АЛГОРИТМ АНАЛИЗА ИСХОДНОГО КОДА C# ДЛЯ ИЗВЛЕЧЕНИЯ СТРУКТУРЫ КРИПТОГРАФИЧЕСКИХ ПРОТОКОЛОВ[1]

Бабенко Л.К.[2], Писарев И.А.[3]

Цель статьи: *разработка алгоритма анализа исходного кода для извлечения упрощенного описания криптографических протоколов в виде структурированной модели с последующей возможностью верификации безопасности криптографических протоколов.*

Метод: *использован метод парсинга исходного кода, который позволяет проводить синтаксический анализ кода с помощью древовидных структур. Так же использован метод представления данных в собственную структуру дерева, которая позволяет удобно хранить все цепочки преобразований данных в ходе исполнения программы.*

Результаты:

В данной работе представлен алгоритм анализа исходного кода языка программирования C# для извлечения структуры криптографических протоколов. Описаны особенности реализации протоколов на практике. В основе алгоритма лежит определение ключевых участков кода, содержащих специфичные для криптографических протоколов конструкции и определение цепочки преобразования переменных от состояния посылки или приема сообщений до их первоначальной инициализации с учетом возможных криптографических преобразований для составления дерева, из которого получается упрощенная структура криптографического протокола. Начата работа над разработкой анализатора исходного кода, написанного на языке программирования C# и использующего парсер Roslyn. Приведен криптографический протокол Нидхема-Шредера. Показана работа анализатора на примере этого протокола. Описано дальнейшее направление работы.

Ключевые слова: *парсинг, дерево, верификация, реализация, Roslyn, цепочки, шифрование, аутентификация.*

DOI: 10.21681/2311-3456-2018-4-46-53

Введение

Криптографические протоколы являются ядром безопасности любой системы. С помощью них осуществляется передача данных, которые нуждаются в защите от третьих лиц. Как правило, криптографический протокол разрабатывается, анализируется с помощью средств формальной верификации и в случае, если признается безопасным, получает свою реализацию на языке программирования, с помощью которого разрабатывается система. Однако, при практической реализации криптографического протокола могут возникнуть ошибки из-за человеческого фактора, допущений, которые необходимы для возможности реализации протокола, которые влекут за собой подрыв его безопасности. Таким образом, получается, что изначально сам протокол считался безопасным, но его реализация на деле не является таковой. Помимо этого, формальная верификация использует довольно абстрактные понятия и не позволяет полностью проанализировать протокол. На данный момент существуют работы, в которых рассматривается проблема извлечения абстрактной модели из исходного кода языков программирования C [1-3], Java [4-6], F# [7-12]. Большая часть из них требует специального стиля программирования для возможности работы данных алгоритмов либо использование дополнительных аннотаций в исходном коде. В работе предлагается анализировать язык программирования C#, работ по которому на данный момент нет. Кроме того, для возможности анализа нет требований к структуре исходного кода и наличия дополнительных спецификаций. Так же выходная модель протокола не будет конкретно подгоняться под существующие модели как в некоторых системах, использующих в дальнейшей формальной верификации инструмент CryptoVerif [13], а будет использована собственная структура, которую в дальнейшем можно будет перевести в любую необходимую модель для сторонних средств верификации либо разработанных самостоятельно.

1. Криптографические протоколы

Криптографический протокол является описанием распределенного алгоритма, в процессе выполнения которого несколько сторон последовательно выполняют определенные действия и обмениваются сообщениями. Любой криптографический протокол должен достигать поставленные цели (свойства) безопасности. Основными из них являются:

Секретность передаваемых данных
Целостность передаваемых данных
Аутентификация сторон

1 Работа выполнена при поддержке гранта Министерства Образования и Науки Российской Федерации № 2.6264.2017/8.9.

2 Бабенко Людмила Климентьевна, доктор технических наук, профессор, Южный Федеральный Университет «ЮФУ», Институт компьютерных технологий и информационной безопасности, г. Таганрог, Россия. E-mail: lkbabenko@sfedu.ru.

3 Писарев Илья Александрович, аспирант, Южный Федеральный Университет «ЮФУ», Институт компьютерных технологий и информационной безопасности, г. Таганрог, Россия. E-mail: ilua.pisar@gmail.com.

Цель «секретность передаваемых данных» задана для того, чтобы злоумышленник не мог узнать содержание передаваемых данных между легальными сторонами. Для достижения этих целей используется асимметричное и симметричное шифрование в зависимости от вида данных. Обычно асимметричные шифры используются для передачи небольшого объема данных, таких как ключи для других шифров, идентификаторы, случайные числа. Симметричные шифры используются для шифрования большого объема данных, однако в отличие от асимметричных шифров, использующих публичный ключ для шифрования и секретный для дешифрования, в них используется общий секретный ключ. Это накладывает необходимость использования предварительного обмена сессионным ключом шифрования либо же его выработка с помощью протоколов выработки общего сессионного ключа, такого как Диффи-Хеллман.

Цель «целостность передаваемых данных» задана для того, чтобы злоумышленник не мог модифицировать передаваемые между легальными сторонами данные. В случае обнаружения нарушения целостности осуществляется либо повторный запрос данных, либо окончание текущей сессии. Для обеспечения целостности данных могут использоваться коды аутентичности сообщения MAC, которые являются одним из режимов шифрования симметричного шифра, HMAC (hash-based message authentication code) механизм обмена данными с использованием секретного ключа и хеш-функций, подпись с помощью секретного ключа и проверка подписи с помощью открытого ключа асимметричного шифра.

Цель «аутентификация сторон» задана для того, чтобы стороны были уверены в том, что ведут взаимодействие в текущей сессии, что сообщения содержат данные настоящего времени и стороны уверены в том, что общаются именно друг с другом. Для обеспечения аутентификации может использоваться запрос-ответная схема, в которой одна сторона посылает случайное число другой стороне, а другая сторона в ответном сообщении посылает это же число, либо некоторую функцию, произведенную над этим числом, которую знает первоначальная сторона. Одной из разновидностей атак на аутентификацию является replay-attack [14]. Она заключается в повторном использовании ранее переданного легального сообщения. Для предотвращения этого могут быть использованы как случайные числа в совокупности с перегенерированием сессионных ключей взаимодействия в начале каждой сессии, так и использование временных меток в сообщении.

Таким образом, при реализации протокола используются криптографические алгоритмы асимметричного, симметричного шифрования в различных режимах, хеширование, коды аутентичности, генераторы случайных чисел. Эти алгоритмы можно реализовать самостоятельно, использовать алгоритмы, присутствующие в стандартной библиотеке, использовать алгоритмы сторонних библиотек. В данной работе рассматривается использование алгоритмов стандартных библиотек .Net Framework. Передача сообщений между сторонами, как правило, реализуется с помощью сокетов. Взаимодействие сторон осуществляется по модели клиент-сервер. Одна из сторон ждет поступающие соединения, вторая их инициирует. Успешным выполнением протокола является состояние, при котором все участвующие в сообщении стороны выполнили свою часть протокола полностью без обрывов соединения.

2. Алгоритм анализа кода

В качестве примера для описания работы алгоритма был взят протокол взаимной аутентификации Нидхема-Шредера с открытым ключом (NSPK) [15].

1. A->B: Epkb (A,NA)
2. B->A: Epka (Na,Nb)
3. A->B: Epkb (Nb)

Протокол состоит из трех сообщений. На первом этапе сторона A посылает свой идентикатор A и случайное число Na, шифрованные асимметричным шифром на открытом ключе B – pkb. Во втором сообщении сторона B посылает полученное случайное число Na от A и свое случайное число Nb, шифрованные на открытом ключе A – pka. В последнем сообщении сторона A отсылает случайное число B, шифрованное на открытом ключе B.

Алгоритм анализа использует парсер исходного кода C# *Roslyn* [16]. С помощью него можно получить древовидную структуру исходного кода, причем возможно использование фильтров. В нашем случае интересны фильтры:

InvocationExpressionSyntax – вызов выражений. Пример – области исходного кода, помеченные красным цветом.

VariableDeclarationSyntax – объявление переменных. Пример в строке исходного кода 9.

AssignmentExpressionSyntax – выражение присваивания. Пример области исходного кода, помеченные желтым цветом.

IfStatementSyntax – утверждение с оператором условия. Пример в строке исходного кода 34.

С помощью фильтров можно получить нужное выражение, после чего можно просмотреть древовидную структуру этого выражения. Например с помощью «*AssignmentExpressionSyntax*» мы можем найти выражение «M1enc1 = RSA.Encrypt(M1, true)». Выведенная линейно древовидная структура выражения представлена на Рис. 1.

Главной целью использования данного парсера является нахождение перехода от одной переменной к другой. В данном случае нас интересует переход M1enc1 -> M1. Достигается это путем поиска данных типа «IdentifierName» совместно с применением черного списка выражений. Например, здесь используется вызов метода «Encrypt», а так же ранее объявленный объект класса асимметричного шифрования «RSA», которые присутствуют в черном списке, а как раз нужные нам M1enc1 и M1 отсюда можно получить, причем при операции присвоения осуществляется поиск с начала списка, где первый элемент будет переменная, которой будет присваиваться значение, а остальные те что ниже и не входят в черный список будет являться новым присваиваемым значением.

В основе алгоритма лежит определение ключевых участков кода, содержащих специфичные для криптографических протоколов конструкции. В конечном счете, задача сводится к определению цепочки преобразования переменных от состояния посылки или приема сообщений (помечено красным) до их первоначальной

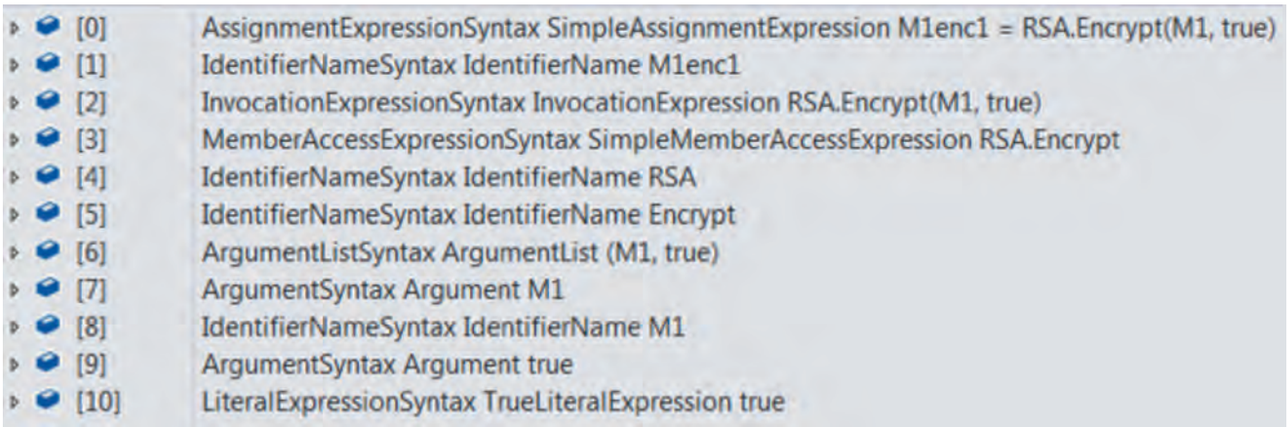

Рис 1. Древовидная структура выражения в линейном виде.

инициализации (помечено зеленым), при этом учитывая возможные криптографические преобразования (помечено желтым). В ходе построения цепочки строится дерево, узлами которого являются переменные с дополнительной информацией о них, включающей определения типа данных для конечных листьев дерева и криптографических алгоритмов в узлах дерева. Структура дерева позволяет описать все цепочки преобразований данных, поскольку данные в сообщении комбинируются различными способами, цепочки могут сильно разветвляться и соединяться.

Ниже представлен фрагмент исходного кода реализации криптографического протокола Нидхема – Шредера NSPK со стороны участника A с помеченными разным цветом ключевыми участками кода.

Определение ключевых участков

Для начала необходимо определить объявление и инициализацию:

Объектов класса *Socket*.

Объектов классов криптографических алгоритмов стандартной библиотеки, таких как алгоритм асимметричного шифрования *RSACryptoServiceProvider*, генератора случайных чисел *RNGCryptoServiceProvider* и т.д.

В представленном фрагменте исходного кода такие конструкции помечены серым цветом. Определяются переменные объекта класса *Socket*: *[socA]*, классов криптографических алгоритмов: *[rng, RSA]*.

Определение порядка обмена сообщениями

Для найденной переменной объекта класса Socket осуществляется поиск конструкций отправки и получения сообщений. В данном случае таких конструкций 3 (помечены красным цветом). На этом этапе можно построить схему взаимодействия следующего вида:

A->B: M1

B->A: M2

A->B: M3

Определение структуры сообщения

Для определения структуры сообщения необходимо построить дерево, в узлах которого содержатся переменные с дополнительной информацией. Рассмотрим пример для определения содержимого первого сообщения. Порядок алгоритма, следующий:

```
1   …
2   Socket socA = new Socket(ipAddress.AddressFamily,
3                   SocketType.Stream, ProtocolType.Tcp);
4
5   socA.Connect(remoteEP);
6
7   RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
8
9   byte A = 132;
10  byte[] NaPrev = new byte[1];
11  rng.GetBytes(NaPrev);
12  byte Na = NaPrev[0];
13  byte Nb;
14
15  byte[] M1 = new byte[2];
16  M1[0] = A;
17  M1[1] = Na;
18  byte[] M1enc;
19  using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
20  {
21          RSA.ImportParameters(rsaPB.ExportParameters(false));
22          M1enc = RSA.Encrypt(M1, true);
23  }
24  socA.Send(M1enc);
```

```
25
26  byte[] M2dec = new byte[256];
27  socA.Receive(M2dec);
28  byte[] M2;
29  using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
30  {
31          RSA.ImportParameters(rsaSA.ExportParameters(true));
32          M2 = RSA.Decrypt(M2dec, true);
33  }
34  if (Na != M2[0])
35  {
36          socA.Shutdown(SocketShutdown.Both);
37          socA.Close();
38          return;
39  }
40  Nb = M2[1];
41
42  byte[] M3 = new byte[1];
43  M3[0] = Nb;
44  byte[] M3enc;
45  using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
46  {
47          RSA.ImportParameters(rsaPB.ExportParameters(false));
48          M3enc = RSA.Encrypt(M3, true);
49  }
50  socA.Send(M3enc);
51
52  socA.Shutdown(SocketShutdown.Both);
53  socA.Close();
```

1. В качестве корня дерева берется выражение посылки первого сообщения socA.Send(M1enc). Необходимо понять содержимое переменной M1enc.

2. Для начала необходимо найти объявление переменной *M1enc* с помощью фильтра VariableDeclarationSyntax. Однако в нашем случае переменная объявлена, но не инициализирована (строка 18). В таком случае используется фильтр *AssignmentExpressionSyntax* и можно обнаружить в строке 22 присвоение значения нашей переменной. В качестве дочернего узла добавляется значение *M1enc* с пометкой «var», что означает, что это просто переменная.

3. Самый простой случай присвоения - когда значение одной переменной присваивается другой. В данном же случае ситуация более сложная. Переменной *M1enc* присваивается значение результата работы метода *Encrypt* у объекта класса асимметричного шифрования *RSACryptoServiceProvider*, который на вход принимает два параметра: что шифровать и флаг использовать ли оптимальное асимметричное шифрование с дополнением *(OAEP padding)*. На текущем этапе запоминаем, что содержимое переменной *M1* было асимметрично зашифровано и присвоено переменной для отправки сообщения 1. В структуре дерева это отображается как добавление дочернего узла *M1* с пометкой *«AsymENC»*, что означает шифрование значения переменной *M1* асимметричным шифром.

4. Аналогично пункту 2 ищем инициализацию переменной *M1*. С помощью первого фильтра можно выяснить, что переменной является одномерный массив, вмещающий в себя 2 элемента (строка 15). С помощью второго фильтра необходимо найти присвоение значений нашему массиву. Это строки 16 и 17. К узлу *M1* добавляются два дочерних элемента *Na* и *A* с пометкой *«var»*.

5. Для переменной A конечное значение можно найти с помощью первого фильтра VariableDeclarationSyntax (строка 9). Здесь происходит статичная инициализация в исходном коде – присваивается значение 132. Человеку достаточно просто понять, что это изначальное значение, но для автоматизированного определения этого факта нужно понять, что перед нами не переменная. Одним из способов решения этой проблемы является повторный поиск конечного значения переменной 132 и поскольку больше в коде конструкции присвоения «132 =» не обнаруживается, такое значение является конечным. В структуре дерева для узла A добавляется конечный лист инициализации «*byte A* = 132;» с пометкой «DATA», что означает наличие каких-то смысловых данных в переменной *A*.

6. Для переменной Na поиск осуществляется далее. В строке 12 обнаруживается, что этой переменной присваивается значение элемента массива *NaPrev*. Добавляется дочерний элемент с пометкой *«var»*.

7. С помощью фильтров далее ищется объявление массива и его инициализация. Инициализация происходит в строке 11 вызовом некоторого метода у переменной rng, которая в свою очередь является объектом класса генератора случайных чисел *RNGCryptoServiceProvider,* таким образом, значение этой переменной определяется как случайное число. В структуре дерева добавляется последний лист *«rng. GetBytes(NaPrev);»* с пометкой *«RANDOM»*, что означает выработку случайного числа.

8. Дальнейший поиск инициализация для текущих листьев ничего не дает, поэтому структура дерева считается конечной. Выходной вид дерева показан на Рис. 2

и он соответствует следующей цепочке: *Send(M1enc)* -> M1enc = E(M1) -> M1 = {A, Na} -> A = 132, Na = rand()

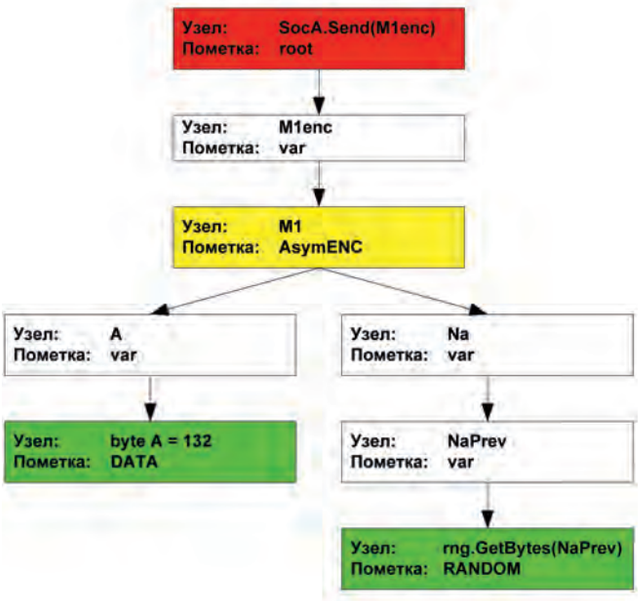

Рис. 2. Выходной вид дерева

9. Полученное дерево необходимо сократить, оставив только корень, узлы с криптографическими функциями и листы с инициализацией. В итоге сокращенное дерево примет вид как на Рис. 3.

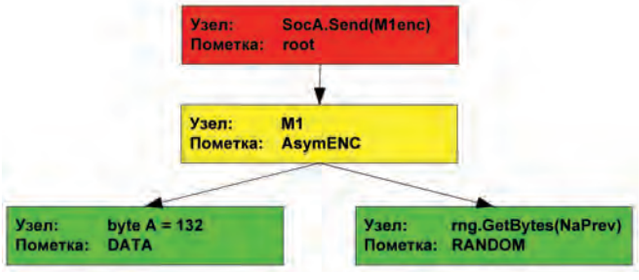

Рис. 3. Сокращенное дерево

10. Из полученного сокращенного дерева необходимо получить упрощенную структуру сообщения. Осуществляется прямой обход в глубину для вывода дерева в линейном виде без использования содержимого узлов. В результате получается следующая структура для первого сообщения: AsymENC(DATA,RANDOM)

11. Аналогично получается структура для остальных сообщений после чего формируется итоговое описание криптографического протокола.

Возврат полученных данных

В первом сообщении протокола обе переменные были получены на посылающей стороне. Однако в дальнейших сообщениях присутствуют части, которые сначала принимаются, а потом отсылаются обратно. Здесь возникает сложность обнаружения таких моментов. Сложность в данном случае возникает в сообщении 3, в котором отсылается значение случайного числа, полученного в сообщении 2. Таким образом, необходимо построить следующую цепочку: Send(M3enc) -> M3enc = E(M3) -> M3 = Nb -> Nb = M2 -> M2 = Dec(M2dec) -> Recv(M2dec). Однако возникает проблема. В сообщении 2 содержится 2 случайных числа, которые являются элементами массива. Сообщение 2 принимается и дешифруется, но необходимо понять какой именно из элементов используется. В данном исходном коде случай наиболее простой, и в нем используется индексация через оператор квадратных скобок. Для этого необходимо найти все обращения к массиву переменной M2. Здесь помимо использования фильтров *VariableDeclarationSyntax*, *AssignmentExpressionSyntax* необходимо использовать фильтр для поиска условий *IfStatementSyntax*, чтобы обнаружить конструкцию в строке 34. Таким образом, в коде используется два момента, в которых обращение идет по индексу 0 и 1 отдельно. Это значит, что элементы массива и есть 2 элемента содержимого сообщения 2. В итоге первоначальным значением для элемента Nb сообщения 3 служит 2 элемент Nb в присланном сообщении 2 B->A: E(Na,Nb). При зеркальном анализе исходного кода стороны B будет выявлено, что изначальное значение элемента Nb в сообщении 2 является случайное число, сгенерированное стороной B. Аналогичным способом происходит обнаружение первоначальной инициализации в сообщении 2 на стороне B, в котором первый элемент Na сообщения 2 является случайным числом, присланным стороной A в первом сообщении A->B: E(A,NA).

Выходная структура протокола

В процессе получения цепочек для поиска исходных инициализаций данных в каждом сообщении отдельно для каждой стороны формируется упрощенная структура криптографического протокола. Эти упрощенные структуры сливаются в одну единую выходную структуру протокола. На данном этапе работы принято, что изначальное определение содержимого сообщения состоит из смысловых данных DATA (идентификаторы сторон, ключи, временные метки, данные, полученные от пользователя) и случайных данных RANDOM (случайно сгенерированные числа). Таким образом, в результате работы описанного алгоритма получается следующий криптографический протокол:

A->B: E(DATA1,RANDOM1)
B->A: E(Recv1{RANDOM1}, RANDOM2)
A->B: E(Recv2{RANDOM2})

Таким образом описывается протокол Нидхема – Шредера. В первом сообщении сторона A посылает шифрованные асимметрично свой идентификатор DATA1 и случайное число RANDOM1. Во втором сообщении сторона B посылает присланное в первом сообщении случайное число Recv1{RANDOM1} и свое случайное число RANDOM2, шифрованные асимметрично. В последнем сообщении сторона A посылает присланное во втором сообщении случайное число Recv2{RANDOM2}. В результате получена структура протокола, которая на данный момент позволяет отразить вид шифрования и содержимое сообщений, состоящее из типов смысловые данные DATA и случайные числа RANDOM.

3. Дальнейшее направление работы

Дальнейшее направление работы в первую очередь включает в себя разбивку смысловых данных DATA на классы:

Идентификаторы сторон
Ключи
Временные метки
Коды аутентичности
Данные, полученные от пользователя

Так же важным моментом является определение принадлежности ключа какой-либо из сторон в случае асимметричного шифрования, и списку сторон в случае симметричного шифрования.

После решения этих задач возможно будет составить полноценную структуру криптографического протокола, которую можно транслировать в языки спецификаций для средств верификации безопасности протоколов, таких как Avispa [17], Scyther[18], CryptoVerif [13], ProVerif [19] и другие. Кроме того, возможно будет использовать динамический анализ реализации криптографического протокола. На основе выходной структуры протокола можно сгенерировать его реализацию в упрощенном виде [20] и уже ее тестировать с помощью динамического анализа, который будет заключаться в проведении реальных атак на аутентификацию, секретность, целостность а так же на корректность исполнения протокола при таких активных атаках.

Заключение

В ходе работы был представлен алгоритм анализа исходного кода языке программирования C# для извлечения структуры криптографических протоколов, основанный на определении ключевых участков кода, содержащих специфичные для криптографических протоколов конструкции и определению цепочки преобразования переменных от состояния посылки или приема сообщений до их первоначальной инициализации для меня кажется наоборот от начальной инициализации до пересылки с учетом возможных криптографических преобразований для составления дерева, из которого получается упрощенная структура криптографического протокола. Приведен пример работы алгоритма анализа исходного кода для реализации криптографического протокола Нидхема-Шредера с открытым ключом. Показана выходная структура криптографического протокола. Для дальнейшей возможности применения формальной верификации протоколов и динамического анализа необходимо дополнительно произвести классификацию смысловых данных и определение принадлежности ключей какой-либо стороне или сторонам.

Рецензент: *Цирлов Валентин Леонидович, кандидат технических наук, доцент Московского государственного технического университета им. Н.Э. Баумана, г. Москва, Россия. E-mail: v.tsirlov@bmstu.ru*

Литература

1. Chaki S., Datta A. ASPIER: An automated framework for verifying security protocol implementations. Computer Security Foundations Symposium, 2009. CSF'09. – IEEE, 2009. – P. 172-185.
2. Goubault-Larrecq J., Parrennes F. Cryptographic protocol analysis on real C code.I nternational Workshop on Verification, Model Checking, and Abstract Interpretation. – Springer, Berlin, Heidelberg, 2005. – P. 363-379.
3. Goubault-Larrecq J., Parrennes F. Cryptographic protocol analysis on real C code. – Technical report, Laboratoire Spécification et Vérification, Report LSV-09-18, 2009.
4. Jürjens J. Using interface specifications for verifying crypto-protocol implementations. Workshop on foundations of interface technologies (FIT). – 2008.
5. Jürjens J. Automated security verification for crypto protocol implementations: Verifying the jessie project.Electronic Notes in Theoretical Computer Science. – 2009. – V. 250. – N 1. – P. 123-136.
6. O'Shea N. Using Elyjah to analyse Java implementations of cryptographic protocols. Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPAWITS-2008). – 2008.
7. Backes M., Maffei M., Unruh D. Computationally sound verification of source code.Proceedings of the 17th ACM conference on Computer and communications security. – ACM, 2010. – P. 387-398.
8. Bhargavan K. et al. Cryptographically verified implementations for TLS. Proceedings of the 15th ACM conference on Computer and communications security. – ACM, 2008. – P. 459-468.
9. Bhargavan K., Fournet C., Gordon A. D. Verified reference implementations of WS-Security protocols.International Workshop on Web Services and Formal Methods. – Springer, Berlin, Heidelberg, 2006. – P. 88-106.
10. Bhargavan K. et al. Verified interoperable implementations of security protocols. ACM Transactions on Programming Languages and Systems (TOPLAS). – 2008. – V. 31. – N 1. – P. 5.
11. Bhargavan K. et al. Verified implementations of the information card federated identity-management protocol. Proceedings of the 2008 ACM symposium on Information, computer and communications security. – ACM, 2008. – P. 123-135.
12. Bhargavan K. et al. Cryptographically verified implementations for TLS. Proceedings of the 15th ACM conference on Computer and communications security. – ACM, 2008. – P. 459-468.
13. Blanchet B., Pointcheval D. Automated security proofs with sequences of games. Annual International Cryptology Conference. – Springer, Berlin, Heidelberg, 2006. – P. 537-554.
14. Syverson P. A taxonomy of replay attacks [cryptographic protocols]. Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings. – IEEE, 1994. – P. 187-191.
15. Needham R. M., Schroeder M. D. Using encryption for authentication in large networks of computers. Communications of the ACM. – 1978. – V. 21. – N. 12 – P. 993-999.
16. Capek P., Kral E., Senkerik R. Towards an empirical analysis of. NET framework and C# language features' adoption. Computational Science and Computational Intelligence (CSCI), 2015 International Conference on. – IEEE, 2015. – P. 865-866.
17. Viganò L. Automated security protocol analysis with the AVISPA tool. Electronic Notes in Theoretical Computer Science. – 2006. – V. 155. – P. 61-86.
18. Cremers C. J. F. The scyther tool: Verification, falsification, and analysis of security protocols. International Conference on Computer Aided Verification. – Springer, Berlin, Heidelberg, 2008. – P. 414-418.
19. Küsters R., Truderung T. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. Computer Security Foundations Symposium, 2009. CSF'09. – IEEE, 2009. – P. 157-171.

20. Jeon C. W., Kim I. G., Choi J. Y. Automatic generation of the C# code for security protocols verified with Casper/FDR. – IEEE, 2005. – P. 507-510.

ALGORITHM OF C# SOURCE CODE ANALYSIS FOR EXTRACTING THE STRUCTURE OF CRYPTOGRAPHIC PROTOCOLS[4]

Babenko L.K.[5], Pisarev I.A.[6]

Article goal: *the development of an algorithm for analyzing the source code to extract a simplified description of cryptographic protocols in the form of a structured model with the subsequent possibility of verifying the security of cryptographic protocols.*

Method: *the source code parsing method is used, which allows parsing code with the help of tree structures. We also used the method of presenting data into our own tree structure, which allows you to conveniently store all the data transformation chains during program execution.*

Results: *This paper presents an algorithm for analyzing the source code of the C # programming language to extract the structure of cryptographic protocols. The features of the implementation of protocols in practice are described. The algorithm is based on the definition of key code sections that contain cryptographic protocol-specific constructions and the definition of a chain of variable transformations from the state of sending or receiving messages to their initial initialization, taking into account possible cryptographic transformations to compose a tree, from which the simplified structure of the cryptographic protocol is obtained. Work has begun on the development of a source code analyzer written in the C # programming language and using the Roslyn parser. The Nidham-Schröder cryptographic protocol is given. The analyzer operation is shown on the example of this protocol. The further direction of work is described.*

Keywords: *parsing, tree, verification, implementation, Roslyn, chains, encryption, authentication.*

References

1. Chaki S., Datta A. ASPIER: An automated framework for verifying security protocol implementations. Computer Security Foundations Symposium, 2009. CSF'09. – IEEE, 2009. – P. 172-185.
2. Goubault-Larrecq J., Parrennes F. Cryptographic protocol analysis on real C code.I nternational Workshop on Verification, Model Checking, and Abstract Interpretation. – Springer, Berlin, Heidelberg, 2005. – P. 363-379.
3. Goubault-Larrecq J., Parrennes F. Cryptographic protocol analysis on real C code. – Technical report, Laboratoire Spécification et Vérification, Report LSV-09-18, 2009.
4. Jürjens J. Using interface specifications for verifying crypto-protocol implementations. Workshop on foundations of interface technologies (FIT). – 2008.
5. Jürjens J. Automated security verification for crypto protocol implementations: Verifying the jessie project.Electronic Notes in Theoretical Computer Science. – 2009. – V. 250. – N 1. – P. 123-136.
6. O'Shea N. Using Elyjah to analyse Java implementations of cryptographic protocols. Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPAWITS-2008). – 2008.
7. Backes M., Maffei M., Unruh D. Computationally sound verification of source code.Proceedings of the 17th ACM conference on Computer and communications security. – ACM, 2010. – P. 387-398.
8. Bhargavan K. et al. Cryptographically verified implementations for TLS. Proceedings of the 15th ACM conference on Computer and communications security. – ACM, 2008. – P. 459-468.
9. Bhargavan K., Fournet C., Gordon A. D. Verified reference implementations of WS-Security protocols.International Workshop on Web Services and Formal Methods. – Springer, Berlin, Heidelberg, 2006. – P. 88-106.
10. Bhargavan K. et al. Verified interoperable implementations of security protocols. ACM Transactions on Programming Languages and Systems (TOPLAS). – 2008. – V. 31. – N 1. – P. 5.
11. Bhargavan K. et al. Verified implementations of the information card federated identity-management protocol. Proceedings of the 2008 ACM symposium on Information, computer and communications security. – ACM, 2008. – P. 123-135.
12. Bhargavan K. et al. Cryptographically verified implementations for TLS. Proceedings of the 15th ACM conference on Computer and communications security. – ACM, 2008. – P. 459-468.
13. Blanchet B., Pointcheval D. Automated security proofs with sequences of games. Annual International Cryptology Conference. – Springer, Berlin, Heidelberg, 2006. – P. 537-554.
14. Syverson P. A taxonomy of replay attacks [cryptographic protocols]. Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings. – IEEE, 1994. – P. 187-191.

4 Работа выполнена при поддержке гранта Министерства Образования и Науки Российской Федерации № 2.6264.2017/8.9.

5 Lyudmila K. Babenko, Dr.Sc., Professor for the Southern Federal University «SFedU», Institute of Computer Technologies and Information Security, Taganrog, Russia. E-mail: lkbabenko@sfedu.ru.

6 Ilya I. Pisarev, Postgraduate at the Southern Federal University «SFedU», Institute of Computer Technologies and Information Security, Taganrog, Russia. E-mail: ilua.pisar@gmail.com.

15. Needham R. M., Schroeder M. D. Using encryption for authentication in large networks of computers. Communications of the ACM. – 1978. – V. 21. – N. 12 – P. 993-999.
16. Capek P., Kral E., Senkerik R. Towards an empirical analysis of. NET framework and C# language features' adoption. Computational Science and Computational Intelligence (CSCI), 2015 International Conference on. – IEEE, 2015. – P. 865-866.
17. Viganò L. Automated security protocol analysis with the AVISPA tool. Electronic Notes in Theoretical Computer Science. – 2006. – V. 155. – P. 61-86.
18. Cremers C. J. F. The scyther tool: Verification, falsification, and analysis of security protocols. International Conference on Computer Aided Verification. – Springer, Berlin, Heidelberg, 2008. – P. 414-418.
19. Küsters R., Truderung T. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. Computer Security Foundations Symposium, 2009. CSF'09. – IEEE, 2009. – P. 157-171.
20. Jeon C. W., Kim I. G., Choi J. Y. Automatic generation of the C# code for security protocols verified with Casper/FDR. – IEEE, 2005. – P. 507-510.