Verifying Indistinguishability of Privacy-Preserving Protocols

KIRBY LINVILL, University of Colorado Boulder, USA
GOWTHAM KAKI, University of Colorado Boulder, USA
ERIC WUSTROW, University of Colorado Boulder, USA

Internet users rely on the protocols they use to protect their private information including their identity and the websites they visit. Formal verification of these protocols can detect subtle bugs that compromise these protections at design time, but is a challenging task as it involves probabilistic reasoning about random sampling, cryptographic primitives, and concurrent execution. Existing approaches either reason about symbolic models of the protocols that sacrifice precision for automation, or reason about more precise computational models that are harder to automate and require cryptographic expertise. In this paper we propose a novel approach to verifying privacy-preserving protocols that is more precise than symbolic models yet more accessible than computational models. Our approach permits *direct-style* proofs of privacy, as opposed to indirect game-based proofs in computational models, by formalizing privacy as *indistinguishability* of possible network traces induced by a protocol. We ease automation by leveraging insights from the distributed systems verification community to create sound synchronous models of concurrent protocols. Our verification framework is implemented in F* as a library we call Waldo. We describe two large case studies of using Waldo to verify indistinguishability; one on the *Encrypted Client Hello* (ECH) extension of the TLS protocol and another on a *Private Information Retrieval* (PIR) protocol. We uncover subtle flaws in the TLS ECH specification that were missed by other models.

CCS Concepts: • **Security and privacy** → **Formal security models**; **Logic and verification**; Information-theoretic techniques; *Security protocols*; • **Theory of computation** → **Automated reasoning**; **Logic and verification**; Programming logic; Equational logic and rewriting.

Additional Key Words and Phrases: Protocol Verification, Indistinguishability, Privacy, Concurrency, Synchronization

ACM Reference Format:
Kirby Linvill, Gowtham Kaki, and Eric Wustrow. 2023. Verifying Indistinguishability of Privacy-Preserving Protocols. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 273 (October 2023), 28 pages. https://doi.org/10.1145/3622849

1 INTRODUCTION

Privacy is an increasingly serious concern on the internet. Exposing users' sensitive data to a third party on the internet makes them susceptible to impersonation, censorship, and offline harm. To counter this threat, several communication and information retrieval protocols have been proposed that offer various guarantees pertaining to the privacy of the communicating parties. The key technology underlying most such protocols is cryptographic encryption. For instance, Transport Layer Security (TLS) is a secure communication protocol that encrypts an application's TCP traffic

Authors' addresses: Kirby Linvill, kirby.linvill@colorado.edu, University of Colorado Boulder, USA; Gowtham Kaki, gowtham.kaki@colorado.edu, University of Colorado Boulder, USA; Eric Wustrow, ewust@colorado.edu, University of Colorado Boulder, USA.

2475-1421/2023/10-ART273
https://doi.org/10.1145/3622849

to prevent a prying observer from learning anything about the information being exchanged. TLS was famously used to extend HTTP with encrypted communications, resulting in the HTTP over TLS (HTTPS) protocol, which has now emerged as the standard for secure communications on the world-wide web [U.S. CIO Council 2016; W3C Technical Architecture Group 2021].

Encryption, however, does not automatically confer the benefits of privacy. Communication protocols often involve complex control flows composed of multiple message exchanges between communicating parties. Even if individual messages are securely encrypted, a sequence of message exchanges induced by a sensitive control flow might still leak private information. Furthermore, encryption itself might be weakened due to violations of hyperproperty pre-conditions, such as reuse of keys and nonces. Consequently, composing secure cryptographic primitives does not necessarily lead to a secure protocol. Indeed, researchers have uncovered several vulnerabilities against previous versions of TLS that exploit the subtle impact of legacy protocol support [Aviram et al. 2016; Möller et al. 2014], complex control flows [Al Fardan and Paterson 2013; Beurdouche et al. 2017; Bhargavan et al. 2014], and the lengths of messages [Rizzo and Duong 2012] to leak sensitive information to an attacker.

Formal verification has the potential to uncover critical vulnerabilities in specifications and implementations of secure communication protocols. The task is challenging, however, as it involves reasoning about pseudo-random functions (PRFs) and cryptographic primitives that make up a secure communication protocol such as TLS. There are currently two major approaches to formal verification. The first approach involves reasoning about protocol specifications while the second involves reasoning about protocol implementations. Formal verification of specifications establishes correctness and security of a protocol in principle. Formal verification of implementations establishes the correctness and security of the implementation, such as the implementation of HTTPS within a particular web browser or web server. Though we focus on formal verification of protocol *specifications* in this paper, our library Waldo can be connected to network, encryption, and random sampling implementations to create a functional implementation.

Reasoning about cryptographic protocol specifications is typically done by either reasoning about *symbolic* or *computational* models. Symbolic models are often reasoned about in the framework of Dolev-Yao [Dolev and Yao 1983], where messages are represented as terms, PRFs and cryptographic primitives are assumed to guarantee perfect randomness and secrecy (resp.), and these assumptions are encoded as axioms in a first-order logic. Security properties are expressed as queries on the ability of an attacker to fully obtain a secret using an equational theory that models the attacker's capabilities. Unfortunately, the symbolic model lacks precision due to its use of deterministic first-order models to overapproximate the probabilistic guarantees of PRFs and cryptographic primitives. While first-order modeling lends itself to automatic reasoning, overapproximating the guarantees from the environment leads to imprecise conclusions (false proofs of inviolability) which may not be valid in practice. As an example, consider a simple communication protocol that encrypts a message together with its sender ID before sending out the encrypted ciphertext. The protocol might want to keep the sender information private to avoid selective censorship. In ProVerif [Blanchet et al. 2016], a verification tool based on the Dolev-Yao model, one can query if an attacker can observe the difference between the ciphertexts $\mathsf{enc}((m, s_1),\ k)$ and $\mathsf{enc}((m, s_2),\ k)$, where s_1 and s_2 are senders sending the same message m, enc is a perfect encryption scheme, and k is the key used for encryption. ProVerif answers the query negatively, meaning that it should be impossible for an attacker to observe the difference between the two because $\mathsf{enc}((m, s_1),\ k)$ and $\mathsf{enc}((m, s_2),\ k)$ are symbolically equivalent; both are distinct bitstrings produced from a perfect encryption scheme. In practice however, the two ciphertexts have different lengths depending on the length of the sender field, thus leaking the identity of the sender as illustrated in Fig. 1.

```
Encrypting message {msg: "This is a test message"; sender: "Alice"}
Encrypted message: OY31M...f4zbe, Length: 30

Encrypting message {msg: "This is a test message"; sender: "Charlie"}
Encrypted message: O4n9I...bqn3e, Length: 32
```

Fig. 1. Example encrypted messages with the same content from two different senders. The encrypted length varies based on the combined length of the message and sender thereby potentially leaking the identity of the sender.

The overapproximation problem of the symbolic model is rectified in the *computational model* of cryptography, which explicitly models the probability of the failure of an encryption scheme in the presence of a polynomially-bounded adversary. Security properties in this model are typically expressed using simulated sessions (i.e., *games*) with an adversary. The demonstration of the inability of the attacker to gain access to the secret in this simulated game with less than a bounded probability ϵ is a *game-based proof* of protocol's inviolability. The precise nature of reasoning in the computational model eliminates false proofs of safety. For instance, CryptoVerif [Blanchet 2008], a verification tool based on the computational model, correctly identifies the problem described in Fig. 1. The gain in precision, however, is offset by the loss in automation as game-based proofs in the computational model often require non-trivial human insight to reduce the game to a known cryptographic assumption. Furthermore, various games can be setup depending on the abilities of the attacker and the required security guarantees. For instance, a game modeling a Chosen Plaintext Attack (CPA) adversary allows the attacker to choose a series of plaintexts to be encrypted by the challenger and observe the corresponding ciphertexts. A variation of a CPA adversary, called Adaptive CPA, allows the attacker to adapt its choices of plaintext based on the ciphertexts observed so far. It is incumbent on the protocol designer to set up an appropriate adversary and game whose proof of inviolability guarantees the protocol's real-world safety. This makes the computational model inaccessible to anyone who is not an expert in cryptography.

Reasoning about the low-level implementations of security protocols is often done in a general purpose theorem prover such as Coq [Team 2023] or F* [Swamy et al. 2016]. Having access to the full generality of a theorem prover allows one to pick elements from the symbolic and computational models and compose them in a way that suits the assumptions and requirements of the verification task at hand. The expressive power of the higher-order logic lets one state security properties directly as probabilistic assertions and construct *direct proofs* of safety instead of indirect game-based proofs. Moreover, successful verification yields a verified executable that can be immediately deployed. On the flip side, however, building formal proofs of correctness for low-level implementations in a higher-order proof system such as Coq's CIC [Bertot and Castran 2010] is extremely hard and labor-intensive. For instance, the end-to-end verification of only part of TLS 1.3, the record layer, in F* required roughly 12,000 lines of annotations and required significant effort from experts in formal methods, security, and cryptography [Delignat-Lavaud et al. 2017]. Such expertise and effort is clearly out of reach for an average application developer who might want to verify a high-level implementation of a protocol but is not necessarily interested in end-to-end verification guarantees.

In this paper we propose an alternative approach to verifying privacy-preserving communication protocols in the presence of a passive attacker that brings together the virtues of the aforementioned approaches into a unified reasoning framework called Waldo. Our approach admits direct specification of privacy properties as *indistinguishability* assertions, which assert the observational

equivalence of protocol traces at the level of their *distributions* rather than individual traces. Cryptographic primitives like encryption are modeled using similar perfect secrecy assumptions as in the symbolic model to ease automation, but messages are modeled as bytestrings to preserve precision just like in computational model tools. Like previous approaches, Waldo operates on the high-level executable specifications of protocols expressed in a bespoke DSL, which can be linked with trusted network and I/O libraries to yield an executable implementation. Waldo's semi-automated reasoning helps developers build direct proofs of privacy by relating distributions of traces generated in distinct runs of the protocol. Waldo relies on Probabilistic Relational Hoare Logic (pRHL) [Barthe et al. 2009] to drive this reasoning. pRHL is a relational program logic for reasoning about relational properties of sequential probabilistic programs. In contrast, protocols such as TLS are concurrent programs with multiple communicating processes. A notable contribution of our approach is to bridge this gap by borrowing techniques from distributed systems verification. In particular, we take inspiration from Lipton's theory of movers [Lipton 1975; v. Gleissenthall et al. 2019] to soundly reduce a composition of concurrent communicating processes to an equivalent sequential program. Lipton's reduction guarantees equivalence of original and reduced programs up to the observable *states*. Since indistinguishability requires equivalence up to observable *traces*, we generalize Lipton's reduction by adopting a more conservative approach to identifying *movers*. We present our variant of Lipton's reduction in Sec. 3. Another distinguishing aspect of our approach is its ability to "factor out" non-determinism into a (small) prefix of the program, allowing equational reasoning to be used to reason about the (large) suffix. This transformation allows Waldo, which is implemented as a library in F*, to bring to bear F*'s formidable proof automation to prove privacy properties of protocols.

Contributions. Our contributions are summarized thus:

- We propose a new approach to verifying privacy-preserving communication protocols that allows for direct specifications using indistinguishability assertions. Our model captures finer-grained information than the symbolic model, while often adopting similar perfect secrecy assumptions to make protocols easier to specify and verify than the computational model.
- We introduce a DSL for specifying privacy-preserving protocols, proof rules for establishing indistinguishability, and a sound reduction of concurrent probabilistic processes to an equivalent sequential probabilistic program (Sec. 3 and Sec. 4).
- We implement the DSL and semi-automated proof system as a library in F* known as Waldo (Sec. 5).
- We use Waldo to verify privacy properties for two important protocols as case studies:

(1) Transport Layer Security (TLS) Encrypted Client Hello (ECH) [Rescorla et al. 2022], an extension to a key component of today's internet infrastructure that can fail to meet its security goals due to insufficient padding and extension orderings that were not uncovered by previous models (Sec. 6).
(2) A Private-Information Retrieval (PIR) scheme from [Chor et al. 1998] in which a user retrieves a record from a set of databases without revealing which record they requested (Sec. 7). PIR is closely related to other important privacy-preserving protocols including secret-sharing schemes [Beimel et al. 2012; Shamir 1979] and multiparty computation [Beimel et al. 2012].

2 KEY IDEAS

We illustrate the challenges with verifying privacy properties and our key ideas to overcome these challenges using the simple protocol shown in Fig. 2a. The protocol is expressed using the π_w

language introduced in Sec. 3 with minor alterations for clarity. The protocol uses One-Time Pad (OTP) encryption, which is a symmetric encryption scheme[1] that works by exclusive or-ing (xor-ing, $\oplus$) a message m with a key k that is at least as long as the message. The original message can be recovered by xor-ing the ciphertext with the same key since $(m \oplus k) \oplus k = m \oplus (k \oplus k) = m$. In isolation, OTP is a *perfect encryption scheme*; it guarantees perfect secrecy of the input message as long as it is encrypted with a key of the same length. For a sequence of messages, i.e., a *trace*, however, OTP might subtly leak information as demonstrated by the protocol in Fig. 2a.

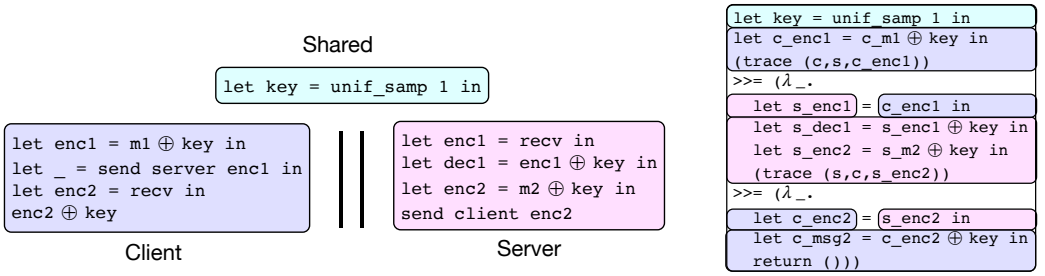
The protocol first generates a shared key (`key`) for use by both the client and server. The key is generated by sampling one byte from the uniform distribution (`unif_samp 1`). This protocol assumes both the messages and the key are a single byte for simplicity, but our approach applies to bytestrings in general. The client then encrypts its message (`m1`) by xor-ing the message with the shared key. It then sends the server the resulting encrypted message. The server receives and decrypts the message, encrypts its response (`m2`) using the same shared key, and sends the encrypted response to the client. Finally, the client decrypts the message it receives from the server. The protocol is parameterized on the messages `m1` and `m2` which should remain secret against an external observer.

A protocol guarantees *perfect secrecy* if an attacker can learn no information about secret inputs from the trace of network activity between the protocol participants. The secret inputs in the protocol shown in Fig. 2a are the messages `m1` and `m2` while the network trace always consists of the encrypted messages `enc1` and `enc2`. Perfect secrecy trivially holds if the network traces are identical for all secret inputs. This is in general impossible as the content of messages often depends on the inputs. We therefore consider statistical identity instead of pair-wise identity. Intuitively, an attacker can learn nothing about the secret inputs by observing the traces if the likelihood of a trace being generated by the protocol is the same regardless of its secret inputs. In other words, the protocol guarantees perfect secrecy if the *probability distribution of traces* generated by the protocol is independent of its secret inputs. We call this property *indistinguishability* as it requires the traces resulting from distinct secret inputs be statistically indistinguishable. Note that indistinguishability does not constrain the concrete probability distributions of traces; it only requires that the distributions be equal regardless of secret inputs. For example, a protocol that simply sends out a random value sampled from a fixed distribution guarantees indistinguishability *regardless of the distribution* as the attacker learns nothing about the secret inputs by observing the network traces.

Despite OTP being a perfect encryption scheme, the protocol in Fig. 2a fails to preserve perfect secrecy, which can be shown as a violation of indistinguishability. The violation occurs because the generated network trace consists of the values $\texttt{m1} \oplus \texttt{key}$ and $\texttt{m2} \oplus \texttt{key}$. An observer could simply xor these two values together to get $\texttt{m1} \oplus \texttt{key} \oplus \texttt{m2} \oplus \texttt{key}$ which, due to the commutative and inverse properties of xor, reduces to $\texttt{m1} \oplus \texttt{m2}$. This result leaks information about the secret input such as which bits of the response `m2` are the same as the initial message `m1`. Consequently, the traces generated by the protocol for different sets of messages are distinguishable. The root cause of this violation is the reuse of keys, which breaks the assumptions of OTP encryption. The protocol can be fixed by generating two shared keys instead of one, and using them to exclusively to encrypt `m1` and `m2` respectively as shown in the functional model in Fig. 3. The fixed protocol ensures that all possible network traces are equally likely regardless of the secret inputs, thus restoring indistinguishability.

While the above informal argument appeals to the intuition, formally proving indistinguishability, even for this simple protocol, is challenging. Firstly, secure communication protocols involve

[1] A symmetric encryption scheme is one where both the communicating parties use the same encryption key.



(a) Example protocol with concurrent client and server processes. (b) Sequential model of the protocol.

Fig. 2. Example protocol (2a) and sequential model (2b). The protocol uses One-Time Pad encryption but subtly leaks information by re-using the shared key. The protocol is parameterized on the secret messages m1 and m2. The sequential model uses a monad that keeps track of the network trace. Random sampling is not yet determinized in this model.

multiple sources of non-determinism, including concurrency and random sampling, which are difficult to reason about. Secondly, indistinguishability is a stochastic property requiring probabilistic reasoning that is hard to automate. Existing approaches at verifying protocol implementations typically use imprecise models that miss trivial information leaks or require the developer to deal with the full generality of these challenges in the context of a theorem prover, which requires considerable expertise in cryptography and a substantial proof effort [Delignat-Lavaud et al. 2017]. See Sec. 8 for a more detailed comparison against current approaches.

In this paper, we adopt an alternative approach to verifying indistinguishability that addresses the aforementioned complexity by systematically reducing concurrent protocol implementations first to sequential models with random sampling and then to deterministic monadic functional programs. The reduction makes the protocol amenable to equational reasoning and type-based techniques which are easier to automate using existing tools. We illustrate this idea on the running example. Our first observation is that, although the protocol in Fig. 2a is a concurrent program with asynchronous communication, we can soundly assume the communication to be synchronous when reasoning about the program. This is because the client's send matches with exactly one recv in the server and vice-versa. Moreover, because recvs are blocking, there are no *racing* communication actions; the latter communication is blocked until the former concludes. Thus, we can soundly model this protocol using the synchronized model shown in Fig. 2b. The sequential model uses a monad that tracks the network trace. ≫= is the monadic bind operator while trace is a monadic function that records a message in the network trace. When reducing from the protocol

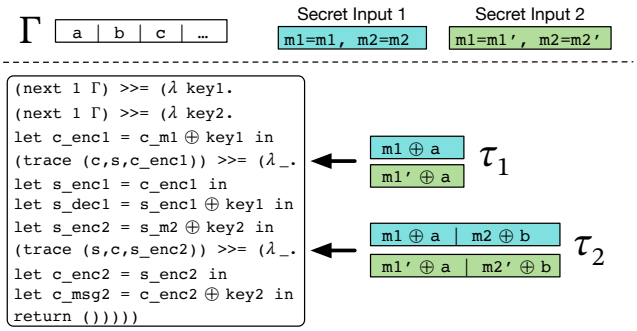


Fig. 3. Symbolic traces for two different runs of the fixed OTP-based protocol with different secret inputs. Proving indistinguishability requires a bijection on the tape Γ that results in equivalence of the final traces. The protocol model uses a monad that tracks both the network trace and random tape to fully determinize execution.

to the sequential model, sends are replaced with `trace` calls and recvs are replaced with the value of the corresponding message. Variables within each process are renamed to avoid clashes. For example, the `enc1` variable in the client process is renamed to `c_enc1` to avoid clashing with the variable in the server process with the same name. This sequential model can be checked for indistinguishability by examining the network traces produced by running the model with an empty initial trace.

Though the sequential model eliminates scheduling non-determinism, verifying indistinguishability for this model still requires reasoning about non-determinism from random sampling. To manage this complexity we reify random sampling as reading from a tape as in the model in Fig. 3. Specifically, `unif_samp` n calls are reified to `next` calls that simply return the next n unread elements from a tape Γ. For example, given the tape Γ and sequential model shown in Fig. 3, the first element of the tape (a) would be bound as `key1`, while the second element of the tape (b) would be bound as `key2`. The elements on the tape are expected to be independently drawn from the same distribution that random sampling would draw from. In this way, randomness is factored out of the program and into the tapes.

With this deterministic formulation of randomness, we can then apply rules derived from Probabilistic Relational Hoare Logic (pRHL) [Barthe et al. 2009] to reason about probabilistic equivalence of our sequential model. Specifically, we apply rules detailed in Sec. 4 to show that the probability of producing a given trace is equivalent for all choices of secret messages `m1` and `m2`. Showing probabilistic equivalence using pRHL requires a suitable bijection on the random tape. The bijection is used to show that for any particular tape that could be sampled, there is an equally likely tape that could be sampled for which running the deterministic program with the first set of secret inputs (`m1` and `m2`) on the original tape produces the same trace as running the deterministic program with the second set of secret inputs (`m1′` and `m2′`) on the bijected tape. When this holds for all possible tapes and for all possible sets of secret inputs, then the bijection shows probabilistic equivalence. When considering a passive attacker, probabilistic equivalence means that any set of secret inputs is equally likely to have produced a given trace. Therefore the attacker cannot infer any information about the secret inputs and the protocol provides indistinguishability.

To determine a suitable bijection for this example we first look at two different runs of the fixed program, which uses a different key for each encryption, with the initial tape Γ and pairs of secret messages (`m1`, `m2`) and (`m1′`, `m2′`) respectively as shown in Fig. 3. The traces (τ) and inputs corresponding to each run are color-coded to make them easier to distinguish. A sufficient bijection to prove indistinguishability must ensure that the final traces, in this case the traces at τ_2, must be equivalent. The elements in the traces differ in terms of which message is encrypted. As a result, a suitable bijection must ensure that $\texttt{m1} \oplus \texttt{a} = \texttt{m1}' \oplus A$ and that $\texttt{m2} \oplus \texttt{b} = \texttt{m2}' \oplus B$ where `a` and `b` are the first and second elements from the original tape while A and B are the first and second elements from the bijected tape. This holds if $A = \texttt{m1}' \oplus \texttt{m1} \oplus \texttt{a}$ and if $B = \texttt{m2}' \oplus \texttt{m2} \oplus \texttt{b}$. A suitable bijection therefore could be a function parameterized on the secret inputs `m1`, `m2`, `m1′`, and `m2′` that only permutes the first two elements of the tape by replacing them with $\texttt{m1}' \oplus \texttt{m1} \oplus \texttt{a}$ and $\texttt{m2}' \oplus \texttt{m2} \oplus \texttt{b}$ respectively where `a` and `b` are the first and second elements of the tape. Note that in the case of the broken protocol which uses the same key (`a` in this case) for both encryptions, proving probabilistic equivalence would require a bijection that ensures that $A = \texttt{m1}' \oplus \texttt{m1} \oplus \texttt{a}$ and $A = \texttt{m2}' \oplus \texttt{m2} \oplus \texttt{a}$. There is no such function that can provide this for all `m1`, `m2`, `m1′`, and `m2′` so it is (correctly) impossible to use our rules to verify probabilistic equivalence for the broken protocol.

As it turns out, the bijection required to verify the fixed protocol is common for protocols that use OTP encryption. Our library Waldo provides a generic form of this bijection (`bij_otp` in Fig. 9), along with bijections for other common models of encryption, that can be used out-of-the-box. Waldo further provides support for implementing the model using a monad that tracks the network

trace and random tape, as well as support for specifying and proving indistinguishability lemmas. Specifying and proving indistinguishability for protocols that can use Waldo's bijections is as simple as the example in Fig. 9 where `indistinguishable_ensures` generates a post-condition for indistinguishability for the protocol while `indistinguishable_lemma` applies the relevant proof rules given the suitable bijection. See Sec. 5 for more details about Waldo's implementation and interface. Waldo is written in F* [Swamy et al. 2016], a dependently-typed language with an OCaml-like syntax that can be used to specify and prove properties about programs. F* sends queries to the Z3 SMT solver to discharge proofs by default. Since the lemma shown in Fig. 9 references a suitable bijection, F* and Z3 successfully verify the lemma without further facts. In this way, verifying indistinguishability is mostly automatic given a suitable bijection, much as sequential program verification can be mostly automatic given suitable loop invariants.

The simple example in this section shows that we handle non-determinism from concurrency through synchronizing the protocol to extract a sound sequential model. It also shows that we handle non-determinism from random sampling by reifying sampling to reading from a tape and by applying rules derived from pRHL. However, not all programs can be soundly synchronized and most are not as trivial to synchronize. Sec. 4 defines what *is* a sound synchronization and introduces rules for computing such a synchronization in the presence of potentially racing sends and branching control flow. The example presented in this section uses OTP encryption which is a simple xor operation that guarantees perfect secrecy as long as the key is never reused. In contrast, most practical cryptographic primitives offer *computational secrecy*, where a polynomial adversary may be able to distinguish between secret inputs but with a negligibly low probability. Unfortunately, precisely modeling computational secrecy assumptions dramatically increases the complexity of reasoning and effort required to discharge proof obligations. Tools like Proverif overcome this problem by *overapproximating* computational secrecy with perfect secrecy [Blanchet et al. 2016]. In other words, practical cryptographic primitives are assumed to provide perfect secrecy. While these assumptions are stronger than more realistic computational assumptions, this approach is nonetheless useful as failing to verify properties under ideal assumptions often exposes flaws in the protocol that are also present under weaker, more realistic assumptions. Furthermore, less effort is typically required to verify properties under these kinds of ideal assumptions. We adopt this approach, but generalize it by letting one control the *extent* of overapproximation between the extremes of symbolic (perfect) secrecy and computational secrecy. For instance, a typical block encryption scheme can be modeled starting with a pair of functions (enc, dec) that satisfy the equation $dec(enc(m, k), k)) = m$, where m is the message and k is the key. To this symbolic model, we add a probabilistic assertion that states that $enc(m, k)$ returns a byte string of length $|m|$ from a probability distribution that is *independent* of m. Note that this is an overapproximation considering that practical encryption schemes do not guarantee the independence of ciphertext content from the message content. However, since the approximation is more precise than simply assuming perfect secrecy, specifically by capturing the length $|m|$ of an encrypted message, it allows for identifying additional protocol-level secrecy violations such as the one shown in Fig. 1. The approximation can be further refined by specifying the length, content, or the probability distribution of the ciphertexts. We thus capture the stochastic nature of cryptographic primitives without fully committing ourselves to the computational model. Such practical aspects of our approach, along with the implementation details of Waldo, are discussed in Sec. 5. We subsequently describe our experience of using Waldo to verify two more complicated protocols: TLS ECH in Sec. 6 and a Private Information Retrieval scheme in Sec. 7.

3 FORMALISM

In this section we formalize our ideas in the context of a probabilistic process calculus called π_w. We then formalize the notion of indistinguishability for π_w programs.

3.1 π_w: Syntax and Operational Semantics

The syntax for π_w is given in Fig. 4. A π_w program is a parallel composition of processes that communicate via message passing. A process is a command (c) or a sequence of commands composed using `let`. Commands include the `send` and `recv` primitives for message passing and a `samp` primitive to sample from a distribution. Commands can be chosen between using the `if then else` command. The identifier i of a process c_i uniquely identifies it among its peers. Name binding is done via `let`. `let` commands are defined to only allow inner commands that cannot be nested (m). Expressions include a unit value ($()$), bytestrings (s), booleans (l), tuples ((v, v)) and their projections (`fst`, `snd`), and user-defined operations (f). The first four constitute the syntactic class of values. Bytestrings are chosen as the core value since messages in an actual program are typically sent between processes as bytestrings. The language is parameterized on the set $f : v \to v$, which could include, for example, a bit-wise xor ($\oplus$) operation that operates on two bytestrings and returns a bytestring. The parallel composition is assumed to be commutative and associative, so $c_1 \parallel c_2 \parallel c_3$ is the same program as $c_2 \parallel c_3 \parallel c_1$.

$b \in$ `Bytes` $\quad x \in$ `Variables` $\quad i, j \in$ `Process Ids` $\quad f \in$ `Bytestring Ops` $\quad l \in \mathbb{B}$

s	$\in$	`Bytestrings`	$::=$	$b \mid b :: s$
v	$\in$	`Values`	$::=$	$s \mid l \mid () \mid (v, v)$
e	$\in$	`Expressions`	$::=$	$v \mid x \mid f\ e \mid$ `fst` $e \mid$ `snd` e
m	$\in$	`SingleCommands`	$::=$	$e \mid$ `samp` $\mid$ `send` $i\ e \mid$ `recv`
c	$\in$	`Commands`	$::=$	$m \mid$ `if` e `then` c_1 `else` $c_2 \mid$ `let` $x = m$ `in` c
P	$\in$	`Processes`	$::=$	$c_i \mid c_i \parallel P$
π	$\in$	`Programs`	$::=$	`let` $x = c$ `in` $\pi \mid P$

Fig. 4. π_w syntax.

The operational semantics of π_w is defined in Fig. 5 using two small-step reduction relations, one for the commands ($\longrightarrow$) and other for the programs ($\longrightarrow\!\!\!\!\!\rightarrow$). We use the notation c_i and $[c]_i$ interchangeably to represent the command c executing in process i. The relations relate execution states σ and commands c_i or programs π. An execution state σ is a tuple of a trace τ, tapes Γ and δ of random bytes, and a map M that maps process identifiers to their message queues. The trace τ records communication events of the form (v, i, j) capturing a message v being sent from process i to j. The trace models what an external observer can see from communicating processes since indistinguishability is defined with respect to such an observer. Concurrent small-step executions often make non-deterministic choices about what processes to reduce next. The scheduling non-determinism, in addition to the non-determinism induced by probabilistic constructs such as `samp`, makes it hard to reason about the relative likelihood of manifesting traces, which is necessary to show indistinguishability of executions. Our operational semantics circumvents this difficulty by reifying non-deterministic choices of sampling and scheduling as a deterministic reads from input tapes Γ and δ of random bytes, respectively. The tapes Γ and δ are assumed to have been sampled from the distributions D_Γ and D_δ that are obtained by lifting an unspecified distribution on individual elements to the tapes (lists) of such elements. To ensure a non-zero probability of sampling a particular tape, tapes Γ and δ are assumed to be of finite but unspecified length $\mathcal{L}$. Note that by appropriately choosing D_δ, one can obtain an adversarial scheduler that prefers one process

over the other. Modeling scheduling non-determinism as probabilistic choice lets us compute the probability of manifesting a trace as a function of the distribution on the tape δ. This in turn helps us formalize indistinguishability (Def. 3.2) precisely, and also show that our approach to synchronization does not change the probability of manifesting a particular trace (Theorem 4.3).

$\boxed{\sigma\ \pi \longrightarrow\!\!\!\!\rightarrow \sigma'\ \pi'}$ $\boxed{\sigma\ c_i \longrightarrow \sigma'\ c_i'}$

$$\tau \in \texttt{Traces} ::= [] \mid (v, i, j) :: \tau \qquad \Gamma, \delta \in \texttt{Tapes} ::= [] \mid b :: \Gamma$$

$$m \in \texttt{Msg. Queue} ::= [] \mid v :: m \qquad M \in \texttt{Msg. Queues} ::= i \mapsto m$$

$$\frac{\delta = i :: \delta' \quad (\tau, \Gamma, \delta', M)\ c_i \longrightarrow \sigma'\ c_i'}{(\tau, \Gamma, \delta, M)\ c_i \parallel P \longrightarrow\!\!\!\!\rightarrow \sigma'\ c_i' \parallel P}\ \text{E-Cong} \qquad \frac{\delta = i :: \delta' \quad \nexists \sigma'.\ (\tau, \Gamma, \delta', M)\ c_i \longrightarrow \sigma'\ c_i'}{(\tau, \Gamma, \delta, M)\ c_i \parallel P \longrightarrow\!\!\!\!\rightarrow (\tau, \Gamma, \delta', M)\ c_i \parallel P}\ \text{E-Refl}$$

$$\frac{M[j] = q \qquad M' = M[j \mapsto q +\!\!+ [v]]}{(\tau, \Gamma, \delta, M)\ [\texttt{send}\ j\ v]_i \longrightarrow ((v, i, j) :: \tau, \Gamma, \delta, M')\ [()]_i}\ \text{E-Send}$$

$$\frac{M[i] = v :: q \qquad M' = [i \mapsto q]}{(\tau, \Gamma, \delta, M)\ [\texttt{recv}]_i \longrightarrow (\tau, \Gamma, \delta, M')\ [v]_i}\ \text{E-Recv} \qquad \frac{\Gamma = b :: \Gamma'}{(\tau, \Gamma, \delta, M)\ [\texttt{samp}]_i \longrightarrow (\tau, \Gamma', \delta, M)\ [b]_i}\ \text{E-Samp}$$

$$\frac{}{\sigma\ [\texttt{let}\ x = v\ \texttt{in}\ c]_i \longrightarrow \sigma\ [c[v/x]]_i}\ \text{E-Let} \qquad \frac{}{\sigma\ [\texttt{if true then}\ c_1\ \texttt{else}\ c_2]_i \longrightarrow \sigma\ [c_1]_i}\ \text{E-IfTrue}$$

Fig. 5. π_w operational semantics. Congruence and tuple projection rules are omitted for brevity.

The E-Cong rule of Fig. 5 shows how the tape δ is used to pick the process to execute next. Tape δ is assumed to be a stream of process ids instead of bytestrings to simplify presentation. In reality the semantics needs to be parameterized on a function `choose` that maps bytestrings to process Ids. The randomly chosen process c_i may not be able to take a step if it is waiting on a message or if is already a value. In such case E-Refl allows skipping c_i and move on to the next choice from δ. Sampling a byte via `samp` corresponds to reading the next byte from the tape Γ (E-Samp). Sending (resp. receiving) messages involves appending to (resp. reading from) the message queue as shown by E-Send (resp. E-Recv). The E-Send rule appends the `send` event to the trace signifying its external observability. Note that the trace grows monotonically capturing the monotonicity of externally visible observations. The E-Recv rule for a given process only applies when there is a message in the queue for that process thereby modeling blocking `recv`s. Together, `send` and `recv` primitives model reliable communication. This reflects the choice made by many privacy-preserving protocols in practice, including the protocols in our case studies, which rely on reliable communication (TCP) and blocking `recv`s. The reduction rules for `let` and `if` are standard. Non-zero and zero bytestrings are interpreted as `true` and `false` values respectively. For brevity, we omit all congruence and tuple projection rules.

As a result of factoring out non-determinism into tapes, the execution is deterministic. We define an execution of a π_w program as its multistep reduction ($\longrightarrow\!\!\!\!\rightarrow^*$) to a value. To focus on execution traces, we overload the notation to write $([], \Gamma, \delta, \emptyset)\ \pi \longrightarrow\!\!\!\!\rightarrow^*\ \tau$ iff there exist Γ', δ', M, and v such that $([], \Gamma, \delta, \emptyset)\ \pi \longrightarrow\!\!\!\!\rightarrow^*\ (\tau, \Gamma', \delta', M)\ v$. The determinism of execution is stated thus:

Theorem 3.1 (Deterministic Execution of π_w)**.** *Forall* Γ, δ, π, *if* $([], \Gamma, \delta, \emptyset)\ \pi \longrightarrow\!\!\!\!\rightarrow^*\ \tau'$ *and* $([], \Gamma, \delta, \emptyset)\ \pi \longrightarrow\!\!\!\!\rightarrow^*\ \tau''$, *then* $\tau' = \tau''$.

Exploiting the determinism of execution, we define a function $\mathcal{F}$ that maps the triple (Γ, δ, π) to its unique execution trace τ, if one exists:

$$\begin{array}{lcll} \mathcal{F}(\Gamma, \delta, \pi) & = & \tau & \texttt{if } ([], \Gamma, \delta, \emptyset)\ \pi \longrightarrow\!\!\!\!\!\rightarrow^* \tau \\ \mathcal{F}(\Gamma, \delta, \pi) & = & \bot & \texttt{otherwise} \end{array}$$

3.2 Indistinguishability

We will now formalize the notion of indistinguishability for π_w programs. Let $\texttt{Tape}_{\mathcal{L}}$ denote the type of tapes that are $\mathcal{L}$ bytes long. A distribution over tapes of length $\mathcal{L}$ is a function $D : \texttt{Tape}_{\mathcal{L}} \rightarrow [0, 1]$ such that $\sum_{t \in \texttt{Tape}_{\mathcal{L}}} D(t) = 1$. Intuitively, a distribution over tapes maps each tape to its corresponding probability. The probability of sampling a particular tape t from a distribution D is therefore:

$$\Pr[X \xleftarrow{\$} D = t] = D(t)$$

When D is clear from the context, we simplify $\Pr[X \xleftarrow{\$} D = t]$ to $\Pr[X = t]$. We assume, without the loss of generality, that D_Γ and D_δ are two distinct distributions over independent and identically distributed tapes of a finite length $\mathcal{L}$.

We consider a passive attacker model, where an attacker can observe a protocol's external interactions, i.e., its network messages, but cannot manipulate them. The attacker also has no access to the protocol's internal state. In the context of the current formal development, a passive attacker can observe the trace τ of a program π's execution, but none of the other artifacts. To help us define the likelihood of observing a trace τ, we introduce an auxiliary *point* function $\mathbf{1}_\tau$ that returns 1 iff its argument is τ and 0 otherwise:

$$\mathbf{1}_\tau\ \tau' \stackrel{\text{def}}{=} \texttt{if } \tau' = \tau \texttt{ then } 1 \texttt{ else } 0$$

Let $\pi \longrightarrow\!\!\!\!\!\rightarrow^* \tau$ iff there exist tapes Γ and δ sampled from D_Γ and D_δ such that $([], \Gamma, \delta, \emptyset)\ \pi \longrightarrow\!\!\!\!\!\rightarrow^* \tau$. The probability of a program π manifesting an execution trace τ is given by:

$$\Pr[\pi \longrightarrow\!\!\!\!\!\rightarrow^* \tau] \quad = \sum_{\Gamma \in \texttt{supp}(D_\Gamma)} \sum_{\delta \in \texttt{supp}(D_\delta)} (\mathbf{1}_\tau\ (\mathcal{F}(\Gamma, \delta, \pi)) * \Pr[X \xleftarrow{\$} D_\Gamma = \Gamma] * \Pr[X \xleftarrow{\$} D_\delta = \delta]$$

where $\texttt{supp}(D)$, the *support* of distribution D, is the set of elements mapped to non-zero probability by D.

Equipped with the above definitions, we now formalize the notion of indistinguishability between a pair of π_w programs π_1 and π_2.[2]

Definition 3.2 (Indistinguishability)**.** A pair of π_w programs π_1 and π_2 are indistinguishable iff both programs are equally likely to manifest any given trace τ when run against the tapes sampled from distributions D_Γ and D_δ. Formally:

$$\pi_1 \equiv_{\langle D_\Gamma, D_\delta \rangle} \pi_2 \quad \stackrel{\text{def}}{=} \quad \forall \tau.\ \Pr[\pi_1 \longrightarrow\!\!\!\!\!\rightarrow^* \tau] = \Pr[\pi_2 \longrightarrow\!\!\!\!\!\rightarrow^* \tau]$$

Assuming fixed distributions D_Γ and D_δ, we omit the subscript and write $\pi_1 \equiv \pi_2$ to denote indistinguishability.

4 PROVING INDISTINGUISHABILITY

We now present our proof technique to prove a pair of π_w programs to be indistinguishable. The cornerstone of the technique is Probabilistic Relational Hoare Logic (pRHL) [Barthe et al. 2009], a program logic to reason about the relationship between a pair of sequential probabilistic programs. As such, the target of pRHL is a sequential imperative language extended with a sampling

[2]Formulating indistinguishability between programs π_1 and π_2 is equivalent to formulating indistinguishability between inputs a and b for a fixed program π (such as the example protocol in Sec. 2). This is because a program could be inlined with its inputs ($\pi(a)$ and $\pi(b)$) to create two distinct programs.

assignment $x \xleftarrow{\$} D$ to sample a value from a distribution D. In contrast, π_w programs are concurrent by definition. One way to bridge the gap would be to generalize the relational reasoning of pRHL to a concurrent setting by pairing it with a concurrent program logic such as Rely-Guarantee. While possible, this approach is unlikely to result in a practical proof technique as it requires simultaneous reasoning about observational equivalence, concurrency, and probabilities, each of which is complicated in its own right. Moreover, we observe that the full generality of a relational concurrent logic is often not needed in practice to establish the indistinguishability of privacy-preserving protocols. While concurrent asynchronous communication is integral to the functionality of such protocols, we observe that the communication can often be *synchronized* to compute an equivalent sequential program without any loss of generality. The key technique that lets us soundly do this transformation is based on Lipton's reduction [Lipton 1975].

4.1 Synchronizing the Asynchronous

Lipton's reduction provides a principled basis for moving a send operation up to its matching receive, thereby fusing a pair of asynchronous communication operations into a single synchronous assignment. In a concurrent execution, Lipton's reduction identifies operations as either *right movers* or *left movers*. The idea is that, in a left-to-right linear trace of a concurrent program, a right (resp. left) mover can be moved right (resp. left) w.r.t the concurrent operations without having an observable effect on the execution state. For instance, in shared memory concurrent programming, `lock` operations are generally considered right movers while `unlocks` are left movers. Lipton's theory of movers makes it possible to soundly move the operations of a process closer together and coalesce them into a single atomic block, thereby reducing the overhead of reasoning about concurrency.

Similar intuitions apply for `send`s and `recv`s in message-passing concurrent programs. For a given trace, a `send` can always be soundly synchronized by moving it up to its matching `recv` without affecting the set of visited states. If this can be done in every possible trace, i.e., if a `send` and a `recv` correspond exclusively to each other in every trace, and if the execution never *blocks* or *loops* between the two operations, then the `send`-`recv` pair can be statically synchronized and replaced with an assignment statement. In [v. Gleissenthall et al. 2019], authors rely on this observation to soundly synchronize and verify the safety properties of distributed programs. We apply similar intuitions in the context of cryptographic communication protocols but we adopt a more conservative approach to synchronization as we are interested in observational equivalence of *traces* as opposed to the safety of the *states* visited.

4.1.1 Soundness of Synchronization. Before we describe *how* to perform synchronization, we first describe *what* constitutes a correct synchronization. Let $\boldsymbol{\lambda}$ be the sequential program resulting from synchronizing a protocol π. Recall that the inputs to π are the tapes Γ and δ which consist of independent, randomly sampled elements. Since $\boldsymbol{\lambda}$ retains the `samp` operations from π, it requires the tape Γ. The tape δ is however no longer required as synchronization eliminates scheduling non-determinism. To let the execution of $\boldsymbol{\lambda}$ simulate π, we structure $\boldsymbol{\lambda}$ such that $\boldsymbol{\lambda}(\Gamma)$ evaluates to a trace τ that *abstracts* the set of traces generated by π; we shall formalize this relationship shortly. Let $\Downarrow$ be the big-step evaluation relation of $\boldsymbol{\lambda}$. We write $\boldsymbol{\lambda}(\Gamma) \Downarrow \tau$ to mean that the sequential program $\boldsymbol{\lambda}$, when executed on the tape Γ, reduces to a value and generates the trace τ. Since the evaluation of $\boldsymbol{\lambda}$ is deterministic, we treat it as a (partial) function, reusing the $\mathcal{F}$ notation from earlier. A correct synchronization $\boldsymbol{\lambda}$ of π should trivially satisfy the following condition:

Remark $\forall \Gamma, \tau.\ \mathcal{F}(\Gamma, \boldsymbol{\lambda}) = \tau$ only if there exists a tape δ such that $\mathcal{F}(\Gamma, \delta, \pi) = \tau$.

While the above is a necessary condition for correct synchronization, it is not sufficient as it only requires $\boldsymbol{\lambda}$ to simulate *an* execution of π resulting from the scheduling decisions captured by

a particular tape δ. For $\boldsymbol{\lambda}$ to be a correct synchronization of π, it has to simulate executions for *all* possible tapes δ. If different scheduling decisions result in a different order of communication actions, then clearly π cannot be reduced to a sequential program $\boldsymbol{\lambda}$ as the latter can only generate a single trace. Our synchronization rules (Fig. 6; discussed later) result in an error in that case. However, even when there are no racing communication actions, different scheduling decisions may result in processes of π consuming different prefixes of the tape Γ, which might result in multiple possible traces that differ in the message values (albeit not the order of the messages). Fortunately, this is not a problem as different interleavings effectively consume different permutations of the tape Γ, and therefore have the same distribution as Γ. Formally, given a tape δ' of scheduling choices, there exists a bijection $f_{\delta'} : \mathtt{Tape}_{\mathcal{L}} \to \mathtt{Tape}_{\mathcal{L}}$ that maps the initial tape Γ to new tape Γ' such that $\mathcal{F}(\Gamma, \delta, \pi) = \mathcal{F}(\Gamma', \delta', \pi) = \mathcal{F}(\Gamma, \boldsymbol{\lambda})$.[3] We capture this intuition as the definition of correct synchronization.

Definition 4.1 (Soundness of Synchronization)**.** A sequential program $\boldsymbol{\lambda}$ is a sound synchronization of π_w program π if and only if: $\forall \Gamma, \tau, \delta.\ \exists f_\delta.\ \mathtt{bijection}(f_\delta) \wedge\ (\mathcal{F}(\Gamma, \boldsymbol{\lambda}) = \tau \Leftrightarrow \mathcal{F}(f_\delta(\Gamma), \delta, \pi) = \tau)$

Our ultimate aim of synchronization is to prove indistinguishability, which is a probabilistic property. We therefore require the synchronized and original programs to generate a trace τ with equal probability. We note that this is indeed the case.

Theorem 4.2. *If $\boldsymbol{\lambda}$ is a sound synchronization of π, then* $\Pr[\boldsymbol{\lambda} \Downarrow \tau] \;=\; \Pr[\pi \longrightarrow\!\!\!\!\!\rightarrow^* \tau]$.

Proof. Let us consider $\Pr[\boldsymbol{\lambda} \Downarrow \tau]$:

$$
\begin{aligned}
&= \sum_{\Gamma \in \mathsf{supp}(D_\Gamma)} (\mathbf{1}_\tau\ (\mathcal{F}(\Gamma, \boldsymbol{\lambda}))) * \Pr[X \xleftarrow{\$} D_\Gamma = \Gamma] \\
&= \sum_{\Gamma \in \mathsf{supp}(D_\Gamma)} \sum_{\delta \in \mathsf{supp}(D_\delta)} (\mathbf{1}_\tau\ (\mathcal{F}(f_\delta(\Gamma), \delta, \pi))) * \Pr[X \xleftarrow{\$} D_\Gamma = \Gamma] * \Pr[X \xleftarrow{\$} D_\delta = \delta] && (Def.\ 4.1) \\
&= \sum_{\Gamma \in \mathsf{supp}(D_\Gamma)} \sum_{\delta \in \mathsf{supp}(D_\delta)} (\mathbf{1}_\tau\ (\mathcal{F}(f_\delta(\Gamma), \delta, \pi))) * \Pr[X \xleftarrow{\$} D_\Gamma = f_\delta(\Gamma)] * \Pr[X \xleftarrow{\$} D_\delta = \delta] && \left(\begin{smallmatrix} permutation\ of \\ independently \\ sampled\ elements \end{smallmatrix}\right) \\
&= \sum_{\Gamma' \in \mathsf{supp}(D_\Gamma)} \sum_{\delta \in \mathsf{supp}(D_\delta)} (\mathbf{1}_\tau\ (\mathcal{F}(\Gamma', \delta, \pi))) * \Pr[X \xleftarrow{\$} D_\Gamma = \Gamma'] * \Pr[X \xleftarrow{\$} D_\delta = \delta] && (renaming) \\
&= \Pr[\pi \longrightarrow\!\!\!\!\!\rightarrow^* \tau]
\end{aligned}
$$

□

As a result of Thm. 4.2 and Def. 3.2, to show $\pi_1 \equiv \pi_2$, it now suffices to show $\forall \tau.\ \Pr[\boldsymbol{\lambda}_1 \Downarrow \tau] = \Pr[\boldsymbol{\lambda}_2 \Downarrow \tau]$, *i.e.*, $\boldsymbol{\lambda}_1 \equiv \boldsymbol{\lambda}_2$, where $\boldsymbol{\lambda}_1$ and $\boldsymbol{\lambda}_2$ are sound synchronizations of π_1 and π_2, respectively. Sec. 4.2 describes a proof technique to establish this equality.

4.1.2 Synchronization Rules. Equipped with the definition of a sound synchronization, we present a system of rewrite rules that gradually rewrite a π_w program π to a sound synchronization $\boldsymbol{\lambda}$. The key rewrite rules are given in Fig. 6 using a combination of rewrite rules ($\rightsquigarrow$) and multistep rewrite rules ($\rightsquigarrow^*$). The rewrite rules rewrite a program π (or command c_i), symbolic network Σ, and synchronized prefix $\boldsymbol{\lambda}$ under the predicate Φ to a reduced program π' (or command c_i'), new symbolic network Σ', and extended synchronized prefix $\boldsymbol{\lambda}'$. The predicate Φ is the predicate that must hold given the branches that have been selected thus far. It can be used to rule out impossible executions. When the program is finally rewritten to `unit`, then $\boldsymbol{\lambda}$ is a synchronous model of the program. The multistep rewrite relation is the reflexive transitive closure of the rewrite relation.

To simplify the rewrite rules, we overload the infix notation :: to represent `snoc`, appending to the end of a list, when clear from the context. We also introduce the `next` function in rule S-Recv which, given an invocation `next` $j\ \Sigma$, selects the first message in Σ destined for the process j. We overload

[3]Since the domain and codomain of the bijection are both the set $\mathtt{Tape}_{\mathcal{L}}$, this bijection is more precisely a permutation.

$\boxed{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, \pi) \rightsquigarrow (\Sigma', \boldsymbol{\lambda}', \pi')}$ $\boxed{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, c_i) \rightsquigarrow (\Sigma', \boldsymbol{\lambda}', c_i')}$

$$\frac{(\Phi_m, k, \_, \_) \in \Sigma \qquad k \neq i \qquad \neg(\Phi \wedge \Phi_m \implies \bot)}{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, [\texttt{send}\ j\ e]_i) \rightsquigarrow \bot}\ \text{S-RacingSend}$$

$$\frac{\begin{array}{c}\Sigma' = \Sigma :: (\Phi, i, j, e) \\ \boldsymbol{\lambda}' = \boldsymbol{\lambda}[\bullet \mapsto \lambda\_.\ \texttt{trace}\ i\ j\ e \gg\!= \bullet] \qquad \forall(\Phi_m, k, \_, \_) \in \Sigma.\ k = i \vee (\Phi \wedge \Phi_m \implies \bot)\end{array}}{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, [\texttt{send}\ j\ e]_i) \rightsquigarrow (\Sigma', \boldsymbol{\lambda}', [\texttt{unit}]_i)}\ \text{S-Send}$$

$$\frac{(\Phi_m, i, j, e) = \texttt{next}\ j\ \Sigma \qquad \Sigma' = \Sigma - (\Phi_m, i, j, e) \qquad \boldsymbol{\lambda}' = \boldsymbol{\lambda}[\bullet \mapsto \lambda\_.\ \texttt{return}\ e \gg\!= \bullet]}{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, [\texttt{recv}]_i) \rightsquigarrow (\Sigma', \boldsymbol{\lambda}', [\texttt{unit}]_i)}\ \text{S-Recv}$$

$$\frac{\boldsymbol{\lambda}' = \boldsymbol{\lambda}[\bullet \mapsto \lambda\_.\ \texttt{samp} \gg\!= \bullet]}{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, [\texttt{samp}]_i) \rightsquigarrow (\Sigma, \boldsymbol{\lambda}', [\texttt{unit}]_i)}\ \text{S-Samp} \qquad \frac{\boldsymbol{\lambda}' = \boldsymbol{\lambda}[\bullet \mapsto \lambda\_.\ \texttt{return}\ e \gg\!= \bullet]}{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, [e]_i) \rightsquigarrow (\Sigma, \boldsymbol{\lambda}', [\texttt{unit}_i])}\ \text{S-Exp}$$

$$\frac{\Phi \vdash (\Sigma, \bullet, [m]_i) \rightsquigarrow (\Sigma', \boldsymbol{\lambda}_i, [\texttt{unit}]_i) \qquad \boldsymbol{\lambda}' = \boldsymbol{\lambda}[\bullet \mapsto \boldsymbol{\lambda}_i[\bullet \mapsto \lambda\, i\_x.\ \texttt{return}\ () \gg\!= \bullet]]}{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, [\texttt{let}\ x = m\ \texttt{in}\ c]_i) \rightsquigarrow (\Sigma', \boldsymbol{\lambda}', [c[i\_x/x]]_i)}\ \text{S-Let}$$

$$\frac{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, [c]_i) \rightsquigarrow (\Sigma', \boldsymbol{\lambda}', [c]_i')}{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, [c]_i \parallel P) \rightsquigarrow (\Sigma', \boldsymbol{\lambda}', [c]_i' \parallel P)}\ \text{S-Par}$$

$$\frac{\Phi \wedge e \vdash (\Sigma, \boldsymbol{\lambda}_0, [c_1]_i \parallel P) \rightsquigarrow^* (\Sigma_1', \boldsymbol{\lambda}_1, \texttt{unit}) \qquad \Phi \wedge \neg e \vdash (\Sigma, \boldsymbol{\lambda}_0, [c_2]_i \parallel P) \rightsquigarrow^* (\Sigma_2', \boldsymbol{\lambda}_2, \texttt{unit})}{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, [\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]_i \parallel P) \rightsquigarrow^* (\Sigma, \boldsymbol{\lambda}[\bullet \mapsto \texttt{if}\ e\ \texttt{then}\ \boldsymbol{\lambda}_1\ \texttt{else}\ \boldsymbol{\lambda}_2], \texttt{unit})}\ \text{S-If}$$

$$\frac{\boldsymbol{\lambda}' = \boldsymbol{\lambda}[\bullet \mapsto \lambda\_.\ \texttt{return}\ ()]}{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, \Big\|\texttt{unit}) \rightsquigarrow (\Sigma, \boldsymbol{\lambda}', \texttt{unit})}\ \text{S-Done} \qquad \frac{\Phi \vdash \bot}{\Phi \vdash (\Sigma, \boldsymbol{\lambda}, \pi) \rightsquigarrow (\Sigma, \boldsymbol{\lambda}, \texttt{unit})}\ \text{S-Impossible}$$

Fig. 6. Key rules to derive a synchronization of a π_w program. A program can be synchronized if $\bot$ can never be derived from the program. () is written as `unit` in these rules for readability. $\boldsymbol{\lambda}_0$ is the initial monadic program `return` () $\gg\!=$ •, where $\gg\!=$ is the monadic bind operation and • represents a hole that is typically filled by the next rule.

the notation $\Sigma - m$ to mean removing the first element from Σ which matches the message m from the list. By appending new messages to the end of Σ and reading and removing the first message for a process, we model reliable in-order message delivery using the list Σ. The synchronized process is built incrementally by filling in holes (•). We overload the notation $\boldsymbol{\lambda}[\bullet \mapsto f]$ to represent replacing the hole • with f.

The predicate Φ is updated monotonically by conjuncting it with taken branch conditions. This is performed by the rule S-If. Intuitively, the S-If rule explores "what-if" scenarios and then emits an expression that selects between them. More precisely, S-If derives a sub-program that models the case that the `then` branch was taken, and a sub-program that models the case that the `else` branch was taken. Each of these sub-programs is derived under the current predicate Φ conjuncted with the branch condition for the then branch or the negation of the branch condition for the else branch. It then combines these two sub-programs as branches of an `if-then-else` expression that branches on the original condition. Unlike the other rewrite rules, S-If is only defined in terms of multi-step rewrites. This formulation reduces the amount of state that must be tracked when deriving the synchronization. A consequence of using multi-step rewrites is that the rule is defined

on a program $c_i \parallel P$ rather than only on the `if-then-else` command in isolation to ensure a sound rewrite. Note that the subsequent program is sound for an arbitrary P because any derivations relating to it can simply be duplicated in both branches. The S-Impossible is then used to rule out execution paths that cannot exist due to mutually exclusive branches.

A synchronization $\boldsymbol{\lambda}$ is a monadic functional program. Specifically it uses a monad, that we call the `W` monad, that tracks both the random tape and the network trace as its hidden state. A synchronization is derived starting from the empty program $\boldsymbol{\lambda}_0$ which is simply `return` $() \gg\!= \bullet$. The model is then incrementally built by filling in the hole $\bullet$. The holes are expected to have the type $'a \rightarrow \mathtt{W}\ 'a$, the type of the second argument to the monadic bind function $\gg\!=$. Subsequent monadic expressions are composed together using binds. When every process that has been run in parallel has been added to the synchronized model as indicated by `unit`, the final hole in the model is then filled using the S-Done rule. The S-Par rule allows for multiple valid synchronizations to be produced since $\parallel$ is commutative and associative reflecting the observation that $\boldsymbol{\lambda}$ simulates *an* execution of π. The synchronization can be proven sound by verifying the conditions given in Theorem 4.3.

Only the rules S-Send and S-Samp alter the state tracked by the monad: the network trace and the random tape. As a result, these rules insert monadic function calls (`trace` and `samp`) respectively to the sequential program $\boldsymbol{\lambda}$. The S-Send rule also checks that the send will not race with any message currently in Σ to avoid unsound rewrites. The S-Recv rule reads and removes the next message intended for the current process from Σ. The `recv` is simply replaced with the content of the message in $\boldsymbol{\lambda}$. All expressions are pushed through unchanged using S-Exp. Let bindings are replaced with the monadic equivalent using binds under rule S-Let. To avoid potential name clashes between processes, variables are renamed with the process ID prepended.

A synchronization of a program π can be generated by rewriting from an empty symbolic network and synchronized prefix. More formally, a synchronization $\boldsymbol{\lambda}$ can be produced from a program π under initial assumptions Φ_0 by performing the multistep rewrite: $\Phi_0 \vdash ([], \boldsymbol{\lambda}_0, \pi) \rightsquigarrow^* (\Sigma', \boldsymbol{\lambda}, \mathtt{unit})$. Φ_0 can contain initial information such as known axioms on cryptographic primitives. In addition to producing a synchronization, the rewrite rules are used to ensure that the synchronization is sound.

Theorem 4.3. *$\boldsymbol{\lambda}$ is a sound synchronization of π under assumptions Φ_0 if and only if: $\Phi_0 \vdash ([], \boldsymbol{\lambda}_0, \pi) \rightsquigarrow^* (\Sigma', \boldsymbol{\lambda}, \mathtt{unit})$, and there does not exist a sequence of rewrites such that $\Phi_0 \vdash ([], \boldsymbol{\lambda}_0, \pi) \rightsquigarrow^* \bot$.*

The proof of Theorem 4.3 follows from Def. 4.1 and induction over the rewrite rules.

As Theorem 4.3 states, a synchronization is unsound only if $\bot$ can be derived using the rules in Fig. 6. This can only occur if two `send`s race using the S-RacingSend rule. A `send` races only if there could exist another message in the network that is from another process and these two messages do not contain mutually exclusive predicates. The first condition, that `send`s can only race if they're from different processes, captures the observation that `send`s in the same process are always executed in program order and therefore do not race given our reliable in-order delivery semantics. The second condition, that two `send`s only race if they don't have mutually exclusive predicates, allows for filtering out impossible executions. For example, consider the program in Fig. 7. This program does not race because if process i sends a message then process j will not send a message, and vice versa. In other words, these `send`s are mutually exclusive because they branch on opposite conditions. By tracking predicates using Φ, the rules ensure that such impossible executions can be ruled out using the S-Impossible rewrite rule. We only check for racing `send`s. `recv`s cannot race because they only modify the state for their own process. Similarly, racing `samp`s

do not have an effect on the likelihood of emitting a particular trace because they simply consume different permutations of the tape Γ and therefore are drawn from the same distribution.

```
[if cond then send j 42 else recv]_i || [if cond then recv else send i 7]_j
```

Fig. 7. Example protocol that does not race since both processes branch on the same condition.

4.2 Proof System

The rewrite rules in Fig. 6 generate a monadic functional program that, via straightforward translation to an imperative setting, is amenable to reasoning via relational Hoare logics such as pRHL [Barthe et al. 2009]. However, functional programs are known to be more suitable for equational reasoning by virtue of their algebraic nature even when modeling computational effects [Gibbons and Hinze 2011]. Reasoning with equational axioms to model common cryptographic assumptions, such as that decryption is the inverse of encryption, is important for further proof automation. Unfortunately, this equational reasoning is more difficult when non-deterministic sampling is interleaved throughout the program. We therefore choose to lift non-determinism to the front of the program by first sampling the tape Γ. We then replace non-deterministic sampling with deterministic reads of Γ, just like in the operational semantics (Fig. 5). We can then reason directly about the deterministic suffix as a functional program. Concretely, we interpret the `W` monad (the monad in Fig. 6) as the computation `(tape * trace) → (option 'a * tape * trace)`, which consumes a prefix of the input tape and appends a suffix to the trace. We thus factor out non-determinism from the `W` monad. Let $\boldsymbol{\lambda}(\Gamma)$ denote the execution of `W` computation $\boldsymbol{\lambda}$ on the tape Γ starting with an empty trace. Also, let $\boldsymbol{\lambda}$ be the functional synchronization generated by the rules in Fig. 6 for program π. With the above interpretation of `W`, the correct synchronization of π is the following program:

$$\texttt{let } \Gamma \texttt{ = sample\_from } D_\Gamma \texttt{ (* } \Gamma \overset{\$}{\leftarrow} D_\Gamma \texttt{ *) in } \boldsymbol{\lambda}(\Gamma)$$

This program has non-determinism only at the top-level; other sampling calls simply read from the pre-sampled tape. As a result pRHL only needs to be applied at the top-level, leaving the rest for straightforward equational reasoning.

Concretely, let π_1 and π_2 be two π_w programs which need to be proven indistinguishable. This in turn requires proving the following indistinguishability property:

$$\texttt{let } \Gamma_1 \texttt{ = sample\_from } D_\Gamma \texttt{ in } \boldsymbol{\lambda}_1(\Gamma_1) \quad \equiv \quad \texttt{let } \Gamma_2 \texttt{ = sample\_from } D_\Gamma \texttt{ in } \boldsymbol{\lambda}_2(\Gamma_2)$$

The top-level expression on both sides is sequential composition (via `let`), which allows the application of pRHL rule R-Seq. The first half of sequential composition is sampling from distribution D_Γ on both sides. Applying pRHL rule R-Rand lets us deduce $\Gamma_2 = f\,\Gamma_1$ for a given bijection f on tapes provided that we can show $\forall(\Gamma \in \mathsf{supp}(D_\Gamma)).D_\Gamma(\Gamma) = D_\Gamma(f\,\Gamma)$. Assuming this can be shown, $\Gamma_2 = f\,\Gamma_1$ can be assumed as pre-condition to prove $\boldsymbol{\lambda}_1(\Gamma_1) = \boldsymbol{\lambda}_2(\Gamma_2)$, which completes the proof of indistinguishability. The proof obligations for indistinguishability are compiled into a proof rule shown in Fig. 8.

5 IMPLEMENTATION

The indistinguishability property formalized in Sec. 3.2 and proof rules introduced in Sec. 4 are implemented in our library Waldo. Waldo is a library in F* for specifying and proving indistinguishability for protocols with a sequential model. In addition, F* supports extraction to OCaml meaning that Waldo's network model, encryption models, and random sampling model can be

$$\frac{\boldsymbol{\lambda}_1 \text{ is a sound synchronization of } \pi_1 \quad \boldsymbol{\lambda}_2 \text{ is a sound synchronization of } \pi_2 \quad f \text{ is a bijection} \quad \forall(\Gamma \in \mathsf{supp}(D_\Gamma)).\, D_\Gamma(\Gamma) = D_\Gamma(f\,\Gamma) \quad \forall(\Gamma \in \mathsf{supp}(D_\Gamma)).\, \boldsymbol{\lambda}_1(\Gamma) = \boldsymbol{\lambda}_2(f\,\Gamma)}{\Pr[\pi_1 \longrightarrow\!\!\!\!\!\rightarrow^* \tau] = \Pr[\pi_2 \longrightarrow\!\!\!\!\!\rightarrow^* \tau]}\ \textsc{PrObsEq}$$

Fig. 8. Proof rule for proving the indistinguishability of sequential programs $\boldsymbol{\lambda}_1$ and $\boldsymbol{\lambda}_2$

swapped out with concrete implementations to create a functioning OCaml implementation that can be executed and tested. We give an overview of Waldo's implementation here but give a more detailed explanation in the supplementary material.

Waldo expects protocols to be modeled as effectful functions of type `pub -> priv -> Wald unit`, where `pub` is a tuple of public arguments that an observer is assumed to know, `priv` is a tuple of private arguments that an observer must not learn anything about, and `Wald` is a *computational effect* that tracks the network trace and models random sampling from a uniform distribution. Since Waldo is specialized to a single-process model, it does not need to track the message queue (M) and tape of choices (δ) that are part of the state in the formalism from Sec. 3. The only state components Waldo needs to model are the random tape and network trace. The effectful functions Waldo works on are implementations of the synchronous models derived from a protocol π, following the rules in Sec. 4, where π is parameterized over tuples of publicly known and private variables.

The `Wald` effect in Waldo is implemented using a monad that operates over a finite tape of a fixed length and a list of network traces. Specifically, Waldo adapts the random monad introduced in [Grimm et al. 2018], specializes it to operate on a read-only tape of bytes, and layers on top a monad to track the network trace. In addition to the standard type constructor, return function, and bind function, the monad also has a sample function that is used to read from the tape and a trace function that is used to append a message to the trace. The monadic function `sample n` is defined so that it returns `None`, indicating an exception, if there are fewer than `n` unread bytes in the tape. Otherwise it returns the next `n` bytes from the tape while advancing the tape forward `n` positions. `trace i j m` simply appends the tuple `(i,j,m)` to the network trace to record that the message `m` was sent from process `i` to process `j`.

PrObsEq, the proof rule given in Fig. 8, is encoded in Waldo as a lemma. However, Waldo specializes the rule to programs that sample from uniform distributions. This reduces the proof obligation to simply providing a suitable bijection since $\forall(\Gamma \in \mathsf{supp}(D_\Gamma)).\, D_\Gamma(\Gamma) = D_\Gamma(f\,\Gamma)$ holds for any bijection when D_Γ is the uniform distribution. We note that this specialization does not reduce the usefulness of Waldo for practical protocols since the secure random-number generators used by cryptographic primitives aim to sample from a uniform distribution. The lemma takes as input two monadic `Wald` computations, a trace, and a bijection on the random tape. The lemma holds if, for all tapes, running the first computation on the tape produces the given trace iff running the second computation on the bijected tape does, and vice versa. If this is the case, the computations are equally likely to producing any given trace since there is an equally likely tape that always results in the computations producing the same trace.

Waldo further encodes the indistinguishability property introduced in Sec. 3.2, but specializes it to protocols that can be modeled using a deterministic function on random tapes that takes a set of public arguments and a set of private arguments. For convenience, Waldo provides both a function that generates an indistinguishability post-condition (`indistinguishable_ensures`) as well as a lemma (`indistinguishable_lemma`) that can be used to prove indistinguishability. For example, Fig. 9 shows the indistinguishability specification of the fixed example OTP protocol, introduced in Sec. 2, using these convenience functions. The specification states that the `otp_proto_indistinguishable` is a lemma parameterized on two tuples of bytes (`m` and `m'`) which represent the secret messages and on

a trace (`trace`). The lemma is valid if it ensures that the post-condition (`indistinguishable_ensures`) holds for all possible instantiations of its parameters and without any preconditions. The post-condition specifically states that the likelihood of producing the given trace (`t`) is equivalent when the program is run with the public argument (`()`)[4] and with each of the private arguments (`m` and `m'`). The lemma is then proven using `indistinguishable_lemma`. It takes the supplied bijection (`bij_otp m m'`) and attempts to fulfill the post-condition using the encoding of the PrObsEq rule. To do this, `indistinguishable_lemma` reifies the effectful `otp_proto` function to reason about its monadic implementation.

```
(** States that the example OTP protocol guarantees indistinguishability for all
    possible pairs of secret messages. *)
let otp_proto_indistinguishable (m: byte & byte) (m': byte & byte) (t: trace)
    : Lemma (ensures indistinguishable_ensures otp_proto () m m' t)
= (* bij_otp is provided by Waldo. It permutes the tape as needed to show
        indistinguishability when using OTP encryption *)
    let bij = (bij_otp m m') in
    indistinguishable_lemma otp_proto bij () m m' t
```

Fig. 9. Indistinguishability lemma and proof in Waldo for the fixed OTP protocol introduced in Sec 2. It states that the protocol model (`otp_proto`) run with secret arguments (`m` or `m'`) and no public arguments (`()`) is indistinguishable with respect to the trace *t*. It can be automatically proven using the indistinguishability lemma and OTP bijection (`bij_otp`) provided by Waldo.

To successfully prove indistinguishability, it suffices to show that there is a bijection on the random tape such that the protocol always produces the same trace for each pair of possible private arguments. Intuitively, since the tapes are sampled from a uniform distribution, applying the bijection to any tape produces a tape from the same distribution. Therefore, showing equality with respect to the bijected tape is sufficient to show equality with respect to the probability distribution. With the proper bijection, the `indistinguishable_lemma` is theoretically sufficient to automatically discharge indistinguishability proofs. We find this is often the case in practice, including for the example in Fig. 9. Waldo's models of cryptographic primitives come equipped with suitable bijections for simple uses. Programmers can use these models of cryptographic primitives out-of-the-box to specify and verify protocols, as done with the `bij_otp` bijection in Fig. 9.

Waldo specifically comes equipped with a model of One-Time Pad encryption as well as models of both symmetric and asymmetric encryption, secure hashing, and secure random number generation. As discussed in Sec. 2, Waldo models the primitives other than One-Time Pad encryption using perfect assumptions modulo lengths. For example, as shown in Fig. 10, encryption is modeled as a function that maps a message `m` and key `k` to a randomly sampled ciphertext with the same length as `m`. Symmetric decryption is then modeled as the function such that `dec (enc m k) k = m`. Programmers can use these cryptographic primitive models out-of-the-box or can introduce their own models that further relax the assumptions Waldo makes at the potential cost of automation.

6 CASE STUDY: TLS ENCRYPTED CLIENT HELLO

To show how Waldo helps verify privacy properties in practice, we use Waldo to verify indistinguishability for the Transport Layer Security (TLS) Encrypted Client Hello (ECH) extension

[4] `()` in this case means there is no public argument

```
let enc (m: bytes) (k: bytes) : Wald (c: bytes{len c == len m}) = sample (len m)
```

Fig. 10. Model of perfect symmetric encryption in Waldo. Encrypting a plaintext `m` under a key `k` simply returns a bytestring of random bytes as long as the input plaintext. The type of `c` is a dependent return type of `bytes` that is valid only if the length of `c` is equivalent to the length of `m`.

[Rescorla et al. 2022] under the assumption that the cryptographic primitives it uses provide perfect secrecy. Even under these strong assumptions, we find the specification insufficient to prove indistinguishability without modifications as detailed in Sec. 6.4.

The TLS protocol is used to protect network traffic for billions of internet users and is best known for its use in HTTP over TLS (HTTPS). The latest version, TLS 1.3, uses authenticated encryption to provide protection from both ciphertext tampering and plaintext observation. Unfortunately, TLS sends unencrypted information during connection establishment, such as the Server Name Indication (SNI) extension, that can be used to determine the domain a user is connecting to, such as twitter.com. This information can be used to track users browsing habits or to censor connections to banned websites. This is not just theoretical, internet censors rely on this information today to block connections to banned sites [Chai et al. 2019; Gatlan 2019].

The TLS Encrypted Client Hello (ECH) extension [Rescorla et al. 2022] aims to solve this weakness by encrypting the sensitive parts of the TLS handshake.[5] It aims to provide 2 primary security goals:

(1) Use of ECH does not weaken the security properties of TLS without ECH.
(2) TLS connection establishment to a host with a specific ECH and TLS configuration is indistinguishable from a connection to any other host with the same ECH and TLS configuration.[6]

This case study focuses on the second goal, verifying that TLS ECH connection establishments are indistinguishable.

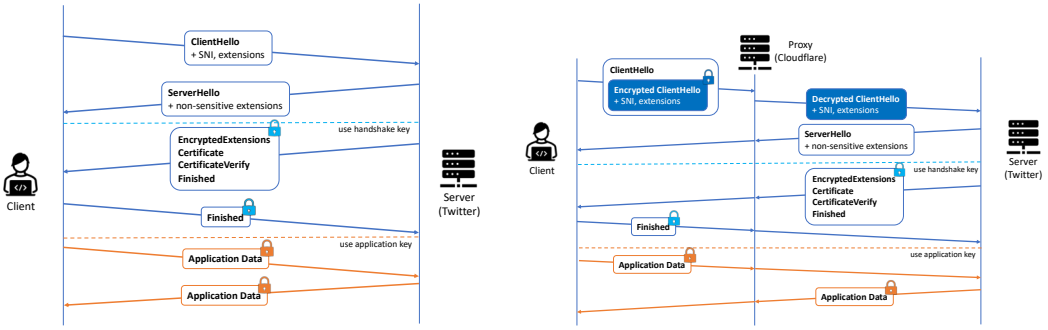
6.1 TLS and TLS ECH Handshakes

At a high level, TLS works by establishing a shared key between a client and server that is then used to encrypt future communication. It also negotiates agreement on a particular configuration to use for the connection. This connection setup and key exchange happens during the TLS handshake.

TLS 1.3 handshakes start with a ClientHello message from the client to server that contains information about configurations the client supports and the kind of connection the client wants to establish. In a typical handshake, the server responds with a ServerHello identifying the encryption configuration selected for the connection. Both hellos are sent unencrypted. The information in the hellos is sufficient for both the client and server to derive a key to protect the rest of their handshake. Thenceforth, the server sends encrypted messages that include information about the selected or supported parameters for the connection (EncryptedExtensions), the server's certificate proving its identity (Certificate), and information required to validate the certificate (CertificateVerify). Finally, both the client and server send a Finished handshake message to conclude the handshake. A typical TLS 1.3 handshake is shown in Fig. 11a.

ECH works by hiding a set of backend servers behind a known proxy, or client-facing server.[7] The client-facing server publishes an ECH configuration advertising its public key and associated

[5]The ECH draft mandates that TLS version 1.3 or later must be used. At present that means that only TLS version 1.3 can be used when using ECH.
[6]The set of hosts which share the same ECH and TLS configuration is referred to as the anonymity set.
[7]ECH allows the client-facing server and backend server to be the same server, but the largest anonymity sets will likely use separate backend and client-facing servers.



(a) Typical TLS 1.3 connection establishment.

(b) Typical TLS ECH connection establishment. The inner ClientHello is shown in filled blue.

Fig. 11. Typical connection establishment for TLS 1.3 both without (11a) and with ECH (11b). The handshake is shown in blue. Encrypted messages are indicated with a lock icon.

metadata.[8] Clients that wish to establish a connection using ECH send an encrypted ClientHello message meant for the backend server using the published public key for the client-facing server. This encrypted ClientHello is called the inner ClientHello. The client sends the encrypted payload within a plaintext ClientHello, called the outer ClientHello, to the client-facing server. The client-facing server decrypts the inner ClientHello extension, and forwards it on to the backend server. The handshake then continues as normal between the client and backend server with the client-facing server forwarding messages between the two. A typical TLS ECH handshake is shown in Fig. 11b.

6.2 Formal Model

TLS ECH connection establishment can be modeled as communication between processes using the formalism introduced in Sec. 3. There are nominally three processes that communicate: the client, the proxy, and the server. However, the protocol can be synchronized following the procedure detailed in Sec. 4. This synchronization is intuitively possible since only one process, either the client, proxy, or server, can send out a message at any given time. The model is parameterized based off shared public arguments (the TLS and ECH configurations) as well as private arguments that an attacker should not be able to differentiate (the domain the connection is meant for). Furthermore, the model relies on cryptographic primitives, including encryption and a secure hash function. Both are assumed to be perfect modulo the length assumptions: the former is assumed to generate a ciphertext of same length as the plaintext and the latter is assumed to generate a hash of fixed length. These assumptions allow for evaluating if the protocol itself provides the guarantees it aims to provide as long as the cryptographic primitives it uses are sufficiently strong.

6.3 Implementation in F*

The synchronized model is implemented in F*, using Waldo, as an effectful function `ech_handshake` that has the type: `configs -> ech_priv_args -> Wald unit`. The function first takes a configuration argument that contains all the configuration information needed for both the client and server including the client's TLS configuration, the client's ECH-specific configuration, the server's TLS configuration, and the server's ECH-specific configuration. It then takes a tuple of private arguments that include the name of the server the client wants to talk to and the server's certificate.

[8]The manner of publishing is out-of-scope of ECH's design but could be distributed through DNS [Schwartz et al. 2021].

Indistinguishability for TLS ECH is specified using the `indistinguishable_ensures` proposition and verified using the `indistinguishable_lemma` lemma provided by Waldo. The bijection Waldo provides for perfect encryption can be used out-of-the-box for this model. The bijection and lemma are sufficient for Z3 to discharge the proof if the protocol has no bugs.

6.4 Formal Verification Results

Unfortunately, the TLS ECH indistinguishability lemma fails to prove without preconditions, even under the assumed use of ideal cryptographic primitives. The model requires several additional preconditions before the indistinguishability lemma can prove. The preconditions represent assumptions that could be bugs with the specification. In particular, the following assumptions are needed:

(assumption already in spec) *The inner ClientHello messages are padded out to the same length.* If the inner ClientHello messages are not padded out to the same length, they can leak information about which server the message is going to. In particular, the Server Name Indication field varies in length between servers for different domains. This is a known potential weakness and the TLS ECH specification provides a padding scheme that SHOULD be followed [Rescorla et al. 2022]. We therefore consider this assumption adequately covered by the specification.

(missing from spec) *The extensions in plaintext messages from the server, including ServerHello and HelloRetryRequest, are always placed in the same order across all servers in the anonymity set.* Neither the TLS 1.3 specification nor the TLS ECH specification specify an order in which extensions should appear in messages. The ServerHello and HelloRetryRequest messages are sent unencrypted so the order in which they present extensions can be seen by an external observer. This means that all servers in an anonymity set must send their extensions in the same order, otherwise they leak which subset of the anonymity set they are in. Since this is addressed by neither the TLS 1.3 nor the TLS ECH spec, we consider this an issue that should be addressed.

(inadequately addressed in spec) *The EncryptedExtensions messages are padded out to the same lengths. The same is also true for the Certificate and CertificateVerify messages.* The encrypted messages sent back by the server, including the EncryptedExtensions, Certificate, and CertificateVerify messages, must also be padded out to the same length. The Certificate message is especially important to pad since the certificates for different sites tend to vary dramatically. For instance, a TLS 1.3 connection to google returned a Certificate message 6343 bytes in length, whereas a connection to twitter returned a Certificate message 2966 bytes in length. The need to pad these messages out is only addressed in the specification with the text: "In addition to padding ClientHelloInner, clients and servers will also need to pad all other handshake messages that have sensitive-length fields. For example, if a client proposes ALPN values in ClientHelloInner, the server-selected value will be returned in an EncryptedExtension, so that handshake message also needs to be padded using TLS record layer padding" [Rescorla et al. 2022]. We believe the current language is inadequate given the ease with which these messages could identify the server. We therefore recommend that the specification describe the leakage that can occur if these messages are not padded and state that these messages SHOULD be padded.

With these assumptions added, the indistinguishability lemma successfully discharges. Therefore, the model shows that TLS ECH provides indistinguishability for connection establishment against a passive observer as long as the protocol is modified so that the assumptions stated above hold. Importantly, our model uncovered required assumptions, such as the same extension ordering requirement for plaintext server messages, that were not captured by previous models [Bhargavan et al. 2022] that used the symbolic model for verification.

7 CASE STUDY: PRIVATE INFORMATION RETRIEVAL

In addition to using Waldo to verify indistinguishability for TLS ECH, it can also be used to verify similar privacy properties. In this section, we verify $m-1$ privacy for a Private Information Retrieval (PIR) scheme. Private Information Retrieval is the problem of querying information from a set of databases containing identical data, without leaking which piece of data the client requests. PIR protocols are used in or strongly related to several other privacy-related protocols including oblivious transfer [Di Crescenzo et al. 2000; Gertner et al. 1998], instance hiding [Beaver and Feigenbaum 1990], multiparty computation [Beimel et al. 2012], secret sharing [Beimel et al. 2012; Shamir 1979], homomorphic encryption [Ishai et al. 2005], and collision-resistant hashing [Ishai et al. 2005]. Though indistinguishability is only proven here for one PIR scheme, the same approach can be extended to other information theoretic schemes and protocols that aim to provide privacy guarantees.

Here we verify a generalization of the two-server scheme to presented in [Chor et al. 1998]. The generalization extends the scheme to m servers and guarantees perfect privacy as long as at least one server does not collude with the others (called $m-1$ privacy). The m servers each have a copy of an n-bit database d. The client wants to learn what the i-th bit of d is set to, without leaking the index i they're interested in. As shown in Fig. 12, the client sends a series of n-bit queries to the servers. Each server responds with a single bit. They determine this bit by bitwise and-ing ($\wedge$) the query they receive with d, then xor-ing ($\oplus_i$) the resulting bits together. The first $m-1$ queries the user sends are randomly sampled from the uniform distribution over n-bit bitstrings. The final query is constructed by bitwise xor-ing ($\oplus$) the previous queries with the query the client would send if they didn't need privacy ($q = 2^i$). The client can then reconstruct the i-th bit of d by xor-ing all the responses together.

7.1 Formal Model

The PIR protocol is modeled as communication between $m+1$ processes where m of the processes are identical modulo the process ID and represent the servers, and the remaining process represents the client. The traces end up alternating between queries and responses. Since all the responses are independent and all the servers are identical modulo the process ID, the order in which each response is received is irrelevant due to symmetry. The $m-1$ privacy property is formulated as indistinguishability when the trace is missing a single query and associated response.

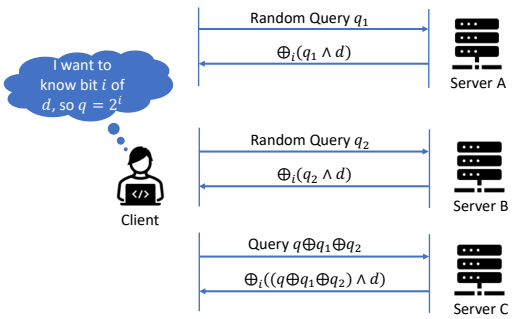


Fig. 12. Simple Private Information Retrieval protocol with $m = 3$ servers, each with a copy of the n-bit database d. The client wants to learn the entry in d corresponding to the bit set in q. The user sends out $m-1$ random n-bit queries, and a final query derived from the previous queries. The servers each respond with a single bit. The client recovers the desired bit of d by XOR-ing the responses together.

7.2 Implementation in F*

Waldo is again used to specify and verify this protocol. First a `pir` function is defined to model the protocol. The function has the signature:

```
(d: lbits n) -> (m: nat{m > 1}) ->
    (i: nat{i < n}) -> WaldBit bit.
```

This function takes an implicit argument for n, a database of n bits, the number of servers m,

and the i-th bit of d the client wants to learn. It then records the queries that would be sent to each server along with the responses the servers would send back and puts them into a list to create the trace. `WaldBit` is the same effect as `Wald`, except it reads random bits from a tape of bits where `Wald` reads random bytes from a tape of bytes. The first $m-1$ queries are each created by randomly sampling n bits from the random tape. The final query is determined by xor-ing the first $m-1$ queries together with the true query (the n-bit representation of 2^i). Each response is computed by bitwise and-ing the query with d and then xor-ing all the resulting bits together to get a single bit. The `pir` function then returns the i-th bit of d, computed by xor-ing all the server responses together.

Proving $m-1$ privacy for this protocol requires reasoning about two different scenarios. In the case that the last query and response are omitted from the trace, every query in the trace is randomly generated. The identity function is a sufficient bijection to prove indistinguishability in this case. In the case that a different query and response are omitted from the trace, the last query in the trace is not randomly generated but instead is created by xor-ing previous queries together. Unfortunately the identity bijection does not suffice in this case. Proving indistinguishability for this case then boils down to showing that there is a bijection that maps the last query for the first run of the program ($q_1 \oplus rq_1 \oplus \cdots \oplus rq_{m-1}$) to the last query for the other run of the program ($q_2 \oplus rq_1 \oplus \cdots \oplus rq_{m-1}$) where q_1 and q_2 are the different true queries for each program and rq_i indicates the i-th random query. Suppose, without loss of generality, the first query and response are picked to omit from the trace. This means excluding rq_1 and its response from the trace. If the bijection $\lambda\, t.\ q_2 \oplus q_1 \oplus t$ is used on the portion of the tape that rq_1 is sampled from and the bijected tape is used only when running the second program, then the last query for the second program would be $q_2 \oplus (q_2 \oplus q_1 \oplus rq_1) \oplus \cdots \oplus rq_{m-1}$ which is equivalent to $q_1 \oplus rq_1 \oplus \cdots \oplus rq_{m-1}$, the last query from the first program. This bijection can then be generalized when omitting the i-th query and response to instead be $\lambda\, t_i.\ q_1 \oplus q_2 \oplus t_i$ where t_i is the portion of the tape used to sample rq_i.

The top-level $m-1$ privacy lemma can then be proven by case analysis using these two bijections. It is proven automatically using the indistinguishability lemma from Waldo along with properties of XOR and AND. The proof confirms that the scheme introduced in [Chor et al. 1998] provides $m-1$ privacy. Furthermore, it shows that Waldo can be used for reasoning about privacy properties beyond simple indistinguishability.

8 RELATED WORK

Many previous works formulate and verify properties similar to the indistinguishability properties we formulate and verify. These properties include secrecy, observational equivalence, and diff-equivalence in the symbolic model, indistinguishability in the computational model, and non-interference. Related work for each of these topics is briefly covered below.

Symbolic Cryptographic Protocol Verification. Cryptographic protocol verification is typically done in either the symbolic model or the computational model. In the symbolic model, also called the Dolev-Yao model [Dolev and Yao 1983], messages and keys are represented as symbolic terms while the protocol is represented using functions along with an equational theory on those terms and functions that model what information an attacker can derive. This symbolic model models an active attacker that is strictly more powerful than the passive attacker we assume, yet it does not implicitly model the potential leakage of partial information, such as the length of a message, that can reduce privacy guarantees. This leads to the situation described in Sec. 1 where a protocol can verify but still leak identifying information.

ProVerif [Blanchet et al. 2016] and Tamarin [Meier et al. 2013] are the most prominent protocol verification tools that work in the symbolic model. DY* [Bhargavan et al. 2021] is another

framework based on the symbolic model that is written in F* like Waldo. However, DY* has the same limitations in precision as other symbolic model tools and, unlike ProVerif and Tamarin, does not support reasoning about observational equivalence. These tools have been used to verify real-world protocols including the Signal protocol [Bhargavan et al. 2021; Kobeissi et al. 2017], TLS 1.3 [Bhargavan et al. 2017; Cremers et al. 2017, 2016], Noise protocols [Girol et al. 2020; Kobeissi et al. 2019], the 5G authentication key exchange protocol [Basin et al. 2018], the Neuchâtel and Norwegian voting protocols [Cortier et al. 2018; Cortier and Wiedling 2017], and the same TLS ECH extension modeled in this paper [Bhargavan et al. 2022]. Of these, only the 5G model [Basin et al. 2018], Noise Tamarin model [Girol et al. 2020], voting protocols [Cortier et al. 2018; Cortier and Wiedling 2017], and TLS ECH model [Bhargavan et al. 2022] consider privacy properties. Both the 5G and TLS ECH models explicitly state they exclude finer-grained information like message length from their models. Furthermore, the 5G model only checks if an adversary could fully learn the anonymous information. The TLS ECH model does checks if an adversary could partially learn the anonymous information through using diff-equivalence properties, but the model still excludes leakage of finer-grained information like message length that can trivially violate these properties. The Noise model and voting models similarly don't consider fine-grained information leakage despite verifying observational equivalence or diff-equivalence. Though it is possible to model fine-grained information leakage in the symbolic model, it must be done explicitly. It is therefore easy for users to incorrectly believe that their protocol guarantees anonymity when verified in the symbolic model. In contrast, our approach is more precise and allows for detecting leakage of additional partial information without explicitly modeling this partial information. It is this capability that allows us to detect that padding and ordering differences in TLS ECH can leak which server a user is talking to.

Computational Cryptographic Protocol Verification. In contrast to the symbolic model, tools in the computational model do model fine-grained partial information leakage. They represent messages as bitstrings and protocols as probabilistic functions on those bitstrings. The computational model is strictly more precise than the symbolic model, but also tends to be harder to verify. The ability to reason about both approximate and quantitative forms of indistinguishability makes the computational model tools more expressive than our approach which currently only allows reasoning about perfect indistinguishability properties between probabilistic programs that sample from a uniform distribution. Unfortunately, writing and verifying protocols in the computational model often requires manual proof effort and experience with cryptographic proofs. In contrast, our approach, in which a protocol is modeled as a deterministic monadic function, allows for a direct specification style that is much closer to the implementations and pseudocode protocol designers already write. Our approach is also able to reduce required manual proof efforts by sequentializing concurrent protocols and using perfect encryption assumptions.

CryptoVerif [Blanchet 2008] and EasyCrypt [Barthe et al. 2011] are the most prominent verification tools in the computational model. They have been used to verify properties for real-world cryptographic protocols including TLS 1.3 [Bhargavan et al. 2017], the Signal protocol [Kobeissi et al. 2017], the WireGuard protocol [Lipp et al. 2019], and the AWS Key Management Service [Almeida et al. 2019]. Additionally, F* and F7 were used to verify parts of TLS 1.2 and 1.3 [Bhargavan et al. 2013; Delignat-Lavaud et al. 2017]. All of these previous works prove some form of indistinguishability, but the computational proofs are all game-based and require careful framing and manual effort. Several of these works [Bhargavan et al. 2017; Kobeissi et al. 2017] leverage both CryptoVerif and ProVerif to combine guarantees from the more labor-intensive proofs about cryptographic primitives with guarantees from more automated proofs in the symbolic model. Our approach could likewise be used to connect higher-level proofs with lower-level guarantees from the computational

model. More information on previous work in both the symbolic and computational models can be found in a recent survey on cryptographic protocol verification [Barbosa et al. 2021].

Non-interference. Our specification of indistinguishability is similar in spirit to the probabilistic non-interference property as defined in [O'Neill et al. 2006], which is derived from the original formulation introduced in [Gray III 1992]. Non-interference properties typically specify that secret inputs (denoted as inputs with a high label) have no observable effect that an unprivileged observer could detect (as determined by outputs with a low label). Probabilistic non-interference generalizes non-interference to include programs with probabilistic choices. The output traces in our models are assumed to be the only observable effects an observer could detect from probabilistic processes, so our indistinguishability property is very similar. However, unlike non-interference properties our indistinguishability property has no notion of labels. Instead we assume every the network trace is observable. Additionally, though we parameterize indistinguishability on common public inputs and potentially differing private inputs, which are essentially equivalent to low and high inputs respectively, our formalism is flexible and does not require this specific parameterization. Finally, our approach to ensuring indistinguishability is very different from the typical approach to ensuring non-interference via information flow control. Information flow control ensures that no private information ever flows into public outputs unless it first goes through appropriate declassification. This is often enforced through a type system. The protocols we consider require sending public information that is derived from private information (e.g. the encrypted message). Previous work handles this restriction by modeling encryption as declassification [Abadi 1999; Broberg et al. 2013; Fournet and Rezk 2008; Kozyri et al. 2022; Laud 2003; Li and Zdancewic 2010; Smith and Alpízar 2006; Volpano 2000; Volpano and Smith 2000; Waye et al. 2017]. However, declassification must explicitly take into account implicit information leakage such as message length and order [Backes and Pfitzmann 2002]. This is the same limitation present in the existing symbolic models of cryptographic verification that allows for users to implicitly leak information that violates indistinguishability. See [Kozyri et al. 2022] for a survey of information flow properties and see [Sabelfeld and Sands 2005] for a survey on declassification in information flow. In contrast, we reason about the probabilities of sending any particular message using rules derived from probabilistic Relational Hoare Logic (pRHL) [Barthe et al. 2009]. Our approach allows for finer-grained reasoning about information in the traces of messages (such as the lengths of the messages) without the protocol designer needing to specify and use appropriate declassification schemes that explicitly account for possible information leakage channels. Instead, the user provides a suitable bijection on tapes to show that each determinized run of a program with a private input corresponds to an equivalent determinized run of the program for all other private inputs.

9 CONCLUSION

Though existing methods in both the symbolic and computational models have been developed to verify privacy-preserving protocols, they are either not precise enough to capture information leakage that compromises privacy, require cryptographic expertise, or require heavy manual proof effort. In this paper we provide an alternate approach that adapts elements of both models as well as insights from the distributed systems verification community. Insights from Lipton's reduction allow for modeling concurrent protocols as simpler synchronous ones that are easier to verify. Reasoning about bytestrings using pRHL allows for more precise reasoning than in the symbolic model while perfect encryption assumptions allow for easier verification than in the computational model. As a result, our approach and implementation provide a precise yet accessible method to specify and verify privacy-preserving protocols including TLS ECH and PIR.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful comments and suggestions. We also thank Jack Wampler for discussions about the TLS Encrypted Client Hello extension, as well as Nikhil Swamy, Tahina Ramananandro, and Guido Martínez for their guidance on F* and preliminary feedback on this work.

REFERENCES

Martin Abadi. 1999. Secrecy by typing in security protocols. *Journal of the ACM (JACM)* 46, 5 (1999), 749–786. https://doi.org/10.1145/324133.324266

Nadhem J. Al Fardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy*. 526–540. https://doi.org/10.1109/SP.2013.42

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Matthew Campagna, Ernie Cohen, Benjamin Gregoire, Vitor Pereira, Bernardo Portela, Pierre-Yves Strub, and Serdar Tasiran. 2019. A Machine-Checked Proof of Security for AWS Key Management Service. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 63–78. https://doi.org/10.1145/3319535.3354228

Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J Alex Halderman, Viktor Dukhovni, et al. 2016. DROWN: Breaking TLS using sslv2. In *25th USENIX Security Symposium (USENIX Security 16)*. 689–706.

Michael Backes and Birgit Pfitzmann. 2002. Computational Probabilistic Non-interference. In *Computer Security — ESORICS 2002*, Dieter Gollmann, Günther Karjoth, and Michael Waidner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–23. https://doi.org/10.1007/s10207-004-0039-7

Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *2021 IEEE Symposium on Security and Privacy (SP)*. 777–795. https://doi.org/10.1109/SP40001.2021.00008

Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology – CRYPTO 2011*, Phillip Rogaway (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 71–90. https://doi.org/10.1007/978-3-642-22792-9_5

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal Certification of Code-Based Cryptographic Proofs. *SIGPLAN Not.* 44, 1 (jan 2009), 90–101. https://doi.org/10.1145/1594834.1480894

David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. 2018. A Formal Analysis of 5G Authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1383–1396. https://doi.org/10.1145/3243734.3243846

Donald Beaver and Joan Feigenbaum. 1990. Hiding instances in multioracle queries. In *STACS 90*, Christian Choffrut and Thomas Lengauer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 37–48. https://doi.org/10.1007/3-540-52282-4_30

Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and Ilan Orlov. 2012. Share Conversion and Private Information Retrieval. In *2012 IEEE 27th Conference on Computational Complexity*. 258–268. https://doi.org/10.1109/CCC.2012.23

Yves Bertot and Pierre Castran. 2010. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions* (1st ed.). Springer Publishing Company, Incorporated.

Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2017. A Messy State of the Union: Taming the Composite State Machines of TLS. *Commun. ACM* 60, 2 (jan 2017), 99–107. https://doi.org/10.1145/3023357

Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. 523–542. https://doi.org/10.1109/EuroSP51992.2021.00042

Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 483–502. https://doi.org/10.1109/SP.2017.26

Karthikeyan Bhargavan, Vincent Cheval, and Christopher Wood. 2022. A Symbolic Analysis of Privacy for TLS 1.3 with Encrypted Client Hello. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 365–379. https://doi.org/10.1145/3548606.3559360

Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. 2013. Implementing TLS with Verified Cryptographic Security. In *2013 IEEE Symposium on Security and Privacy*. 445–459. https://doi.org/10.1109/SP.2013.37

Karthikeyan Bhargavan, Antoine Delignat Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre Yves Strub. 2014. Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS. In *2014 IEEE Symposium on Security and*

Privacy. 98–113. https://doi.org/10.1109/SP.2014.14

Bruno Blanchet. 2008. A Computationally Sound Mechanized Prover for Security Protocols. *IEEE Transactions on Dependable and Secure Computing* 5, 4 (2008), 193–207. https://doi.org/10.1109/TDSC.2007.1005

Bruno Blanchet et al. 2016. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends® in Privacy and Security* 1, 1-2 (2016), 1–135. https://doi.org/10.1561/3300000004

Niklas Broberg, Bart van Delft, and David Sands. 2013. Paragon for practical programming with information-flow control. In *Asian Symposium on Programming Languages and Systems*. Springer, 217–232. https://doi.org/10.1007/978-3-319-03542-0_16

Zimo Chai, Amirhossein Ghafari, and Amir Houmansadr. 2019. On the importance of encrypted-SNI (ESNI) to censorship circumvention. In *9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19)*.

Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private Information Retrieval. *J. ACM* 45, 6 (nov 1998), 965–981. https://doi.org/10.1145/293347.293350

Veronique Cortier, David Galindo, and Mathieu Turuani. 2018. A Formal Analysis of the Neuchatel e-Voting Protocol. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 430–442. https://doi.org/10.1109/EuroSP.2018.00037

Véronique Cortier and Cyrille Wiedling. 2017. A formal analysis of the Norwegian E-voting protocol. *Journal of Computer Security* 25, 1 (2017), 21–57. https://doi.org/10.3233/JCS-15777

Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1773–1788. https://doi.org/10.1145/3133956.3134063

Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. 2016. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 470–485. https://doi.org/10.1109/SP.2016.35

Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Beguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. 2017. Implementing and Proving the TLS 1.3 Record Layer. In *2017 IEEE Symposium on Security and Privacy (SP)*. 463–482. https://doi.org/10.1109/SP.2017.58

Giovanni Di Crescenzo, Tal Malkin, and Rafail Ostrovsky. 2000. Single Database Private Information Retrieval Implies Oblivious Transfer. In *Advances in Cryptology — EUROCRYPT 2000*, Bart Preneel (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138. https://doi.org/10.1007/3-540-45539-6_10

Danny Dolev and Andrew Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory* 29, 2 (1983), 198–208. https://doi.org/10.1109/TIT.1983.1056650

Cédric Fournet and Tamara Rezk. 2008. Cryptographically sound implementations for typed information-flow security. *ACM SIGPLAN Notices* 43, 1 (2008), 323–335. https://doi.org/10.1145/1328897.1328478

Sergiu Gatlan. 2019. South Korea is Censoring the Internet by Snooping on SNI Traffic. *Bleeping Computer* (13 Feb 2019). https://www.bleepingcomputer.com/news/security/south-korea-is-censoring-the-internet-by-snooping-on-sni-traffic/

Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. 1998. Protecting data privacy in private information retrieval schemes. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 151–160. https://doi.org/10.1145/276698.276723

Jeremy Gibbons and Ralf Hinze. 2011. Just Do It: Simple Monadic Equational Reasoning. *SIGPLAN Not.* 46, 9 (sep 2011), 2–14. https://doi.org/10.1145/2034574.2034777

Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David Basin. 2020. A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols. In *29th USENIX Security Symposium (USENIX Security 20)*. 1857–1874.

James W Gray III. 1992. Toward a mathematical foundation for information flow security. *Journal of Computer Security* 1, 3-4 (1992), 255–294. https://doi.org/10.3233/JCS-1992-13-405

Niklas Grimm, Kenji Maillard, Cédric Fournet, Cătălin Hriţcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella-Béguelin. 2018. A monadic framework for relational verification: applied to information security, program equivalence, and optimizations. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 130–145. https://doi.org/10.1145/3167090

Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. 2005. Sufficient Conditions for Collision-Resistant Hashing. In *Theory of Cryptography*, Joe Kilian (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 445–456. https://doi.org/10.1007/978-3-540-30576-7_24

Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. 2017. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 435–450. https://doi.org/10.1109/EuroSP.2017.38

Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. 2019. Noise Explorer: Fully automated modeling and verification for arbitrary Noise protocols. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE,

356–370. https://doi.org/10.1109/EuroSP.2019.00034

Elisavet Kozyri, Stephen Chong, and Andrew C. Myers. 2022. Expressing Information Flow Properties. *Foundations and Trends® in Privacy and Security* 3, 1 (2022), 1–102. https://doi.org/10.1561/3300000008

Peeter Laud. 2003. Handling encryption in an analysis for secure information flow. In *European Symposium on Programming*. Springer, 159–173. https://doi.org/10.1007/3-540-36575-3_12

Peng Li and Steve Zdancewic. 2010. Arrows for secure information flow. *Theoretical computer science* 411, 19 (2010), 1974–1994. https://doi.org/10.1016/j.tcs.2010.01.025

Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. 2019. A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 231–246. https://doi.org/10.1109/EuroSP.2019.00026

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (dec 1975), 717–721. https://doi.org/10.1145/361227.361234

Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN prover for the symbolic analysis of security protocols. In *International conference on computer aided verification*. Springer, 696–701. https://doi.org/10.1007/978-3-642-39799-8_48

Bodo Möller, Thai Duong, and Krzysztof Kotowicz. 2014. This POODLE bites: exploiting the SSL 3.0 fallback. *Security Advisory* 21 (2014), 34–58.

K.R. O'Neill, M.R. Clarkson, and S. Chong. 2006. Information-flow security for interactive programs. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. 12 pp.–201. https://doi.org/10.1109/CSFW.2006.16

E. Rescorla, K. Oku, N. Sullivan, and C.A. Wood. 2022. *TLS Encrypted Client Hello*. Internet-Draft draft-ietf-tls-esni-15. IETF Secretariat. https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni-15

Juliano Rizzo and Thai Duong. 2012. The CRIME Attack. In *EKOparty Security Conference*.

A. Sabelfeld and D. Sands. 2005. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. 255–269. https://doi.org/10.1109/CSFW.2005.15

B. Schwartz, M. Bishop, and E. Nygren. 2021. *Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs)*. Internet-Draft draft-ietf-dnsop-svcb-https-08. IETF Secretariat. https://datatracker.ietf.org/doc/html/draft-ietf-dnsop-svcb-https-08

Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (nov 1979), 612–613. https://doi.org/10.1145/359168.359176

Geoffrey Smith and Rafael Alpízar. 2006. Secure information flow with random assignment and encryption. In *Proceedings of the fourth ACM workshop on Formal methods in security*. 33–44. https://doi.org/10.1145/1180337.1180341

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. 2016. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 256–270. https://doi.org/10.1145/2837614.2837655

The Coq Development Team. 2023. The Coq Proof Assistant. https://doi.org/10.5281/zenodo.1003420

U.S. CIO Council. 2016. HTTPS. https://https.cio.gov/. [Online; accessed 31-March-2023].

Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend Synchrony: Synchronous Verification of Asynchronous Distributed Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 59 (jan 2019), 30 pages. https://doi.org/10.1145/3290372

D. Volpano. 2000. Secure introduction of one-way functions. In *Proceedings 13th IEEE Computer Security Foundations Workshop. CSFW-13*. 246–254. https://doi.org/10.1109/CSFW.2000.856941

Dennis Volpano and Geoffrey Smith. 2000. Verifying secrets and relative secrecy. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 268–276. https://doi.org/10.1145/325694.325729

W3C Technical Architecture Group. 2021. Web HTTPS. https://www.w3.org/2001/tag/doc/web-https. [Online; accessed 31-March-2023].

Lucas Waye, Pablo Buiras, Owen Arden, Alejandro Russo, and Stephen Chong. 2017. Cryptographically secure information flow control on key-value stores. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1893–1907. https://doi.org/10.1145/3133956.3134036

Received 2023-04-14; accepted 2023-08-27