Automated Proofs for Diffie–Hellman–based key exchanges

Long Ngo, Colin Boyd and Juan González Nieto
Information Security Institute
Queensland University Of Technology
GPO Box 2434, Brisbane QLD 4001, Australia
{lt.ngo, c.boyd, j.gonzalez nieto}@qut.edu.au

Abstract—We present an automated verification method for security of Diffie–Hellman–based key exchange protocols. The method includes a Hoare-style logic and syntactic checking. The method is applied to protocols in a simplified version of the Bellare–Rogaway–Pointcheval model (2000). The security of the protocol in the complete model can be established automatically by a modular proof technique of Kudla and Paterson (2005).

I. INTRODUCTION

Key exchange protocols are a very important mechanism to establish secure communication channels. In general, conducting security analysis manually for key exchange is time-consuming and easy to get wrong. Consequently, formal analysis with tool support has been popular and successful for at least 20 years. One famous example is when Lowe [1] used the tool FDR and found an attack against the Needham–Schroeder public–key protocol, which had been believed to be secure for many years.

Most protocol analysis tools have treated cryptography in an idealized manner, using the Dolev–Yao model. Meanwhile, the cryptographic research community has developed computational models for security of key exchange protocols and produced numerous proofs by hand. However, manual security proofs for cryptographic primitives are similarly easy to get wrong and many flaws in security proofs were found after the proofs have been published [2], [3]. As a consequence, automated methods for proof generation and verification for cryptographic primitives have been considered a promising advance. So far, little research effort has been spent for automated proofs for key exchange. This paper is one step towards filling the gap.

Designing an automated procedure for verifying protocol security in a powerful cryptographic model is not easy. The first difficulty is that we need a computational formalism, i.e. probability and complexity must be included. However, formal methods have not been applied widely for computational security models typically used in the cryptographic research community. The second difficulty is that even if we can make such a formalism, it may not ease the pain but could make things more complicated [4]. The main reason is that a computational adversary is so powerful that there are many things to be considered together at one time. Therefore too much formality would cause great complexity for deriving and understanding the proofs. That is why the cryptography community has not been so interested in using formal methods to solve their problems. We take these obstacles into consideration during this work.

Contribution: In this work, we are interested in automated verification for security of Diffie–Hellman based key exchange in the Bellare–Rogaway (BR) model [5], a typical class of key exchange in a typical model. The basic idea of our approach is as follows. We define a programming language for specifying protocols formally. Then we define an assertion language to describe some properties that a protocol run may satisfy. We will show that those properties guarantee the security of the protocol. Finally, we design a logic to verify if a protocol satisfies the properties or not. Verification with the logic can be done automatically. However, the automated verification would be very complicated if we made it directly for the BR model, because the adversary is too powerful.

We reduce the complexity of the verification by using a “divide and conquer” strategy. First, the work is based on the modular proof technique by Kudla and Paterson [6], which already divides the manual analysis into separate steps. The basic idea of their technique is that the analysis in the Bellare–Rogaway model can be done by using a simpler model, called cNR-mBR. Consequently, we restrict the verification to the cNR-mBR model only. Second, our verification uses a Hoare-style logic, whose basic idea is verifying a whole algorithm by tracking the state after each step. It means that the whole analysis is done by analyzing every step inductively, assuming that protocols are functional.

Unfortunately, the original Hoare logic is limited to verifying independent sequential programs, but key exchange protocols are distributed. In some cases, verifying the algorithm of each party separately is enough to conclude the security. But, in general, the relation between two parties must be analyzed in order to conclude the security. To overcome this problem, we propose a syntactic checking technique, whose idea is borrowed from the way Hoare logic has been improved to verify communicating sequential process [7], [8], [9]. However, we have to adapt the technique to suit the malicious communicating environment. We do not syntactically match any received message with a
sent message, because the network is not trusted. Instead, we syntactically match signing and verifying commands, because it is unlikely that the adversary can fake a signature. Last but not least, reasoning using our logic can be fully automated. We also have implemented a reasoning tool and tested it on several protocols. The source code is small and the tool can produce the proofs in a short time.

Related work: Automated proofs in cryptographic models have attracted a certain interest from previous researchers. One line is to use automated tools designed for a symbolic, i.e. Dolev-Yao, model and then show that the resulting proof implies the proof we want in a cryptographic model. Our work follows a different approach, which is to make automated proofs directly in cryptographic models. We will mention here only work in this direction.

Courant et al. [10] designed a Hoare-style logic to verify computational invariants, e.g. indistinguishability in an asymmetric encryption scheme. Later, the logic has been extended to cover symmetric block ciphers by Gagné et al. [11] and has been generalized into Computational Indistinguishability Logic by Barthe et al. [12]. Those logics are different from this one in that they focus on cryptographic primitives, while this logic is for protocols.

Blanchet [13] designed a variant of $\pi$-calculus to formalise games, and developed CryptoVerif, a tool that can automatically transform games using game-hopping techniques, thereby freeing the human from the mundane parts of the proof. CryptoVerif can be potentially extended to cover many types of protocols, but it is hard to make it fully automated, i.e. manual guidance is required in non-trivial situations. In addition, CryptoVerif covers only static corruptions and has not yet been shown to work on Diffie-Hellman protocols.

Datta et al. [14] tuned the Computational Protocol Composition Logic for verifying key exchange protocols, resulting in security proofs in the older version of Bellare-Rogaway model [15]. The major difference between their work and ours is that they verify trace properties, whereas we check non-trace ones. Furthermore, it is not clear how reasoning in their logic can be for protocols.

We are not aware of much work that takes advantage of the simplicity in modular proofs. In earlier work, we have used the same modular proof as we do here and proposed an approach to automate proofs [16]. This earlier work linked a Dolev-Yao model used in an automated tool with the simplified model. The proof made by the automated tool implies security in the simplified model, and then security in the full model can be achieved by compiling the protocol. The basic idea is the same as in this paper, but the previous work is different in the way that security proofs of public-key-based protocols are achieved indirectly, via a Dolev-Yao tool. In this paper we establish proofs directly for Diffie-Hellman-based protocols.

The rest of the paper is structured as follows:

- Section 2 explains some preliminaries for the work.
- Section 3 discusses how we model the protocol execution in an adversarial environment.
- Section 4 defines some properties that a protocol running in the model may satisfy.
- Section 5 introduces a Hoare-style logic for checking those properties.
- Section 6 shows how we can additionally use a syntactic checking to improve the result.
- Section 7 describes the implementation of our tool.
- Section 8 concludes the paper.

II. PRELIMINARIES

In this section we define some necessary primitives and their security requirements. Our work covers key exchange protocols based on the Diffie-Hellman scheme with signature schemes used to authenticate messages.

A. Diffie-Hellman problems

Let $p$ and $q$ be primes where $q|p-1$ and $|q| = k$, and let $G$ be a multiplicative subgroup of $\mathbb{Z}_p^*$ of order $q$. Let $g$ be a generator of $G$. Notice that by $g^a$, we mean $g^a \mod p$. The computational, decisional and gap problems of the Diffie-Hellman instance $(p, q, g)$ are defined as follows.

- **Computational Diffie-Hellman problem (CDH)**
  Given $g^a, g^b \in G$ where $a, b \in \mathbb{Z}_q$, output $g^{ab}$.

- **Decisional Diffie-Hellman problem (DDH)**
  Given $g^a, g^b, g^c \in G$ where $a, b \in \mathbb{Z}_q$, determine whether or not $g^c = g^{ab}$.

- **Gap Diffie-Hellman problem (GDH)**
  Given $g^a, g^b \in G$ where $a, b \in \mathbb{Z}_q$ and an oracle that solves the DDH problem of $G$, output $g^{ab}$.

We assume that the above problems are hard, i.e. there is no probabilistic polynomial-time (PPT) algorithm to solve them.

B. Signature Scheme

Definition 1: A signature scheme is a triple of algorithms:

- A PPT algorithm $G_{Sig}: (pk, sk) \leftarrow G_{Sig}(1^k)$ where $pk, sk$ are called public and private keys respectively.
- A PPT algorithm $S_{Sig}: s \leftarrow S_{Sig}(sk, m)$ where $m, s \in \{0, 1\}^*$ and $m, s$ are called message and signature respectively.
- A deterministic PT algorithm $V_{Sig}$ such that $V_{Sig}(pk, m, s) \in \{\text{true}, \text{false}\}$.

We demand that $V_{Sig}(pk, m, S_{Sig}(sk, m)) = \text{true}$. If $V_{Sig}(pk, m, s)$ is true then $s$ is a valid signature of $m$.

Definition 2: A signature scheme is existentially unforgeable under adaptive chosen message attacks (EF-CMA) if for every PPT algorithm $A$, which is given a public key and a signing oracle, the probability that $A$ produces a valid signature of a message not previously sent to the signing oracle is negligible.
III. PROTOCOL MODELLING

This section describes how we model a protocol running in an adversarial environment, what we mean by security, and a modular proof technique that allows us to simplify the model. Our work focuses on that simplified model, called cNR-mBR. We also present a simple programming language for specifying a protocol in that model.

A. A communication model for key exchange protocols

We define a variant of the Bellare–Rogaway model [5] for key exchange protocols (see Figure 1). Our model is the same as the mBR model by Kudla and Paterson [6], except that we do not consider any corrupted oracle to be fresh (since we do not model key compromise impersonation).

Let \( k \) be the security parameter and \( U \) denote the set of \( p(k) \) participant IDs, each of which may have up to \( o(k) \) instances. We use \( \Pi_U \) to denote the oracle of the \( i \)th instance of \( U \). Every \( \Pi_U \) has a partner ID \( pid \), which is the ID of another arbitrary party.

How a party oracle behaves depends on the protocol specification, defined by two algorithms \( \mathcal{I}(\text{is}k) \) and \( \mathcal{R}(\text{rsk}) \) : \( c_i \) and \( c_r \), which are initiator and responder code respectively. Each party oracle executes only one of these algorithms once and uses the output \( \text{is}k \) or \( \text{rsk} \) as its session key (for more details, see Section III-D).

At any time, an oracle \( \Pi_U \) is in one of the following states:

- **Accepted**: When the oracle has received a set of properly constructed messages to make a session key and the oracle accepts the key. It holds a role \( \text{role} \in \{\text{initiator}, \text{responder}\} \), a session ID \( \text{sid} \) and a session key \( \text{seskey} \), which is the output of the executed algorithm, i.e. \( \mathcal{I}(\text{is}k) \) or \( \mathcal{R}(\text{rsk}) \).
- **Rejected**: When the oracle has decided not to establish a session key and has aborted the protocol.
- **Revealed**: When the oracle has answered a Reveal query from the adversary.
- **Corrupted**: When the oracle has answered a Corrupt query from the adversary.
- **State \(*\)**: An oracle is in this state if it is not in any state above.

**Partnership.** Two oracles \( \Pi_U \), holding \( \text{seskey}_{\Pi_U}, \text{sid}_{\Pi_U}, \text{pid}_{\Pi_U} \) and \( \Pi_V \), holding \( \text{seskey}_{\Pi_V}, \text{sid}_{\Pi_V}, \text{pid}_{\Pi_V} \) are said to be partners if they have accepted and:

1. \( \text{sid}_{\Pi_U} = \text{sid}_{\Pi_V}, \text{seskey}_{\Pi_U} = \text{seskey}_{\Pi_V}, \text{pid}_{\Pi_U} = V, \text{pid}_{\Pi_V} = U \);
2. \( \text{role}_{\Pi_U} = \text{initiator} \) and \( \text{role}_{\Pi_V} = \text{responder} \) or vice versa;
3. no other oracle has accepted with session ID equals \( \text{sid}_{\Pi_U} \).

**Freshness.** An oracle \( \Pi_U \) is fresh if it and its partner \( \Pi_V \) (if any) are not in the state Revealed and neither \( U \) nor \( V \) is in the state Corrupted.

**Protocol execution.** The execution of the protocol is modelled by the following experiment. There are a challenger \( C \) and an adversary \( E \). \( C \) maintains a string \( pubinf \), which will be given to \( E \). Given an EF-CMA signature scheme \( (G_{\text{Sig}}, S_{\text{Sig}}, V_{\text{Sig}}) \) and a random oracle \( H, C \) executes the following initial steps:

1. **Preparing Diffie–Hellman–based keys**: \( C \) creates a Diffie–Hellman–instance \( (p, q, g) \) (as explained in Section II-A). Then for each party \( U \), \( C \) picks a random secret \( dhseckey \) as the secret key, makes \( g^{dhseckey} \) as the public key \( dhpubkey \). \( C \) makes \( (p, q, g), dhseckey \) of \( U \) and \( dhpubkey \) of \( V \) accessible to any oracle \( \Pi_U \), whose partner is \( V \).

2. **Preparing signature keys**: For each party \( U \), \( C \) runs \( G_{\text{Sig}} \) on input \( 1^k \) to get a pair \( (sigseckey, sigpubkey) \). \( C \) makes \( sigseckey \) of \( U \) and \( sigpubkey \) of \( V \) accessible to any oracle \( \Pi_U \), whose partner is \( V \).

3. **Preparing public information**: First, for every party \( U \), \( C \) adds the public key \( dhpubkey \) and \( sigpubkey \) are added into \( pubinf \). Second, for every oracle \( \Pi_U \), \( C \) adds the string \( pubinf \) and \( pubinf \). Thirdly, the generated Diffie–Hellman instance \( (p, q, g) \) is added into \( pubinf \). Finally, the algorithms \( G_{DH}, G_{\text{Sig}}, S_{\text{Sig}}, V_{\text{Sig}}, \) the random oracle \( H \) and the string \( pubinf \) are made available for all party oracles and the adversary.

4. **Starting the experiment**: \( C \) starts the adversary \( E \) and allows it to query party oracles.

Now \( E \) can make the following queries:

- **Send** \((U, i, M)\): \( E \) gives the oracle \( \Pi_U \) a message \( M \). \( \Pi_U \) assumes that \( M \) is from its partner and acts according to the protocol. For initiating an oracle \( \Pi_U \), \( E \) can make a special Send query \( \lambda \), which tells \( \Pi_U \) to set its \text{role} = \text{initiator} \) and executes the code for initiator, i.e. \( \mathcal{I}(\text{is}k) \). If \( \Pi_U \) did not receive a message \( \lambda \) as the first message, \text{role} is set to be \text{responder} and the code for responder, i.e. \( \mathcal{R}(\text{rsk}) \), will be executed.
**Reveal**($U, i$): $E$ uses this query to obtain the session key of $\Pi^r_U$ (if any).

**Corrupt**($U$): This allows $E$ to learn $U$’s long-term keys, i.e. $dhseckey$ and $sigseckey$.

**Test**($U, i$): Only once during the experiment, $E$ can make a **Test** query to an oracle $\Pi^r_U$, which must be in **Accepted** state, still fresh and already terminated. Then $C$ chooses a random bit $b$. If $b = 0$ then $\Pi^r_U$ outputs a randomly chosen session key, otherwise it outputs its real session key, which is the output of the terminated algorithm, i.e. $\mathcal{I}(isk)$ or $\mathcal{R}(rsk)$. $E$ can continue querying, but not reveal or corrupt the test oracle or its partner.

Finally, $E$ outputs his guess $b'$ for $b$. $E$’s advantage, denoted as $\text{Advantage}_E(k)$, is $|\frac{1}{2} - \Pr[b' = b]|$.

**Remark 1**: Party oracles communicate through the adversary and do not share any variable, so they are message-driven. We assume that there is only one party oracle that is active at one time. Whenever the adversary asks a **Send**($U, i, M$) query, $\Pi^r_U$ is active until it finishes processing $M$ and terminates, or rejects, or becomes inactive, waiting for the next message. After that, it is up to the adversary to activate another party oracle by making a new **Send** query.

**B. Definition of Security**

A benign adversary is one who just relays messages between parties without any modification. Then security for authenticated key exchange (AKE) is defined as follows.

**Definition 3**: Given a security parameter $k$, a protocol is a secure AKE protocol if:

1) in the presence of a benign adversary, two oracles running the protocol accept and hold the same session key and session ID, and the session key is distributed uniformly at random on $\{0,1\}^k$; and
2) for any adversary $E$, $\text{Advantage}_E(k)$ is negligible.

**C. A modular proof**

The idea of modular proof by Kudla and Paterson [6] is essential for our work. Following that technique, a protocol $\Pi$ defined in the full model described above can be first proven secure in the simpler model called cNR–mBR. (The name cNR–mBR stands for “computational Non-Reveal modified Bellare Rogaway”. The simpler model requires the adversary to solve a computational rather than a decisional problem, without using any **Reveal** queries.) Then we can promote such a protocol to one secure in the full model as long as the protocol $\Pi$ produces a **session string** which is input to a hash function, modelled as a random oracle, to finally compute a hashed session key.

**Definition 4**: [6] If $\Pi$ is a key exchange protocol and there exists an adversary $E$, who participates in the above experiment with $\Pi$, and with non-negligible probability (in security parameter $k$) can make any two oracles $\Pi^r_U$ and $\Pi^r_V$ accept and hold the same session key when they are not partners, then we say the $\Pi$ has **weak partnering**. Otherwise $\Pi$ has **strong partnering**.

Note that we can always ensure a protocol $\Pi$ has strong partnering by adding the partnering information, i.e. session IDs and partner IDs [6].

**Definition 5**: Suppose $\Pi$ is a key exchange protocol. The **session string decisional problem** for protocol $\Pi$ is: given all information accessible to $\Pi^r_U$ except the Diffie–Hellman secret key $dhseckey$, $\Pi^r_U$’s transcript $T^{\Pi^r_U}$ and a string $s$, decide whether $s$ is the session string of $\Pi^r_U$ or not.

**Definition 6**: The cNR–mBR model is the same as the model above, except:

- the adversary cannot make any **Reveal** query;
- instead of a normal **Test** query, the adversary selects an accepted and fresh oracle $\Pi^r_U$, whose session key is $seskey$, and outputs seskey’. Then $\text{Advantage}_E(k) = \Pr[seskey' = seskey]$.

In order to use the cNR–mBR model, given a protocol $\Pi$, we define a protocol $\pi$ to be the same as $\Pi$, except that the session string of $\Pi$ is the session key of $\pi$.

**Theorem 1**: [6] Suppose that a key exchange protocol $\Pi$ uses a hash function $H$ to compute a hashed session key on completion of the protocol and $\Pi$ has strong partnering. If the cNR–mBR-security of the related protocol $\pi$ is probabilistic polynomial time reducible to the hardness of the CDH problem, and the session string decisional problem for $\Pi$ is polynomial time reducible to the decisional problem of $f$, then the security of $\Pi$ is probabilistic polynomial time reducible to the hardness of the GDH problem, assuming that $H$ is a random oracle.\(^1\)

This modular proof makes our work easier. From now on, we focus just on the cNR–mBR model. The security of a protocol can be established later by adding partnering information, i.e. session IDs and partner IDs [6], and hashing the session key.

**D. Protocol specification in the cNR–mBR model**

In Section III-A we have shown how we model the communication between party oracles. That model is the same for every protocol. However, party oracles of different protocols act differently, depending on the protocol specifications. In this section we define a simple programming language for specifying protocol in a formal way. Notice that the language is for the cNR–mBR model, therefore there will be no command for hashing.

A protocol specification $\pi$ is defined as $\pi = (\mathcal{I}(isk) : c_i, \mathcal{R}(rsk) : c_r)$, where $\mathcal{I}(isk) : c_i$ and $\mathcal{R}(rsk) : c_r$ are the initiator and responder algorithm respectively. When executed, $\mathcal{I}(isk)$ or $\mathcal{R}(rsk)$ runs every command inside it and the value of $isk$ or $rsk$ will be used as the session key.

\(^1\)By CDH and GDH problems, we mean the CDH and GDH problems of the Diffie–Hellman instance $(p, q, g)$ used in the experiment.
1) Specification syntax: The syntax of the language is defined in Table I. To avoid confusion, we require that the initiator and responder codes of a protocol use disjoint sets of variable names. The language allows us to express protocols based on the basic flow of the original DH protocol, but enhanced by an authentication mechanism (signature or public key). However, the language does not yet support hashing commands, which may allow us to express protocol public key). However, the language does not yet support hashing commands, which may allow us to express protocol

Example 1: The “no-hashing” version of the protocol 4 of Blake-Wilson et al. [19], denoted by $\tau_4$, is written is our programming language.

2) Semantics of the commands: Now we describe formally how an oracle $\Pi_U$ behaves, given a protocol specification. The semantics of a command is defined by showing how it affects the current local state of a party oracle.

Recall that the following information is accessible to party oracle $\Pi_U$: its own ID oid, the partner ID pid, DH-related information $(p, q, g)$, dhseckey and dhpubkey_partner, signature-related information sigseckey, sigpubkey_partner, and two algorithms $S_{\text{Sig}}, V_{\text{Sig}}$. Let $\text{orcinf}$ denote all of this information.

In addition to programming variables, each party oracle $\Pi_U$ maintains some auxiliary variables, which provide information for a security reduction in next sections:

- rejected: This variable can be true or false. It is to know whether $\Pi_U$ has rejected or not.
- $\mathbb{N}$: This is the set of nonces generated by $\Pi_U$ (including the static DH secret key). It tells us what values may not be given when we make a reduction from a protocol to the CDH problem.
- $\mathbb{T}$: This variable holds all the messages sent/received by $\Pi_U$ and all the DH public values generated by $\Pi_U$, in chronological order. It tells us what values will be given when we make a reduction from a protocol to the CDH problem.

An oracle state $S$ of a party oracle is a map from any variable (both programming and auxiliary variables) to a string in $\{0, 1\}^*$. A command executed by $\Pi_U$ will transit the current oracle state $S$ of $\Pi_U$ to a distribution of new oracle states. Given an oracle state $S$, let $S'(x)$ denote the bitstring value kept in the storage of the variable $x$ and $S[x \mapsto u]$ denote that the value of $x$ is set to be $u$. These notations are extended to a list of variables $l$ in a similar way. Notice that $\Pi_U$ can execute a command if its variable rejected is not true. The transition is defined formally (together with informal explanations) in Table III.

IV. AN ASSERTION LANGUAGE

In the previous section we have defined an experiment for key exchange. During the experiment, every party oracle changes from one local state to another after executing every command. If we could deal with oracle local states

<table>
<thead>
<tr>
<th>Command $c ::=$</th>
<th>Informal description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := \text{mkDHSec}()$</td>
<td>make a DH secret value, i.e. a nonce</td>
</tr>
<tr>
<td>$y := \text{mkDHPub}(x)$</td>
<td>compute a DH public value, i.e. $g^a$ where $a$ is a nonce</td>
</tr>
<tr>
<td>$z := \text{mkDHPub}(y, x)$</td>
<td>compute a DH key, i.e. $g^{ab}$ where $a, b$ are nonces</td>
</tr>
<tr>
<td>$x, y := \text{getStaticDHKey}()$</td>
<td>return pre-generated DH secret and public values</td>
</tr>
<tr>
<td>$x, y := \text{getIDs}()$</td>
<td>return the oracle and its partner’s IDs</td>
</tr>
<tr>
<td>$z := \text{sign}(x, y)$</td>
<td>$x$ is a signature when we sign the list 1 of variables</td>
</tr>
<tr>
<td>$\text{verisign}(i, x)$</td>
<td>verify if $x$ is a signature of the list 1 of variables</td>
</tr>
<tr>
<td>$z := \text{concat}(x, y)$</td>
<td>$z$ is concatenation of $x$ and $y$</td>
</tr>
<tr>
<td>$z := \text{add}(x, y)$</td>
<td>$z = x + y$</td>
</tr>
<tr>
<td>$z := \text{multiply}(x, y)$</td>
<td>$z = x \cdot y$</td>
</tr>
<tr>
<td>$\text{send}(1) \mid \text{receive}(1)$</td>
<td>send or receive a list of variables</td>
</tr>
<tr>
<td>$c; c$</td>
<td>a sequence of commands</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\mathcal{N}(x) : c$</th>
</tr>
</thead>
</table>

where

- $c$ is a command.
- $x, y, z$ are programming variables.
- 1 is a list of programming variables.

Table I

SYNTAX OF PROTOCOL SPECIFICATION

<table>
<thead>
<tr>
<th>$\mathcal{I}(\text{ink})$</th>
<th>$\mathcal{R}(\text{rsk})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{ir} := \text{mkDHSec}()$</td>
<td>$\text{rsk} := \text{mkDHSec}()$</td>
</tr>
<tr>
<td>$\text{igr} := \text{mkDHPub}(\text{ir})$</td>
<td>$\text{rgr} := \text{mkDHPub}(\text{rsk})$</td>
</tr>
<tr>
<td>$\text{send}(\text{igr})$</td>
<td>$\text{send}(\text{rgr})$</td>
</tr>
<tr>
<td>$\text{is, ip} := \text{getStaticDHKey}()$</td>
<td>$\text{rs, rp} := \text{getStaticDHKey}()$</td>
</tr>
<tr>
<td>$\text{isk1} := \text{mkDHKey}(\text{rgr1}, \text{is})$</td>
<td>$\text{rsk1} := \text{mkDHKey}(\text{rp, rsk})$</td>
</tr>
<tr>
<td>$\text{isk2} := \text{mkDHKey}(\text{ip, ir})$</td>
<td>$\text{rsk2} := \text{mkDHKey}(\text{igr1, rs})$</td>
</tr>
<tr>
<td>$\text{isk} := \text{concat}(\text{isk1}, \text{isk2})$</td>
<td>$\text{rsk} := \text{concat}(\text{rsk1}, \text{rsk2})$</td>
</tr>
</tbody>
</table>

Table II

PROTOCOL 4 OF BLAKE-WILSON ET AL. [19] IN OUR PROGRAMMING LANGUAGE
\[x := \text{mkDHSec}(\cdot)\] \((S, \text{orcinf}) = [u \leftarrow U; \forall x \rightarrow (S(N) \cup \{u\}), T \rightarrow (S(T), y^u)]\)

\((u \text{ is chosen randomly, } x \text{ evaluates to } u, \text{ then } N, T \text{ are updated})\

\[y := \text{mkDHPub}(x)\] \((S, \text{orcinf}) = S[y \rightarrow g^{S(y)}]\)

\((y \text{ evaluates to } g^y)\

\[z := \text{mkDHKey}(y, x)\] \((S, \text{orcinf}) = S[z \rightarrow S(y)^{S(x)}]\)

\((z \text{ evaluates to } y^z)\

\[x, y := \text{getStaticDHKey}()\] \((S, \text{orcinf}) = S[x \rightarrow \text{dhseckey}, y \rightarrow \text{dhpubkey}_{\text{partner}}, N \rightarrow (S(N) \cup \{\text{dhseckey}\}), T \rightarrow (S(T), g^{\text{dhseckey}})]\)

\((x \text{ evaluates to the DH secret key, } y \text{ evaluates to the partner’s DH public key})\

\[x, y := \text{getIDs}()\] \((S, \text{orcinf}) = S[x \rightarrow \text{oid}, y \rightarrow \text{pid}]\)

\((x \text{ evaluates to the oracle ID, } y \text{ evaluates to its partner’s ID})\

\[x := \text{sign}(1)\] \((S, \text{orcinf}) = S[x \rightarrow S_{\text{Sig}}(\text{sigseckey}, S(1))]\)

\((x \text{ evaluates to a signature, signed on 1})\

\[\text{verisign}(1, x)\] \((S, \text{orcinf}) = \begin{cases} S[\text{rejected} \rightarrow \text{true}] & \text{if } v = \text{false, where } v \leftarrow V_{\text{Sig}}(\text{sigpubkey}_{\text{partner}}, S(1), S(x)) \\ S & \text{otherwise} \end{cases}\)

\((\text{verify if } x \text{ is a signature signed by the partner on 1 or not})\

\[x := \text{concat}(x, y)\] \((S, \text{orcinf}) = S[z \rightarrow S(x)||S(y)]\)

\((z \text{ evaluates to the concatenation of } x \text{ and } y)\

\[x := \text{add}(x, y)\] \((S, \text{orcinf}) = S[z \rightarrow (S(x) + S(y)) \mod p]\)

\((z \text{ evaluates to } x + y)\

\[x := \text{multiply}(x, y)\] \((S, \text{orcinf}) = S[z \rightarrow (S(x) \ast S(y)) \mod p]\)

\((z \text{ evaluates to } x \ast y)\

\[\text{send}(1)\] \((S, \text{orcinf}) = S[T \rightarrow S(T, 1)]\)

\((T \text{ is updated. The content } S(1) \text{ is given to the adversary as the answer for the most recent adversarial query } \text{Send}(U, i, M).)\

\[\text{receive}(1)\] \((S, \text{orcinf}) = \begin{cases} S[1 \rightarrow M, T \rightarrow S(T, 1)] & \text{if } |M| = 1 \\ S[\text{rejected} \rightarrow \text{true}] & \text{otherwise} \end{cases}\)

\((\text{The oracle } \Pi_U^i \text{ becomes inactive and waits until it is reactivated by a new } \text{Send}(U, i, M) \text{ command from the adversary (see Remark 1). } 1 \text{ evaluates to } M)\

\[\text{[c1}; c_2][S, \text{orcinf}] = [c_2][[c_1][S, \text{orcinf}], \text{orcinf}]\)

\((c_2 \text{ is executed after } c_1)\

\[\text{[N}(x) : c][S, \text{orcinf}] = [c][S, \text{orcinf}]\).

\((\text{Executing the algorithm } \text{N} \text{ means executing c and x is the output})

Table III

SEMANTICS OF THE PROGRAMMING LANGUAGE

The assertion language would be pretty as simple as a typical Hoare-style logic. Unfortunately, the adversary interacts with a set of algorithms running in parallel, therefore we must consider global invariants. We define an experiment state to be the set of isolated states of all party oracles. Since every party oracle is probabilistic, at every moment, the experiment is in a probabilistic state, i.e. a distribution of states. We will show that if the way a protocol changes the experiment state satisfies some properties, then the protocol security is reduced to CDH problem.

There are several points to be considered to make the assertion language formally captures the properties that guarantee a reduction. First, we must be able to simulate any party oracle, without using any nonce (DHSim). Second, a DH public value that oracle receives must be from an honest oracle (DHPubOut). Third, when the adversary gives the session key, we must be able to solve the CDH problem (DHSol). DHPub and DHPubln are used only to establish other predicates.

A. Syntax

\[
\psi ::= \text{DHSec}(<\Pi_U^i; x; bv>) \mid \text{DHPubln}(<\Pi_U^i; y>) \mid \text{DHPubOut}(<\Pi_U^i; y>) \mid \text{DHSol}(<\Pi_U^i; z; y; x>) \mid \text{DHSim}(<\Pi_U^i; g>)
\]

\[
\varphi ::= \text{true} \mid \psi \mid \varphi \land \psi
\]

where

- \(\Pi_U^i\) is the \(i^{th}\) instance of party \(U\).
• x, y, z are variables used in party oracles.
• bv and gv are sets of variables (bv and gv stand for “bad variables” and “good variables”).

B. Semantics

The semantics of an assertion \( \varphi \) is defined by showing when a distribution of experiment states \( X \) satisfies \( \varphi \), written \( X \models \varphi \). To avoid ambiguity, we write \( [x]_{\Pi_U} \) for the variable \( x \) inside the party oracle \( \Pi_U \).

- \( X \models \varphi \) true.
- \( X \models \varphi_1 \land \varphi_2 \) iff \( X \models \varphi_1 \) and \( X \models \varphi_2 \).
- \( X \models \text{DHSim}(\Pi_U; x; bv) \) iff there exists an algorithm \( B \) such that

\[
\Pr[S \leftarrow X, x \leftarrow \text{T.snd} : \varphi_1 \land \varphi_2] \text{ is overwhelming, where } \varphi_1 = B(S([x]_{\Pi_U})) \text{ and } \varphi_2 = S([y]_{\Pi_U})
\]

with \([\text{vars}]_{\Pi_U} \) is the set of all variables in \( \Pi_U \).

Informally, \( \varphi_1 \) means that the variable \( x \) of \( \Pi_U \) has been assigned a nonce \( n \). \( \varphi_2 \) means that if we are given the set \( N \) of nonces (except the nonce stored in \( x \)), the current transcript and all public information, we can still compute every variable in \( \text{vars} \) except ones in the set \( bv \).

- \( X \models \text{DHPubln}(\Pi_U; y) \) iff there exists an algorithm \( B \) such that

\[
\Pr[S \leftarrow X, j \leftarrow \{1, \ldots, o(k)\}, n \leftarrow S([y]_{\Pi_U}) \in \text{g}^n] \text{ is non-negligible, where } V \text{ is } \Pi_U \text{’s partner and } o(k) \text{ is the number of instances of a party.}
\]

Informally, this predicate means that the variable \( y \) inside \( \Pi_U \) must equal \( g^n \) where \( n \) is a nonce used in one instance of \( \Pi_U \)‘s partner.

- \( X \models \text{DHSol}(\Pi_U; z; y; x) \) iff there exists an algorithm \( B \) such that

\[
\Pr[S \leftarrow X, m = S([y]_{\Pi_U}), n = S([x]_{\Pi_U}) : \varphi_1 \land \varphi_2]
\]

is overwhelming, where \( \varphi_1 = S([x]_{\Pi_U}) \in S([N]_{\Pi_U}) \) and \( \varphi_2 = B(S([N]_{\Pi_U}) \setminus \{S([x]_{\Pi_U})\}, S([z]_{\Pi_U}), \text{pubinf} = n^m) \).

Informally, \( \varphi_1 \) means that \( x \) is a nonce, while \( \varphi_2 \) means that given \( z \), all nonces (except \( x \)), the current transcript and all public information, we can output \( y \). In other words, we can solve a DH instance with challenge values \( y \) and \( g^z \).

- \( X \models \text{DHSec}(\Pi_U; x; bv) \) iff there exists two algorithms \( B \) and \( B’ \) such that

\[
\Pr[S \leftarrow X, x \leftarrow \text{T.snd} : \varphi_1 \land \varphi_2] \text{ is overwhelming, where }
\]

\[
\varphi_1 = B(S([\text{T.pre}(x)]_{\Pi_U}), \text{pubinf} = S([x]_{\Pi_U})
\]

\[
\varphi_2 = B’(S([T]_{\Pi_U}), \text{pubinf} = S([gv]_{\Pi_U})).
\]

\( \text{T.snd} \) denotes the list obtained by projecting \( T \) to all messages sent from \( \Pi_U \).

\( \text{T.pre}(x) \) denotes the part of \( T \) from the beginning to before \( x \).

Informally, \( \varphi_1 \) means that we can always simulate the party oracle \( \Pi_U \). It says that we can always compute any sent message using the public information and the part before the message in the transcript, which also includes all DH public values generated before the message. On the other hand, \( \varphi_2 \) is just to keep track of the variables we can simulate. It says that all variables in \( gv \) can be computed without using any DH secret values but just the transcript, which also includes all DH public values and all public information. This is useful because when an oracle sends a message, it must be the current \( gv \) for the current DHSim continues to hold.

C. Link between the invariants and the security definition

As we have explained about the choice of the assertion language in the beginning of Section IV, we define our invariants so that they imply the security of analyzed protocols, i.e. a security reduction is guaranteed. Now we explain the link between the invariant and the reduction in more details.

Definition 7: A protocol \( \pi \) satisfies \( \varphi_1[c]_{\Pi_U} \varphi_2 \), written \( \pi \models \varphi_1[c]_{\Pi_U} \varphi_2 \) iff for any party oracle \( \Pi_U \) and any adversary \( E \) the following statement holds:

If \( \Pi_U \) starts executing \( c \) from a distribution of experiment states that satisfies \( \varphi_1 \) and after that \( \Pi_U \) is still fresh and not in the state \( \text{Rejected} \), then the distribution of new experiment states satisfies \( \varphi_2 \).

Lemma 1: If a distribution of experiment states satisfies \( \varphi \) which has \( \Pi_U \) as its first argument, then as long as \( \Pi_U \) does not execute any more commands while the adversary and other oracles are still running, the new distributions of experiment states always satisfy \( \varphi \).

Proof: We have five possible cases. If \( \varphi \) is one of DHSec(\( \Pi_U; x; bv \)), DHPubln(\( \Pi_U; y \)), DHSol(\( \Pi_U; z; y; x \))
or DHSim(Π\textsubscript{U}; gv), then \( \varphi \) is related to the local state of \( \Pi_i \) only. Therefore if the local state remains unchanged, then new distributions of experiment states still satisfy \( \varphi \).

Only if \( \varphi \) is DHPubOut(Π\textsubscript{U}; y), then \( \varphi \) is related to variables outside \( \Pi_i \). However, for \( \varphi \) to be satisfied we only require that: with non-negligible probability the variable \( y \) inside \( \Pi_i \) equals \( g^a \) where \( n \) is a nonce used in one instance of \( \Pi_i \)'s partner. And because values of variables are unchanged once they are created, DHPubOut(Π\textsubscript{U}; y) is still satisfied.

Now we are ready to define the main theorem of our work.

**Theorem 2:** Suppose we have a functional protocol \( \pi = (I(\text{iask}) : c_i, R(rsk) : c_r) \). If \( \pi \) satisfies the following statements:

- \( \{\text{DHSim}(\Pi_i^j; 0)\} [I(\text{iask}) : c_i |_{\Pi_i^j} \{\text{DHSim}(\Pi_i^j; gv) \} \land \text{DHSol}(\Pi_i^j; \text{iask}; y; x) \land \text{DHPubOut}(\Pi_i^j; y)] \) for some variables \( x, y \) and a set \( y \) of variables;
- \( \{\text{DHSim}(\Pi_i^j; 0)\} [R(\text{rsk}) : c_r |_{\Pi_i^j} \{\text{DHSim}(\Pi_i^j; gv) \} \land \text{DHSol}(\Pi_i^j; rsk; y'; x') \land \text{DHPubOut}(\Pi_i^j; y')] \) for some variables \( x', y' \) and a set \( y' \) of variables;
- then the cNR–mBR security of \( \pi \) is probabilistic polynomial time reducible to the hardness of the CDH.

**Proof:** Now we show that given the adversary \( E \) whose Advantage\textsubscript{E}(k) is non-negligible, we can construct an adversary \( D \) who can solve any CDH instance with non-negligible probability. Suppose that \( \pi \) gives the, \( g^a, g^b \) from the CDH instance, now \( D \) has to output \( g^{ab} \).

The intuition of the proof is as follows. We will inject \( g^a, g^b \) into two party oracles, where the second one is a partner instance of the first one. DHSim predicates guarantee that we can simulate them correctly. DHSol and DHPubOut guarantee that later on we can get \( g^{ab} \) from the session key of the first oracle (the session key will be given by the adversary as defined in the experiment).

The reduction works as follows. \( D \) runs the challenger \( C \) as defined. After all setting-up steps have been done, \( D \) chooses randomly a party oracle \( \Pi_i^j \), whose partner is \( V \). During the experiment, if \( \Pi_i^j \) becomes unfresh, then \( D \) gives up. But with a non-negligible probability \( \Pi_i^j \) remains fresh until the end of the experiment (because \( E \) must keep at least one oracle fresh). We assume that \( \Pi_i^j \) remains fresh until the end of the experiment. \( D \) decides randomly to simulate \( \Pi_i^j \) as an initiator or responder \(^3\). The probability that \( D \) chooses the correct role for \( \Pi_i^j \) is \( 1/2 \). Assume that \( D \) chooses correctly, there will be two cases here.

1. The first case happens with probability \( 1/2 \) where \( D \) chooses \( \Pi_i^j \) to be an initiator. \( \Pi_i^j \) executes the initiator code as normally with two exceptions. The first exception is that if \( \Pi_i^j \) executes as if \( x \) has the value \( a \), i.e. we use \( g^a \) when necessary and skip any command that requires \( a \). Meanwhile, notice that \( \pi = \{\text{DHSim}(\Pi_i^j; 0)\} [I(\text{iask}) : c_i |_{\Pi_i^j} \{\text{DHSim}(\Pi_i^j; gv) \}, i.e. there is an algorithm \( B_1 \) that can compute any sent message using the public information and the previous transcript. The second exception is that \( D \) uses \( B_1 \) to compute messages that are sent out (\( B_1 \) must have enough information to operate because we have \( g^a \), i.e. the previous transcript is always built correctly). On the other hand, \( D \) picks a random instance \( \Pi_i^j \), where \( V \) is \( \Pi_i^j \)'s partner. Then \( b \) replaces another nonce uniformly chosen in \( \Pi_i^j \). Note that we can also still compute any sent message from \( \Pi_i^j \) regardless that \( \Pi_i^j \) is an initiator or responder, because we also have \( \pi = \{\text{DHSim}(\Pi_i^j; 0)\} [R(\text{rsk}) : c_r |_{\Pi_i^j} \{\text{DHSim}(\Pi_i^j; gv) \} \). Overall, with an overwhelming probability, \( \Pi_i^j \) is simulated correctly where \( x \) has the value \( a \), and \( \Pi_i^j \) is also simulated correctly where one of its nonces is \( b \). It means that with a non-negligible probability, \( E \) will output the correct session key of the test oracle (because Advantage\textsubscript{E}(k) is non-negligible in the normal experiment).

Assume that \( E \) wins the experiment, the probability that \( E \) chooses \( \Pi_i^j \) to test is \( \frac{1}{2} \). Also, because the experiment is simulated correctly with an overwhelming probability and \( \pi = \{\text{DHSim}(\Pi_i^j; 0)\} [I(\text{iask}) : c_i |_{\Pi_i^j} \{\text{DHPubOut}(\Pi_i^j; y) \} \), \( y \) must have the value \( g^b \) with a non-negligible probability.

Assuming that \( E \) chooses \( \Pi_i^j \) to test, because \( \pi = \{\text{DHSim}(\Pi_i^j; 0)\} [I(\text{iask}) : c_i |_{\Pi_i^j} \{\text{DHPubOut}(\Pi_i^j; y) \} \), we can always have an algorithm \( B_2 \) that given \( S(\text{rsk}) \) and \( S(\text{iask}) \) output the value of \( S(y) \), which equals \( g^{ab} \) with a non-negligible probability. To sum up, we can extract the value \( g^{ab} \) with a non-negligible probability finally.

1. The second case happens also with probability 1/2 where \( D \) chooses \( \Pi_i^j \) to be a responder. Everything happens in the same way as in the first case.

It means that \( D \) can solve any CDH instance with non-negligible probability. That concludes the proof.

V. PROOF SYSTEM

So far we have explained what we want to know, but not how to find it. Although we know that if some invariants hold then our protocol is secure, we have not shown how to check if those invariants hold or not. In this section we develop a proof system, which allows us to verify the invariants. Firstly, we build a set of axioms, i.e. statements that are always true. Then, we show some inference rules, which allow us to reason on a protocol specification, i.e. find new knowledge using the set of axioms.
Our proof system follows the idea of Hoare logic which has been used to verify sequential programs. Note that party oracles execute their sequential code locally without any shared variable, but they communicate via a network. Our logic is sound even when we verify each party specification separately, i.e. an invariant for a party oracle can be established no matter what the adversary and other oracles do. This is important because it makes reasoning simple. However, since a protocol is an agreement between two parties, if we verify each one without considering the other, what the logic can verify is limited. In Section VI we will discuss the solution for this problem.

A. Axioms

Table IV presents a set of axioms, which tell us how the experiment state changes after a command is executed. They do not depend on any specific protocol. The axioms follow the form $\varphi_1[c]\Pi_i\varphi_2$, from Definition 7, where $c$ is only one command.

**Lemma 2:** The axioms given in Table IV are sound.

*Proof:* Proofs are given in Appendix A. ■

B. Proof rules

The following Sequential Composition rule can be used for reasoning using the axioms above.

$\varphi_1[c_1]c_2\varphi_2 \vdash \varphi_1[c_1]c_2\varphi_3$

**Lemma 3:** The Sequential Composition rule is sound.

*Proof:* Because $\{\varphi_1\}[c_1]\Pi_i\{\varphi_2\}$ and $\{\varphi_2\}[c_2]\Pi_i\{\varphi_3\}$ hold, we only have to show that after $\Pi_i$ executes $c_1$, and stops in a distribution of states that satisfies $\varphi_2$, the distribution of states continues to satisfy $\varphi_2$ until $\Pi_i$ starts to execute $c_2$. This is true according to Lemma 1. ■

**Example 2:** The proof outline for $\pi$ in Example 1 is presented as follows (for readability, unnecessary predicates are removed).

**Initiator:**

\[
\begin{align*}
I(isk) : & \\
ir := mk\text{DHSec}(); & \quad \mathcal{R}(rsk) : \\
igr := mk\text{DHPub}(ir) ; & \quad \text{receive}(igr) ; \\
send(ir) ; & \quad \text{send}(igr) ; \\
receive(rg1) ; & \quad \text{receive}(rsk) ; \\
isk := mk\text{DHKey}(rsk1, ir) ; & \quad \text{mkDHSec}(igr1, rsk) ; \\
ir := mk\text{DHPub}(ir) ; & \quad \text{mkDHKey}(igr1, ir) ; \\
igr := mk\text{DHSec}(igr1) ; & \quad \text{DHPubOut}(igr1, ip) \wedge \text{DHSec}(igr1, ir, \{ir, isk\}) \\
\end{align*}
\]

\[\begin{align*}
\mathcal{I}(isk) : & \\
ir := mk\text{DHSec}(); & \quad \mathcal{R}(rsk) : \\
igr := mk\text{DHPub}(ir) ; & \quad \text{receive}(igr) ; \\
send(ir) ; & \quad \text{send}(igr) ; \\
receive(rg1) ; & \quad \text{receive}(rsk) ; \\
isk := mk\text{DHKey}(rsk1, ir) ; & \quad \text{mkDHSec}(igr1, rsk) ; \\
ir := mk\text{DHPub}(ir) ; & \quad \text{mkDHKey}(igr1, ir) ; \\
igr := mk\text{DHSec}(igr1) ; & \quad \text{DHPubOut}(igr1, ip) \wedge \text{DHSec}(igr1, ir, \{ir, isk\}) \\
\end{align*}\]

\[\begin{align*}
\mathcal{I}(isk) : & \\
ir := mk\text{DHSec}(); & \quad \mathcal{R}(rsk) : \\
igr := mk\text{DHPub}(ir) ; & \quad \text{receive}(igr) ; \\
send(ir) ; & \quad \text{send}(igr) ; \\
receive(rg1) ; & \quad \text{receive}(rsk) ; \\
isk := mk\text{DHKey}(rsk1, ir) ; & \quad \text{mkDHSec}(igr1, rsk) ; \\
ir := mk\text{DHPub}(ir) ; & \quad \text{mkDHKey}(igr1, ir) ; \\
igr := mk\text{DHSec}(igr1) ; & \quad \text{DHPubOut}(igr1, ip) \wedge \text{DHSec}(igr1, ir, \{ir, isk\}) \\
\end{align*}\]
<table>
<thead>
<tr>
<th>Description</th>
<th>Rule ID</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonce generation</td>
<td>A_NG1</td>
<td>{true}[x := \text{mkDHSec}(\Pi_i); \text{DHSec}(\Pi_i); x; {x}]</td>
</tr>
<tr>
<td></td>
<td>A_NG2</td>
<td>{\varphi}[x := \text{mkDHSec}(\Pi_i); \varphi] if \varphi is a DHPubln, DHPubOut, DHSol, DHSim predicate, or DHSec(\Pi_i'; x'; bv) where x' is not x.</td>
</tr>
<tr>
<td>DH public value generation</td>
<td>A_PVG1</td>
<td>{\text{DHSec}(\Pi_i); x; bv}[y := \text{mkDHPub}(x)[\Pi_i]; {\text{DHPubln}(\Pi_i); y \land \text{DHSec}(\Pi_i); x; bv}]</td>
</tr>
<tr>
<td></td>
<td>A_PVG2</td>
<td>{\text{DHSim}(\Pi_i); gv}[y := \text{mkDHPub}(x)[\Pi_i]; {\text{DHSim}(\Pi_i); gv \cup {y}}]</td>
</tr>
<tr>
<td></td>
<td>A_PVG3</td>
<td>{\varphi}[y := \text{mkDHPub}(x)[\Pi_i]; \varphi] if \varphi is a DHSec, DHPubln, DHPubOut or DHSol predicate</td>
</tr>
<tr>
<td>DH shared key generation</td>
<td>A_KG1</td>
<td>{\text{DHSec}(\Pi_i); x; bv}[z := \text{mkDHKey}(y, x)[\Pi_i]; {\text{DHSec}(\Pi_i); x; bv \cup {z}} \land \text{DHSol}(\Pi_i); y; x}]</td>
</tr>
<tr>
<td></td>
<td>A_KG2</td>
<td>{\text{DHSim}(\Pi_i); gv}[z := \text{mkDHKey}(y, x)[\Pi_i]; {\text{DHSim}(\Pi_i); gv \cup {z}}] if x, y \in \text{gv}</td>
</tr>
<tr>
<td></td>
<td>A_KG3</td>
<td>{\varphi}[z := \text{mkDHKey}(y, x)[\Pi_i]; \varphi] if \varphi is a DHPubln, DHPubOut, DHSol, DHSec(\Pi_i'; x'; bv) predicate, where x' is not x.</td>
</tr>
<tr>
<td>Static pre-computed DH keys</td>
<td>A_GSK1</td>
<td>{true}[y, x := \text{getStaticDHKey}(\Pi_i); {\text{DHSec}(\Pi_i); x; {x}} \land \text{DHPubOut}(\Pi_i); y}]</td>
</tr>
<tr>
<td></td>
<td>A_GSK2</td>
<td>{\varphi}[y, x := \text{getStaticDHKey}(\Pi_i); {\text{DHSim}(\Pi_i); gv \cup {y}}]</td>
</tr>
<tr>
<td></td>
<td>A_GSK3</td>
<td>{\varphi}[y, x := \text{getStaticDHKey}(\Pi_i); \varphi] if \varphi is a DHPubln, DHPubOut, DHSol or DHSec(\Pi_i'; x'; bv) where x' is not x. predicate</td>
</tr>
<tr>
<td>oracles ID</td>
<td>A_GID1</td>
<td>{\text{DHSim}(\Pi_i); gv}[x, y := \text{getIDs}(\Pi_i); {\text{DHSim}(\Pi_i); gv \cup {x, y}}]</td>
</tr>
<tr>
<td></td>
<td>A_GID2</td>
<td>{\varphi}[x, y := \text{getIDs}(\Pi_i); \varphi] if \varphi is a DHSec, DHPubln, DHPubOut or DHSol predicate</td>
</tr>
<tr>
<td>Signing</td>
<td>A_SGN1</td>
<td>{\text{DHSim}(\Pi_i); gv}[x := \text{sign}(1)[\Pi_i]; {\text{DHSim}(\Pi_i); gv \cup {x}}] if 1 \subseteq \text{gv}</td>
</tr>
<tr>
<td></td>
<td>A_SGN2</td>
<td>{\text{DHSim}(\Pi_i); gv}[x := \text{sign}(1)[\Pi_i]; {\text{DHSim}(\Pi_i); gv}] if 1 \not\subseteq \text{gv}</td>
</tr>
<tr>
<td></td>
<td>A_SGN3</td>
<td>{\text{DHSec}(\Pi_i); y; bv}[x := \text{sign}(1)[\Pi_i]; {\text{DHSec}(\Pi_i); y; bv \cup {x}}] if 1 \cap \text{bv} \neq \emptyset</td>
</tr>
<tr>
<td></td>
<td>A_SGN4</td>
<td>{\text{DHSec}(\Pi_i); y; bv}[x := \text{sign}(1)[\Pi_i]; {\text{DHSec}(\Pi_i); y; bv}] if 1 \cap \text{bv} = \emptyset</td>
</tr>
<tr>
<td></td>
<td>A_SGN5</td>
<td>{\varphi}[x := \text{sign}(1)[\Pi_i]; \varphi] if \varphi is a DHPubln, DHPubOut or DHSol predicate</td>
</tr>
<tr>
<td>Signature verification</td>
<td>A_VFS1</td>
<td>{\varphi}[\text{verisign}(1, x)[\Pi_i]; \varphi] if \varphi is a DHSec, DHPubln, DHPubOut, DHSol or DHSim predicate</td>
</tr>
<tr>
<td>Concatenation</td>
<td>A_CON1</td>
<td>{\text{DHSol}(\Pi_i); u; v}[x := \text{concat}(x, y)[\Pi_i]; {\text{DHSol}(\Pi_i); z; u; v} \land \text{DHSol}(\Pi_i); w; u; v}] if u is x or y</td>
</tr>
<tr>
<td></td>
<td>A_CON2</td>
<td>{\text{DHSol}(\Pi_i); u; v}[x := \text{concat}(x, y)[\Pi_i]; {\text{DHSol}(\Pi_i); w; u; v}] if u is not x and y</td>
</tr>
<tr>
<td></td>
<td>A_CON3</td>
<td>{\text{DHSim}(\Pi_i); gv}[x := \text{concat}(x, y)[\Pi_i]; {\text{DHSim}(\Pi_i); gv \cup {z}}] if x and y \in \text{gv}</td>
</tr>
<tr>
<td></td>
<td>A_CON4</td>
<td>{\text{DHSim}(\Pi_i); gv}[x := \text{concat}(x, y)[\Pi_i]; {\text{DHSim}(\Pi_i); gv}] if x or y \not\in \text{gv}</td>
</tr>
<tr>
<td></td>
<td>A_CON5</td>
<td>{\text{DHSec}(\Pi_i); x'; bv}[x := \text{concat}(x, y)[\Pi_i]; {\text{DHSec}(\Pi_i); x'; bv \cup {z}}] if x or y \in \text{bv}</td>
</tr>
<tr>
<td></td>
<td>A_CON6</td>
<td>{\text{DHSec}(\Pi_i); x'; bv}[x := \text{concat}(x, y)[\Pi_i]; {\text{DHSec}(\Pi_i); x'; bv}] if x and y \not\in \text{gv}</td>
</tr>
<tr>
<td></td>
<td>A_CON7</td>
<td>{\varphi}[x := \text{concat}(x, y)[\Pi_i]; \varphi] if \varphi is a DHPubln, DHPubOut predicate</td>
</tr>
<tr>
<td>Addition or Multiplication</td>
<td>A_COM1</td>
<td>{\text{DHSol}(\Pi_i); u; v} \land \text{DHSec}(\Pi_i); v; bv}[z := c(x, y)[\Pi_i]; {\text{DHSol}(\Pi_i); z; u; v} \land \text{DHSol}(\Pi_i); w; u; v}] if u is x and y \not\in \text{bv}, or u is y and x \not\in \text{bv}</td>
</tr>
<tr>
<td></td>
<td>A_COM2</td>
<td>{\text{DHSol}(\Pi_i); u; v}[z := c(x, y)[\Pi_i]; {\text{DHSol}(\Pi_i); w; u; v}] if u is not x and y</td>
</tr>
<tr>
<td></td>
<td>A_COM3</td>
<td>{\text{DHSim}(\Pi_i); gv}[z := c(x, y)[\Pi_i]; {\text{DHSim}(\Pi_i); gv \cup {z}}] if x and y \in \text{gv}</td>
</tr>
<tr>
<td></td>
<td>A_COM4</td>
<td>{\text{DHSim}(\Pi_i); gv}[z := c(x, y)[\Pi_i]; {\text{DHSim}(\Pi_i); gv}] if x or y \not\in \text{gv}</td>
</tr>
<tr>
<td></td>
<td>A_COM5</td>
<td>{\text{DHSec}(\Pi_i); x'; bv}[z := c(x, y)[\Pi_i]; {\text{DHSec}(\Pi_i); x'; bv \cup {z}}] if x or y \in \text{bv}</td>
</tr>
<tr>
<td></td>
<td>A_COM6</td>
<td>{\text{DHSec}(\Pi_i); x'; bv}[z := c(x, y)[\Pi_i]; {\text{DHSec}(\Pi_i); x'; bv}] if x and y \not\in \text{gv}</td>
</tr>
<tr>
<td></td>
<td>A_COM7</td>
<td>{\varphi}[z := \text{concat}(x, y)[\Pi_i]; \varphi] if \varphi is a DHPubln, DHPubOut predicate</td>
</tr>
<tr>
<td>Sending</td>
<td>A_SND1</td>
<td>{\text{DHSim}(\Pi_i); gv}[\text{send(sentvars)}[\Pi_i]; {\text{DHSim}(\Pi_i); gv}] if sentvars \subseteq \text{gv}</td>
</tr>
<tr>
<td></td>
<td>A_SND2</td>
<td>{\varphi}[\text{send(sentvars)}[\Pi_i]; \varphi] if \varphi is a DHSec, DHPubln, DHPubOut or DHSol predicate</td>
</tr>
<tr>
<td>Receiving</td>
<td>A_RCV1</td>
<td>{\text{DHSim}(\Pi_i); gv}[\text{receive(receivedvars)}[\Pi_i]; {\text{DHSim}(\Pi_i); gv \cup \text{receivedvars}}]</td>
</tr>
<tr>
<td></td>
<td>A_RCV2</td>
<td>{\varphi}[\text{receive(receivedvars)}[\Pi_i]; \varphi] if \varphi is a DHSec, DHPubln, DHPubOut or DHSol predicate</td>
</tr>
</tbody>
</table>

Table IV

The list of axioms of our logic
A DHPubOut predicate cannot be established because no party receives an authenticated public DH value. In fact, the DH protocol is vulnerable to a man-in-the-middle attack.

**Remark 3:** Our logic is sound but not complete, since the axioms cannot cover all possible cases. Thus, if the logic cannot show the security of a protocol, it does not mean the protocol is insecure.

### VI. Improvement by Syntactic Checking

Notice that using our logic, we can only be sure that a DH public value really belongs to an honest oracle if it is a public key. Therefore, a predicate DHPubOut may not be established just by using the logic. Our idea to solve this problem is that, if a DH public value is signed and verified in a way that we can always match, then we can be sure that it is from an honest oracle. We borrow this idea from the way Hoare logic has been improved to verify communicating sequential process [7], [8], [9].

Now we describe our syntactic checking algorithm. Given a list \( l \) of variables, let \( l[t] \) denote the \( t \)th variable in \( l \) (starting from 1). Also, by saying that the value of a list \( l \) equals the value of another list \( l' \), we mean they have the same number of elements and the value of elements in the same position are equal. Given a protocol \( \pi = (\mathcal{I}(\text{isk}), \mathcal{R}(\text{rsk})) \), two numbers \( s \) and \( p \), the checking procedure \( \text{syncheck}(\pi, s, p) \) runs as follows:

- in every list \( l \) that is the input of sign command in \( \mathcal{I}(\text{isk}) \) and has size \( s \), get \( l[p] \);
- if for every such variable \( l[t] \), we have \( \pi \models \{ \mathcal{DHSim}(\Pi_{l'}, \emptyset) \} \mathcal{I}(\text{isk}) : c_{l_{l'}} \{ \mathcal{DHPubIn}^{\Pi_{l'}} ; 1[l[t]] \} \), then continue, otherwise return false;
- repeat the above process with \( \mathcal{R}(\text{rsk}) \) instead of \( \mathcal{I}(\text{isk}) \);
- return true.

**Theorem 3:** Suppose we have a protocol \( \pi = (\mathcal{I}(\text{isk}) : c_{l}, \mathcal{R}(\text{rsk}) : c_{r}) \). Given two arbitrary numbers \( s \) and \( p \), if \( \text{syncheck}(\pi, s, p) \) returns true, then \( \pi \models \{ \text{true} \} c_{l} ; \text{verisign}(l', s) ; c_{2}^{\Pi_{l'}} \{ \mathcal{DHPubOut}(\Pi_{l'} ; 1[p]) \} \) holds for any \( c_{l}, c_{2} \), and the list \( l' \), whose size is \( s \).

**Proof:** The intuitive idea of this theorem is as follows. When a fresh party oracle receives a message with a valid signature, the message must really be from one instance of its partner. However, the message may have been signed by an initiator or responder (the adversary is malicious). In both cases, if there is one message element at a fixed position that is a DH public value, then the counterpart in the received message must also be a DH public value. It helps us to establish a DHPubOut predicate.

Now we describe the proof in more details. Suppose that \( \Pi_{l'} \) has executed the code \( c_{l} ; \text{verisign}(l', s) ; c_{2} \) and is still fresh. Let \( V \) be the partner of \( \Pi_{l'} \). Notice that we assume our signature scheme is EF-CMA secure.

Firstly, we will show that with overwhelming probability the value of \( l' \) equals a list signed by an instance of \( V \). Assume that it is not correct, i.e. with a non-negligible probability \( l' \) equals no list signed by an instance of \( V \). Now we can construct an algorithm \( B \) that can win the EF-CMA game with our signature. We do the experiment as normally with its adversary \( E \) except that we use the signing oracle to sign anything that any instance of \( V \) wants to sign. After \( \Pi_{l'} \) has executed the code \( c_{l} ; \text{verisign}(l', s) ; c_{2} \), \( s \) must be a valid signature for \( l' \). However, with a non-negligible probability \( l' \) equals no list signed by an instance of \( V \), i.e. signed by the signing oracle. Therefore \( B \) wins the EF-CMA game which is a contradiction.

Now assume that with overwhelming probability the value of \( l' \) equals a list signed by an instance of \( V \). Because to be equal, two lists must have the same number of elements, we consider only a set \( sl \) of signed lists that have the same size with \( l' \). According to the condition, for every element \( e \) at a position \( p \) in every list in \( sl \), the value of \( e \) must equal a DH public value generated by the party oracle that contains \( e \). Thus, the value of \( l'[p] \) must equal a DH public value generated in one instance of \( V \). It means that \( \pi \models \{ \text{true} \} c_{l} ; \text{verisign}(l', s) ; c_{2}^{\Pi_{l'}} \{ \mathcal{DHPubOut}(\Pi_{l'} ; 1[p]) \} \).

**Example 4:** This technique can be applied for a modified STS protocol [20] specified in Table VI. Using the proof system in the Section V, we conclude:

- \( \pi_{MSTS} \models \{ \mathcal{DHSim}(\Pi_{l'} ; 0) \} \mathcal{I}(\text{isk}) : c_{l}^{\Pi_{l'}} \{ \mathcal{DHSim}(\Pi_{l'} ; \{ \text{igr}, \text{rgr1}, \text{rsl}, \text{oid}, \text{ipid}, \text{is} \}) \} \mathcal{DHSim}(\Pi_{l'} ; \text{isk}; \text{igr1}; \text{ir}) \mathcal{DHPubIn}(\Pi_{l'} ; \text{igr}) \} \)
- \( \pi_{MSTS} \models \{ \mathcal{DHSim}(\Pi_{l'} ; 0) \} \mathcal{R}(\text{rsk}) : c_{l}^{\Pi_{l'}} \{ \mathcal{DHSim}(\Pi_{l'} ; \{ \text{igr}, \text{rgr}, \text{roid}, \text{rapid}, \text{rs}, \text{isl} \}) \} \mathcal{DHSim}(\Pi_{l'} ; \text{rsk}; \text{igr1}; \text{rr}) \mathcal{DHPubIn}(\Pi_{l'} ; \text{rgr}) \} \)

These statements are not enough to conclude security because some DHPubOut predicates are missing. Now we try to establish them. By syntactically checking, we find that \( \{ \text{oid}, \text{ipid}, \text{igr} \} \) and \( \{ \text{roid}, \text{rapid}, \text{rgr} \} \) are the only lists signed in the initiator and responder codes respectively. Also, the sizes of these lists are 3, and \( \text{igr} \) and \( \text{rgr} \) are
in the position 3 in each list. Furthermore, $\pi_{mSTS}$ satisfies 
\{DHSim($\Pi_i^\prime$; $\emptyset$), $\{Z(\text{isk}) : c_1; \Pi_i^\prime\}$, $\{\text{DHPubl}($\Pi_i^\prime$; $\text{igr})\}$
\{DHSim($\Pi_i^\prime$; $\emptyset$), $\{R(\text{rsk}) : c_r; \Pi_i^\prime\}$, $\{\text{DHPubl}($\Pi_i^\prime$; $\text{rgr})\}$
also. According to Theorem 3, we have $\pi_{mSTS}$ satisfies
\{true\}$c_1; \text{verisign}(l^\prime, s); c_2; \Pi_i^\prime\}$, $\{\text{DHPublOut}($\Pi_i^\prime$; $l^\prime$'[3])}
holds for any $c_1$ and $c_2$, and $l^\prime$, whose size is 3. Fortunately, there is a command $\text{verisign}(\text{ipid}, \text{oid}, \text{rgr}, \text{rsl}; 1)$;
in $\{Z(\text{isk}) : c_1$ and $\text{verisign}(\text{rpid}, \text{roid}, \text{igr}, 1)$; in $\{R(\text{rsk}) : c_r$. Thus, finally we have
- $\pi_{mSTS} = \{\text{DHSim}($\Pi_i^\prime$; $\emptyset$), $\{Z(\text{isk}) : c_1; \Pi_i^\prime\}$
- $\{\text{DHSim}($\Pi_i^\prime$; $\{\text{igr}, \text{rgr}, \text{rsl}, \text{oid}, \text{ipid}, \text{is})\}$
- $\{\text{DHSol}($\Pi_i^\prime$; $\text{isk}, \text{rgr}, 1\}i\}$
- $\{\text{DHPublOut}($\Pi_i^\prime$; $\text{rgr})\}$. 

**Conclusion:** According to Theorem 2, these results imply that the security of $\pi_{mSTS}$ in the cNR-mBR model is reducible to the CDH problem. Furthermore, according to Theorem 1, the security of the compiled version of $\pi_{mSTS}$ is reducible to the GDH problem.

**Remark 4:** We are not aware of any security proof for the original protocol $\pi_{mSTS}$ in the BR model, although it has been done with the CK model [21]. However, notice that the compiled version of $\pi_{mSTS}$ here is a bit different from the original one, since the partnering information is added and the session key is hashed.

**VII. IMPLEMENTING AN AUTOMATED TOOL**

The automatic verification procedure presented in this paper has been implemented. Given two algorithms, including the initiator and responder codes, we check if the conditions are met or not by reasoning on the Hoare-style logic forwardly. After reasoning with the logic, the tool will conduct the syntactic checking to make the final conclusion.

Reasoning on Hoare-style logics can be fully automated. Also, our syntactic check can be implemented easily. Thus, the tool is fully automated, i.e., without any guidance. For an algorithm, the verification is likely exponential in the number of commands. However, a typical protocol has approximately ten lines of codes, so the verification can be done very efficiently. The output is printed in LHlXand compiled to a pdf file. It makes the proof readable and checkable by human. The tool is available in the author’s website: http://www.isi.qut.edu.au/research/projects/dhverif.zip.

Notice that the tool can only produce security proofs in the cNR-mBR model. Security proofs in the BR model is then established by the modular proof. The tool has been tested with KEA [22], Modified STS [20], MTI A(0) [23], Unified Model [24], ISO-9798-3 [25] and “no hashing” version of protocol 3 and 4 of [19]. We believe there are still more suitable protocols that can be tested.

**VIII. CONCLUSION AND FUTURE WORK**

We have developed a method for verifying security of Diffie–Hellman-based key exchange protocols. The method includes a Hoare-style logic and an additional syntactic check. The logic is computationally sound but not complete. We also implemented the tool and tested it successfully. Although our work focused only on Diffie–Hellman key exchange in the BR model, the method can be extended to a larger class of protocols or to a different model.

There are some future directions to extend the work. Firstly, we may cover more Diffie–Hellman–based protocols like Naxos [17] or HMQV [18], which use Diffie–Hellman schemes in a different way that is not considered here. This direction may also include extension of commands, e.g., allowing parties to hash, XOR, etc. Secondly, the method can be extended to other types of key exchange, such as RSA–based ones, because the modular proof of Kudla and Paterson [6] is still applicable in that case. Thirdly, the syntactic checking can be improved, e.g., involving type information. Fourthly, another modular proofs could be considered, e.g., CK model [21]. Finally, the predicate could be revised in order to have tighter proofs or even allow exact security, although it would reduce the simplicity of the logic.

**Acknowledgements:** We thank Gilles Barthe and Yassine Lakhnech for helping to correct a number of errors and omissions. This research was partially funded by the Australian Research Council through Discovery Project DP0773348.


APPENDIX A

THE SOUNDNESS OF THE AXIOMS

We prove the soundness of axioms. Due to the obviousness of most of axioms, we only sketch the proofs.

1) Nonce generation:

• (A_NG1) \( \{ \text{true} \} [x := \text{mkDHSec}(\Pi_i); \text{DHSec}(\Pi_i; x; \{ x \})] \)

Proof: It is obvious because the semantics of the command mkDHSec(), which generates a new nonce for x. And also, because x has just been created, x is the only variable that cannot be computed without knowing x.

• (A_NG2) \( \{ \varphi \} [x := \text{mkDHSec}(\Pi_i); \{ \varphi \} ] \text{ if } \varphi \text{ is a DHPubIn, DHPubOut, DHSol, DHSim predicate, or DHSec}(\Pi_i; x'; bv) \text{ where } x' \text{ is not x.} \)

Proof: Because x is a new variable, this command cannot affect \( \varphi \) if it is a DHPubIn, DHPubOut or DHSol predicate. If \( \varphi = \text{DHSec}(\Pi_i; x'; bv) \) where \( x' \) is not x, then \( \varphi \) still holds because x is still computable when only \( x' \) is missing. On the other hand, this command does not send out any message, therefore it cannot affect any DHSim predicate.

2) DH public value generation:

• (A_PVG1) \( \{ \text{DHSec}(\Pi_i; x; bv) \} [y := \text{mkDHPub}(\Pi_i); \{ \text{DHPubIn}(\Pi_i; y) \wedge \text{DHSec}(\Pi_i; x; bv) \}] \)

Proof: DHSec(\Pi_i; x; bv) continues to hold because this command just generates a new variable, and \( y \notin bv \) because the value of y must be recorded into T according to the semantics of the command. On the other hand, DHPubIn(\Pi_i; y) holds according to the semantics of the command and the predicate.

• (A_PVG2) \( \{ \text{DHSim}(\Pi_i; gv) \} [y := \text{mkDHPub}(\Pi_i); \{ \text{DHSim}(\Pi_i; gv \cup \{ y \}) \}] \)

Proof: This command does not send out any message, therefore the is no problem with computing any sent message. In addition, the value of y must be recorded into T according to the semantics of the command, so y must be added into gv.

• (A_PVG3) \( \{ \varphi \} [y := \text{mkDHPub}(\Pi_i; \{ \varphi \} ) \text{ if } \varphi \text{ is a DHSec, DHPubIn, DHPubOut or DHSim predicate.} \)
Proof: Notice that y is a new variable, therefore the command cannot affect any DHPubn, DHPubOut, DHSol predicate. In addition, y cannot be in any set bv in any DHSec($\Pi_i^j; x; \{bv\}$) predicate, because its value is a DH public value. Thus, the command cannot affect any DHSec either.

3) DH key generation:

- (A_KG1) 
  \{DHSec($\Pi_i^j; x; bv$)\}$z := mkDHKey(y, x)$$\}\Pi_i^j$
  \{DHSec($\Pi_i^j; x; bv \cup \{z\}$) $\land$ DHSol($\Pi_i^j; z; y; x$)\}

  Proof: Because x must be in bv, z must be added into bv, i.e. z cannot be computed without x. And DHSol($\Pi_i^j; z; y; x$) holds according to the semantics of the command and the definition of the predicate.

- (A_KG2) 
  \{DHSim($\Pi_i^j; gv$)\}$z := mkDHKey(y, x)$$\}\Pi_i^j$
  \{DHSim($\Pi_i^j; gv \cup \{z\}$) $\land$ DHSol($\Pi_i^j; gv \cup \{z\}$) $\land$ DHSol($\Pi_i^j; z; y; x$)\}

  Proof: DHSim($\Pi_i^j; gv \cup \{z\}$) holds because this command does not send out any message, and when x, y $\in$ gv then z must be added into gv (since x, y contain enough information for computing z).

- (A_KG3) \{\phi\}$z := mkDHKey(y, x)$$\}\Pi_i^j$ \{\phi$\}$ if $\phi$ is a DHPubn, DHPubOut, DHSol, DHSim($\Pi_i^j; gv$) predicate, where x $\notin$ gv or DHSec($\Pi_i^j; x'; bv$), where x' is not x.

  Proof: Since z is a new variable, any DHPubn, DHPubOut, DHSol predicate is not affected. Because this command does not send out any message, and x $\notin$ gv, DHSim($\Pi_i^j; gv$) still holds without changing its gv. If $\phi$ is a DHSec($\Pi_i^j; x'; bv$), where x' is not x, then DHSec($\Pi_i^j; x'; bv$) is not affected because x' is independent from x.

4) Getting the static public and secret key:

- (A_GSK1) \{true\}$y, x := getStaticDHKey()$$\}\Pi_i^j$ \{DHSec($\Pi_i^j; x; \{x\}$) $\land$ DHPubOut($\Pi_i^j; y$)\}

- (A_GSK2) 
  \{DHSim($\Pi_i^j; gv$)\}$y, x := getStaticDHKey()$$\}\Pi_i^j$
  \{DHSim($\Pi_i^j; gv \cup \{y\}$)\}

- (A_GSK3) \{\phi\}$y, x := getStaticDHKey()$$\}\Pi_i^j$ \{\phi$\}$ if $\phi$ is a DHPubn, DHPubOut, DHSol or DHSec($\Pi_i^j; x'; bv$) where x' is not x predicate.

  Proof: Proofs for these axioms are similar to proofs for nonce generation. However, in this case we establish a new predicate DHPubOut($\Pi_i^j; y$) because the value of y is actually the public key of $\Pi_i^j$'s partner.

5) Getting IDs:

- (A_GID1) 
  \{DHSim($\Pi_i^j; gv$)\}$y, x := getIDs()$$\]\Pi_i^j$
  \{DHSim($\Pi_i^j; gv \cup \{x, y\}$)\}

6) Signing:

- (A_SGN1) 
  \{DHSim($\Pi_i^j; gv$)\}$y := \sign(1)$$\}\Pi_i^j$ \{DHSim($\Pi_i^j; gv \cup \{x\}$)\}

  Proof: Since the IDs of the oracle and its partner are in pubinf, DHSim still holds, but x, y are added into gv.

- (A_GID2) \{\phi\}$y, x := getIDs()$$\}\Pi_i^j$ \{\phi$\}$ if $\phi$ is a DHSec, DHPubn, DHPubOut or DHSol predicate.

  Proof: Because this command assigns values to new variables from pubinf, it does not affect any DHSec, DHPubn, DHPubOut or DHSol predicate.

- (A_SGN2) 
  \{DHSim($\Pi_i^j; gv$)\}$y := \sign(1)$$\}\Pi_i^j$ \{DHSim($\Pi_i^j; gv \cup \{x\}$)\} if 1 $\subseteq$ gv.

  Proof: DHSim continues to hold because the command does not send any message. In addition, if 1 $\subseteq$ gv then x can also be computed without the generate nonces.

- (A_SGN3) 
  \{DHSim($\Pi_i^j; y; bv$)\}$y := \sign(1)$$\}\Pi_i^j$ \{DHSim($\Pi_i^j; y; bv \cup \{x\}$)\} if 1 $\subseteq$ bv.

  Proof: Like the proof above, DHSim still holds. But in this case we cannot guarantee to be able to compute x.

- (A_SGN4) 
  \{DHSec($\Pi_i^j; y; bv$)\}$y := \sign(1)$$\}\Pi_i^j$ \{DHSec($\Pi_i^j; y; bv \cup \{x\}$)\} if 1 $\subseteq$ bv.

  Proof: This command just creates a new variable x, so DHSec still holds. However, if there is any variable in 1 is in bv, then we do not know how to compute x, i.e. x is added into bv.

- (A_SGN5) \{\phi\}$y := \sign(1)$$\}\Pi_i^j$ \{\phi$\}$ if $\phi$ is a DHPubn, DHPubOut or DHSol predicate.

  Proof: This command just creates a new variable x, so it does not affect any DHPubn, DHPubOut or DHSol predicate.

7) Verifying signature:

- (A_VFS1) \{\phi\}$\{\versign(1, x)\}$$\}\Pi_i^j$ \{\phi$\}$ if $\phi$ is a DHSec, DHPubn, DHPubOut, DHSol or DHSim predicate.

  Proof: The only one variable that this command may affect is rejected, which has been created in the beginning. Therefore, any $\phi$ continue to hold with no change.

8) Concatenation:

- (A_CON1) 
  \{DHSol($\Pi_i^j; w; u; v$)\}$z := \concat(x, y)$$\]\Pi_i^j$ \{DHSol($\Pi_i^j; z; u; v$) $\land$ DHSol($\Pi_i^j; w; u; v$) if w is x or y\}

  Proof: DHSol($\Pi_i^j; w; u; v$) still holds because this command does not affect any of its variables. But we have a new predicate: DHSol($\Pi_i^j; z; u; v$), because given z we can always extract w with an overwhelming probability.
9) Addition or Multiplication: In the following axioms, the command c can be add or multiply.

(A_COM1) \{DHSol(Π_j^i; v; u; v) \land DHSec(Π_j^i; v; bv)\}[z := c(x, y)]_{Π_j^i} \{DHSol(Π_j^i; v; u; v)\}
if w is x and y \notin bv, or w is y and x \notin bv.

(A_COM2) \{DHSol(Π_j^i; w; u; v)\}[z := c(x, y)]_{Π_j^i} \{DHSec(Π_j^i; w; u; v)\}
if w is not x and y.

(A_COM3) \{DHSim(Π_j^i; gv)\}[z := c(x, y)]_{Π_j^i} \{DHSim(Π_j^i; gv)\}
if x and y \in gv.

(A_COM4) \{DHSim(Π_j^i; gv)\}[z := c(x, y)]_{Π_j^i} \{DHSim(Π_j^i; gv)\}
if x or y \notin gv.

(A_COM5) \{DHSec(Π_j^i; x'; bv)\}[z := c(y, x)]_{Π_j^i} \{DHSec(Π_j^i; x'; bv \cup \{z\})\}
if x or y \in bv.

(A_COM6) \{DHSec(Π_j^i; x'; bv)\}[z := c(y, x)]_{Π_j^i} \{DHSec(Π_j^i; x'; bv)\} if x
and y \notin bv.

(A_COM7) \{ϕ\}[z := concat(x, y)]_{Π_j^i} \{ϕ\} if ϕ is a
DHPubIn, DHPubOut predicate.

Proof:
The proofs are similar to the proofs for concatenation. However, it is a bit more complex with the axiom A_COM1.
In this axiom, if DHSec(Π_j^i; x; u; v) holds, then we need to be able to compute y without knowing v. The reason is that it helps to extract x from z. Notice that if z := c(x, y), there exists a PPT algorithm to extract x given z and y. In the case of addition, it is simple by subtracting y from z. In the case of multiplication, we can do it by computing the inverse of y.

10) Sending:

(A_SND1) \{DHSim(Π_j^i; gv)\}[send(sentvars)]_{Π_j^i} \{DHSim(Π_j^i; gv)\}
if sentvars \subset gv

Proof: The proof works inductively. A DHSim holds only when we can compute any sent message using the public information and the part before the message in the transcript. For any sent message before this command, it is possible because DHSim(Π_j^i; gv) holds for the state distribution before this command. Because sentvars \subset gv, everything in sentvars can also be computed as well.

(A_SND2) \{ϕ\}[send(sentvars)]_{Π_j^i} \{ϕ\} if ϕ is a DHSec, DHPubIn, DHPubOut or DHSol predicate.

Proof: Since the command sends information out and does not change any variable, any DHSec, DHPubIn, DHPubOut or DHSol predicate is not affected.

11) Receiving:

(A_RCV1) \{DHSim(Π_j^i; gv)\}[receive(receivedvars)]_{Π_j^i}
\{DHSim(Π_j^i; gv \cup receivedvars)\}

Proof: Because Π_j^i just receives more information, the DHSim still holds, but gv must be extended to include receivedvars.

(A_RCV2) \{ϕ\}[receive(receivedvars)]_{Π_j^i} \{ϕ\} if ϕ is a DHSec, DHPubOut or DHSol predicate.

Proof: Since the command receives information, i.e., creates new variables, and does not change any previous one, any DHSec, DHPubIn, DHPubOut or DHSol predicate is not affected.