MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 31, NO. 4, 2024
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Using TLA+/TLC for Modeling and Verification of Cryptographic Protocols

M. V. Neyzov[1], E. V. Kuzmin[2]

DOI: 10.18255/1818-1015-2024-4-446-473

[1]Institute of Automation and Electrometry SB RAS, Novosibirsk, Russia
[2]P.G. Demidov Yaroslavl State University, Yaroslavl, Russia

MSC2020: 68Q60
Research article
Full text in Russian

Received November 6, 2024
Revised November 25, 2024
Accepted November 30, 2024

Interacting in open networks carries certain risks. To ensure the information security of network interaction participants, cryptographic protocols (CrP) are used. High levels of security can be achieved through their formal verification. A common formal method for verifying CrP is model checking.

In this work, we propose using the TLA+/TLC toolset to check models of CrP. This toolset is widely applied in various practical fields. The protocol model is defined in the TLA+ specification language, as well as the required security properties in the form of invariants. The model of a protocol describes its behavior as a transition system containing all possible states of the protocol model and transitions between them. The TLC model checker is employed to automatically verify that the model meets the required properties. The task of verifying CrP has its specifics. This study proposes three modeling techniques that take into account the specifics of both the task and the TLA+/TLC toolset being used. The first technique involves replacing a system consisting of an arbitrary number of agents with a three-agent system. This simplifies the model and reduces its state space. The second technique is related to representing transmitted messages as a hierarchical structure, allowing encrypted messages to be nested within others. The third technique consists of optimizing the model to improve the performance of the TLC model checker by defining a function that generates only those elements leading to transitions between states in the model. These techniques simplify the model and reduce verification time. We demonstrate the application of these results on a simple protocol example — the Needham-Schroeder public key authentication protocol. After detecting a known vulnerability in the original protocol by using TLC, we model and verify an improved version. Verification results show that the new version of the protocol does not have this vulnerability.

Keywords: protocol vulnerability; security properties; authentication; Needham-Schroeder protocol; asymmetric encryption system; formal verification; verification duration; model checking; transition system; protocol model; modeling techniques; generating function

INFORMATION ABOUT THE AUTHORS

Neyzov, Maxim V. (corresponding author) | ORCID iD: 0009-0000-6893-6137. E-mail: neyzov.max@gmail.com
Researcher

Kuzmin, Egor V. | ORCID iD: 0000-0003-0500-306X. E-mail: kuzmin@uniyar.ac.ru
Head of the Chair of Theoretical Informatics, Dr. Sc.

Funding: State task IAaE SB RAS, project No. 122031600173-8; Yaroslavl State University (project VIP-016).

For citation: M. V. Neyzov and E. V. Kuzmin, "Using TLA+/TLC for modeling and verification of cryptographic protocols", *Modeling and Analysis of Information Systems*, vol. 31, no. 4, pp. 446–473, 2024. DOI: 10.18255/1818-1015-2024-4-446-473.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 31, № 4, 2024
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Применение TLA+/TLC для моделирования и верификации криптографических протоколов

М. В. Нейзов[1], Е. В. Кузьмин[2]

DOI: 10.18255/1818-1015-2024-4-446-473

[1]Институт автоматики и электрометрии СО РАН, Новосибирск, Россия
[2]Ярославский государственный университет им. П.Г. Демидова, Ярославль, Россия

УДК 004.942+004.056
Научная статья
Полный текст на русском языке

Получена 6 ноября 2024 г.
После доработки 25 ноября 2024 г.
Принята к публикации 30 ноября 2024 г.

Взаимодействие в открытых сетях несёт определённые риски. Для обеспечения информационной безопасности участников сетевого взаимодействия используют криптографические протоколы. Высокие гарантии безопасности могут быть достигнуты в результате их формальной верификации. Распространённым формальным методом верификации криптографических протоколов является метод проверки модели.

В работе для проверки модели криптографических протоколов предлагается использовать инструментальное средство TLA+/TLC, широко применяемое на практике в различных прикладных областях. На языке спецификации TLA+ задаётся модель протокола, а также требуемые свойства безопасности в форме инвариантов. Модель протокола описывает его поведение в виде системы переходов, содержащей все возможные состояния модели протокола и переходы между ними. Для проведения автоматической проверки соответствия модели требуемым свойствам задействуется верификатор TLC. Задача верификации криптографических протоколов имеет свою специфику. Настоящее исследование предлагает три приёма моделирования, учитывающих особенности данной задачи и используемого инструментария TLA+/TLC. Первый приём моделирования состоит в замене системы, состоящей из произвольного количества агентов, на трёхагентную систему. Это позволяет упростить модель и уменьшить её пространство состояний. Второй приём связан с представлением передаваемых сообщений в виде иерархической структуры — это даёт возможность вкладывать одни зашифрованные сообщения в другие. Третий приём состоит в оптимизации модели с целью повышения производительности верификатора TLC. Это выполняется путем задания функции, порождающей множество только тех элементов, которые приводят к переходам между состояниями в модели. В итоге предложенные приёмы позволяют упростить модель и снизить время её верификации. Применение результатов демонстрируется на примере простого протокола — протокола Нидхема-Шредера для аутентификации с открытым ключом. После обнаружения верификатором TLC известной уязвимости этого протокола выполняется моделирование и верификация его доработанной версии. Результаты верификации показывают, что новая версия протокола не имеет данной уязвимости.

Ключевые слова: уязвимость протокола; свойства безопасности; аутентификация; протокол Нидхема-Шредера; ассиметричная система шифрования; формальная верификация; продолжительность верификации; проверка модели; система переходов; модель протокола; приёмы моделирования; порождающая функция

ИНФОРМАЦИЯ ОБ АВТОРАХ

Нейзов, Максим Вячеславович (автор для корреспонденции)	ORCID iD: 0009-0000-6893-6137. E-mail: neyzov.max@gmail.com Исследователь
Кузьмин, Егор Владимирович	ORCID iD: 0000-0003-0500-306X. E-mail: kuzmin@uniyar.ac.ru Заведующий кафедрой теоретической информатики, доктор физ.-мат. наук

Финансирование: Госзадание ИАиЭ СО РАН, проект № 122031600173-8; ЯрГУ (проект VIP-016).

Для цитирования: M. V. Neyzov and E. V. Kuzmin, “Using TLA+/TLC for modeling and verification of cryptographic protocols”, *Modeling and Analysis of Information Systems*, vol. 31, no. 4, pp. 446–473, 2024. DOI: 10.18255/1818-1015-2024-4-446-473.

Введение

Криптографический протокол (КрП) — коммуникационный протокол, в котором для защиты информации применяются криптографические алгоритмы [1, 2]. Коммуникации между агентами и их действия осуществляются согласно определённым в протоколе правилам. Протокол может фиксировать алгоритмы, используемые агентами для вычислений, тип и порядок передаваемых сообщений.

Существует множество КрП различного назначения. Условно их можно классифицировать по предоставляемым *услугам защиты*. В [3] выделяются пять базовых услуг:

1. Аутентификация (сторон или источника данных);
2. Управление доступом;
3. Конфиденциальность;
4. Целостность данных;
5. Безотказность (от факта отправления или получения сообщения).

КрП может быть направлен на реализацию как одной, так и нескольких услуг защиты. При этом важно обеспечить отсутствие *уязвимостей* [4] — недостатков, которые могут быть использованы для реализации угроз безопасности информации. Некоторые гарантии отсутствия уязвимостей даёт *теоретическая стойкость* [1] протокола — математически обоснованная способность противостоять возможным атакам. Для обоснования стойкости выполняется построение математических моделей протокола и требуемого свойства. Процедура установления соответствия между этими моделями называется *формальной верификацией* [5]. Широкое распространение получили две техники [6, 7]: *дедуктивная* верификация (доказательство теорем) и верификация методом *проверки модели* (model checking) [8]. Для их поддержки существует множество инструментальных средств — как специализирующихся на КрП [2], так и общего назначения.

Настоящая работа посвящена верификации КрП методом проверки модели с помощью инструментального средства TLA+/TLC [9]. На языке спецификации TLA+ будут задаваться модели протокола и требуемых свойств. Для проведения автоматической проверки соответствия этих двух моделей будет применяться верификатор TLC. Модель поведения протокола — *система переходов*, отражающая все возможные состояния модели протокола и переходы между ними. Формализация требуемого свойства протокола представляет собой инвариант на языке TLA+.

Верификация методом проверки модели сопряжена с некоторыми трудностями: «взрыв» в пространстве состояний и длительное время верификации. Настоящая работа направлена на преодоление этих сложностей и предлагает три приёма моделирования.

Первый приём — использование трёхагентной системы вместо системы из n агентов. Уменьшение количества агентов упрощает процесс моделирования и потенциально уменьшает пространство состояний модели.

Второй приём — представление сообщений в виде иерархической структуры. Иерархия вложенности позволяет достаточно легко формировать и обрабатывать содержимое зашифрованных сообщений. При этом происходит абстрагирование от самой процедуры шифрования.

Третий приём — оптимизация модели с помощью порождающей функции. Данная оптимизация связана с особенностью работы инструмента проверки модели TLC и позволяет повысить его производительность (снизить время верификации). После оптимизации спецификация TLA+ остаётся информативной и удобочитаемой, так как ограничения, накладываемые протоколом на сообщения, сконцентрированы в определении порождающей функции.

Данные приёмы моделирования продемонстрированы на примере простого протокола — протокола Нидхема-Шредера для аутентификации с открытым ключом [10] (далее NS-протокол). Цель протокола — обеспечить взаимную *аутентификацию сторон* [11] (агентов). Аутентификация — проверка агентом того, что взаимодействующий с ним агент именно тот, за кого себя выдаёт. NS-протокол использует *асимметричную систему шифрования* [12], где для шифрования и расшиф-

рования используется пара взаимосвязанных ключей: *открытый* и *закрытый*. Если сообщение зашифровано открытым ключом, то расшифровать его можно только соответствующим закрытым ключом, и наоборот, если сообщение зашифровано закрытым ключом, то расшифровать его можно только соответствующим открытым ключом.

Известно, что NS-протокол обладает уязвимостью. После её обнаружения было предложено несколько доработок данного протокола. В связи с этим в настоящей работе поставлена следующая задача моделирования: построить модель классического NS-протокола и автоматически обнаружить его уязвимость, далее построить модель доработанного NS-протокола и автоматически доказать отсутствие этой уязвимости.

Содержание работы. Раздел 1 содержит описание двух версий NS-протокола: классическую и доработанную. Приведён анализ уязвимости классической версии. Раздел 2 представляет собой краткое введение в язык спецификации TLA+ и инструмент проверки модели TLC. Приведён обзор наиболее известных промышленных применений данного инструментария. В разделе 3 представлены приёмы моделирования. В разделе 4 осуществляется разработка и верификация модели классического NS-протокола, в разделе 5 — модели доработанного NS-протокола. В заключительном разделе подведён итог работы.

1. Протокол Нидхема-Шредера для аутентификации с открытым ключом

В 1978 году Нидхем и Шредер предложили протокол для аутентификации с открытым ключом [10]. Агент A инициирует сеанс связи с агентом B для проведения взаимной аутентификации с помощью доверенного посредника T. Протокол определяет семь этапов взаимодействия: одна группа этапов предназначена для получения открытых ключей, другая — непосредственно для аутентификации [13]. Далее нас будет интересовать только последняя группа. Протокол предусматривает три этапа для аутентификации [13]:

$$1.\ A \longrightarrow B : \{\, Na, a \,\}_{Kb} \tag{1}$$

$$2.\ B \longrightarrow A : \{\, Na, Nb \,\}_{Ka} \tag{2}$$

$$3.\ A \longrightarrow B : \{\, Nb \,\}_{Kb} \tag{3}$$

Здесь $A \longrightarrow B$ указывает на передачу сообщения от агента A к агенту B, далее через двоеточие идёт само сообщение вида $\{M\}_{Kx}$, которое означает, что данные M зашифрованы с помощью открытого ключа агента x. На первом этапе (1) агент A посылает сообщение агенту B, зашифрованное открытым ключом Kb агента B. Сообщение содержит сгенерированное агентом A одноразовое случайное число Na (nonce [1], далее *нонс*), а также имя отправителя a. Агент B с помощью своего закрытого ключа расшифровывает полученное сообщение и из его содержимого понимает, что с ним пытается установить связь агент A. На втором этапе (2) наоборот, агент B посылает сообщение агенту A, также зашифрованное открытым ключом Ka агента A. Сообщение содержит нонс Na и сгенерированный агентом B нонс Nb. Агент A по нонсу Na удостоверяется в том, что сообщение получено именно от агента B, так как только он мог расшифровать сообщение содержащее этот нонс. На третьем этапе (3) агент A посылает агенту B зашифрованное открытым ключом Kb агента B сообщение, содержащее нонс Nb. Теперь агент B по полученному нонсу Nb удостоверяется в том, что сообщение получено именно от агента A. Таким образом, оба агента аутентифицируют друг друга.

В 1995 году Лоу обнаружил уязвимость данного протокола [14]. То есть ошибка, допущенная разработчиками протокола, была замечена только спустя 17 лет. Чтобы продемонстрировать найденную уязвимость протокола потребуется три агента: A — инициатор связи, B — ответчик, P — злоумышленник. Атака на протокол, предложенная Лоу (далее атака Лоу) представлена на рис. 1. Здесь злоумышленник P осуществляет параллельно два сеанса связи: один с агентом A, другой — с агентом B.

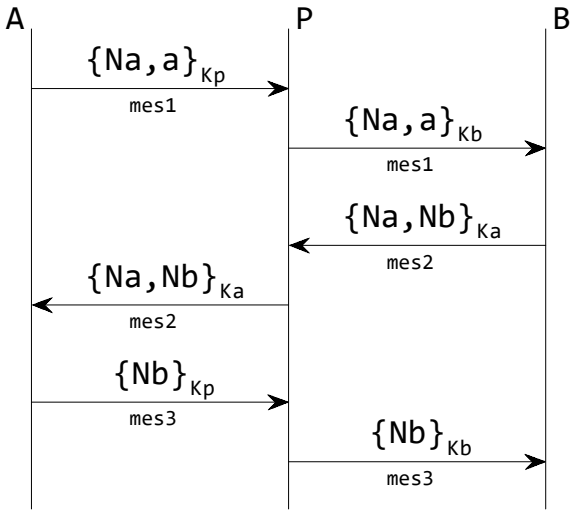

Fig. 1. Agent interaction diagram: Low's attack on the Needham-Schroeder protocol

Рис. 1. Диаграмма взаимодействия агентов: атака Лоу на протокол Нидхема-Шредера

Обозначим *mes1* тип сообщений, передаваемых на первом этапе. Аналогично *mes2* и *mes3* — типы сообщений, передаваемые на втором и третьем этапах соответственно. При атаке на протокол (рис. 1) изначально агент A посылает агенту P сообщение типа *mes1*, зашифрованное открытым ключом Kp агента P. Далее агент P расшифровывает полученное сообщение своим закрытым ключом. Извлечённые из сообщения данные пересылает агенту B, шифруя сообщение типа *mes1* открытым ключом Kb агента B. Агент B по содержимому полученного сообщения полагает, что с ним пытается установить связь агент A. Поэтому агент B высылает ответное сообщение типа *mes2* агенту A, шифруя сообщение открытым ключом Ka агента A. Злоумышленник (агент P) перехватывает это сообщение, но не может его расшифровать, так как не знает закрытый ключ агента A. Далее агент P пересылает перехваченное сообщение адресату — агенту A. Он уже расшифровывает это сообщение типа *mes2*. Полученные данные соответствуют ожиданиям агента A: первый элемент данных — его нонс Na, второй элемент — нонс Nb, который теперь агент A считает нонсом, сгенерированным агентом P, что не так. Далее агент A пересылает данный нонс Nb агенту P, шифруя сообщение типа *mes3* его открытым ключом Kp. Теперь злоумышленник (агент P) может расшифровать полученное сообщение и извлечь из него нонс Nb. Далее агент P пересылает этот нонс Nb агенту B в виде сообщения типа *mes3*. Агент B получает и расшифровывает данное сообщение. В его содержимом обнаруживает ожидаемые данные (свой нонс Nb), поэтому теперь считает, что установил связь с агентом A и пара нонсов Na, Nb известна только им двоим, что не так. Теперь злоумышленник (агент P) может взаимодействовать с агентом B, маскируясь под агента A. Агент B будет уверен в том, что он взаимодействует с агентом A. Если агент B — это банк, то злоумышленник (агент P) может снять деньги со счета агента A. Банк (агент B) будет уверен в том, что деньги со своего счёта снял именно агент A.

После обнаружения вышеприведённой уязвимости для её устранения было предложено множество различных доработок протокола, связанных с изменением формата передаваемых сообщений. Рассмотрим одну из доработанных версий протокола Нидхема-Шредера [13]:

$$1.\ A \to B : \left\{[Na, a]_{Ka}\right\}_{Kb} \tag{4}$$

$$2.\ B \to A : \left\{Na, [Nb]_{Kb}\right\}_{Ka} \tag{5}$$

$$3.\ A \to B : \left\{[Nb]_{Ka}\right\}_{Kb} \tag{6}$$

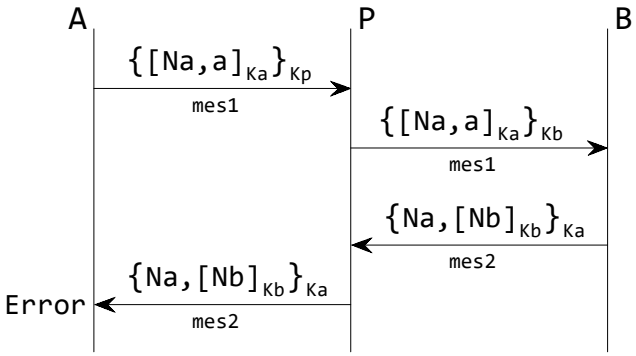

Fig. 2. Agent interaction diagram: Impossibility of Lowe's attack on the modified Needham-Schroeder protocol

Рис. 2. Диаграмма взаимодействия агентов: невозможность атаки Лоу на доработанный протокол Нидхема-Шредера

Здесь $\{M\}_{Kx}$ означает, что данные M зашифрованы с помощью открытого ключа агента x, а $[M]_{Kx}$ — зашифрованы с помощью его закрытого ключа. На первом этапе (4) агент A передаёт сообщение агенту B. Данное сообщение типа *mes1* формируется следующим образом: пересылаемые данные (нонс Na и имя a) агент A изначально шифрует своим закрытым ключом, далее открытым ключом агента B. Соответственно агент B для расшифровки полученного сообщения сначала применяет свой закрытый ключ, а потом открытый ключ агента A. На втором этапе (5) агент B отвечает агенту A сообщением типа *mes2*. Агент B свой нонс Nb шифрует закрытым ключом, далее зашифрованный нонс Nb вместе с нонсом Na шифрует открытым ключом агента A. Агент A извлекает данные из полученного сообщения. На третьем этапе (6) агент A высылает агенту B его нонс Nb, предварительно зашифрованный сначала своим закрытым ключом, а потом открытым ключом агента B. Данное сообщение имеет тип *mes3*.

Атака Лоу на данную версию протокола невозможна, так как на втором этапе взаимодействия (при получении сообщения типа *mes2*) агент A попытается расшифровать данные $[Nb]_{Kb}$ открытым ключом агента P и получит ошибку [13] (см. рис. 2). После чего сеанс связи с агентом P прекращается.

Формальная верификация методом проверки модели позволила бы обнаружить уязвимость первой версии протокола — атака Лоу могла быть найдена автоматически во время проектирования протокола, а не спустя 17 лет. Верификация доработанной версии протокола может доказать (или опровергнуть) отсутствие уязвимости при определённых упрощениях и допущениях. Для проведения процедуры верификации необходимо построить модель протокола. Для моделирования и верификации будем использовать средство TLA+/TLC.

2. Инструментальное средство моделирования и верификации TLA+/TLC

TLA+ [9] — язык моделирования (формальной спецификации), предназначенный для задания модели поведения в виде системы переходов. Формализм TLA+ основан на теории множеств и логики предикатов первого порядка, расширенной темпоральными операторами. Спецификация на языке TLA+ представляет собой темпоральную формулу и задаёт систему переходов, которая удовлетворяет этой формуле. В спецификации на языке TLA+ описываются все возможные переходы с помощью *действий* [15] — предикатов над парой состояний, образующих переход. Например, предикат *inc*, определённый как $a' = a + 1$, задаёт все переходы, в которых переменная a увеличивается на единицу. Здесь a' — значение переменной a в следующем состоянии.

Если спецификация на TLA+ задаёт конечную систему переходов, то она может быть верифицирована с помощью инструмента проверки модели TLC [9]. Среда разработки «TLA+ Toolbox» [16] предназначена для работы с TLA+/TLC.

Язык TLA+ получил широкое распространение и применяется во многих областях. Компания Intel использовала его для проверки протокола когерентности кэша своих чипов [17], Microsoft — для проверки протокола когерентности памяти Xbox 360 [9]. Также компания Microsoft использует TLA+ при разработке своих облачных сервисов Azure, например, базы данных Azure Cosmos DB [18]. TLA+ применяют в компании Amazon при разработке облачных сервисов Amazon Web Services [19]. Язык TLA+ использовался для моделирования и верификации межсетевых экранов [20], операционной системы реального времени OpenComRTOS [21], протоколов консенсуса HotStuff [22, 23] и Raft [24], протокола формирования команды [25], протокола обмена сообщениями MQTT [26], протокола вещания Zab [27], распределенной системы координации ZooKeeper [28].

3. Приёмы моделирования

3.1. Использование трёхагентной системы

В общем случае моделируемая система состоит из n агентов, где m агентов ($m < n$) для достижения своих целей могут иметь не соответствующее протоколу поведение. Такие агенты являются *злоумышленниками*. Однако нет необходимости моделировать поведение всех n агентов. Всегда можно выделить только пару честных агентов, которые пытаются установить сеанс связи между собой, и одного злоумышленника с определёнными возможностями согласно *модели угрозы Долева-Яо* [29]. Такой злоумышленник должен уметь перехватывать все сообщения, имитировать работу сразу нескольких агентов (честных и нечестных) и маскироваться под другого агента, тогда пара честных агентов будет находиться в такой же ситуации, если бы они находились в системе из n агентов. Предложенная схема трёхагентной системы представлена на рис. 3, где A и B — честные агенты, причём A — инициатор (initiator) связи, B — ответчик (responser), а P — агент-злоумышленник (intruder). Злоумышленник P всегда перехватывает сообщение при его передаче от A к B. Далее P может переслать сообщение для B в неизменном виде, либо послать своё собственное сообщение. Аналогичная ситуация возникает при передаче сообщения в обратном направлении от B к A.

Если проблема проявится в такой системе, то она будет присутствовать и в системе из n агентов. Если в трёхагентной системе проблема отсутствует, то данный результат можно распространить на систему из k агентов, где $k \leq n$ и зависит от настроенных параметров агента P. Использование трёхагентной системы обеспечивает простоту и минималистичность модели — нет необходимости моделирования системы из n агентов.

Подобная трёхагентная система использовалась при верификации NS-протокола с помощью инструмента проверки модели Spin [30]. Отличие нашей системы состоит в том, что мы не ограничиваем возможности злоумышленника — в целом он может представлять собой всю враждебную среду, а также других честных агентов. В приведённой далее спецификации TLA+ (см. раздел 4) злоумышленник воспроизводит поведение ровно одного агента, так как этого достаточно для выявления уязвимости классического NS-протокола. Отметим, что верификатор подтверждает выполнение свойства лишь при определённых ограничениях, заложенных в модели злоумышленника. Поэтому может потребоваться расширение его возможностей для ослабления данных ограничений.

С одной стороны, действия злоумышленника согласно модели Долева-Яо никак не ограничены, с другой стороны, отправка агентам произвольных сообщений не является результативной. Большая

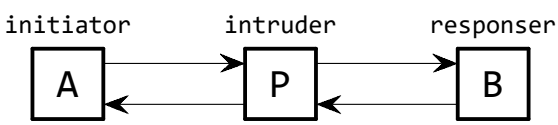

Fig. 3. Three-agent system diagram

Рис. 3. Схема трёхагентной системы

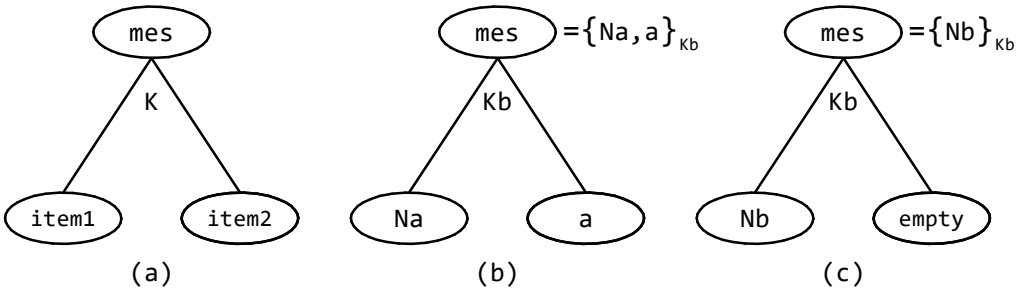

Fig. 4. Simple message: structure (a), examples (b,c)

Рис. 4. Простое сообщение: структура (a), примеры (b,c)

часть из них будет приводить к прекращению сеанса связи, что не способствует поиску атаки на протокол. Для сокращения поискового пространства возможных атак имеет смысл ограничить нерезультативные посылки сообщений агентам.

С этой целью в [31] модель злоумышленника основывается на метаданных, полученных в результате инспекции передаваемых сообщений. В настоящей работы мы предлагаем модель злоумышленника, который будет ограничен передачей только результативных (не прерывающих сеанс связи) сообщений. Результативность сообщения будет определяться на основании текущего состояния агентов. Технически это будет реализовано с помощью порождающей функции (см. раздел 3.3), которая возвращает множество таких сообщений.

3.2. Представление сообщений в виде иерархической структуры

Для передачи и обработки сообщений в модели нужно выбрать способ их представления и кодирования. При моделировании потребуется работа со всеми сообщениями, представленными в (1), (2), (3), (4), (5), (6). Определим структуру *простого* сообщения (см. рис. 4a). Любое простое сообщение зашифровано ключом K и содержит в себе до двух элементов данных (*item1*, *item2*). Чтобы расшифровать сообщение и получить доступ к его элементам данных (спуститься из вершины *mes* к вершинам *item1* и *item2*) необходимо располагать ключом расшифровки, соответствующим ключу шифрования K. Структура сообщений (1) и (3) представлена на рис. 4b и рис. 4c соответственно. Обозначим «empty» пустой элемент, т. е. отсутствие данных.

Теперь определим структуру *составного* сообщения (см. рис. 5), составленного из простых. Корневую вершину простого сообщения назовём *узлом*. Простое сообщение, представленное узлом *node1*, в качестве элементов данных содержит ссылки на узлы *node2* и *node3*, которые, в свою очередь, представляют два других простых сообщения. Каждое простое сообщение зашифровано своим ключом, указанным под узлом.

Структура сообщений (4) и (5) представлена на рис. 6a и рис. 6b соответственно. Обозначим «not encrypted» отсутствие ключа шифрования, т. е. сообщение не зашифровано.

Таким образом, любое требуемое в рамках рассматриваемого протокола сообщение может быть представлено в виде иерархической структуры, хранящей все необходимые данные.

3.3. Оптимизация модели с помощью порождающей функции

В TLA+ спецификациях достаточно часто используется выражение вида:

$$\backslash E\ x\ \backslash in\ X:\ Q(x,\ a), \tag{7}$$

где \E — квантор существования, \in — отношение принадлежности, X — некоторое множество. Его элементами могут быть, например, числовые значения, векторы значений, где каждый компонент вектора принадлежит определённому множеству. Также X может содержать более сложные струк-

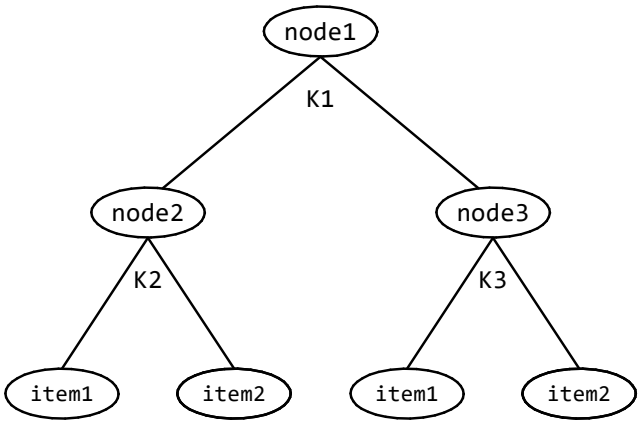

Fig. 5. Structure of a complex message

Рис. 5. Структура составного сообщения

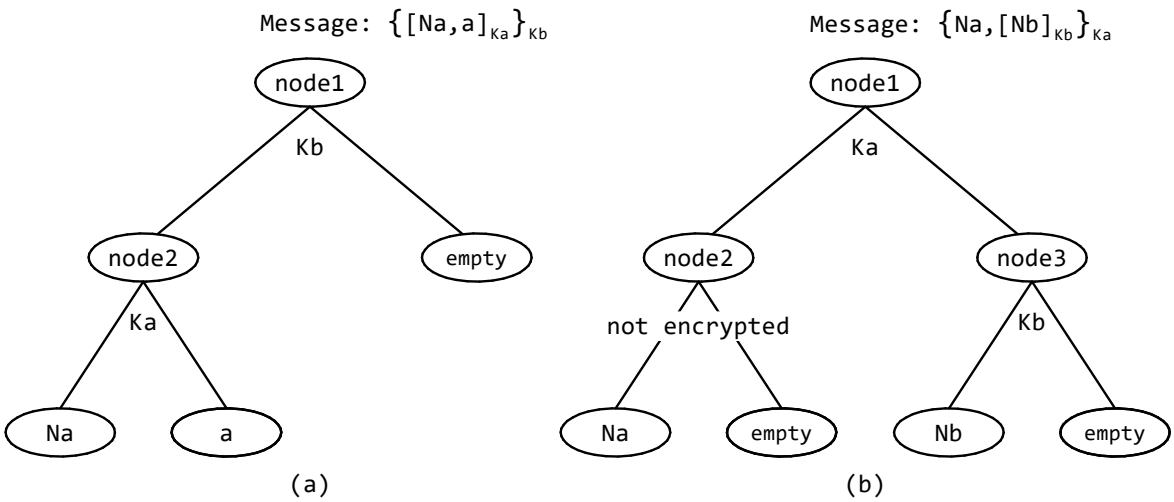

Fig. 6. An example of a complex message consisting of: two messages (a), three messages (b)

Рис. 6. Пример составного сообщения состоящего из: двух сообщений (a), трёх сообщений (b)

турированные данные (например, деревья). Q — n-местный предикат, параметр a — значение вектора (длины $n-1$) переменных предиката Q. В TLA+ при определении предиката Q также могут фигурировать и переменные модели (неявные переменные предиката Q). Обозначим b значение вектора используемых переменных модели. Таким образом, фактически предикат Q зависит от двух параметров a и b. Например, предикат $x\_add(y)$, определённый как $x' = x + y$, фактически зависит от двух параметров: y и вектора (x, x'). Тогда с учётом принятых обозначений выражению (7) в TLA+ соответствует математическое выражение

$$\exists x \in X : Q(x, a, b). \tag{8}$$

Инструмент проверки модели TLC оценивает выражение (8) путем перебора всех элементов множества X, каждый раз определяя истинность предиката $Q(x, a, b)$ [15]. Для всех $x \in X$, удовлетворяющих $Q(x, a, b)$, вычисляется переход в модели. Остальные $x \in X$, не удовлетворяющие $Q(x, a, b)$, игнорируются. В итоге при фиксированных a и b предикат $Q(x, a, b)$ задаёт множество $M_Q = \{x \in X \mid Q(x, a, b)\}$, для всех элементов которого и только для них вычисляются переходы в мо-

дели. Если мощность множества X намного больше мощности множества M_Q:

$$|X| \gg |M_Q|, \tag{9}$$

то верификатор TLC будет слишком много времени тратить вхолостую на обход множества $X \setminus M_Q$, который не приносит никаких результатов. Таким образом, условие (9) является *критерием неэффективности* использования выражения (7).

При разработке TLA+ спецификации предикат $Q(x, a, b)$ можно попытаться разделить (если это уже не сделано) на два предиката $P(x, a, b)$ и $A(x, a, b)$, где первый — определяет только условия (без действий), второй — по возможности только действия (оценивает переходы между состояниями). При фиксированных a и b предикат $P(x, a, b)$ задаёт множество $M_P = \{x \in X \mid P(x, a, b)\}$. Задача при разделении $Q(x, a, b)$ на два предиката — обеспечить *максимальное приближение сверху* множества M_P к множеству M_Q, т. е. либо совпадение множеств: $M_P = M_Q$, либо сопоставимость их мощностей: $|M_P| \approx |M_Q|$ при $M_Q \subset M_P$. После разделения $Q(x, a, b)$ на два предиката выражение (8) примет вид

$$\exists x \in X : P(x, a, b) \wedge A(x, a, b),$$

что при фиксированных параметрах a и b эквивалентно

$$\exists x \in M_P : A(x, a, b). \tag{10}$$

Далее множество M_P, зависимое от параметров a и b, будем определять с помощью порождающей функции вида

$$F\colon AB \to 2^X, \tag{11}$$

где AB — множество значений вектора определённых переменных модели и предиката Q, 2^X — булеан (множество всех подмножеств) множества X. Выражение (10) с учётом порождающей функции (11) примет вид

$$\exists x \in F(a, b) : A(x, a, b). \tag{12}$$

В выражении (12) функция F порождает множество тех и только тех элементов из X, которые гарантированно приводят к переходам (в том числе и по петле) в модели согласно действиям, определённым в $A(x, a, b)$. В этом случае даже при условии (9) верификатор TLC будет эффективно работать, так как ему не требуется обходить элементы из X, которые не приводят к переходам в модели.

Математическому выражению (12) соответствует TLA+ выражение

$$\text{\E x \in F(a, b): A(x, a).} \tag{13}$$

Заметим, что в (13) предикат A неявно зависит от параметра b.

Мы предлагаем при условии (9) в TLA+ спецификации вместо выражения (7) использовать выражение (13) для повышения производительности верификатора TLC.

4. Разработка и верификация модели классического NS-протокола

При проектировании протокола для подтверждения его корректности может потребоваться проверка различных свойств. С этой целью разработаем модель протокола, которая также должна позволять обнаружить и атаку Лоу.

Проследим по шагам процесс разработки TLA+ спецификации (полный текст доступен по ссылке [32]), задающей модель классического протокола. Подключим модули для работы с натуральными числами и конечными множествами:

```
EXTENDS Naturals, FiniteSets
```

Определим множество имён (идентификаторов) агентов и множество их уникальных нонсов:

```
ID == { "a", "b", "p" }
Nonces == { "Na", "Nb", "Np" }
```

Множество открытых и множество закрытых ключей:

```
Pub_keys ==  { "pub_key_a",  "pub_key_b",  "pub_key_p"  }
Priv_keys == { "priv_key_a", "priv_key_b", "priv_key_p" }
```

Множество всех ключей вместе, а также это же множество со специальным ключом, означающим отсутствие шифрования:

```
Keys == Pub_keys \cup Priv_keys
Keys_ == Keys \cup {"not_encrypted"}
```

Множество ключей, доступных для агента p:

```
p_AvailableKeys == Pub_keys \cup {"priv_key_p", "not_encrypted"}
```

Соответствие открытых и закрытых ключей:

```
pub_to_priv == [ pub_key_a |-> "priv_key_a",
                 pub_key_b |-> "priv_key_b",
                 pub_key_p |-> "priv_key_p" ]
priv_to_pub == [ priv_key \in Priv_keys |->
                 CHOOSE pub_key \in Pub_keys: pub_to_priv[pub_key] = priv_key ]
```

Соответствующий ключ для расшифровки (открытому ключу соответствует закрытый и наоборот):

```
decrypt_key == [ key \in Keys_ |->
                 CASE key \in Pub_keys  -> pub_to_priv[key]
                   [] key \in Priv_keys -> priv_to_pub[key]
                   [] OTHER             -> "not_encrypted"
               ]
```

Соответствие ID и открытых ключей:

```
id_to_pub == [ a |-> "pub_key_a",
               b |-> "pub_key_b",
               p |-> "pub_key_p" ]
```

Множество состояний агента a:

```
StatesA == { "send_mes1", "wait_mes2", "send_mes3", "finalA" }
```

Функция следующего состояния агента a:

```
next_a_state == [ send_mes1 |-> "wait_mes2",
                  wait_mes2 |-> "send_mes3",
                  send_mes3 |-> "finalA",
                  finalA    |-> "finalA" ]
```

Множество состояний агента b:

```
StatesB == { "wait_mes1", "send_mes2", "wait_mes3", "finalB" }
```

Функция следующего состояния агента b:

```
next_b_state == [ wait_mes1 |-> "send_mes2",
                  send_mes2 |-> "wait_mes3",
                  wait_mes3 |-> "finalB",
                  finalB    |-> "finalB" ]
```

Ссылки на узлы в сообщениях:

```
NodesRef == {"node2", "node3"}
```

Множество передаваемых элементов данных (empty — пустой элемент):

```
DataItems == ID \cup Nonces \cup NodesRef \cup {"empty"}
```

Номера элементов данных в сообщении:

```
ItemsNum == { "i1", "i2" }
```

Множество простых сообщений (согласно рис. 4а):

```
SimpleMessages ==
[ key  : Keys_,     \* Encryption key.
  item : [ItemsNum -> DataItems]
]
```

Множество передаваемых сообщений:

```
Messages == {<<mes>> : mes \in SimpleMessages} \cup
            (SimpleMessages \X SimpleMessages) \cup
            (SimpleMessages \X SimpleMessages \X SimpleMessages)
```

Передаваться могут простые сообщения — однокомпонентные векторы (строка 1), а также составные сообщения — двух- и трёхкомпонентные векторы (строки 2 и 3 соответственно). Таким образом, все сообщения кодируются в виде векторов. Проверка, что сообщение является составным:

```
is_complex(mes) == Cardinality(DOMAIN mes) > 1
```

Зафиксируем три типа сообщений:

```
MesTypes == { "mes1", "mes2", "mes3" }
```

Объявим переменные модели:

```
VARIABLES
    p_input_mes,    \* Incoming messages of agent p.
    p_simple_mes,   \* Simple (incoming, disassembled) messages of agent p.
    p_Nonces,       \* Nonces known to agent p.
    a_state,        \* Current state of agent a.
    b_state,        \* Current state of agent b.
    a_input_nonce,  \* Nonce received by agent a.
    b_input_nonce,  \* Nonce received by agent b.
    b_input_id,     \* ID received by agent b.
    a_connect_to,   \* Agent for communication selected by agent a.
    msg_type,       \* Message type (variable for debugging).
    msg_kind,       \* Message kind: created/existing (variable for debugging).
    sended_mes      \* Sended message (variable for debugging).
```

Все входящие сообщения агента *p* будем сохранять в множестве *p_input_mes*. Далее их будем разбирать на простые и сохранять в множестве *p_simple_mes*. Нонсы, полученные агентом *p* из расшифрованных сообщений, будут сохраняться в множестве *p_Nonces*. Агенты *a* и *b*, получая и посылая сообщения, изменяют своё текущее состояние, которое будет храниться в переменных *a_state* и *b_state* соответственно. Переменные *a_input_nonce* и *b_input_nonce* предназначены для сохранения полученных нонсов агентами *a* и *b* соответственно. Переменная *b_input_id* — для хранения имени агента, с которым установил связь агент *b*. Переменная *a_connect_to* — для хранения имени агента, с которым выбрал установить связь агент *a*. Оставшиеся три переменные предназначены для отладки и повышения информативности состояния модели. Переменная *msg_type* отражает тип передаваемого сообщения, а переменная *msg_kind* — вид сообщения для *p*: полученное от агента (*a* или *b*), посланное агенту (*a* или *b*), при этом созданное агентом *p* или существующее в его входном буфере *p_input_mes*. В переменной *sended_mes* будем фиксировать посланное сообщение. Все переменные модели объединим в кортеж:

```
vars == << p_input_mes, p_simple_mes, p_Nonces, a_state, b_state, a_input_nonce, b_input_nonce,
           b_input_id, a_connect_to, msg_type, msg_kind, sended_mes >>
```

Определим инвариант типа для данных переменных:

```
TypeInv ==
    /\ p_input_mes \subseteq Messages
    /\ p_simple_mes \subseteq SimpleMessages
    /\ p_Nonces \subseteq Nonces
    /\ a_state \in StatesA
    /\ b_state \in StatesB
    /\ a_input_nonce \in Nonces \cup {"empty"}
    /\ b_input_nonce \in Nonces \cup {"empty"}
    /\ b_input_id \in ID \cup {"empty"}
    /\ a_connect_to \in (ID \ {"a"})
    /\ msg_type \in MesTypes \cup {"empty"}
    /\ msg_kind \in {"from_agent", "created", "existing", "empty"}
    /\ sended_mes \in Messages
```

Зададим инициализацию данных переменных:

```
Init ==
    /\ p_input_mes = {}
    /\ p_simple_mes = {}
    /\ p_Nonces = {"Np"}
    /\ a_state = "send_mes1"
    /\ b_state = "wait_mes1"
    /\ a_input_nonce = "empty"
    /\ b_input_nonce = "empty"
    /\ b_input_id = "empty"
    /\ a_connect_to \in (ID \ {"a"})
    /\ msg_type = "empty"
    /\ msg_kind = "empty"
    /\ sended_mes = empty_mes
```

Изначально множества *p_input_mes* и *p_simple_mes* пусты. Агент *p* располагает только своим нонсом *Np*. Агент *a* желает послать сообщение типа *mes1*, агент *b* находится в состоянии его ожидания. Агентам *a* и *b* неизвестны нонсы агентов, с которыми они будут взаимодействовать. Агенту *b* (ответчику) изначально неизвестен идентификатор агента, который попытается установить с ним связь. В начальном состоянии агент *a* выбирает любой идентификатор агента (кроме своего) для установки сеанса связи. Посланное сообщение установим пустым и незашифрованным, тип и вид которого не определены. Пустое незашифрованное сообщение:

```
empty_mes == <<[ key |-> "not_encrypted", item |-> [ i1 |-> "empty", i2 |-> "empty" ] ]>>
```

Зададим отношение переходов модели:

```
Next ==
    \/ p_decrypt
    \/ mes_transfer_p
    \/ p_transfer_mes
    \/ UNCHANGED vars
```

Допустимы следующие взаимоисключающие действия: агент *p* расшифровывает входящие сообщения, выполняется передача сообщения агенту *p* (от агента *a* или *b*), агент *p* передаёт сообщение (агенту *a* или *b*), ничего не происходит и значения переменных не изменяются (переход в модели по петле). Только три вышеуказанных действия приводят к переходам между различными состояниями модели, что отражается в атомарном изменении значений её переменных.

Определим действие расшифровки сообщений:

```
p_decrypt ==
    /\ extract_simple_messages
    /\ extract_nonces_from_ExtractedMes
    /\ msg_type' = "empty"
    /\ msg_kind' = "empty"
    /\ sended_mes' = empty_mes
    /\ UNCHANGED << p_input_mes, a_state, b_state, a_input_nonce, b_input_nonce,
                    b_input_id, a_connect_to >>
```

Расшифровка сообщений — одновременно извлечение простых сообщений (из входящих) и извлечение из них нонсов. При расшифровке агентом *p* извлекаются только нонсы, так как остальная информация ему всегда известна. При расшифровке не происходит передачи сообщений между агентами, поэтому переменные для отладки имеют такие же значения, как и при инициализации. В *UNCHANGED* указаны все переменные, которые не были затронуты действием *p_decrypt*. Далее для данного действия локально (внутри конструкции *LET*/*IN*) раскроем все используемые определения. Извлечение простых сообщений из *p_input_mes* — пополнение множества простых сообщений извлеченными сообщениями:

```
extract_simple_messages == p_simple_mes' = p_simple_mes \cup ExtractedMes
```

Извлеченные сообщения — сообщения извлеченные из составных и не составных сообщений:

```
ExtractedMes == ExtractedFromComplexMes \cup ExtractedFromNotComplexMes
```

Множество простых сообщений, извлеченных из составных сообщений:

```
ExtractedFromComplexMes ==
    { mes \in SimpleMessages:
        \E complex_mes \in ComplexMes:
            /\ contains_mes(complex_mes, mes)
            /\ can_disassemble(complex_mes) }
```

Составные сообщения из *p_input_mes*:

```
ComplexMes == { mes \in p_input_mes : is_complex(mes) }
```

Проверка, что составное сообщение содержит простое сообщение:

```
contains_mes(complex_mes, simple_mes) ==
    \E index \in (DOMAIN complex_mes): complex_mes[index] = simple_mes
```

Составное сообщение может быть разобрано на простые, если агент *p* имеет соответствующий ключ для расшифровки сообщения, представленного узлом 1 (см. рис. 5):

```
can_disassemble(mes) == decrypt_key[mes[1].key] \in p_AvailableKeys
```

Множество простых сообщений, извлеченных из не составных сообщений (из однокомпонентного вектора извлекаем значение этой компоненты):

```
ExtractedFromNotComplexMes == { mes[1] : mes \in NotComplexMes }
```

Не составные сообщения из *p_input_mes*:

```
NotComplexMes == { mes \in p_input_mes : ~is_complex(mes) }
```

Извлечение нонсов из полученных простых сообщений — пополнение множества нонсов доступных агенту *p* извлеченными нонсами:

```
extract_nonces_from_ExtractedMes == p_Nonces' = p_Nonces \cup extracted_Nonces
```

Нонсы, полученные из извлеченных простых сообщений:

```
extracted_Nonces ==
    { nonce \in Nonces:
        \E mes \in ExtractedMes:
            /\ contains_item(mes, nonce)
            /\ can_decrypt(mes) }
```

Простое сообщение содержит элемент данных:

```
contains_item(mes, itm) == \E i \in ItemsNum: mes.item[i] = itm
```

Простое сообщение может быть расшифровано, если агент p располагает соответствующим ключом для его расшифровки:

```
can_decrypt(mes) == decrypt_key[mes.key] \in p_AvailableKeys
```

Определим следующее действие: передача сообщения агенту p — это передача сообщения от агента a или агента b:

```
mes_transfer_p ==
    \/ a_transfer_p
    \/ b_transfer_p
```

Передача сообщения от агента a к агенту p — одновременно посылка сообщения агентом a и её приём агентом p (вид сообщения для p — получено от агента):

```
a_transfer_p ==
    /\ a_send
    /\ p_receive
    /\ msg_kind' = "from_agent"
    /\ UNCHANGED << p_simple_mes, p_Nonces, b_state, a_input_nonce, b_input_nonce,
                    b_input_id, a_connect_to >>
```

Посылка сообщения агентом a:

```
a_send ==
    /\ a_state \in { "send_mes1", "send_mes3" }
    /\ a_state' = next_a_state[a_state]
```

Агент a должен находиться в состоянии посылки сообщения. При посылке сообщения агент a переходит в следующее состояние согласно протоколу.

Приём сообщения агентом p:

```
p_receive ==
    \/ /\ a_state = "send_mes1"
       /\ msg_type' = "mes1"
       /\ p_input_mes' = p_input_mes \cup {mes1}
       /\ sended_mes' = mes1

    \/ /\ a_state = "send_mes3"
       /\ msg_type' = "mes3"
       /\ p_input_mes' = p_input_mes \cup {mes3}
       /\ sended_mes' = mes3
```

Агент p от агента a может получить сообщение двух типов: *mes1* либо *mes3* (см. рис. 1). Агент p принимает то сообщение от агента a, которое он ему посылает. Приём заключается в пополнении буфера *p_input_mes*. Возможные сообщения от агента a:

```
mes1 == <<[ key |-> id_to_pub[a_connect_to], item |-> [ i1 |-> "Na",          i2 |-> "a" ] ]>>
mes3 == <<[ key |-> id_to_pub[a_connect_to], item |-> [ i1 |-> a_input_nonce, i2 |-> "empty" ]]>>
```

Сообщения зашифрованы открытым ключом того агента, которого агент *a* выбрал для сеанса связи. Содержимое сообщений *mes1* и *mes3* см. в (1) и (3) соответственно. В $mes3.item[i1]$ агент *a* пересылает полученный нонс для подтверждения.

Передача сообщения от агента *b* к агенту *p* — одновременно посылка сообщения агентом *b* и её приём агентом *p*:

```
b_transfer_p ==
    /\ b_send
    /\ p_receive
    /\ msg_kind' = "from_agent"
    /\ UNCHANGED << p_simple_mes, p_Nonces, a_state, a_input_nonce, b_input_nonce,
                    b_input_id, a_connect_to >>
```

Посылка сообщения агентом *b*:

```
b_send ==
    /\ b_state = "send_mes2"
    /\ msg_type' = "mes2"
    /\ b_state' = next_b_state[b_state]
```

Агент *b* должен находиться в состоянии посылки сообщения. Агент *b* может послать только сообщение типа *mes2* (см. рис. 1). При посылке сообщения агент *b* переходит в следующее состояние согласно протоколу.

Приём сообщения агентом *p* заключается в пополнении буфера *p_input_mes*:

```
p_receive ==
    /\ p_input_mes' = p_input_mes \cup {mes2}
    /\ sended_mes' = mes2
```

Содержание сообщения *mes2*:

```
mes2 == <<[ key |-> id_to_pub[b_input_id], item |-> [ i1 |-> b_input_nonce, i2 |-> "Nb" ] ]>>
```

Сообщение зашифровано открытым ключом того агента, идентификатор которого был получен агентом *b* заранее. Содержимое сообщения *mes2* см. в (2) В $mes2.item[i1]$ агент *b* пересылает полученный нонс для подтверждения.

Определим следующее действие: передача сообщения от агента *p* — это передача сообщения одного из трёх возможных типов:

```
p_transfer_mes ==
    \/ p_transfer_mes1
    \/ p_transfer_mes2
    \/ p_transfer_mes3
```

Передача агентом *p* сообщения типа *mes1*:

```
p_transfer_mes1 ==
    /\ \E mes \in AcceptedMessages("mes1"): /\ p_send(mes)
                                            /\ b_receive(mes)
    /\ UNCHANGED << p_input_mes, p_simple_mes, p_Nonces, a_state, a_input_nonce, a_connect_to >>
```

Сообщение типа *mes1* принимает только агент *b*, поэтому факт передачи — посылка агентом *p* и получение агентом *b* сообщения, допустимого при такой передаче.

Посылка сообщения агентом *p* — посылка существующего или созданного сообщения:

```
p_send(mes) ==
    /\ \/ send_existing
       \/ send_created
    /\ sended_mes' = mes
```

Локально (внутри конструкции *LET/IN*) определим эти два действия. Посылка существующего сообщения возможна, если оно было получено ранее:

```
send_existing ==
    /\ mes \in p_input_mes
    /\ msg_kind' = "existing"
```

Посылка созданного сообщения возможна, если агент p в состоянии его создать:

```
send_created ==
    /\ can_create_transfer(mes)
    /\ msg_kind' = "created"
```

Передаваемое сообщение может быть создано агентом p из имеющихся данных:

```
can_create_transfer(mes) ==
    /\ mes \in Messages
    /\ can_create_simple(mes[1])
    /\ \A i \in (DOMAIN mes) \ {1}:
            \/ can_create_simple(mes[i])
            \/ mes[i] \in p_simple_mes
```

Сообщение должно быть соответствующего типа, агент p может создать простое сообщение, представленное узлом 1. Если сообщение составное (имеет другие узлы), то для них агент p либо может создать простое сообщение, представленное этим узлом, либо имеет такое сообщение в своём буфере *p_simple_mes*.

Простое сообщение может быть создано агентом p из имеющихся данных:

```
can_create_simple(mes) ==
    /\ mes \in SimpleMessages
    /\ mes.key \in p_AvailableKeys
    /\ mes.item["i1"] \in p_AvailableItems
    /\ mes.item["i2"] \in p_AvailableItems
```

Сообщение должно быть соответствующего типа, агент p должен располагать ключом, которым оно зашифровано. Элементы данных, присутствующие в сообщении, должны быть доступны агенту p.

Доступные агенту p элементы данных:

```
p_AvailableItems == (DataItems \ Nonces) \cup p_Nonces
```

Доступны все элементы, кроме нонсов, за исключением нонсов, ставших известными агенту p.

Для *p_transfer_mes1* локально (используя *LET/IN*) определим получение сообщения агентом b:

```
b_receive(mes) ==
    /\ b_state        = "wait_mes1"
    /\ msg_type'      = "mes1"
    /\ b_input_nonce' = mes[1].item["i1"]
    /\ b_input_id'    = mes[1].item["i2"]
    /\ b_state'       = next_b_state[b_state]
```

Агент b должен находиться в состоянии ожидания этого сообщения. Получение сообщения состоит в извлечении из него нонса и идентификатора агентом b, а также в переходе в следующее состояние согласно протоколу.

Определим следующее действие — передача агентом p сообщения типа *mes2*:

```
p_transfer_mes2 ==
    /\ \E mes \in AcceptedMessages("mes2"): /\ p_send(mes)
                                            /\ a_receive(mes)
    /\ UNCHANGED << p_input_mes, p_simple_mes, p_Nonces, b_state,
                    b_input_nonce, b_input_id, a_connect_to >>
```

Сообщение типа *mes2* принимает только агент *a*, поэтому факт передачи — посылка агентом *p* и получение агентом *a* сообщения, допустимого при такой передаче.

Для *p_transfer_mes2* локально (используя *LET/IN*) определим получение сообщения агентом *a*:

```
a_receive(mes) ==
    /\ a_state        = "wait_mes2"
    /\ msg_type'      = "mes2"
    /\ a_input_nonce' = mes[1].item["i2"]
    /\ a_state'       = next_a_state[a_state]
```

Агент *a* должен находиться в состоянии ожидания этого сообщения. Получение сообщения состоит в извлечении из него нонса агентом *a*, а также в переходе в следующее состояние согласно протоколу.

Определим следующее действие — передача агентом *p* сообщения типа *mes3*:

```
p_transfer_mes3 ==
    /\ \E mes \in AcceptedMessages("mes3"): /\ p_send(mes)
                                            /\ b_receive(mes)
    /\ UNCHANGED << p_input_mes, p_simple_mes, p_Nonces, a_state, a_input_nonce,
                    b_input_nonce, b_input_id, a_connect_to >>
```

Сообщение типа *mes3* принимает только агент *b*, поэтому факт передачи — посылка агентом *p* и получение агентом *b* сообщения, допустимого при такой передаче.

Для *p_transfer_mes3* локально (используя *LET/IN*) определим получение сообщения агентом *b*:

```
b_receive(mes) ==
    /\ b_state   = "wait_mes3"
    /\ msg_type' = "mes3"
    /\ b_state'  = next_b_state[b_state]
```

Агент *b* должен находиться в состоянии ожидания этого сообщения. При его получении агент *b* переходит в следующее состояние согласно протоколу.

В определении действия *p_transfer* используется функция *AcceptedMessages*, порождающая множество допустимых для передачи сообщений:

```
AcceptedMessages(type) ==
    CASE type = "mes1" -> AcceptedMessages1
      [] type = "mes2" -> AcceptedMessages2
      [] type = "mes3" -> AcceptedMessages3
      [] OTHER -> {}
```

Функция имеет один аргумент — тип сообщения. Она возвращает множество всех сообщений указанного типа, которые будут приняты соответствующим агентом.

Множество принимаемых агентом *b* сообщений типа *mes1*:

```
AcceptedMessages1 ==
    LET
        Domain == (Nonces \ {"Nb"}) \X (ID \ {"b"})

        create_mes == [ <<nonce, id>> \in Domain |->
                        <<[ key |-> "pub_key_b", item |-> [ i1 |-> nonce, i2 |-> id ] ]>> ]
    IN
    { create_mes[vec] : vec \in Domain }
```

По определению функция возвращает множество сообщений (последняя строка), где каждое из них порождается на основе вектора из множества *Domain* с помощью отображения *create_mes*. Каждый вектор из множества *Domain* — уникальный набор значений параметров, от которых зависит передаваемое сообщение. Параметрами сообщения типа *mes1* являются нонс и идентификатор. Агент *b*

ожидает получить в сообщении любые нонсы и идентификаторы, кроме своих собственных. Это отражено в определении множества *Domain*. Отображение *create_mes* конструирует сообщение на основе заданного нонса и идентификатора. Агент *b* ожидает получить сообщение, которое зашифровано только его открытым ключом.

Множество принимаемых агентом *a* сообщений типа *mes2*:

```
AcceptedMessages2 ==
    LET
        Domain == (Nonces \ {"Na"})

        create_mes == [ nonce \in Domain |->
                        <<[ key |-> "pub_key_a", item |-> [ i1 |-> "Na", i2 |-> nonce ] ]>> ]
    IN
    { create_mes[nonce] : nonce \in Domain }
```

Отображение *create_mes* порождает сообщения зашифрованные только открытым ключом агента *a*. В качестве первого элемента данных всегда передаётся нонс *Na* для подтверждения. В противном случае сообщение не будет принято агентом *a*. Параметром сообщения типа *mes2* является только второй нонс. В его качестве агент *a* ожидает получить любой нонс, кроме своего. Это отражено в определении множества *Domain*. Отображение *create_mes* конструирует сообщение на основе заданного нонса. Функция *AcceptedMessages2* возвращает множество сообщений, где каждое из них порождается на основе нонса из множества *Domain* с помощью отображения *create_mes*.

Множество принимаемых агентом *b* сообщений типа *mes3*:

```
AcceptedMessages3 ==
    { <<[ key |-> "pub_key_b", item |-> [ i1 |-> "Nb", i2 |-> "empty" ] ]>> }
```

По определению функция возвращает множество, состоящее из одного сообщения, которое зашифровано открытым ключом агента *b*. В качестве первого элемента данных передаётся нонс *Nb* для подтверждения. Второй элемент данных отсутствует. В противном случае сообщение не будет принято агентом *b*.

Обоснование использования порождающей функции выполним согласно критерию (9). Оценим мощность множества передаваемых сообщений *Messages* и мощность множества сообщений, порождаемых функцией *AcceptedMessages*. Мощность множества всех ключей:

$$|Keys\_| = |Pub\_keys| + |Priv\_keys| + |\{not\_encrypted\}| = 3 + 3 + 1 = 7.$$

Мощность множества элементов данных:

$$|DataItems| = |ID| + |Nonces| + |NodesRef| + |\{empty\}| = 3 + 3 + 2 + 1 = 9.$$

Простые сообщения типа *SimpleMessages* содержат ключ шифрования и два элемента данных. Мощность множества простых сообщений:

$$|SimpleMessages| = |Keys\_| \cdot |DataItems| \cdot |DataItems| = 7 \cdot 9 \cdot 9 = 567.$$

По определению множество *Messages* — множество векторов длины один, два и три, где компонентами являются простые сообщения. Мощность множества передаваемых сообщений:

$$|Messages| = |SimpleMessages|^1 + |SimpleMessages|^2 + |SimpleMessages|^3 = 567^1 + 567^2 + 567^3 = 182\,606\,319 \approx 2^{27,4}$$

Мощности множеств, порождаемых функцией *AcceptedMessages*:

$$|AcceptedMessages(mes1)| = |Nonces \setminus \{Nb\}| \cdot |ID \setminus \{b\}| = 2 \cdot 2 = 4;$$
$$|AcceptedMessages(mes2)| = |Nonces \setminus \{Na\}| = 2;$$
$$|AcceptedMessages(mes3)| = 1.$$

Таким образом, критерий (9) выполняется для сообщений любого типа:

$$\forall type \in MesTypes : |Messages| \gg |AcceptedMessages(type)|.$$

Верификация модели. Введём пять вспомогательных утверждений.

1. Агенты *a* и *b* завершили процесс аутентификации:

```
authentication_completed == a_state = "finalA" /\ b_state = "finalB"
```

2. Агент *a* считает, что взаимодействовал с агентом *b*:

```
A_think_connected_B == a_connect_to = "b"
```

3. Агент *b* считает, что взаимодействовал с агентом *a*:

```
B_think_connected_A == b_input_id = "a"
```

4. Агенты *a* и *b* обменялись своими нонсами:

```
AB_swap_nonces == a_input_nonce = "Nb" /\ b_input_nonce = "Na"
```

5. Нонсы агентов *a* и *b* не скомпрометированы (неизвестны злоумышленнику):

```
AB_nonces_not_compromised == "Na" \notin p_Nonces /\ "Nb" \notin p_Nonces
```

Проверим наличие в модели нормального сеанса связи между агентами *a* и *b*. Для этого определим инвариант, утверждающий, что аутентификация между агентами *a* и *b* не может успешно завершиться:

```
Authentication_AB_not_successful ==
LET Authentication_AB_successful ==
    /\ authentication_completed
    /\ A_think_connected_B
    /\ B_think_connected_A
    /\ AB_swap_nonces
    /\ AB_nonces_not_compromised
IN ~Authentication_AB_successful
```

Здесь «~» — знак логического отрицания. Условия успешного завершения аутентификации: агенты *a* и *b* завершили процесс аутентификации, агент *a* считает, что взаимодействовал с агентом *b*, агент *b* считает, что взаимодействовал с агентом *a*, агенты *a* и *b* обменялись своими нонсами, нонсы агентов *a* и *b* не скомпрометированы.

Верификатор TLC находит контрпример, демонстрирующий возможность успешного завершения аутентификации агентов *a* и *b* за шесть шагов:

```
1. a_transfer_p \*(from a to p)
    sended_mes = <<[key |-> "pub_key_b", item |-> [i1 |-> "Na", i2 |-> "a"]]>>
2. p_transfer_mes1 \*(from p to b)
    msg_kind = "existing"
    sended_mes = <<[key |-> "pub_key_b", item |-> [i1 |-> "Na", i2 |-> "a"]]>>
3. b_transfer_p \*(from b to p)
    sended_mes = <<[key |-> "pub_key_a", item |-> [i1 |-> "Na", i2 |-> "Nb"]]>>
```

```
4. p_transfer_mes2 \*(from p to a)
    msg_kind = "existing"
    sended_mes = <<[key |-> "pub_key_a", item |-> [i1 |-> "Na", i2 |-> "Nb"]]>>
5. a_transfer_p \*(from a to p)
    sended_mes = <<[key |-> "pub_key_b", item |-> [i1 |-> "Nb", i2 |-> "empty"]]>>
6. p_transfer_mes3 \*(from p to b)
    msg_kind = "existing"
    sended_mes = <<[key |-> "pub_key_b", item |-> [i1 |-> "Nb", i2 |-> "empty"]]>>
```

После каждого шага (действия) указаны значения только интересующих нас переменных (остальные не показаны). Из последовательности шагов видно, что агент *p* просто пересылает полученные им сообщения далее, выполняя роль посредника.

Проверим, всегда ли *взаимная аутентификация* между агентами *a* и *b* является *безопасной*. Данное утверждение формализуем как конъюнкцию двух высказываний, подобно тому, как это было сделано при проверке NS-протокола с использованием инструмента FDR [33].

1. Если агент *a* аутентифицировал агента *b*, то и агент *b* безопасно аутентифицировал агента *a*:

```
Secure_authentication_AB ==
(authentication_completed /\ A_think_connected_B) =>
    /\ B_think_connected_A
    /\ AB_swap_nonces
    /\ AB_nonces_not_compromised
```

Если аутентификация завершена и агент *a* считает, что взаимодействовал с агентом *b*, то и агент *b* считает, что взаимодействовал с агентом *a* при этом они обменялись своими нонсами и данные нонсы не скомпрометированы.

2. Если агент *b* аутентифицировал агента *a*, то и агент *a* безопасно аутентифицировал агента *b*:

```
Secure_authentication_BA ==
(authentication_completed /\ B_think_connected_A) =>
    /\ A_think_connected_B
    /\ AB_swap_nonces
    /\ AB_nonces_not_compromised
```

Если аутентификация завершена и агент *b* считает, что взаимодействовал с агентом *a*, то и агент *a* считает, что взаимодействовал с агентом *b* при этом они обменялись своими нонсами и данные нонсы не скомпрометированы.

Таким образом, *безопасная взаимная аутентификация* происходит тогда и только тогда, когда оба агента одновременно аутентифицируют друг друга, соблюдая правила протокола (обмениваются нонсами и не разглашают их).

Верификатор TLC подтверждает выполнение свойства *Secure_authentication_AB* и опровергает выполнение свойства *Secure_authentication_BA*, предоставляя контрпример, демонстрирующий нарушение за восемь шагов:

```
1. a_transfer_p \*(from a to p)
    a_connect_to = "p"
    sended_mes = <<[key |-> "pub_key_p", item |-> [i1 |-> "Na", i2 |-> "a"]]>>
2. p_decrypt
    p_Nonces = { "Na", "Np" }
3. p_transfer_mes1 \*(from p to b)
    msg_kind = "created"
    sended_mes = <<[key |-> "pub_key_b", item |-> [i1 |-> "Na", i2 |-> "a"]]>>
4. b_transfer_p \*(from b to p)
    sended_mes = <<[key |-> "pub_key_a", item |-> [i1 |-> "Na", i2 |-> "Nb"]]>>
5. p_transfer_mes2 \*(from p to a)
    msg_kind = "existing"
```

```
    sended_mes = <<[key |-> "pub_key_a", item |-> [i1 |-> "Na", i2 |-> "Nb"]]>>
6. a_transfer_p \*(from a to p)
    sended_mes = <<[key |-> "pub_key_p", item |-> [i1 |-> "Nb", i2 |-> "empty"]]>>
7. p_decrypt
    p_Nonces = { "Na", "Nb", "Np" }
8. p_transfer_mes3 \*(from p to b)
    msg_kind = "created"
    sended_mes = <<[key |-> "pub_key_b", item |-> [i1 |-> "Nb", i2 |-> "empty"]]>>
```

Из последовательности шагов видно, что это и есть атака Лоу (см. рис. 1). Таким образом, с помощью верификатора TLC была найдена уже известная уязвимость классического NS-протокола.

Отметим, что для проверки вышеуказанных свойств на персональном компьютере с процессором Intel Core i5-3570 3.4 ГГц и 8 ГБ оперативной памяти потребовалось несколько секунд. Без оптимизации модели (без порождающей функции *AcceptedMessages*) время верификации кардинально увеличивается — верификация не оптимизированной модели была прервана по истечении шести часов. Таким образом, предложенная оптимизация модели, как и ожидалось, сокращает время верификации.

5. Разработка и верификация модели доработанного NS-протокола

Модель доработанного протокола получим из уже имеющейся модели путем внесения в неё изменений, связанных с доработкой (полный текст TLA+ спецификации доступен по ссылке [32]). Необходимость модификации модели следует из изменения формата передаваемых сообщений.

Изменим сообщения от агента *a* (определены локально для действия *a_transfer_p*):

```
mes1 == << [ key |-> id_to_pub[a_connect_to], item |-> [ i1 |-> "node2", i2 |-> "empty" ] ],
           [ key |-> "priv_key_a",            item |-> [ i1 |-> "Na",    i2 |-> "a"     ] ] >>

mes3 == << [ key |-> id_to_pub[a_connect_to], item |-> [ i1 |-> "node2",       i2 |-> "empty" ] ],
           [ key |-> "priv_key_a",            item |-> [ i1 |-> a_input_nonce, i2 |-> "empty" ] ] >>
```

Изменим сообщение от агента *b* (определено локально для действия *b_transfer_p*):

```
mes2 == << [ key |-> id_to_pub[b_input_id], item |-> [ i1 |-> "node2",       i2 |-> "node3" ] ],
           [ key |-> "not_encrypted",       item |-> [ i1 |-> b_input_nonce, i2 |-> "empty" ] ],
           [ key |-> "priv_key_b",          item |-> [ i1 |-> "Nb",          i2 |-> "empty" ] ] >>
```

Модифицируем локально определённое действие получения сообщения типа *mes1* агентом *b*:

```
p_transfer_mes1 ==
    LET b_receive(mes) ==
        /\ b_state        = "wait_mes1"
        /\ msg_type'      = "mes1"
        /\ b_input_nonce' = mes[2].item["i1"]
        /\ b_input_id'    = mes[2].item["i2"]
        /\ b_state'       = next_b_state[b_state]
        ...
```

Нонс и идентификатор агент *b* теперь извлекает из сообщения, представленного вторым узлом.

Модифицируем локально определённое действие получения сообщения типа *mes2* агентом *a*:

```
p_transfer_mes2 ==
    LET a_receive(mes) ==
        /\ a_state        = "wait_mes2"
        /\ msg_type'      = "mes2"
        /\ a_input_nonce' = mes[3].item["i1"]
        /\ a_state'       = next_a_state[a_state]
        ...
```

Нонс агент *a* теперь извлекает из сообщения, представленного третьим узлом.

Изменим множество принимаемых агентом *b* сообщений типа *mes1*:

```
AcceptedMessages1 ==
LET
    id_for(priv_key) ==
        CHOOSE id \in (ID \ {"b"}): id_to_pub[id] = decrypt_key[priv_key]

    D1 == (Priv_keys \ {"priv_key_b"})
    D2 == { <<priv_key, id_for(priv_key)>> : priv_key \in D1 }
    D3 == D2 \X (Nonces \ {"Nb"})
    Domain == { <<d[1][1], d[2], d[1][2]>> : d \in D3 }

    create_mes ==
        [ <<priv_key, nonce, id>> \in Domain |->
                << [ key |-> "pub_key_b", item |-> [ i1 |-> "node2", i2 |-> "empty" ] ],
                   [ key |-> priv_key,    item |-> [ i1 |-> nonce,   i2 |-> id      ] ] >> ]
IN
{ create_mes[vec] : vec \in Domain }
```

Функция *AcceptedMessages1* возвращает множество сообщений (последняя строка), где каждое из них порождается с помощью отображения *create_mes* на основе вектора из множества *Domain*. Отображение *create_mes* порождает составные сообщения, где одна часть полей сообщения имеет фиксированное значение, другая — может варьироваться. Первое сообщение всегда зашифровано открытым ключом *pub_key_b*, первый элемент данных содержит ссылку на второй узел, второй элемент данных отсутствует, иначе агент *b* не примет сообщение. Поля второго сообщения могут варьироваться — обозначим их как параметры: закрытый ключ (*priv_key*), нонс (*nonce*) и идентификатор (*id*). Все допустимые значения вектора параметров (*priv_key*, *nonce*, *id*) представляют множество *Domain* — сконструируем его.

Множество допустимых закрытых ключей *D1* — все закрытые ключи, кроме ключа *priv_key_b*, так как агент *b* не ожидает получить сообщение зашифрованное его закрытым ключом. Параметр *id* не может быть произвольным и зависит от закрытого ключа. Определим множество допустимых значений вектора параметров (*priv_key*, *id*) как *D2*. Первая компонента *priv_key* вектора принадлежит множеству *D1*, а вторая (*id*) определяется для значения *priv_key* с помощью функции *id_for*(*priv_key*). Данная функция выбирает подходящий идентификатор — идентификатор того агента, чьим открытым ключом расшифровывается сообщение.

Последний параметр *nonce* — неизвестное для агента *b* число, значение которого он не может ни с чем связать, поэтому оно может быть любым, кроме своего собственного значения *Nb*. Это отражено в определении множества *D3*, которое содержит все допустимые сочетания значений всех трёх параметров. На основании *D3* сконструируем множество *Domain* путем извлечения и нужной перестановки значений параметров.

Множество принимаемых агентом *a* сообщений типа *mes2*:

```
AcceptedMessages2 ==
LET
    Domain == (Nonces \ {"Na"})
    pub_key_for_decrypt == id_to_pub[a_connect_to]
    priv_key == CHOOSE key \in Priv_keys: decrypt_key[key] = pub_key_for_decrypt

    create_mes ==
        [ nonce \in Domain |->
            << [ key |-> "pub_key_a",     item |-> [ i1 |-> "node2", i2 |-> "node3" ] ],
               [ key |-> "not_encrypted", item |-> [ i1 |-> "Na",    i2 |-> "empty" ] ],
               [ key |-> priv_key,        item |-> [ i1 |-> nonce,   i2 |-> "empty" ] ] >> ]
```

```
IN
{ create_mes[nonce] : nonce \in Domain }
```

Функция *AcceptedMessages2* возвращает множество сообщений, порождённых с помощью отображения *create_mes* на основе нонсов из множества *Domain*. Отображение *create_mes* порождает составные сообщения, состоящие из трёх простых. Первое простое сообщение зашифровано открытым ключом *pub_key_a*, чтобы агент *a* мог его расшифровать. Первый и второй элементы данных — ссылки на второе и третье сообщение соответственно. Второе сообщение не зашифровано и содержит один элемент данных — нонс *Na*, так как агент *a* ожидает увидеть именно его. Поля третьего сообщения могут варьироваться — обозначим их как параметры: закрытый ключ (*priv_key*) и нонс (*nonce*). В этом сообщении агент *a* примет любой нонс, кроме своего собственного нонса *Na* — это отражено в определении множества *Domain*. Закрытый ключ *priv_key*, которым зашифровано третье сообщение, должен быть выбран таким образом, чтобы соответствующий ключ для расшифровки этого сообщения совпадал с ключом, которым будет пользоваться агент *a* при расшифровке (*pub_key_for_decrypt*). Известно, что агент *a* будет использовать открытый ключ того агента, с которым он решил установить связь.

Множество принимаемых агентом *b* сообщений типа *mes3*:

```
AcceptedMessages3 ==
LET
    pub_key_for_decrypt == id_to_pub[b_input_id]
    priv_key == CHOOSE key \in Priv_keys: decrypt_key[key] = pub_key_for_decrypt
IN
IF b_input_id # "empty" THEN
    { << [ key |-> "pub_key_b", item |-> [ i1 |-> "node2", i2 |-> "empty" ] ],
         [ key |-> priv_key,    item |-> [ i1 |-> "Nb",    i2 |-> "empty" ] ] >> }
ELSE {}
```

Функция *AcceptedMessages3* возвращает пустое множество, если агенту *b* ещё неизвестен агент для установки связи (*b_input_id* = *empty*), иначе функция возвращает множество, состоящее из одного сообщения. Это составное сообщение содержит в себе два простых. Первое зашифровано открытым ключом *pub_key_b*, чтобы агент *b* мог его расшифровать. Первый элемент данных ссылается на второе сообщение. Закрытый ключ шифрования второго сообщения может варьироваться — обозначим его как параметр *priv_key*. Данный параметр не может быть любым и определяется следующим образом: ключ, используемый агентом *b* при расшифровке (*pub_key_for_decrypt*), должен совпадать с ключом, который соответствует закрытому ключу *priv_key*. При расшифровке агент *b* будет использовать открытый ключ того агента, который устанавливал с ним связь. Второе сообщение содержит только нонс *Nb*, так как именно его ожидает увидеть агент *b*.

Оценка мощностей порождаемых множеств. Мощность множества *D1*:

$$|D1| = |Priv\_keys \setminus \{priv\_key\_b\}| = 2.$$

Мощность множества *D2*: $|D2| = |D1|$, так как каждому элементу из множества *D1* по построению (с помощью функции *id_for*) соответствует один элемент из множества *D2*. Мощность множества *D3*:

$$|D3| = |D2| \cdot |Nonces \setminus \{Nb\}| = 2 \cdot 2 = 4.$$

Мощность множества *Domain*: $|Domain| = |D3|$, так как все элементы из множества *D3* переводятся в элементы множества *Domain*. Мощности множеств, порождаемых функцией *AcceptedMessages*:

$$|AcceptedMessages(mes1)| = |Domain| = 4;$$
$$|AcceptedMessages(mes2)| = |Nonces \setminus \{Na\}| = 2;$$
$$|AcceptedMessages(mes3)| = 1.$$

Таким образом, второй вариант функции *AcceptedMessages* порождает множества с такими же мощностями, как и в первом варианте этой функции, несмотря на то, что число параметров, от которых зависит содержимое сообщений, увеличилось.

Верификация модели. При проверке свойства *Authentication_AB_not_successful* найденный контрпример демонстрирует возможность успешного завершения аутентификации агентов *a* и *b* за шесть шагов:

```
1. a_transfer_p \*(from a to p)
    sended_mes = << [key |-> "pub_key_b",  item |-> [i1 |-> "node2", i2 |-> "empty"]],
                    [key |-> "priv_key_a", item |-> [i1 |-> "Na",    i2 |-> "a"]] >>
2. p_transfer_mes1 \*(from p to b)
    msg_kind = "existing"
    sended_mes = << [key |-> "pub_key_b",  item |-> [i1 |-> "node2", i2 |-> "empty"]],
                    [key |-> "priv_key_a", item |-> [i1 |-> "Na",    i2 |-> "a"]] >>
3. b_transfer_p \*(from b to p)
    sended_mes = << [key |-> "pub_key_a",     item |-> [i1 |-> "node2", i2 |-> "node3"]],
                    [key |-> "not_encrypted", item |-> [i1 |-> "Na",    i2 |-> "empty"]],
                    [key |-> "priv_key_b",    item |-> [i1 |-> "Nb",    i2 |-> "empty"]] >>
4. p_transfer_mes2 \*(from p to a)
    msg_kind = "existing"
    sended_mes = << [key |-> "pub_key_a",     item |-> [i1 |-> "node2", i2 |-> "node3"]],
                    [key |-> "not_encrypted", item |-> [i1 |-> "Na",    i2 |-> "empty"]],
                    [key |-> "priv_key_b",    item |-> [i1 |-> "Nb",    i2 |-> "empty"]] >>
5. a_transfer_p \*(from a to p)
    sended_mes = << [key |-> "pub_key_b",  item |-> [i1 |-> "node2", i2 |-> "empty"]],
                    [key |-> "priv_key_a", item |-> [i1 |-> "Nb",    i2 |-> "empty"]] >>
6. p_transfer_mes3 \*(from p to b)
    msg_kind = "existing"
    sended_mes = << [key |-> "pub_key_b",  item |-> [i1 |-> "node2", i2 |-> "empty"]],
                    [key |-> "priv_key_a", item |-> [i1 |-> "Nb",    i2 |-> "empty"]] >>
```

Данный сеанс связи отличается от сеанса связи в первой версии протокола только форматом передаваемых сообщений. В первой версии протокола — простые сообщения, во второй — составные.

Верификатор TLC подтверждает, что оставшиеся свойства выполняются:

- безопасная аутентификация агента *a* агентом *b* (*Secure_authentication_AB*);
- безопасная аутентификация агента *b* агентом *a* (*Secure_authentication_BA*).

Таким образом, выполнение последнего свойства показывает, что для доработанной версии протокола атака Лоу невозможна, а выполнение обоих свойств гарантирует безопасную взаимную аутентификацию между двумя честными агентами *a* и *b*. В итоге, если агенты следуют правилам протокола, то никакой злоумышленник не сможет их «обмануть» (привести к небезопасной взаимной аутентификации). Максимум, что может злоумышленник, — прервать сеанс связи между агентами *a* и *b*.

Время проверки каждого свойства занимает несколько секунд. Без оптимизации модели (без порождающей функции *AcceptedMessages*) время верификации кардинально увеличивается — верификация не оптимизированной модели была прервана по истечении шести часов. В итоге проделанная оптимизация модели с помощью порождающей функции имеет положительный эффект.

Заключение

В статье предложено использовать средство TLA+/TLC для моделирования и верификации криптографических протоколов. Результат работы продемонстрирован на примере простого криптографического протокола — протокола Нидхема-Шредера для аутентификации с открытым ключом. С помощью верификатора TLC была найдена его уже известная уязвимость. Далее была промоделирована доработанная версия протокола Нидхема-Шредера и проведена её верификация, которая показала отсутствие уязвимости, присутствующей в классической (изначальной) версии протокола.

В качестве результата работы представлено три приёма моделирования:

1. Использование трёхагентной системы;
2. Представление сообщений в виде иерархической структуры;
3. Оптимизация модели с помощью порождающей функции.

Трёхагентная система позволяет избежать построения модели, состоящей из n агентов, где параметр $n > 3$. Система состоит из двух честных агентов (инициатор и ответчик), а также одного нечестного агента — злоумышленника. Предполагается, что универсальная модель злоумышленника позволяет ему выступать в различных ролях, в том числе позволяет имитировать сразу несколько честных и нечестных агентов одновременно. Различные возможности и параметры модели злоумышленника влияют на правомерность сделанных обобщений типа «если это верно для трёхагентной системы, то верно и для n-агентной системы». Изучение влияния параметров модели злоумышленника на правомерность различных обобщений — задача для будущих работ.

В настоящей работе при моделировании протокола Нидхема-Шредера агент-злоумышленник располагал только одним собственным идентификатором и одним нонсом, что не даёт ему возможности для полноценной имитации нескольких агентов. Однако это позволило ему провести атаку Лоу, которая предполагает одновременное проведение сразу двух сеансов связи с разными агентами.

В целом трёхагентная система обеспечивает минималистичность модели (уменьшает пространство возможных состояний), снижает её сложность, что положительно сказывается при проведении верификации методом проверки модели.

Агенты взаимодействуют посредством передачи сообщений. Для представления сообщений в модели предложена *иерархическая структура*. Это позволяет формировать сложные зашифрованные сообщения и достаточно просто их обрабатывать. Простое сообщение имеет два уровня иерархии: верхний — зашифрованное сообщение целиком, нижний — элементы данных, зашифрованные в сообщении. Составное сообщение выстраивается из простых путем соединения их вершин определённым образом. При моделировании классического протокола Нидхема-Шредера достаточно простых сообщений. В доработанной версии протокола используется шифрование вложенных в передаваемое сообщение элементов данных. Такие сообщения моделировались как составные.

При определённых условиях верификатор TLC неэффективно (длительное время) оценивает выражения с квантором существования. В настоящей работе данные условия формализованы в виде критерия неэффективной работы верификатора TLC. При выполнении данного критерия предлагается делать оптимизацию модели, которая заключается в замене множества, ограничивающего квантор существования, на функцию, порождающую множество только тех элементов, которые приводят к переходу между состояниями модели. В этом случае верификатор не будет тратить время на перебор лишних элементов. Это обеспечивает сокращение времени верификации.

В итоге предложенные приёмы позволяют упростить модель и снизить время её верификации.

References

[1] *PNST 799-2022. Information Technologies. Cryptographic Protection of Information. Terms and Definitions*, in Russian. [Online]. Available: https://protect.gost.ru/document.aspx?control=7&id=246680.

[2] D. Basin, C. Cremers, and C. Meadows, “Model Checking Security Protocols”, in *Handbook of Model Checking*, Springer, 2018, ch. 22, pp. 727–762. DOI: 10.1007/978-3-319-10575-8_22.

[3] *GOST R ISO 7498-2-99. Information technology. Open systems interconnection. Basic reference model. Part 2. Security Architecture*, in Russian. [Online]. Available: https://protect.gost.ru/document.aspx?control=7&id=131456.

[4] *GOST R 56545-2015. Information protection. Vulnerabilities in information systems. Rules of vulnerabilities description*, in Russian. [Online]. Available: https://protect.gost.ru/document.aspx?control=7&id=201374.

[5] M. Roggenbach, S. A. Shaikh, and H. N. Nguyen, "Formal Verification of Security Protocols", in *Formal Methods for Software Engineering: Languages, Methods, Application Domains*, Springer, 2022, pp. 395–451, ISBN: 978-3-030-38800-3. DOI: 10.1007/978-3-030-38800-3_8.

[6] M. Pourpouneh and R. Ramezanian, "A Short Introduction to Two Approaches in Formal Verification of Security Protocols: Model Checking and Theorem Proving", *ISeCure*, vol. 8, no. 1, pp. 3–24, 2016. DOI: 10.22042/isecure.2016.8.1.1.

[7] C. Meadows, "Formal Analysis of Cryptographic Protocols", in *Encyclopedia of Cryptography, Security and Privacy*, Springer, 2019, pp. 1–3, ISBN: 978-3-642-27739-9. DOI: 10.1007/978-3-642-27739-9_876-2.

[8] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*, 1st. Springer, 2018, vol. 10, ISBN: 3319105744. DOI: 10.1007/978-3-319-10575-8.

[9] *TLA+ home*. [Online]. Available: https://lamport.azurewebsites.net/tla/tla.html.

[10] R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers", *Communications of the ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[11] *ISO/IEC 9798-1:2010. Information technology – Security techniques – Entity authentication – Part 1: General*. [Online]. Available: https://www.iso.org/ru/standard/53634.html.

[12] *ISO/IEC 11770-3:2021. Information security – Key management – Part 3: Mechanisms using asymmetric techniques*. [Online]. Available: https://www.iso.org/ru/standard/82709.html.

[13] W. Mao, *Modern Cryptography: Theory and Practice*. Williams Publishing House, 2005, p. 768, in Russian, ISBN: 5-8459-0847-7.

[14] G. Lowe, "An Attack on the Needham-Schroeder Public-Key Authentication Protocol", *Information Processing Letters*, vol. 56, no. 3, pp. 131–133, 1995.

[15] L. Lamport, *Specifying Systems: the TLA+ Language and Tools for Hardware and Software Engineers*, 1st. Addison-Wesley, 2002, p. 364, ISBN: 0-321-14306-X.

[16] M. A. Kuppe, L. Lamport, and D. Ricketts, "The TLA+ Toolbox", *Electronic Proceedings in Theoretical Computer Science*, vol. 310, pp. 50–62, 2019. DOI: 10.4204/eptcs.310.6.

[17] R. Beers, "Pre-RTL Formal Verification: an Intel Experience", in *Proceedings of the 45th annual Design Automation Conference*, 2008, pp. 806–811. DOI: 10.1145/1391469.1391675.

[18] F. Hackett, J. Rowe, and M. A. Kuppe, "Understanding Inconsistency in Azure Cosmos DB with TLA+", in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, 2023, pp. 1–12. DOI: 10.1109/ICSE-SEIP58684.2023.00006.

[19] C. Newcombe, T. Rath, F. Zhang, and et al., "How Amazon Web Services Uses Formal Methods", *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, 2015. DOI: 10.1145/2699417.

[20] Y.-M. Kim and M. Kang, "Formal Verification of SDN-based Firewalls by using TLA+", *IEEE Access*, vol. 8, pp. 52 100–52 112, 2020. DOI: 10.1109/ACCESS.2020.2979894.

[21] E. Verhulst, R. T. Boute, J. M. S. Faria, and et al., *Formal Development of a Network-Centric RTOS: Software Engineering for Reliable Embedded Systems*, 1st. Springer, 2011, ISBN: 978-1-4419-9735-7. DOI: 10.1007/978-1-4419-9736-4.

[22] V. A. Kukharenko, K. V. Ziborov, R. F. Sadykov, and et al., "Innochain: a Distributed Ledger for Industry with Formal Verification on All Implementation Levels", *Modeling and Analysis of Information Systems*, vol. 27, no. 4, pp. 454–471, 2020, in Russian. DOI: 10.18255/1818-1015-2020-4-454-471.

[23] V. Kukharenko, K. Ziborov, R. Sadykov, and et al., "Verification of Hotstuff BFT Consensus Protocol with TLA+/TLC in an Industrial Setting", in *SHS Web of Conferences*, Springer, vol. 93, 2021, pp. 77–95. DOI: 10.1051/shsconf/20219301006.

[24] H. Guo, Y. Ji, and X. Zhou, "The Development of a TLA+ Verified Correctness Raft Consensus Protocol", in *Web and Big Data*, Springer, vol. 14965, 2024, pp. 459–469. DOI: 10.1007/978-981-97-7244-5_40.

[25] R. Niyogi and A. Nath, "Formal Specification and Verification of a Team Formation Protocol using TLA+", *Software: Practice and Experience*, vol. 54, no. 6, pp. 961–984, 2024. DOI: 10.1002/spe.3307.

[26] A. Jandoubi, M. T. Bennani, O. Mosbahi, and A. El Fazziki, "Analyzing MQTT Attack Scenarios: A Systematic Formalization and TLC Model Checker Simulation", in *Evaluation of Novel Approaches to Software Engineering*, vol. 1, SciTePress, 2024, pp. 370–378. DOI: 10.5220/0012625600003687.

[27] J.-Q. Yin, H.-B. Zhu, and Y. Fei, "Specification and Verification of the Zab Protocol with TLA+", *Journal of Computer Science and Technology*, vol. 35, pp. 1312–1323, 2020. DOI: 10.1007/s11390-020-0538-7.

[28] L. Ouyang, X. Sun, R. Tang, and et al., *Multi-Grained Specifications for Distributed System Model Checking and Verification*, 2024. arXiv: 2409.14301 [cs.DC].

[29] D. Dolev and A. Yao, "On the Security of Public Key Protocols", *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983. DOI: 10.1109/TIT.1983.1056650.

[30] T. Keerthan Kumar and S. Ramu, "Formal Verfication of Security Protocol Using Spin Tool", in *International Conference on Advances in Information Technology*, IEEE, 2019, pp. 393–399. DOI: 10.1109/ICAIT47043.2019.8987376.

[31] S. Basagiannis, P. Katsaros, and A. Pombortsis, "An Intruder Model with Message Inspection for Model Checking Security Protocols", *Computers & Security*, vol. 29, no. 1, pp. 16–34, 2010. DOI: 10.1016/j.cose.2009.08.003.

[32] *Needham-Schroeder Public Key Protocol Model Checking with TLA+/TLC*. [Online]. Available: https://github.com/MaximNeyzov/NSPK-model-checking.

[33] G. Lowe, "Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR", in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 1055, Springer, 1996, pp. 147–166, ISBN: 978-3-540-49874-2. DOI: 10.1007/3-540-61042-1_43.