message payloads that are signed differently or encrypted with different keys.

Secure DNS variant. Next, we present a three-party protocol: an adaptation of the DNSCurve [20] secure DNS protocol proposed by D. J. Bernstein. In DNSCurve, when a user wants to resolve an IP address by name, she first requests a response from a local DNS cache. That cache makes a signed, encrypted request to an authoritative server, and if the server can resolve the name, it returns a response, which the local DNS cache forwards back to the user. In our adaptation, we assume a single DNS cache and authoritative server that can always resolve requested IP addresses, eliding some particulars of the way in which keys are managed in the actual DNSCurve protocol.[7]

Secret aggregation. The third protocol we examine is a four-party protocol where three parties trust a server to perform a sensitive computation on data aggregated from each of the parties. This example is a bit contrived but further illustrates embedding arbitrary computations within protocol definitions. Here, each user sends an encrypted message containing their salary to the server, which then computes and reports the average of the results.

Network authentication. As a final example, we developed a simple protocol for mutual authentication over an insecure network using a trusted third party. This protocol demonstrates an alternative bootstrapping mechanism for distributing public signing keys based on the trusted identity of a single server rather than pairwise secrets established between all parties. In our implementation here, two users who want to communicate securely ask the server for a (symmetric) key they can use to establish a secure communication channel.

B. Discussion

In this section, we discuss how *SPICY* fares verifying the protocols from §IV-A, placed into context with some of the popular tools in this space, specifically ProVerif [21] and Tamarin [22]. We defer a more detailed analysis of related work to the next section and focus here on verifier performance versus tool implementation and usage complexity.

SPICY presents a programming model that should be fairly familiar to developers, particularly those with functional-programming experience. By leveraging a mixed embedding within Gallina, we are able to present small DSLs while still allowing developers to use more complex programming-language features. For example, the example protocols include features like pattern matching, server loops, associative map lookups, and arithmetic calculations, by appealing to the Gallina metalanguage rather than extending our formal DSLs. This kind of capability allows *SPICY* to handle *stateful protocols*, something that ProVerif cannot handle (though Tamarin can). The programming experience feels very much like writing secure-messaging protocols, with the added benefits of the safety checks that *SPICY* provides. Both Tamarin and ProVerif present an arguably less familiar programming interface (additionally requiring explicit setup for how the adversary could impact the modeled protocol), though they currently handle a wider variety of protocols than *SPICY* can (something we hope to address in future work).

TABLE I: Evaluation Protocol Metrics

Protocol	Characteristics			Performance	
	Parties	LoC	LoS	Time (min)	Mem. (GB)
PGP variant	2	12	8	1	2
Secure DNS var.	3	23	15	3	4
Aggregation	4	10	16	16	14
Network auth.	3	21	13	36	30

LoC: lines of code, LoS: lines of specification

SPICY successfully proves all example protocols correct, with reasonable performance numbers, taking minutes to tens of minutes and using a couple of GB to tens of GB of memory. A quick glance at the summary performance numbers in Table I shows that the labeled-step nondeterminism of Secret aggregation and Network authentication increases the proving cost in both time and memory. Though all four of these protocols could be modeled in both Tamarin and ProVerif with better performance characteristics, we note that *SPICY* both uses a fairly naïve model-checking procedure and implements this procedure as a tactic in Coq, running into bottlenecks in Coq's tactic engine (e.g., certain primitive tactics requiring time quadratic in goal size or worse). While the assurance guarantees that we obtain by implementing the model-checking procedure in Coq are solid, the performance characteristics are not on par with standalone tools. Ultimately, Coq was not optimized for this kind of use case, and performance bottlenecks incurred via the Coq implementation could be overcome by implementing as a standalone tool (like both Tamarin and ProVerif). As an estimate of the difficulty to implement a standalone version of *SPICY*, in comparison to other popular tools, we use the SCC [23] program to estimate the lines of code, shown in Table II. Though the full implementation of *SPICY* with complete proofs is ~ 25 kLoC, we have eliminated theorem proofs and nonessential theorem statements since they would not be present in a standalone implementation.

TABLE II: Implementation complexity

Tool	Implementation Language	kLoC
SPICY	Coq	4
Tamarin	Haskell	23
ProVerif	OCaml	44

kLoC: thousands of lines of code

The point here is that *SPICY* demonstrates feasibility of small, well-designed languages to build developer tooling for

[7]For example, DNSCurve servers publish their public keys as parts of their hostnames.

writing safe cryptographic protocols. A few example protocols show the limits of the model-checking method, and indeed we explored this method as something of a worst case for application of our new operational semantics. We conclude that eliminating the adversary has restricted state spaces by enough that even brute-force methods can validate interesting protocols. To improve performance, follow-on efforts would use more powerful reasoning tools, e.g. relational program logics, on top of our languages as well as creating a more efficient implementation, perhaps though extracting the analysis from Coq into a lower-level language.

C. Limitations

Let us step back for a moment to discuss some broader limitations of the current implementation of *SPICY*. We have shown (§IV-B) that, despite limiting both the implementation and specification languages, we can implement and prove correct a wide variety of interesting cryptographic protocols. There are, however, extensions we envision that would expand the implementable protocols. *SPICY* cannnot, for example, handle protocols which allow communication amongst potentially dishonest individuals. In this initial work, we require honest messages to be signed, and the security analysis exploits this restriction. We would have to think carefully about how to relax this requirement, and we think it would be a good direction for future work. Additionally, we currently examine only single protocol executions, while some protocols (like Signal) rely on particular mechanisms to ensure their security when multiple sessions are running. When we extend this work with more powerful reasoning capabilities (beyond simple model checking), it would be appropriate to examine this kind of use case as well.

In summary, we view this work as a first step towards a new way of performing protocol analysis. Future work could extend our languages, reasoning capabilities, and adversarial models so that we can examine an even wider range of protocols and threat models. We think this research direction is an exciting one and look forward to exploring it even further.

V. Related Work

Recent work [10] provides a taxonomy of tools in the computer-aided-cryptography (CAC) space, categorizing them based on their focus (protocol design, correctness and optimization, or deployment) and cryptographic model (symbolic or computational). *SPICY* fits into the design-level, symbolic category. In comparison to other tools in this category, the kinds of protocols that *SPICY* can analyze sit at a higher level of abstraction.

The big idea in *SPICY* is that we can formally codify what it means for a cryptographic protocol to be secure into a pair of languages specialized for this purpose. Using these languages, we develop a top-level Strong Preservation Theorem which justifies disregarding the adversary as long as the protocol developer follows these formal safety rules (which are checked by *SPICY*). Cortier et al. [24] proposed a related technique with very much the flavor of compiling the *SPICY Ideal World* specifications into *Real World* implementations. Specifically, they transform a cryptographic protocol automatically into one that is safe from an active adversary (they also handle unbounded numbers of protocol executions, which *SPICY* does not yet handle). Their approach is a bit more heavy-handed than ours, requiring all messages to be encrypted (rather than just signed) and imposing stronger PKI assumptions (whereas protocols in *SPICY* can be written to bootstrap identity management). Developers can write cryptographic *applications* in *SPICY* and produce executable code, while Cortier et al. stick to traditional sequential message-sending charts.

Sprenger and Basin [25] present a similar idea to ours in that they verify protocols via refinement, much like we have the *Real World* implementation refine the *Ideal World* specification. Though their approach has larger scope in terms of protocols and security arguments, it requires specific formal-methods work for each new protocol to be verified. Each new simulation argument involves cleverness in finding the right simulation relations and invariants, followed by a manual tactic-based proof. In contrast, we provide to programmers what we hope is an intuitive property similar to type safety, which can be understood independently of formal methods. Fully automated security proofs follow in some cases that we demonstrated. Unlike *SPICY*, Sprenger and Basin's protocols do not seem designed to be executed in real deployments.

Other symbolic cryptographic protocol analyzers, like *SPICY*, have substantial automated verification steps. Unlike *SPICY*, the object of their verification is to prove security properties *per protocol*, always in a world with an attacker. Tools backed by model checking search for "failed" states in which the adversary has violated a desired property, while others (e.g., Maude-NPA [26], Tamarin [22], and Sapic [27]) consume models of adversarial behavior and descriptions of failure states. In either case, adversarial activity balloons the search space, though these tools use an array of specialized techniques to manage the complexity. Other tools take a language-based approach similar to ours, treating the adversary as a program executing arbitrary code. The F7 family [28], [29] employs a refinement type system with which the programmer can encode a variety of security properties. Adversary programs are ascribed a distinguished type, and proving safety amounts to type checking (which is discharged via an SMT solver). Like *SPICY*, scyther-proof [30] formalizes the operational semantics for its language in a proof assistant, Isabelle/HOL; however, the proof-generation algorithm that exercises those semantics always does so in a world with an adversary (c.f. $\mathfrak{R} \otimes \mathcal{A}_{\text{code}}$ from §III-C).

Regarding the ability to input programs within a *familiar language* and generate *executable code*, the tools based on F# (like F7 and fs2pv [31]) are the most similar to *SPICY*. Other tools tend to express protocols with an (arguably) difficult syntax for nonexperts. Like F7 and fs2pv, *SPICY* stands out in that we have a relatively straightforward path to executable protocol code. Unlike some other CAC tools (e.g., Fiat Cryptography [14]), *SPICY* does not focus on the correct, efficient implementation of low-level cryptographic code. We have

ideas on how to extend *SPICY*'s code-generation capabilities but leave that to future work. To our knowledge in comparing with this category, *SPICY*'s mechanisms of *message patterns* and automatic nonce tracking are novel and facilitate full isolation of the adversary away from observations by honest parties, simplifying proofs.

The story for *equational theories* is a bit more nuanced, as many tools include at least partial support. Equational theories are used, e.g., to define algebraic properties of cryptographic functions or refine adversary capabilities. *SPICY* currently does not allow users to extend the language in this way.

We see analogies (e.g. in the way we handle nonce tracking automatically) to work on cryptographic compilers (e.g. [32], [33], [34]) that take suitably annotated programs and compile them to run in interesting models like secure multiparty computation. However, the mission of those tools is to start with ordinary programs and add elements of secure distributed execution automatically (modulo certain annotations), while *SPICY* exposes distributed computation directly, at a higher level of abstraction than usual (i.e., with no adversary) while giving protocol designers more freedom to optimize.

There has been significant past work using proof assistants like Coq to prove the security of cryptographic protocols. One of the best-known frameworks is CertiCrypt [35], which performs computational-model proofs, therefore providing stronger assurance than with symbolic-model proofs but also requiring substantial human effort to write the proofs (which are then checked automatically). The follow-on tool EasyCrypt [15] is a standalone implementation, not a library within a proof assistant, which enables more automation while growing the trusted code base. Other proof-assistant-based frameworks in this family (based on sequences of games with proofs of probabilistic refinement) include CryptHOL [36] and the Foundational Cryptography Framework (FCF) [37]. A general concern in these frameworks is explicit modeling of which computations the adversary could perform in polynomial time, while our framework justifies dropping the adversary from proofs (albeit without computational-hardness bounds). Some exercises have been carried out connecting these frameworks to lower-level functional correctness. For instance, Beringer et al. [38] connected an HMAC security proof in FCF to a correctness proof for a C implementation fed to a verified compiler. We are eager to study this kind of connection for our own work.

VI. Conclusions and Future Work

SPICY is a new toolchain that enables practitioners not versed in formal methods *or* cryptography to design and develop secure cryptographic protocols. We have demonstrated through a suite of examples that *SPICY* can be used to create secure designs of a variety of protocols – though the scope is intentionally not *all* protocols, any more than Java should be considered the right language for all programming. Rather, we carefully designed a language that enforces certain pieces of conventional protocol-design wisdom using certain conservative mechanisms, therefore guaranteeing that the adversary-free intuitions of everyday programming are sound for developers to apply.

The past few decades have seen a thriving design ecosystem for type-safe programming languages, with new features appearing to strike new balances between flexibility, performance, and safety. We hope to see the same for adversary-safe languages. From the starting point of *SPICY*, adding lower-level crypto primitives and allowing for ephemeral keys would allow analysis of more complex protocols. Relaxing restrictions on how cryptographic operations can be combined within messages, as discussed in PGP variant, would permit analysis of a variety of modern protocols. Increasing the complexity of the protocol languages would also increase the value of further automation of the correspondence between the simple *Ideal World* and intermediate *Ideal World* implementations. Naïve model checking is only the beginning of reasoning tools taking advantage of adversary safety. We hope to explore both classic model-checking optimizations (e.g., partial-order reduction and modularity) and alternative proof techniques like relational program logics. Finally, we would like to extend the code-generation capabilities, providing a proved connection to efficient low-level protocol-implementation code.

Acknowledgment

This material is based upon work supported by the Dept of the Air Force under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Dept of the Air Force.

References

[1] N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt, "On the Security of RC4 in TLS," in *Proceedings of the 22nd USENIX Security Symposium (SEC)*. USENIX Association, pp. 305–320.

[2] N. J. AlFardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," in *Proceedings of the 2013 IEEE Symposium on Security & Privacy (S&P)*. Institute of Electrical and Electronics Engineers (IEEE), pp. 526–540.

[3] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt, "DROWN: Breaking TLS Using SSLv2," in *Proceedings of the 25th USENIX Security Symposium (SEC)*. USENIX Association, pp. 689–706.

[4] 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 *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 15. Association for Computing Machinery, pp. 5–17.

[5] R. Chandramouli and S. Rose, "Secure Domain Name System (DNS) Deployment Guide."

[6] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Proceedings of the 2006 Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*. Springer International Publishing, pp. 1–20.

[7] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient Cache Attacks on AES, and Countermeasures," vol. 23, pp. 37–71.

[8] B. Brumley, M. Barbosa, D. Page, and F. Vercauteren, "Practical Realisation and Elimination of an ECC-Related Software Bug Attack," in *Proceedings of the 2012 Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*. Springer International Publishing, pp. 171–186.

[9] T. C. D. Team, "The coq proof assistant." [Online]. Available: https://doi.org/10.5281/zenodo.4501022
[10] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "SoK: Computer-Aided Cryptography," in *Proceedings of the 2021 IEEE Symposium on Security & Privacy (S&P)*. Institute of Electrical and Electronics Engineers (IEEE), pp. 123–141.
[11] E. Rescorla. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3. [Online]. Available: https://tools.ietf.org/html/rfc8446
[12] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate," in *Proceedings of the 2017 IEEE Symposium on Security & Privacy (S&P)*. Institute of Electrical and Electronics Engineers (IEEE), pp. 483–503.
[13] B. Beurdouche, F. Kiefer, and T. Taubert. Verified Cryptography for Firefox 57. Mozilla Blog. [Online]. Available: https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/
[14] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises," in *Proceedings of the 40th IEEE Symposium on Security & Privacy (S&P)*. Institute of Electrical and Electronics Engineers (IEEE), pp. 1202–1219.
[15] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-Aided Security Proofs for the Working Cryptographer," in *Proceedings of the 31st Annual Cryptology Conference (CRYPTO)*. Springer International Publishing, pp. 71–90.
[16] D. Dolev and A. C. Yao, "On the Security of Public Key Protocols," vol. 29.
[17] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using crash hoare logic for certifying the fscq file system," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. Association for Computing Machinery, pp. 18–37.
[18] A. Chlipala, "Skipping the binder bureaucracy with mixed embeddings in a semantics course (functional pearl)," in *Proceedings of the 26th ACM SIGPLAN International Conference on Functional Programming (ICFP'21)*.
[19] D. Atkins, W. Stallings, and P. Zimmermann. RFC1991: PGP Message Exchange Formats. [Online]. Available: https://tools.ietf.org/html/rfc1991
[20] D. J. Bernstein. DNSCurve: Usable Security for DNS. DNSCurve Protocol Website. [Online]. Available: https://dnscurve.org/
[21] B. Blanchet, "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif," vol. 1, pp. 1–135.
[22] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The TAMARIN Prover for the Symbolic Analysis of Security Protocols," in *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*. Springer International Publishing, pp. 696–701.
[23] B. Boyter, "Sloc Cloc and Code (scc)." [Online]. Available: https://github.com/boyter/scc
[24] V. Cortier, B. Warinschi, and E. Zălinescu, "Synthesizing secure protocols," in *Computer Security – ESORICS 2007*, J. Biskup and J. López, Eds. Springer Berlin Heidelberg, pp. 406–421.
[25] C. Sprenger and D. Basin, "Developing security protocols by refinement," in *7th ACM Conference on Computer and Communications Security (CCS 2010)*. Association for Computing Machinery (ACM), pp. 361–374.
[26] S. Escobar, C. Meadows, and J. Meseguer, *Foundations of Security Analysis and Design V*. Springer International Publishing, ch. Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties, pp. 1–50.
[27] S. Kremer and R. Künnemann, "Automated Analysis of Security Protocols with Global State," in *Proceedings of the 2014 IEEE Symposium on Security & Privacy (S&P)*. Institute of Electrical and Electronics Engineers (IEEE), pp. 163–178.
[28] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement Types for Secure Implementations," in *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*. Institute of Electrical and Electronics Engineers (IEEE), pp. 17–32.
[29] M. Backes, C. Hriţcu, and M. Maffei, "Union, Intersection, and Refinement Types and Reasoning About Type Disjointness for Secure Protocol Implementations," vol. 22, pp. 301–353.
[30] S. Meier, C. Cremers, and D. Basin, "Strong Invariants for the Efficient Construction of Machine-Checked Protocol Security Proofs," in *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF)*. Institute of Electrical and Electronics Engineers (IEEE), pp. 231–245.
[31] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified Interoperable Implementations of Security Protocols," in *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*. Institute of Electrical and Electronics Engineers (IEEE), pp. 152–166.
[32] D. Darais, I. Sweet, C. Liu, and M. Hicks, "A language for probabilistically oblivious computation," in *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL. Association for Computing Machinery (ACM), pp. 1–31.
[33] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. Association for Computing Machinery, pp. 87–101.
[34] Y. Ishai, M. Prabhakaran, and A. Sahai, "Founding cryptography on oblivious transfer – efficiently," in *Advances in Cryptology – CRYPTO 2008*, D. Wagner, Ed. Springer Berlin Heidelberg, pp. 572–591.
[35] G. Barthe, B. Grégoire, and S. Zanella-Béguelin, "Formal Certification of Code-Based Cryptographic Proofs," in *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Association for Computing Machinery (ACM), pp. 90–101.
[36] D. Basin, A. Lochbihler, and S. R. Sefidgar, "CryptHOL: Game-Based Proofs in Higher-Order Logic," vol. 33, pp. 494–566.
[37] A. Petcher and G. Morrisett, "The Foundational Cryptography Framework," in *Proceedings of the 4th International Conference on Principles of Security and Trust - Volume 9036*. Springer International Publishing, pp. 53–72.
[38] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel, "Verified Correctness and Security of OpenSSL HMAC," in *Proceedings of the 24th USENIX Security Symposium (SEC)*. Springer International Publishing, pp. 207–221.

APPENDIX

In this appendix, we show the source code for the protocols described in the case studies of §IV; including the specification code, the implementation code, and the initial configuration of the channels and keys for each user. In order to make the examples easier to read, we have eliminated some of the "ceremony" that is required in the current implementation of our languages (e.g., abbreviating some function calls, eliminating others, and dropping constructors of datatypes such as messages and permissions). We adopt the same convention for keys as §II (pk and sk represent asymmetric public and private keys, respectively; and κ represents a symmetric key). We use uppercase letters to name users (so, the first user in each protocol is $\hat{u}_A$). We label user-to-user channels with subscripts indicating source and destination: so, a channel from $\hat{u}_A$ to $\hat{u}_B$ is called $\mathrm{ch}_{\mathrm{AB}}$. All keys have subscripts that indicate their "ownership," where appropriate. Function calls that we do use are described along with each protocol. Initial configurations include the initial channel permissions (for specifications) and owned keys (for implementations).

User	Initial Configuration	Specification	Implementation
$\hat{u}_A$	$[\mathrm{ch}_{\mathrm{AB}} \mapsto rw, \mathrm{ch}_{\mathrm{BA}} \mapsto r\text{-}]$ $\{\mathsf{pk}^{\sigma}_{\mathrm{A}}, \mathsf{sk}^{\sigma}_{\mathrm{A}}, \mathsf{pk}^{\varepsilon}_{\mathrm{A}}, \mathsf{sk}^{\varepsilon}_{\mathrm{A}}, \mathsf{pk}^{\sigma}_{\mathrm{B}}\}$	$\mathrm{ch}_{\mathrm{S}} \leftarrow$ ***Recv*** $\mathrm{ch}_{\mathrm{BA}}$ $\Sigma \leftarrow$ ***Recv*** $(\mathrm{ch}_{\mathrm{S}} \cap \mathrm{ch}_{\mathrm{BA}})$ ***Return*** Σ	$c_1 \leftarrow$ **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{B}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{A}}$) $\kappa^{\varepsilon}_{\mathrm{S}} \leftarrow$ **Decrypt** c_1 $c_2 \leftarrow$ **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{B}}$ $\kappa^{\varepsilon}_{\mathrm{S}}$) $\Sigma \leftarrow$ **Decrypt** c_2 **Return** Σ
$\hat{u}_B$	$[\mathrm{ch}_{\mathrm{AB}} \mapsto r\text{-}, \mathrm{ch}_{\mathrm{BA}} \mapsto rw]$ $\{\mathsf{pk}^{\sigma}_{\mathrm{A}}, \mathsf{pk}^{\varepsilon}_{\mathrm{A}}, \mathsf{pk}^{\sigma}_{\mathrm{B}}, \mathsf{sk}^{\sigma}_{\mathrm{B}}\}$	$\mathrm{ch}_{\mathrm{S}} \leftarrow$ ***CreateChannel*** $\_ \leftarrow$ ***Send*** $(\mathrm{ch}_{\mathrm{S}} \mapsto \mathsf{rw})$ $\mathrm{ch}_{\mathrm{BA}}$ $\Sigma \leftarrow$ ***Gen*** $\_ \leftarrow$ ***Send*** Σ $(\mathsf{chs} \cap \mathrm{ch}_{\mathrm{BA}})$ ***Return*** Σ	$\kappa^{\varepsilon}_{\mathrm{S}} \leftarrow$ **GenerateKey** Sym Encryption $c_1 \leftarrow$ **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{B}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{A}}$ $\hat{u}_A$ $\kappa^{\varepsilon}_{\mathrm{S}}$ $\_ \leftarrow$ **Send** $\hat{u}_A$ c_1 $\Sigma \leftarrow$ **Gen** $c_2 \leftarrow$ **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{B}}$ $\kappa^{\varepsilon}_{\mathrm{S}}$ $\hat{u}_A$ Σ $\_ \leftarrow$ **Send** $\hat{u}_A$ c_2 **Return** Σ

Fig. 11: Our PGP variant definition is fairly similar to the SECRET SHARING PROTOCOL we described in §II. $\hat{u}_B$ establishes the secure channel (symmetric key) over which $\hat{u}_B$ can send the secret (Σ). As was stated in §IV, one difference between our implementation and that of the true PGP protocol is that we separately encrypt the generated symmetric key and the message payload, sending them in consecutive messages rather than a single one. Initial configurations contain preshared (asymmetric) keys and channels.

User	Initial Configuration	Specification	Implementation
$\hat{u}_A$	$[\mathrm{ch}_{\mathrm{AB}} \mapsto \text{-}w, \mathrm{ch}_{\mathrm{BA}} \mapsto r\text{-}]$ $\{\mathsf{pk}^{\sigma}_{\mathrm{A}}, \mathsf{sk}^{\sigma}_{\mathrm{A}}, \mathsf{pk}^{\varepsilon}_{\mathrm{A}}$ $,\mathsf{sk}^{\varepsilon}_{\mathrm{A}}, \mathsf{pk}^{\sigma}_{\mathrm{B}}, \mathsf{pk}^{\varepsilon}_{\mathrm{B}}\}$	serverLoop n_{iter} v_{ret} (m ← ***Recv*** $\mathrm{ch}_{\mathrm{BA}}$ let ip := match names ? m with | None => 0 | Some a => a end in _ ← ***Send*** ip $\mathrm{ch}_{\mathrm{AB}}$ ***Return*** ip)	serverLoop n_{iter} v_{ret} (c ← **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{B}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{A}}$) m ← **Decrypt** c let ip := match names ? m with | None => 0 | Some a => a end in ipC ← **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{A}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{B}}$ $\hat{u}_B$ ip _ ← **Send** $\hat{u}_B$ ipC **Return** ip)
$\hat{u}_B$	$[\mathrm{ch}_{\mathrm{AB}} \mapsto r\text{-}, \mathrm{ch}_{\mathrm{BA}} \mapsto \text{-}w$ $,\mathrm{ch}_{\mathrm{BC}} \mapsto \text{-}w, \mathrm{ch}_{\mathrm{CB}} \mapsto r\text{-}]$ $\{\mathsf{pk}^{\sigma}_{\mathrm{A}}, \mathsf{pk}^{\varepsilon}_{\mathrm{A}}, \mathsf{pk}^{\sigma}_{\mathrm{B}}, \mathsf{sk}^{\sigma}_{\mathrm{B}}$ $,\mathsf{pk}^{\varepsilon}_{\mathrm{B}}, \mathsf{sk}^{\varepsilon}_{\mathrm{B}}, \mathsf{pk}^{\sigma}_{\mathrm{C}}, \mathsf{pk}^{\varepsilon}_{\mathrm{C}}\}$	req ← ***Recv*** $\mathrm{ch}_{\mathrm{CB}}$ _ ← ***Send*** req $\mathrm{ch}_{\mathrm{BA}}$ ip1 ← ***Recv*** $\mathrm{ch}_{\mathrm{AB}}$ _ ← ***Send*** ip1 $\mathrm{ch}_{\mathrm{BC}}$ ***Return*** ip1	reqc ← **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{C}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{B}}$) req ← **Decrypt** reqc $c_1 \leftarrow$ **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{B}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{A}}$ $\hat{u}_1$ req _ ← **Send** $\hat{u}_1$ c_1 hostC ← **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{A}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{B}}$) host ← **Decrypt** hostC $c_2 \leftarrow$ **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{B}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{C}}$ $\hat{u}_3$ host _ ← **Send** $\hat{u}_3$ c_2 **Return** host
$\hat{u}_C$	$[\mathrm{ch}_{\mathrm{BC}} \mapsto r\text{-}, \mathrm{ch}_{\mathrm{CB}} \mapsto \text{-}w]$ $\{\mathsf{pk}^{\sigma}_{\mathrm{B}}, \mathsf{pk}^{\varepsilon}_{\mathrm{B}}, \mathsf{pk}^{\sigma}_{\mathrm{C}}$ $,\mathsf{sk}^{\sigma}_{\mathrm{C}}, \mathsf{pk}^{\varepsilon}_{\mathrm{C}}, \mathsf{sk}^{\varepsilon}_{\mathrm{C}}\}$	_ ← ***Send*** hostname $\mathrm{ch}_{\mathrm{CB}}$ ip1 ← ***Recv*** $\mathrm{ch}_{\mathrm{BC}}$ ***Return*** ip1	c ← **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{C}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{B}}$ $\hat{u}_2$ hostname _ ← **Send** $\hat{u}_2$ c hostC ← **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{B}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{C}}$) host ← **Decrypt** hostC **Return** host

Fig. 12: Our implementation of the Secure DNS variant includes a few interesting features. In this protocol $\hat{u}_A$ is the DNS server. As a server, we expect it to loop indefinitely, taking in requests. Neither of our languages support loops, so we implemented generic looping constructs within Gallina functions which allow us to construct servers that perform the same operation repeatedly. In this example, the `serverLoop` functions take in two numbers, the maximum number of loop iterations and a value to return upon loop exit. Gallina does not allow infinitely executing functions, but without much effort we are able to implement something that behaves like one by choosing arbitrarily large "loop iteration values." A last note about this example: the function call `names ? m` represents a lookup of the hostname in the DNS database, modeled here as a retrieval from an associative-map data structure for simplicity.

User	Initial Configuration	Specification	Implementation
$\hat{u}_A$	$[\mathrm{ch_{AD}} \mapsto \text{-}w]$ $\{\mathsf{pk}^{\sigma}_{\mathrm{A}}, \mathsf{sk}^{\sigma}_{\mathrm{A}}, \mathsf{pk}^{\varepsilon}_{\mathrm{D}}\}$	_ ← ***Send*** sal_1 $\mathrm{ch_{AD}}$ ***Return*** sal_1	c ← **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{A}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{D}}$ $\hat{u}_4$ sal_1 _ ← **Send** $\hat{u}_4$ c **Return** sal_1
$\hat{u}_B$	$[\mathrm{ch_{BD}} \mapsto \text{-}w]$ $\{\mathsf{pk}^{\sigma}_{\mathrm{B}}, \mathsf{sk}^{\sigma}_{\mathrm{B}}, \mathsf{pk}^{\varepsilon}_{\mathrm{D}}\}$	_ ← ***Send*** sal_2 $\mathrm{ch_{BD}}$ ***Return*** sal_2	c ← **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{B}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{D}}$ $\hat{u}_4$ sal_2 _ ← **Send** $\hat{u}_4$ c **Return** sal_2
$\hat{u}_C$	$[\mathrm{ch_{CD}} \mapsto \text{-}w]$ $\{\mathsf{pk}^{\sigma}_{\mathrm{C}}, \mathsf{sk}^{\sigma}_{\mathrm{C}}, \mathsf{pk}^{\varepsilon}_{\mathrm{D}}\}$	_ ← ***Send*** sal_3 $\mathrm{ch_{CD}}$ ***Return*** sal_3	c ← **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{C}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{D}}$ $\hat{u}_4$ sal_3 _ ← **Send** $\hat{u}_4$ c **Return** sal_3
$\hat{u}_D$	$[\mathrm{ch_{AD}} \mapsto r\text{-}, \mathrm{ch_{BD}} \mapsto r\text{-}$ $,\mathrm{ch_{CD}} \mapsto r\text{-}]$ $\{\mathsf{pk}^{\sigma}_{\mathrm{A}}, \mathsf{pk}^{\sigma}_{\mathrm{B}}, \mathsf{pk}^{\sigma}_{\mathrm{C}}, \mathsf{pk}^{\varepsilon}_{\mathrm{D}}, \mathsf{sk}^{\varepsilon}_{\mathrm{D}}\}$	m_1 ← ***Recv*** $\mathrm{ch_{AD}}$ m_2 ← ***Recv*** $\mathrm{ch_{BD}}$ m_3 ← ***Recv*** $\mathrm{ch_{CD}}$ ***Return*** $(m_1+m_2+m_3)$ / 3	salC1 ← **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{A}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{D}}$) salC2 ← **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{B}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{D}}$) salC3 ← **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{C}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{D}}$) sal_1 ← **Decrypt** salC1 sal_2 ← **Decrypt** salC2 sal_3 ← **Decrypt** salC3 **Return** $(sal_1+sal_2+sal_3)$ / 3

Fig. 13: Secret aggregation is another example of performing computations with Gallina code. The idea here is that each of the users $\hat{u}_A, \hat{u}_B, \hat{u}_C$ wants to perform some calculation over some secret data (say their salary). They trust a third party $\hat{u}_D$ to perform that computation and not leak the raw data. In principle, we could have implemented a much more complicated calculation.

User	Initial Configuration	Specification	Implementation
$\hat{u}_A$	$[\mathrm{ch_A} \mapsto rw, \mathrm{ch_{AC}} \mapsto \text{-}w$ $,\mathrm{ch_{CA}} \mapsto r\text{-}]$ $\{\mathsf{pk}^{\sigma}_{\mathrm{A}}, \mathsf{sk}^{\sigma}_{\mathrm{A}}, \mathsf{pk}^{\varepsilon}_{\mathrm{A}}$ $,\mathsf{sk}^{\varepsilon}_{\mathrm{A}}, \mathsf{pk}^{\sigma}_{\mathrm{C}}, \mathsf{pk}^{\varepsilon}_{\mathrm{C}}\}$	_ ← ***Send*** $\hat{u}_B$ $\mathrm{ch_{AC}}$ (_, $\mathrm{ch_S}$) ← ***Recv*** $\mathrm{ch_{CA}}$ n ← ***Gen*** _ ← ***Send*** n ($\mathrm{ch_A} \cap \mathrm{ch_S}$) ***Return*** n	c_1 ← **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{A}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{C}}$ $\hat{u}_3$ $\hat{u}_2$ _ ← **Send** $\hat{u}_C$ c_1 c_2 ← **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{C}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{A}}$) (_, $\kappa^{\varepsilon}_{\mathrm{S}}$) ← **Decrypt** c_2 n ← **Gen** c_3 ← **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{A}}$ $\kappa^{\varepsilon}_{\mathrm{S}}$ $\hat{u}_2$ n _ ← **Send** $\hat{u}_B$ c_3 **Return** n
$\hat{u}_B$	$[\mathrm{ch_B} \mapsto rw, \mathrm{ch_{BC}} \mapsto \text{-}w$ $,\mathrm{ch_{BA}} \mapsto r\text{-}]$ $\{\mathsf{pk}^{\sigma}_{\mathrm{B}}, \mathsf{sk}^{\sigma}_{\mathrm{B}}, \mathsf{pk}^{\varepsilon}_{\mathrm{B}}$ $,\mathsf{sk}^{\varepsilon}_{\mathrm{B}}, \mathsf{pk}^{\sigma}_{\mathrm{C}}, \mathsf{pk}^{\varepsilon}_{\mathrm{C}}\}$	_ ← ***Send*** $\hat{u}_A$ $\mathrm{ch_{BC}}$ (_, $\mathrm{ch_S}$) ← ***Recv*** $\mathrm{ch_{CB}}$ n ← ***Recv*** ($\mathrm{ch_A} \cap \mathrm{ch_S}$) ***Return*** n	c_1 ← **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{B}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{C}}$ $\hat{u}_3$ $\hat{u}_1$ _ ← **Send** $\hat{u}_3$ c_1 c_2 ← **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{C}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{A}}$) ($\mathsf{pk}^{\sigma}_{\mathrm{A}}$, $\kappa^{\varepsilon}_{\mathrm{S}}$) ← **Decrypt** c_2 c_3 ← **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{A}}$ $\kappa^{\varepsilon}_{\mathrm{S}}$) n ← **Decrypt** c_3 **Return** n
$\hat{u}_C$	$[\mathrm{ch_A} \mapsto r\text{-}, \mathrm{ch_B} \mapsto r\text{-}$ $,\mathrm{ch_{AC}} \mapsto r\text{-}, \mathrm{ch_{CA}} \mapsto \text{-}w$ $,\mathrm{ch_{BC}} \mapsto r\text{-}, \mathrm{ch_{BA}} \mapsto \text{-}w]$ $\{\mathsf{pk}^{\sigma}_{\mathrm{A}}, \mathsf{pk}^{\varepsilon}_{\mathrm{A}}, \mathsf{pk}^{\sigma}_{\mathrm{B}}, \mathsf{pk}^{\varepsilon}_{\mathrm{B}}$ $,\mathsf{pk}^{\sigma}_{\mathrm{C}}, \mathsf{sk}^{\sigma}_{\mathrm{C}}, \mathsf{pk}^{\varepsilon}_{\mathrm{D}}, \mathsf{sk}^{\varepsilon}_{\mathrm{D}}\}$	m_1 ← ***Recv*** $\mathrm{ch_{AC}}$ m_2 ← ***Recv*** $\mathrm{ch_{BC}}$ $\mathrm{ch_S}$ ← ***CreateChannel*** _ ← ***Send*** ($\mathrm{ch_A} \mapsto$ r- , $\mathrm{ch_S} \mapsto$ rw) $\mathrm{ch_{CB}}$ _ ← ***Send*** ($\mathrm{ch_B} \mapsto$ r- , $\mathrm{ch_S} \mapsto$ rw) $\mathrm{ch_{CA}}$ ***Return*** 1	c_1 ← **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{A}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{C}}$) c_2 ← **Recv** (SignedEncrypted $\mathsf{pk}^{\sigma}_{\mathrm{B}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{C}}$) m_1 ← **Decrypt** c_1 m_2 ← **Decrypt** c_2 $\kappa^{\varepsilon}_{\mathrm{S}}$ ← **GenerateKey** SymKey Encryption c_3 ← **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{C}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{A}}$ $\hat{u}_1$ ($\mathsf{pk}^{\sigma}_{\mathrm{B}}$, $\kappa^{\varepsilon}_{\mathrm{S}}$) c_4 ← **SignEncrypt** $\mathsf{sk}^{\sigma}_{\mathrm{C}}$ $\mathsf{pk}^{\varepsilon}_{\mathrm{B}}$ $\hat{u}_2$ ($\mathsf{pk}^{\sigma}_{\mathrm{A}}$, $\kappa^{\varepsilon}_{\mathrm{S}}$) _ ← **Send** $\hat{u}_2$ c_4 _ ← **Send** $\hat{u}_1$ c_3 **Return** 1

Fig. 14: In Network authentication, $\hat{u}_C$ acts as the server, awaiting requests from the other two parties to establish a secure communication channel. When both messages are received, $\hat{u}_C$ generates a symmetric key and shares it with them. The users then use the key to establish a secure communication channel.