Them's Just Semantics: Towards Reasoning About Adversarial Protocol Interaction

KIRBY LINVILL, University of Colorado Boulder, USA

We consider the problem of verifying cryptographic protocol equivalence with respect to a Dolev-Yao adversary. Verifying protocol equivalence is the key to common cryptographic proof techniques, such as simulation-based proofs. Unfortunately, existing proof approaches to verify protocol equivalence either require significant manual proof effort or impose restrictions that make them unsuitable for these cryptographic proof techniques. In this paper, we lay the foundations for reducing verification effort required to prove protocol equivalence in general. Specifically, we define adversarial equivalence, a general definition of protocol equivalence from the perspective of an active symbolic adversary. We further provide a collecting semantics for protocols that captures the information an adversary can learn from interacting with a protocol. These definitions serve as the foundation upon which sound, over-approximate abstractions can be defined to simplify and automate the verification of adversarial equivalence.

1 Introduction

This paper focuses on the problem of formally defining cryptographic protocol equivalence under interaction by an active, Dolev-Yao [6] style, adversary. The formal definition, along with concrete and collecting semantics for these protocols, is a step towards developing sound abstractions and automated techniques to verify this equivalence.

Cryptographic protocols, protocols that aim to provide security properties using cryptographic primitives such as encryption and digital signatures, are notoriously difficult to design. For example, the classic Needham-Schroeder public-key protocol [11] aims to establish a mutually authenticated connection between two parties over a network using asymmetric encryption and digital signatures. Unfortunately, the protocol fails to provide mutual authentication when an adversary can interact with the protocol [8]; a fact that was only discovered 17 years later. It is therefore critical that cryptographic protocols be *formally specified* and *verified* to ensure they meet both their correctness and security requirements *before* the protocols are widely used.

Cryptographic protocol specifications are often given as equivalence with another protocol or arguments. For example, an encryption scheme is semantically secure if, given a ciphertexts and two plaintexts of the same length, an attacker is equally likely to believe each plaintext was encrypted to create the ciphertext. Likewise, a cryptographic protocol may be proven secure by showing it behaves equivalently, from the perspective of an adversary, to an ideal specification in which security is assumed and a simulator [7]. Proving a cryptographic protocol secure then reduces to proving equivalence between two protocols from the perspective of an adversary that interacts with the protocols.

To ensure cryptographic protocols are secure against realistic attacks, it is important to model the strongest realistic adversary when reasoning about protocol equivalence. The strongest such adversary for protocols that communicate over a network is a man-in-the-middle adversary that can observe, intercept, and modify all network traffic. An adversary also has computational capabilities that allow it to derive new information from the information it observes. In other words, the strongest realistic adversary is an active computational network adversary.

However, verifying a cryptographic protocol in the presence of such an adversary is a challenging task as it involves reasoning about probabilities, feasible information an attacker could derive from interacting with the protocol, and the arithmetic used to implement cryptographic primitives. To simplify verification, cryptographic primitives can be modeled algebraically using an equational

Author's Contact Information: Kirby Linvill, University of Colorado Boulder, Boulder, Colorado, USA, kirby.linvill@colorado.edu.

theory. For example, the encryption of a message m using key k can be modeled algebraically using the term $enc(m, k)$, while the (symmetric) decryption of such a term using the same key can be captured by the equation $dec(enc(m, k), k) = m$. An adversary is then modeled with an explicit knowledge containing the set of terms it knows, the ability to observe and control network communication, and the ability to derive new terms from its knowledge using the equational theory. This approach is known as a Dolev-Yao [6], or Symbolic, model of cryptography.

Unfortunately, existing symbolic proof approaches are either insufficient for proving the equivalence properties needed for simulation-style proofs, or require significant manual proof efforts. Specifically, current approaches to proving symbolic equivalence of two protocols require the protocols to be identical modulo the values they send [2–4, 12]. This approach is typically only used to show two processes are equivalent modulo different inputs, hence the name *diff-equivalence*. Unfortunately, diff-equivalence's requirements are too strong to prove equivalence in simulation style specifications, which necessarily include extra transitions related to private communication between the ideal specification and simulator. Proving a more general equivalence between two protocols currently relies on explicitly proving bisimilarity, which involves a significant proof effort.

An automated yet general approach to verifying protocol equivalence with respect to an active symbolic adversary is needed. In this paper, I take the first step towards such an approach by defining adversarial equivalence, a general definition of equivalence for protocols from the perspective of an active symbolic adversary. The definition is given with respect to a collecting semantics that captures the information an adversary can learn from interacting with a protocol. These definitions serve as a foundation upon which sound, over-approximate abstractions can be defined. Such abstractions may then be used to simplify and automate the verification of adversarial equivalence.

Contributions. Specifically, this paper makes the following contributions:

- An actor-style syntax (Section 3) and concrete semantics (Section 4) for protocols interacting with an active symbolic adversary.
- A collecting semantics for protocols (Section 5.2) that captures adversarial interaction properties. These properties describe how the adversary interacted with the protocol, the observed effects, and the attacker's knowledge.
- A definition of adversarial equivalence (Section 5.3) based on the collecting semantics.

2 Overview

In this section we motivate the key modeling choices behind the definition of adversarial equivalence.

Consider the simple protocols shown in Figure 1. Protocol `send_1_2` takes two messages as input and an output channel `out`, generates a fresh key k that is not known to the adversary, and then proceeds to send out the encrypted first message $enc(m_1, k)$, followed by the encrypted second message $enc(m_2, k)$, followed by the first message m_1. When the red line is included, the protocol also sends out the key k used to encrypt the messages. Protocol `send_2_1` is identical except that it sends out the encrypted second message first, followed by the encrypted first message. An adversary can distinguish between the two protocols when the red line (`send` k `out`) is included, but not when the red line is excluded.

2.1 Distinguishing Traces

An adversary can distinguish between two protocols if it can observe behavior or learn facts from one protocol that it could not have plausibly observed or learned from the other protocol. The focus for this paper is on an adversary that can observe messages sent over network channels,

```
protocol send_1_2 (m_1, m_2, out) := {
    k := fresh();
    send enc(m_1,k) out;
    send enc(m_2,k) out;
    send m_1 out;
    send k out
}
```

```
protocol send_2_1 (m_1, m_2, out) := {
    k := fresh();
    send enc(m_2,k) out;
    send enc(m_1,k) out;
    send m_1 out;
    send k out
}
```

Fig. 1. Two protocols, specified in the DYAct language from Section 3, that can be distinguished by an adversary when the red line is included, but behave equivalently when it is removed. They differ only in the order in which they send out their encrypted messages.

but the model can be generalized to other forms of information leakage. When a message is sent over the network, the adversary learns which channel the message was sent on and the contents of the message itself. From its current knowledge, the adversary can also derive new facts using its computational capabilities. Given its interactions, observations, and derivations, the adversary then only distinguishes between two protocols if the interactions, observations, and derivations *must* have come interacting with one protocol, and not the other.

Simply observing different encrypted messages from protocols, such as the messages $enc(m_1, k)$ from protocol send_1_2 and $enc(m_2, k)$ from protocol send_2_1, is not sufficient to distinguish between them. This is because a secure symmetric encryption schemes must provide a property known as semantic security. An encryption scheme is semantically secure if no adversary can determine which of two possible messages, of a given length, were used to produce a ciphertext, based only on knowing the ciphertext and messages. In a symbolic model of cryptography, this semantic security property is captured by defining an indistinguishability relation on terms ($\sim$, defined in Figure 2c) in which two encrypted terms are always statically indistinguishable, regardless of their contents. Other statically indistinguishable terms include identical terms and pairs of indistinguishable terms. Note that encrypted terms are only *statically* indistinguishable since the adversary may deduce more information about the encrypted messages, including their contents, by trying to decrypt the messages or supply the terms to a protocol.

Differentiating between protocol send_1_2 and protocol send_2_1 hinges on the distinguishable terms the adversary can derive after decrypting the first (or second) encrypted message using the learned key k. Therefore, a first attempt to capture relevant information to distinguish traces might be to capture the set of terms an adversary learns during a protocol execution or can derive from its knowledge. However, both protocols leak the same set of terms by the time the key is leaked, so any term the adversary can derive from one protocol it can also derive from the other.

An adversary's ability to differentiate between the two protocols relies not just on *if* it learns a fact, but on *how* it learns that fact. Specifically, the adversary can differentiate between the two protocols by decrypting the first encrypted message (the 0^{th} term learned) with the leaked key (the 3^{rd} term learned), and comparing it to the leaked message (the 2^{nd} term learned). The key here is the action the adversary takes to derive a new term (decryption, and an equality test) and the index of the terms it uses. We therefore represent these same interactions using the labels $derive(dec, [0, 3])$ and $derive(eq, [2, 4])$ to indicate that the adversary used its decryption capability on the 0^{th} and 3^{rd} learned terms, followed by its equality comparison capability. Notably these labels capture exactly how the adversary derived new terms, in a knowledge-agnostic manner that applies to both protocols. After the sequence of derivations, the attacker will have derived m_1 and

$$k \in Keys \qquad d \in Data$$

$$\begin{array}{lcl} t & \in & Terms & ::= & k \mid d \mid (t,t) \mid enc(t,k) \mid true \mid false \\ f & \in & TermFunctions & ::= & dec(t,k) \mid fst(t) \mid snd(t) \mid eq(t,t) \end{array}$$

(a) Syntax for terms and term functions for a minimal symmetric encryption model.

$$\begin{array}{lcl} dec(enc(t,k_e),k_d) & := & \text{if } k_d = k_d \text{ then } t \text{ else } \bot \\ fst((t_1,t_2)) & := & t_1 \\ snd((t_1,t_2)) & := & t_2 \\ eq(t_1,t_2) & := & t_1 = t_2 \end{array}$$

(b) Denotation of term functions for minimal symmetric encryption model.

$$\frac{}{t \sim t}\ \textsc{IndRefl} \qquad \frac{t_1 \sim t_1' \qquad t_2 \sim t_2'}{(t_1,t_2) \sim (t_1',t_2')}\ \textsc{IndPair} \qquad \frac{}{enc(t,k) \sim enc(t',k')}\ \textsc{IndEnc}$$

(c) Static term indistinguishability relation. Defines when an adversary cannot statically distinguish between terms from across executions.

Fig. 2. Minimal symbolic model of symmetric encryption.

true for the left protocol, and m_2 and $false$ for the right protocol. Even if the adversary does not originally know the first and second messages, it only needs to check the results of the equality test to distinguish between the two protocols. Specifically, $true \nsim false$.

2.2 Adversary Capabilities

In addition to being able to derive new facts from their current knowledge, an active network adversary is able to control and observe network traffic. An adversary's interactions with a protocol then consist of sending messages to the protocol over network channels, and observing messages sent by the protocol over network channels. To simplify modeling this control, network communication is modeled as protocol sends, that simply add sent messages to the adversary's knowledge, and adversary sends, that add terms from the adversary's knowledge to the network. Since the network is modeled using channels, adversary sends can be modeled using the label $send(c,i)$ to indicate the adversary is sending the i^{th} term in its knowledge over channel c.

As previously mentioned, the adversary can derive new facts from its current knowledge using its computational capabilities. The adversary is parametric on these capabilities. For this example, it suffices to consider a minimal symbolic model of symmetric encryption (Figure 2) where terms consist of keys, abstract data terms, pairs, encrypted terms, and boolean values. These terms can be manipulated using functions for decryption, pair projection, and equality testing (Figure 2b). Decryption only succeeds if the correct key is provided, otherwise it returns $\bot$ to denote that decryption failed. Equality testing is necessary since any adversary should always be able to check if two terms it knows are the same. We refer to the constructors and term functions an adversary can use as its computational capabilities, denoted $\mathcal{A}$. Deriving a new term from $\mathcal{A}$'s knowledge is then modeled using the label $derive(f,\bar{i})$ where f is a term derivation function such that $f \in \mathcal{A}$ and $\bar{i}$ is a list of indices into the knowledge to apply f on.

2.3 System Model

The complete system model then includes four kinds of transitions modeling both protocol and adversary actions. The adversaries actions are:

- sending known terms over network channels, with the label $send(c, i)$
- deriving new terms from its knowledge, with the label $derive(f, \bar{i})$

while the protocol's actions are broken up into:

- internal actions that produce no visible effects, e.g. `k := fresh()`, with the label τ
- external actions that model sending a message over the adversary-controlled network by using the label c, and adding the sent term directly to the adversary's knowledge

2.4 Checking Indistinguishability

Determining that an adversary *could never* distinguish between two protocols then requires considering all possible adversarial interactions, observations, and derivations. Two protocols are considered adversarially equivalent, with respect to adversary $\mathcal{A}$, if any possible sequence of interactions with and observations of one protocol can be matched by a sequence of interactions and observations of the other protocol, such that any term the adversary can derive could plausibly have been derived from the other protocol (i.e. is *indistinguishable* with a term from the other protocol). Specifically, the sequence of interactions and observations is captured by the non-internal transition labels, denoted by α. The sequence of observed and derived terms by the adversary is then tracked using the adversary's knowledge κ. Two protocols π_l and π_r, with system state $(\alpha, \kappa_l, \_)$ and $(\alpha, \kappa_r, \_)$ respectively, are then adversarially equivalent if for every valid sequence of interactions and observations α and knowledge κ_l derived for one protocol, there exists a knowledge κ_r derived for the other protocol such that the two knowledges are statically equivalent (denoted $\kappa_l \approx \kappa_r$), and vice versa. Two knowledges are statically equivalent if they are pairwise indistinguishable, i.e. if for every index i, $\kappa_l[i] \sim \kappa_r[i]$. Note that while the knowledges must be statically equivalent, not identical, while the interactions and observations must be identical. These states are sufficient to reason about adversarial interaction properties, introduced in Section 5.1 and are therefore the basis for the collecting semantics defined in Section 5.2.

The example `send_1_2` and `send_2_1` protocols are adversarially equivalent when the red line is excluded, since the collected properties cannot contain distinguishable knowledges without being able to decrypt the encrypted messages. Verification of this property is left out of scope of this paper.

3 DYAct: A Language for Protocol Specifications

In this section we introduce the syntax for `DYAct`, a simple language for specifying protocols that can interact with an adversarially controlled network. The syntax for `DYAct` is given in Figure 3. It is parametric on the set of terms $\mathcal{T}$ and functions $\mathcal{F}$ on those terms, defined by a model of cryptographic primitives. At the top level, a program consists of a set of protocol definitions (`protocol` x $(\overline{x})$:= $\{p\}$). Each protocol definition provides a name for the protocol, a list of parameters that will be bound when the protocol is instantiated, and a body that may consist of variable assignments ($x := s$), handler definitions (`handler` x `on` s `when` e `do` s), and subprotocol inclusions (`run x` $(\overline{e})$). Protocols may contain other subprotocols to allow for easy composition. This naturally allows for defining protocols with multiple participants, where each participant is modeled as a separate subprotocol.

The rest of the syntax is fairly standard with one notable exception: the type of values is parametric on the type of terms. This allows the language to be instantiated with different algebraic terms, such as $enc(m, k)$ for symmetric encryption or $sign(m, sk)$ for digital signatures. The language

$$
\begin{array}{llll}
d & \in & \mathit{Definitions} & ::= \ \texttt{protocol}\ x\ (\overline{x}) := \{p\} \mid d; d \\
p & \in & \mathit{Bodies} & ::= \ s \mid p; p \mid \texttt{handler}\ x\ \texttt{on}\ s\ \texttt{when}\ e\ \texttt{do}\ s \mid \texttt{run}\ x\ (\overline{e}) \\
s & \in & \mathit{Statements} & ::= \ x := s \mid s; s \mid \texttt{receive}\ e \mid \texttt{send}\ e\ e \mid \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{fresh} \mid e \\
e & \in & \mathit{Expressions} & ::= \ x \mid v \mid f(\overline{e}) \\
v & \in & \mathit{Values} & ::= \ \mathit{true} \mid \mathit{false} \mid t \\
f & \in & \mathit{TermFunctions} & ::= \ \mathcal{F} \\
t & \in & \mathit{Terms} & ::= \ \mathcal{T}
\end{array}
$$

Fig. 3. DYAct $(\mathcal{T}, \mathcal{F})$ Syntax. The language is parametric on a set of terms $\mathcal{T}$ and functions $\mathcal{F}$ that together represent cryptographic operations.

requires the boolean values *true* and *false* which are used to evaluate control flow in the case of handler conditions and `if` statements.

4 Concrete Semantics for DYAct Protocols and Adversaries

The semantics of the language is given as separate operational semantics for protocol definitions (Section 4.1), bodies (Section 4.2), statements (Section 4.3), and expressions. The evaluation of a protocol conceptually occurs in two different phases: initialization and execution. First, during initialization, protocol definitions and bodies are evaluated to produce the initial state for the system. The protocol definition and body semantics are only relevant for initialization. Second, during execution, the system non-deterministically executes either a protocol handler or an adversary action. Protocol handlers are atomic, and either fully execute or do not execute at all. The system semantics, which include the combined protocol and adversary semantics, are given denotationally as a labeled transition system in Figure 7, where the labels indicate the interaction or observable effect from the transition. Non-termination is only captured at the system level since the protocol semantics are all given in a big-step style to a new state. We further describe each of the semantics below.

4.1 Protocol Definition Semantics (Figure 4)

$$\boxed{\Gamma, d \Downarrow \Gamma'}$$

$$\Gamma \in \mathit{Variables} \rightharpoonup \mathit{Variables}^{*} \times \mathit{ProtocolBodies}$$

$$\frac{}{\Gamma, \texttt{protocol}\ x\ (\overline{x'}) := \{p\} \Downarrow \Gamma[x \mapsto (\overline{x'}, p)]}\ \textsc{Def} \qquad \frac{\Gamma, d \Downarrow \Gamma' \qquad \Gamma', d' \Downarrow \Gamma''}{\Gamma, d; d' \Downarrow \Gamma''}\ \textsc{DefSeq}$$

Fig. 4. Protocol Definition Semantics

Protocol definitions enable protocol reuse and composition by allowing protocols to be instantiated as sub-protocols within other protocols. For example, a one-way confidential communication protocol may be defined as a protocol, then instantiated twice as sub-protocols to allow for bidirectional confidential communication.

Protocol definitions simply associate a protocol body p with a name x and list of parameters $\overline{x'}$. The Def rule then stores the argument names and body along with the protocol's name in the top-level environment Γ. Γ can be used to look up defined protocols for invocation as a sub-protocol.

The protocol body is not interpreted until the protocol is instantiated with a sequence of arguments, ensuring the body is executed with the correct arguments.

4.2 Protocol Body Semantics (Figure 5)

$$\boxed{\Gamma, C \vdash (\sigma, p) \Downarrow \delta, \Sigma, \pi} \qquad \boxed{\Gamma, C \vdash (\sigma, p) \Downarrow (\alpha, \kappa), (\sigma', \overline{\Sigma_\mu}), (H, \overline{\pi_\mu})}$$

$$\mathit{LocalState} = \mathit{Variables} \rightharpoonup \mathit{Values} \qquad \mathit{ProtocolState} = \mathit{LocalState} \times \mathit{ProtocolState}^*$$

$$\mathit{Handlers} = \mathit{Statements} \times \mathit{Expressions} \times \mathit{Statements} \qquad \mathit{Protocols} = \mathcal{P}(\mathit{Handlers}) \times \mathit{Protocols}^*$$

$$\Gamma \in \mathit{Variables} \rightharpoonup \mathit{Variables}^* \times \mathit{ProtocolBodies} \qquad C \subseteq \mathit{Values} \qquad \delta \in \mathit{Effects}^* \times \mathit{Values}^*$$

$$\sigma \in \mathit{LocalState} \qquad \Sigma \in \mathit{ProtocolState} \qquad H \subseteq \mathit{Handlers} \qquad \pi \in \mathit{Protocols}$$

$$\frac{C \vdash \langle \emptyset, \sigma, \emptyset, s \rangle \Downarrow \langle \emptyset, \sigma', \delta, v \rangle}{\Gamma, C \vdash (\sigma, s) \Downarrow \delta, (\sigma', \emptyset), (\emptyset, \emptyset)} \textsc{ ProtoStmt}$$

$$\frac{}{\Gamma, C \vdash (\sigma, \texttt{handler}\ x\ \texttt{on}\ s\ \texttt{when}\ e\ \texttt{do}\ s') \Downarrow (\sigma, \emptyset), (\{(s, e, s')\}, \emptyset)} \textsc{ Handler}$$

$$\frac{\langle \sigma, \overline{e} \rangle \Downarrow \overline{v} \qquad \Gamma[x] = (\overline{x'}, p_\mu) \qquad \Gamma, C \vdash (\emptyset[\overline{x'} \mapsto \overline{v}], p_\mu) \Downarrow \delta_\mu, \Sigma_\mu, \pi_\mu}{\Gamma, C \vdash (\sigma,\ \texttt{run}\ x\ (\overline{e})) \Downarrow \delta_\mu, (\sigma, [\Sigma_\mu]), (\emptyset, [\pi_\mu])} \textsc{ Sub-protocol}$$

$$\frac{\Gamma \vdash, C(\sigma, p) \Downarrow (\alpha', \kappa'), (\sigma', \overline{\Sigma_\mu}'), (H', \overline{\pi_\mu}') \qquad \Gamma \vdash, C(\sigma', p') \Downarrow (\alpha'', \kappa''), (\sigma'', \overline{\Sigma_\mu}''), (H'', \overline{\pi_\mu}'')}{\Gamma \vdash, C(\sigma, p; p') \Downarrow (\alpha' +\!\!+\ \alpha'', \kappa' +\!\!+\ \kappa''), (\sigma'', \overline{\Sigma_\mu}' +\!\!+\ \overline{\Sigma_\mu}''), (H' \cup H'', \overline{\pi_\mu}' +\!\!+\ \overline{\pi_\mu}'')} \textsc{ ProtoSeq}$$

Fig. 5. Protocol Body Semantics

When a protocol is instantiated with an appropriate number of arguments, the protocol body is then executed to generate a tuple of the form $((\alpha, \kappa), (\sigma, \overline{\Sigma_\mu}), (H, \overline{\pi_\mu}))$ where σ is the internal state for this protocol, $\overline{\Sigma_\mu}$ is the list of states for each sub-protocol, H is the set of handlers defined in this protocol, $\overline{\pi_\mu}$ is the list of sub-protocol handlers, and α and κ are the sequences of observable effects and values emitted to the adversary that are produced by the statement semantics (Section 4.3). The internal states are simply a partial map from variables to values. The handlers are represented as a tuple of the trigger statement s, enabling condition e, and action statement s'. These handlers are only interpreted later during system execution. We alternatively refer to the tuple $(\sigma, \overline{\Sigma_\mu})$ as the protocol state Σ, the tuple $(H, \overline{\pi_\mu})$ as the protocol π, and the tuple (α, κ) as the effects δ. The body semantics are defined with respect to a definition environment Γ, which maps protocol variables to their parameter lists and bodies, and a set of external channels C that is used by the statement semantics introduced in Section 4.3.

Both the protocol state Σ and protocol π are recursively defined, containing both the local state/handlers defined in the protocol alongside those of its sub-protocols. Conceptually, protocols and sub-protocols operate on their own respective internal states, providing encapsulation and disambiguation in the case of clashing variable names between sub-protocols. Values can be passed to sub-protocols via parameters at instantiation. The body is executed only once for each protocol instance: when the protocol is instantiated.

Statements in the body (rule ProtoStmt), along with the arguments to the protocol (rule Sub-protocol), are used to generate the initial protocol state. Handler definitions are preserved as-is in the protocol value (rule Handler) for later interpretation. Specifically, a handler's trigger statement s, enabling expression e, and action statement s' are packaged into the tuple (s, e, s'). Sub-protocol values are defined by recursive application of the protocol body semantics (rule Sub-protocol). The arguments and body for the sub-protocol are found in the top-level environment Γ. Notably, the initial state for the sub-protocols consists only of the arguments to the sub-protocol. Variables cannot be implicitly inherited from a parent's scope.

Note that the `run` command is used to compose different protocols together. Protocols parameterized by channels can then have their channels connected by instantiating the protocols with the same value for their connected channels. For example, connecting the *io* channel of $P(io, net)$ with the *net* channel of $Q(io, net)$, can be done by the protocol body $c := \texttt{fresh};\ \texttt{run}\ P(c, net);\ \texttt{run}\ Q(io, c)$.

Sequential composition of protocol bodies is carried out by propagating the state changes (σ to σ''), and by separately computing the local handlers H, sub-protocol states $\overline{\Sigma_\mu}$, and sub-protocols $\overline{\pi_\mu}$ before unioning the handlers and concatenating the lists together. Intuitively, handlers are represented as sets because they all operate on the same state, so duplicate handlers will always offer the same possible transitions. Duplicates can therefore be eliminated without a loss of generality. In contrast, sub-protocols all have their own independent pieces of state. Duplicate sub-protocol values could evolve to different states, thus presenting different possible transitions. Since we cannot soundly eliminate duplicate sub-protocols, we track lists of sub-protocols in order to preserve duplicates. Lists also allow for keeping track of sub-protocol state and handlers by using the same index.

4.3 Statement Semantics (Figure 6)

Statements may be executed either as part of protocol initialization (Section 4.2) or in a handler during system execution (Section 4.4). The statement semantics are given as the big-step judgment $C \vdash \langle N, \sigma, \delta, s \rangle \Downarrow \langle N', \sigma', \delta', v \rangle$. This judgement represents a protocol statement s executing with protocol state σ, shared network state N, effects δ, and external channels C. Sequential composition (rule Seq) and variable assignment (rule Assign) are standard for effectful statements. The main interesting semantics are those that concern interacting with the network and generating fresh values.

Sending a message over an external channel (rule SendExternal) does not change the shared network state N, but rather adds the sent message and channel to the effects δ, which later get added to the adversary's knowledge when constructing the system semantics (Section 4.4). The effects are then visible to an external adversary which may opt to add the observed sent message to the network, or may opt to ignore it, in effect intercepting or dropping the message. Sending a message over an internal (i.e. not external) channel instead directly adds the message to the shared network, modeling communication that is hidden and uncontrolled by the adversary (rule SendInternal). The network state consists of a map from channels to bags of messages. Receiving a message over a channel is then non-deterministic, as any message in the channel may be received (rule Receive). Note that receiving a message is only possible if there is at least one message in the channel; effectively modeling statements that block until enough messages are available to execute all receive statements in the handler.

Protocols should be able to generate values that are unknown to the adversary in order to model operations like random sampling that are common in cryptographic protocols. In DYAct this modeled by the Fresh rule, that describes the generation of fresh values. It does not prescribe a

$$\boxed{C \vdash \langle N, \sigma, \delta, s \rangle \Downarrow \langle N', \sigma', \delta', v \rangle}$$

$$Networks = Values \mapsto \mathcal{B}(Values) \qquad ProtoState = Variables \rightharpoonup Values$$

$$C \subseteq Values \qquad N \in Networks \qquad \sigma \in ProtoState \qquad \delta \in Effects^* \times Values^*$$

$$\frac{\langle \sigma, e \rangle \Downarrow v \qquad \langle \sigma, e' \rangle \Downarrow c \qquad c \in C}{C \vdash \langle N, \sigma, (\alpha, \kappa), \texttt{send}\ e\ e' \rangle \Downarrow \langle N, \sigma, (\alpha \mathbin{++} [c], \kappa \mathbin{++} [v]), \epsilon \rangle}\ \textsc{SendExternal}$$

$$\frac{\langle \sigma, e \rangle \Downarrow v \qquad \langle \sigma, e' \rangle \Downarrow c \qquad c \notin C}{C \vdash \langle N, \sigma, \delta, \texttt{send}\ e\ e' \rangle \Downarrow \langle N[c \mapsto N[c] + \{v\}], \sigma, \delta, \epsilon \rangle}\ \textsc{SendInternal}$$

$$\frac{\langle \sigma, e \rangle \Downarrow c \qquad v \in N[c]}{C \vdash \langle N, \sigma, \delta, \texttt{receive}\ e \rangle \Downarrow \langle N[c \mapsto N[c] - \{v\}], \sigma, \delta, v \rangle}\ \textsc{Receive}$$

$$\frac{fresh\ v}{C \vdash \langle N, \sigma, \delta, \texttt{fresh} \rangle \Downarrow \langle N, \sigma, \delta, v \rangle}\ \textsc{Fresh}$$

$$\frac{C \vdash \langle N, \sigma, \delta, s \rangle \Downarrow \langle N', \sigma', \delta', v \rangle \qquad C \vdash \langle N', \sigma', \delta', s' \rangle \Downarrow \langle N'', \sigma'', \delta'', v' \rangle}{C \vdash \langle N, \sigma, \delta, s \rangle \Downarrow \langle N'', \sigma'', \delta'', v' \rangle}\ \textsc{Seq}$$

$$\frac{C \vdash \langle N, \sigma, \delta, s \rangle \Downarrow \langle N', \sigma', \delta', v \rangle}{C \vdash \langle N, \sigma, \delta, x := s \rangle \Downarrow \langle N', \sigma'[x \mapsto v], \delta', \epsilon \rangle}\ \textsc{Assign}$$

Fig. 6. Statement Semantics

method for generating fresh values, but instead just requires that the generated values be fresh (i.e. unused anywhere else in the protocols and adversary).

4.4 Adversary-Controlled System Semantics (Figure 7)

With a semantics for protocol initialization (Sections 4.1 and 4.2) and protocol statement execution (Section 4.3), we can now define the full system semantics for protocol execution under an external adversary. A semantics for the execution of a protocol π under an external adversary is given by the judgment $C, \mathcal{A}, \pi \vdash \langle \alpha, \kappa, N, \Sigma \rangle \longrightarrow \langle \alpha', \kappa', N', \Sigma' \rangle$. Importantly, the system semantics are defined with respect to a protocol value π, not a protocol definition d.

The system semantics only model system execution after initialization. The initial system state $\langle \alpha_0, \kappa_0, N_0, \Sigma_0 \rangle$ can be constructed from the definition and body semantics, given a main protocol name x_0, arguments $\overline{v_0}$, and external channels C. Specifically, the definition environment Γ is first constructed from the protocol definitions d using the protocol definition semantics (Section 4.1). Then, the main protocol body p (where $\Gamma[x_0] = (\overline{x}, p)$) is executed with the initial environment $[\overline{x} \mapsto \overline{v_0}]$ using the protocol body semantics (Section 4.2) to get the initial observed effects α_0, initial knowledge κ_0, and initial protocol state Σ_0. For simplicity, we assume the initial network N_0 is empty.

$$\boxed{C, \mathcal{A}, \pi \vdash \langle \alpha, \kappa, N, \Sigma \rangle \longrightarrow \langle \alpha', \kappa,' N', \Sigma' \rangle}$$

$$\boxed{C, \mathcal{A}, (H, \overline{\pi_\mu}) \vdash \langle \alpha, \kappa, N, (\sigma, \overline{\Sigma_\mu}) \rangle \longrightarrow \langle \alpha', \kappa', N', (\sigma', \overline{\Sigma_\mu}')) \rangle}$$

$$Networks = Values \rightharpoonup \mathcal{B}(Values) \qquad ProtoState = Variables \rightharpoonup Values$$

$$Handlers = Statements \times Predicates \times Statements$$

$$Protocols = ProtoState \times \mathcal{P}(Handlers) \times \mathcal{B}(Protocols)$$

$$C \subseteq Values \qquad \mathcal{A} \subseteq Values^* \rightharpoonup Values \qquad \alpha \in Effects^*$$

$$\kappa \in Values^* \qquad N \in Networks \qquad \pi \in Protocols$$

$$\frac{\begin{array}{c}(s, e, s') \in H \qquad C \vdash \langle N, \sigma, [\,], s \rangle \Downarrow \langle N', \sigma', \delta', v \rangle \\ \langle \sigma', e \rangle \Downarrow \top \qquad C \vdash \langle N', \sigma', \delta', s' \rangle \Downarrow \langle N'', \sigma'', \delta'', v' \rangle \qquad \delta'' = (\alpha''_\delta, \kappa''_\delta)\end{array}}{C, \mathcal{A}, (H, \overline{\pi_\mu}) \vdash \langle \alpha, \kappa, N, (\sigma, \overline{\Sigma_\mu}) \rangle \longrightarrow \langle \alpha \mathbin{+\!\!+} \alpha''_\delta, \kappa \mathbin{+\!\!+} \kappa''_\delta, N'', (\sigma'', \overline{\Sigma_\mu}) \rangle} \textsc{ExecHandler}$$

$$\frac{i \in \mathrm{dom}(\overline{\pi_\mu}) \qquad C, \mathcal{A}, \overline{\pi_\mu}[i] \vdash \langle \alpha, \kappa, N, \overline{\Sigma_\mu}[i] \rangle \longrightarrow \langle \alpha', \kappa', N', \sigma'_\mu \rangle}{C, \mathcal{A}, (H, \overline{\pi_\mu}) \vdash \langle \alpha, \kappa, N, (\sigma, \overline{\Sigma_\mu}) \rangle \longrightarrow \langle \alpha', \kappa', N', (\sigma, \overline{\Sigma_\mu}[i \mapsto \sigma'_\mu]) \rangle} \textsc{ExecSubprocess}$$

$$\frac{f \in \mathcal{A} \qquad i \in \bar{i} \implies i \in \mathrm{dom}(\kappa)}{C, \mathcal{A}, \pi \vdash \langle \alpha, \kappa, N, \Sigma \rangle \longrightarrow \langle \alpha \mathbin{+\!\!+} [derive(f, \bar{i})],\ \kappa \mathbin{+\!\!+} [f(\kappa\,[\,\bar{i}\,])], N, \Sigma \rangle} \textsc{AdversaryDerive}$$

$$\frac{c \in C \qquad i \in \mathrm{dom}(\kappa)}{C, \mathcal{A}, \pi \vdash \langle \alpha, \kappa, N, \Sigma \rangle \longrightarrow \langle \alpha \mathbin{+\!\!+} [send(c, i)],\ \kappa, N[c \mapsto N[c] + \{\kappa[i]\}], \Sigma \rangle} \textsc{AdversarySend}$$

Fig. 7. Adversary-Controlled System Semantics

The actions an adversary can take are captured by the AdversarySend and AdversaryDerive rules. Given current knowledge κ, the adversary can add values it knows to the network N (rule AdversarySend). It can derive new values by applying one of its derivation functions, represented by the set $\mathcal{A}$, to values it already knows (rule AdversaryDerive). The specific interaction the adversary made with the protocol (*send* or *derive*) is recorded in the trace of interactions and observed effects α. α is not used for execution, but is rather later used to define adversarial equivalence between two protocols under interaction with the same adversary. Note that adversary interactions that reference terms in the attacker's knowledge κ only reference terms by their index. This restriction is important in defining adversarial equivalence, since the adversary's specific knowledge may differ between two protocol executions while still remaining statically indistinguishable to the adversary.

Separately, the protocol π, represented as a tuple of the local handlers H and sub-protocols $\overline{\pi_\mu}$, may execute one of its handlers (rule ExecHandler) or one of its sub-protocols' handlers (rule ExecSubprocess). ExecSubprocess is defined recursively, so that any protocol transition must include exactly one invocation of ExecHandler. A protocol step updates the current protocol state Σ, along with adversary's knowledge κ, observed effects α, and network N. The adversary does not directly control handler execution, but can influence it by choosing to add messages to the network N. Handler execution is modeled as atomic, so the effects from executing the statements in a handler are either all applied (if the handler is enabled and can succesfully execute) or not applied. The semantics for the system is given in a small-step style since the system evolution is not expected to terminate and may infinitely take steps. The system semantics are non-deterministic, so many different transitions may occur from a given state.

5 Adversarial Equivalence and Collecting Semantics

In this section we ultimately define when two `DYAct` protocols are adversarially equivalent with respect to a given adversary $\mathcal{A}$. This definition relies on a collecting semantics for `DYAct` protocols and adversaries (Section 5.2) and the notion of adversarial interaction properties (Section 5.1).

5.1 Adversarial Interaction Properties

Adversarial equivalence is a relational property defined on two sets of adversarial interaction properties. Adversarial interaction properties are designed to capture sufficient information for two purposes: ① to model the interactions and observations an adversary has with a protocol and any information it can derive from those interactions, and ② to model system execution. The first purpose is necessary to reason about what an adversary has learned from interacting with a protocol, while the second purpose is necessary to reason about possible system executions that produce the relevant interactions, observations, and knowledge.

Definition 5.1 (Adversarial Interaction Property). An adversarial interaction property is a tuple of the form $(\alpha, \kappa, N, \Sigma)$ where α is a sequence of adversary interactions with and observable effects from a protocol, κ is a sequence of values representing the adversary's current knowledge, N is the current network state, and Σ is the current protocol state.

5.2 Collecting Adversarial Interaction Properties

All possible behaviors of a protocol π interacting with an adversary $\mathcal{A}$ can be captured as a set of forward-reachable adversarial interaction properties P. This set is defined using a collecting semantics (Figure 8), denoted $\{\!\![\![\mathcal{A}, \pi]\!]\!\!\}$, that is defined with respect to a set of initial system states P_0 and set of external channels C. Note that the system states $\langle \alpha, \kappa, N, \Sigma \rangle$ and adversarial interaction properties $(\alpha, \kappa, N, \Sigma)$ contain the same pieces of information and can be easily converted between. The collecting semantics is precisely defined as the least fixed point of the set of properties reachable via the system step relation (Section 4.4) starting from the given initial system states. A least fixed point exists by the Knaster-Tarski fixed point theorem since the set of adversarial interaction properties forms a complete lattice under set inclusion, and the *reachNext* function used below is monotonic.

Fig. 8. Collecting Semantics for `DYAct` Protocols and Adversaries

$$\mathit{reachNext}\ C\ \mathcal{A}\ \pi\ P_0 = \lambda\, P.\, P_0 \cup P \cup \left\{ (\alpha', \kappa', N', \Sigma') \;\middle|\; \begin{array}{l} C, \mathcal{A}, \pi \vdash \langle \alpha, \kappa, N, \Sigma \rangle \rightarrow \langle \alpha', \kappa', N', \Sigma' \rangle \\ \wedge\ (\alpha, \kappa, N, \Sigma) \in P \end{array} \right\}$$

$$\{\!\![\![\mathcal{A}, \pi]\!]\!\!\}\ C\ P_0 = \mathrm{lfp}\ (\mathit{reachNext}\ C\ \mathcal{A}\ \pi\ P_0)$$

In general, the set of reachable adversarial interaction properties is unbounded since given even a single term and a tuple constructor, an adversary can generate an unbounded number of distinct terms by repeatedly applying the constructor. As a result the set of collected properties is typically unbounded and therefore not computable. Despite this restriction, the collecting semantics still provide a precise semantics upon which to define adversarial equivalence. The collecting semantics are further useful as the foundation upon which sound over-approximate abstractions can be defined.

5.3 Adversarial Equivalence

Intuitively, two protocols Π_1 and Π_2 are adversarially equivalent for adversary $\mathcal{A}$ and external channels C when any observable behavior and derivable knowledge from one protocol can be matched for the other protocol, and vice versa. We use the notation Π to refer to the packaged set of initial system states for a protocol, P_0, as well as the protocol value that contains the handlers for the protocol, π. The intuition for this matching in one direction (from protocol Π_1 to protocol Π_2) is formalized in the adversarial preorder relation (Definition 5.2), while adversarial equivalence (Definition 5.4) simply requires the preorder to hold in both directions.

Adversarial preorder between Π_1 and Π_2 holds when the exact same sequence of adversary interactions and observable effects α from Π_1 can be produced by Π_2, and the adversary's knowledge after those interactions are statically equivalent. The possible interactions, effects, and knowledge are captured by the set of forward-reachable adversarial interaction properties produced by the collecting semantics $\{\!\![\![\mathcal{A}, \pi]\!]\!\!\}$. The static equivalence condition captures the intuition that while an adversary can unambiguously determine how it interacted with a protocol, it may not be able to distinguish between different pieces of information it has derived from those interactions dependent on the model of cryptographic primitives. In particular, to model semantically secure encryption schemes, two encrypted messages are considered statically indistinguishable to an adversary, regardless of their contents.

The definition relies on a static equivalence relation $\sim_\kappa$ between adversary knowledges κ_1 and κ_2, which is just the pairwise lifting of the indistinguishability relation $\sim_t$ on terms defined by the cryptographic model. Specifically, we say $\kappa_1 \sim_\kappa \kappa_2$ holds if and only if $\mathrm{dom}(\kappa_1) = \mathrm{dom}(\kappa_2)$ and for every index i, if $i \in \mathrm{dom}(\kappa_1)$ then $\kappa_1[i] \sim_t \kappa_2[i]$. We overload the notation $\sim$ to refer to either $\sim_\kappa$ or $\sim_t$ depending on the context.

Definition 5.2 (Adversarial Preorder). A protocol Π_1 $(\mathcal{A}, C)$-precedes Π_2, or equivalently $(\pi_1, P_{1,0})$ $(\mathcal{A}, C)$-precedes $(\pi_2, P_{2,0})$, denoted $\Pi_1 \lesssim_{\mathcal{A},C} \Pi_2$, if for every forward-reachable adversarial interaction property $(\alpha, \kappa_1, N_1, \Sigma_1) \in \left(\{\!\![\![\mathcal{A}, \pi_1]\!]\!\!\}\ C\ P_{1,0}\right)$ there exists a forward-reachable adversarial interaction property $(\alpha, \kappa_2, N_2, \Sigma_2) \in \left(\{\!\![\![\mathcal{A}, \pi_2]\!]\!\!\}\ C\ P_{2,0}\right)$ such that $\kappa_1 \sim \kappa_2$.

Since the adversarial preorder relation is parametric on the static equivalence relation $\sim_\kappa$ for the given cryptographic model, it is only a preorder if $\sim_\kappa$ is also a preorder. In practice, $\sim_\kappa$ should be an equivalence relation.

Lemma 5.3. *$\lesssim_{\mathcal{A},C}$ is a preorder (reflexive and transitive), if $\sim_\kappa$ is a preorder.*

Proof. $\lesssim_{\mathcal{A},C}$ is trivially reflexive since $\sim_\kappa$ is reflexive (by virtue of being a preorder). To show $\lesssim_{\mathcal{A},C}$ is also transitive, consider protocols Π, Π', and Π'' such that $\Pi \lesssim_{\mathcal{A},C} \Pi'$ and $\Pi' \lesssim_{\mathcal{A},C} \Pi''$. For every property $(\alpha, \kappa, N, \Sigma) \in (\{\!\![\![\mathcal{A}, \pi]\!]\!\!\}\ C\ P_0)$, there must exist some property $(\alpha, \kappa', N', \Sigma') \in \left(\{\!\![\![\mathcal{A}, \pi']\!]\!\!\}\ C\ P_0'\right)$ such that $\kappa \sim \kappa'$. Similarly, for the property $(\alpha, \kappa', N', \Sigma') \in \left(\{\!\![\![\mathcal{A}, \pi']\!]\!\!\}\ C\ P_0'\right)$, there must exist some property $(\alpha, \kappa'', N'', \Sigma'') \in \left(\{\!\![\![\mathcal{A}, \pi'']\!]\!\!\}\ C\ P_0''\right)$ such that $\kappa' \sim \kappa''$. Because $\sim_\kappa$ is transitive, $\kappa \sim \kappa'$, and $\kappa' \sim \kappa''$, then $\kappa \sim \kappa''$. Since $(\alpha, \kappa'', N'', \Sigma'') \in \left(\{\!\![\![\mathcal{A}, \pi'']\!]\!\!\}\ C\ P_0''\right)$ and $\kappa \sim \kappa''$, then $\Pi \lesssim_{\mathcal{A},C} \Pi''$. □

Given the definition of adversarial preorder, adversarial equivalence is simply defined as adversarial preorder in both directions.

Definition 5.4 (Adversarial Equivalence). Two protocols Π_1 and Π_2 are *adversarially equivalent* with respect to an adversary $\mathcal{A}$ for external channels C, denoted $\Pi_1 \approx_{\mathcal{A},C} \Pi_2$, if $\Pi_1 \lesssim_{\mathcal{A},C} \Pi_2$ and $\Pi_2 \lesssim_{\mathcal{A},C} \Pi_1$.

Lemma 5.5. *$\approx_{\mathcal{A},C}$ is an equivalence relation, if $\sim_\kappa$ is an equivalence relation.*

Proof. When $\sim_\kappa$ is an equivalence relation, then $\lesssim_{\mathcal{A},C}$ is reflexive and transitive by Lemma 5.3. The conjunction of preorders is reflexive and transitive, so $\approx_{\mathcal{A},C}$ is reflexive and transitive. $\Pi' \approx_{\mathcal{A},C} \Pi$ is defined as $\Pi' \lesssim_{\mathcal{A},C} \Pi$ and $\Pi \lesssim_{\mathcal{A},C} \Pi'$. Since conjunction is symmetric, this is equivalent to $\Pi \lesssim_{\mathcal{A},C} \Pi'$ and $\Pi' \lesssim_{\mathcal{A},C} \Pi$. Therefore $\approx_{\mathcal{A},C}$ is symmetric. Since $\approx_{\mathcal{A},C}$ is reflexive, transitive, and symmetric, it is an equivalence relation. ◻

6 Related Work

6.1 Comparison with Other Equivalences

In this paper we introduce the novel property adversarial equivalence for protocols interacting with an active symbolic adversary. Other notions of protocol equivalence have been introduced in prior work. The Calculus of Communicating Systems (CCS) defined observational equivalence without a notion of an adversary [10]. The applied pi calculi added the notion of observational equivalence with respect to an active adversary, where the adversary is represented using the context [1]. The applied pi calculus also defined labeled bisimilarity [1] and showed that labeled bisimilarity and observational equivalence coincide. The notions of trace equivalence introduced in [5] and multiset rewrite observational equivalence in [3] are very similar to the definition we give of adversarial equivalence. Trace equivalence is defined based on traces, which is analogous to our definition based on collected adversarial interaction properties. However, both trace equivalence and multiset rewrite observational equivalence treat receiving a message over a channel as an observational effect. We note that in practice a man-in-the-middle adversary is only able to intercept and send messages over a network, they are not able to determine when a message is read. Additionally, while we treat derivation steps as separate transitions, both trace equivalence and multiset rewrite observational equivalence are defined with respect to labels that include the recipes used to derive terms, effectively rolling many derivations into a single transition.

6.2 Comparison with Other Protocol Models

The models used in CCS [10], variants of the applied pi calculus [1, 5], and tamarin [3, 9] all model protocols algebraically as we do here. CCS and applied pi calculus variants further model communication between protocol participants as synchronous communication, whereas our model of communication is asynchronous, a more realistic choice. Tamarin's model of communication with an adversary is asynchronous, but other forms of communication between protocol participants need to be manually modeled with custom rewrite rules. The applied pi calculus variants typically model adversaries as contexts and include derivable terms as part of the definition of static equivalence, while we explicitly model adversaries and adversary derivations using explicit transitions in our semantics. Tamarin likewise models adversaries explicitly, but derivations are rolled into single rewrites using recipes rather than using several single-step derivations as we do.

7 Conclusion

In this paper, we introduced adversarial equivalence, a general definition of cryptographic protocol equivalence from the perspective of an active symbolic adversary. Adversarial equivalence is defined with respect to a collecting semantics that captures the interactions and observations the adversary makes with the protocol, as well as the knowledge adversary can learn from interacting with a protocol. Unlike prior work, our definitions do not treat receiving messages as observable effects and explicitly model adversary derivations, allowing for a more realistic model of adversarial interaction and observation. Furthermore, the collecting semantics we introduce serve as a foundation upon which abstractions can be explored and proved sound. Such abstractions may then be used to simplify and automate the verification of adversarial equivalence. Ultimately, this work is a first step towards reducing the effort required to prove general cryptographic protocol equivalence, such as simulation-style proofs.

References

[1] Abadi, M., Blanchet, B., Fournet, C.: The applied pi calculus: Mobile values, new names, and secure communication. J. ACM **65**(1) (Oct 2017). https://doi.org/10.1145/3127586, https://doi.org/10.1145/3127586

[2] Baelde, D., Delaune, S., Jacomme, C., Koutsos, A., Moreau, S.: An interactive prover for protocol verification in the computational model. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 537–554 (2021). https://doi.org/10.1109/SP40001.2021.00078

[3] Basin, D., Dreier, J., Sasse, R.: Automated symbolic proofs of observational equivalence. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. p. 1144–1155. CCS '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2810103.2813662, https://doi.org/10.1145/2810103.2813662

[4] Blanchet, B., Abadi, M., Fournet, C.: Automated verification of selected equivalences for security protocols. The Journal of Logic and Algebraic Programming **75**(1), 3–51 (2008). https://doi.org/https://doi.org/10.1016/j.jlap.2007.06.002, https://www.sciencedirect.com/science/article/pii/S1567832607000549, algebraic Process Calculi. The First Twenty Five Years and Beyond. III

[5] Cheval, V., Comon-Lundh, H., Delaune, S.: Trace equivalence decision: negative tests and non-determinism. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. p. 321–330. CCS '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2046707.2046744, https://doi.org/10.1145/2046707.2046744

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

[7] Lindell, Y.: How to Simulate It – A Tutorial on the Simulation Proof Technique, pp. 277–346. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-57048-8_6, https://doi.org/10.1007/978-3-319-57048-8_6

[8] Lowe, G.: An attack on the needham- schroeder public- key authentication protocol. Information processing letters **56**(3) (1995)

[9] Meier, S., Schmidt, B., Cremers, C., Basin, D.: The tamarin prover for the symbolic analysis of security protocols. In: International conference on computer aided verification. pp. 696–701. Springer (2013)

[10] Milner, R.: A calculus of communicating systems, Lecture Notes in Computer Science, vol. 92. Springer Berlin, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3

[11] Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12), 993–999 (Dec 1978). https://doi.org/10.1145/359657.359659, https://doi.org/10.1145/359657.359659

[12] Santiago, S., Escobar, S., Meadows, C., Meseguer, J.: A formal definition of protocol indistinguishability and its verification using maude-npa. In: Mauw, S., Jensen, C.D. (eds.) Security and Trust Management. pp. 162–177. Springer International Publishing, Cham (2014)