УДК 004.052.42

# Формальная верификация реализации хэш-функции «Стрибог» с «Группой Астра»*

*Кондратьев Д.А. (Институт систем информатики им. А.П. Ершова СО РАН)*

*Бояндин Л.К. (Новосибирский государственный университет)*

*Гончар Г.Е. (Новосибирский государственный университет)*

*Марченко В.В. (Московский государственный университет им. М.В. Ломоносова, Институт системного программирования им. В.П. Иванникова РАН)*

*Обухова А.А.*

*Разбитнова Ю.Ю. (Новосибирский государственный университет)*

*Хованская А.С.*

*Янбулатов Д.Р. (Новосибирский государственный университет)*

*при поддержке Математического центра в Академгородке, соглашение с Министерством науки и высшего образования Российской Федерации № 075-15-2025-349

Обеспечение доверия к реализациям криптостойких алгоритмов является актуальной задачей современного программирования. К правильности работы таких программных систем предъявляются повышенные требования, поэтому для доказательства корректности таких программ относительно спецификаций применяют дедуктивную верификацию. В данной статье описана работа в прогрессе по доказательству корректности реализации хэш-функции «Стрибог» из ядра Linux относительно ГОСТ Р 34.11-2012. Данная работа стартовала на проекте «Формальная верификация реализации хэш-функции «Стрибог» с «Группой Астра»» на Большой Математической Мастерской 2025 года. В качестве результатов работы мы презентуем формализацию ГОСТ Р 34.11-2012 в системе интерактивного доказательства Rocq. Данная формализация является функциональной спецификацией любой реализации хэш-функции «Стрибог». Также мы презентуем задание спецификаций для базовых функций реализации хэш-функции «Стрибог» из ядра Linux и методы упрощения доказательства таких функций с помощью задания набора лемм о связи структур данных из функциональной спецификации и из данной реализации.

*Ключевые слова*: дедуктивная верификация, логика Хоара, функциональная спецификация, хэш-функция «Стрибог», ГОСТ Р 34.11-2012, Rocq, Verified Software Toolchain

## 1. Введение

В настоящее время хэш-функции активно применяются в программных системах, где ошибки недопустимы, например, системах управления правами доступа,
блокчейн-системах, системах электронной цифровой подписи и т.д. Поэтому, актуальной задачей является обеспечение доверия к реализациям хэш-функций. ГОСТ Р 34.11-2012 [4] установил текущий стандарт хэширования в Российской Федерации, основанный на хэш-функции «Стрибог», где для вычисления хэша каждой 512-битовой части входных данных применяется функций сжатия на базе преобразования XLPS, в котором используются xor (X), линейное преобразование (L), подстановка (P) и нелинейное преобразование (S) [3, 24, 37, 54]. Особенно важно проверить корректность реализации хэш-функции «Стрибог» из ядра Linux [27], активному применению которой в российском программном обеспечении способствуют наличие отечественного стандарта ГОСТ Р 34.11-2012 на данную функцию и широкое внедрение в России операционных систем на базе ядра Linux, например, операционной системы Astra Linux [14], разработанной Группой Астра. Так как к корректности реализаций подобных функций предъявляются повышенные требования, то в таких случаях тестирования не достаточно. Для доказательства корректности программ применяется формальная верификация [1, 2, 8, 12, 25]. На первоначальном этапе верифи-

кации важно проверить корректность программы относительно функциональных свойств, описывающих взаимосвязь между входными и выходными данными. Для доказательства корректности программ относительно функциональных свойств применяется такой вид формальной верификации, как дедуктивная верификация программ [11, 13, 17, 30, 33].

Дедуктивная верификация основана на логике Хоара [22, 23, 31, 35], позволяющей свести задачу проверки корректности программ относительно спецификаций к задаче доказательства формул, называемых условиями корректности программы. В качестве спецификаций используются предусловия, являющиеся ограничениями на входные данные программы, и постусловия, представляющие собой ограничения на выходные данные программы и на связь входных и выходных данных программы. Упомянутое сведение осуществляется на основе правил из логики Хоара для всех конструкций языка программирования. Такой набор правил называется аксиоматической семантикой языка программирования. На базе аксиоматических семантик реализуют системы дедуктивной верификации программ, в которых для доказательства условий корректности применяют системы интерактивного доказательства теорем [34].

Важным видом аксиоматической семантики является прямое прослеживание [28], которое основано на применяемых к самой первой инструкции в программе правилах, что позволяет при выводе условий корректности продвигаться по инструкциям программы в порядке их исполнения. В случае, когда инструкция программы содержит работу с памятью через указатели, применяется такой специальный вид логики Хоара, как сепарационная логика [36, 49, 50, 52, 53]. Такая логика позволяет упростить работу с памятью с помощью разделения памяти на отдельные непересекающиеся участки и независимую работу с такими участками.

В данной статье описана работа в прогрессе по проекту дедуктивной верификации реализации хэш-функции «Стрибог» из ядра Linux. В качестве верифицируемой функциональной спецификации используется ГОСТ Р 34.11-2012. Отметим, что на текущем этапе проекта мы не верифицируем криптографические свойства [15, 18, 32, 42, 57] из-за приоритетности верификации функциональных свойств. Доказательство корректности выполняется с помощью системы дедуктивной верификации Verified Software Toolchain (VST) [20, 21, 26], основанной на погружении программы и ее спецификаций в среду интерактивной системы доказательства теорем Rocq (ранее называлась Coq) [51]. Для генерации условий корректности в системе VST используется прямое прослеживание и

сепарационная логика.

Данная работа выполняется на проекте «Формальная верификация реализации хэш-функции «Стрибог» с «Группой Астра»» на Большой Математической Мастерской 2025 года. Заказчиком данного проекта является Тимофей Юрьевич Черганов из Группы Астра. Куратором проекта является Дмитрий Александрович Кондратьев, к.ф.-м.н., научный сотрудник ИСИ СО РАН и старший преподаватель НГУ. В команду проекта входят авторы статьи, а именно:

- Бояндин Лев Константинович
- Гончар Глеб Евгеньевич
- Марченко Вадим Витальевич
- Обухова Алиса Андреевна
- Разбитнова Юлия Юрьевна
- Хованская Анна Станиславовна
- Янбулатов Денис Ринатович

Таким образом, вместе с куратором, проект выполняют восемь исполнителей. Коллектив проекта представлен на рисунке 1. Дополнительной задачей проекта является создание коллектива, способного выполнять задачи по дедуктивной верификации С-программ.

Основная задача проекта состоит в формальной верификации реализации хэш-функции «Стрибог» из ядра Linux и разработка подхода к формальной верификации реализаций криптографических функций. В данной статье мы презентуем такие результаты нашей работы, как формализацию ГОСТ Р 34.11-2012 в системе Rocq и задание лемм для теории предметной области, что может быть применимо в других подобных исследованиях. Результаты выполнения работы общедоступны в репозитории проекта [55].

**Обзор родственных работ** В качестве основной родственной работы отметим статью про дедуктивную верификацию хэш-функции SHA-256 в системе VST [19]. В данной работе, также как и в нашем исследовании, верифицировались не криптографические, а функциональные свойства, заданные стандартом на хэш-функцию. Однако из-за разницы стандартов в нашем случае в системе Rocq потребовалось задавать новые типы данных для хранения битовых векторов. Отметим, что мы использовали более гибкое представление для битовых векторов, параметризованное по их длине.

Также в качестве родственного исследования рассмотрим работы по формальной ве-

*Рис. 1. Коллектив проекта «Формальная верификация реализации хэш-функции «Стрибог» с «Группой Астра»»*

рификации такой части операционной системы Astra Linux, как система управления правами доступа [5–7]. Данные работы основаны на таком методе формальной верификации, как проверка моделей (model checking), который основан на проверке выполнение свойств на модели, описывающей состояния программной системы и переходы между ними. С одной стороны, такой подход, в отличие от дедуктивной верификации, позволяет автоматически проводить формальную верификацию. С другой стороны, в отличие от дедуктивной верификации, используемая при таком подходе абстракция системы не позволяет верифицировать низкоуровневые детали в виде реализации лежащих в основе хэш-функций. Для дедуктивной верификации частей операционной системы Astra Linux применяют систему дедуктивной верификации AstraVer [9, 29, 43, 45, 56]. Но используемые в системе AstraVer для доказательства условий корректности такие автоматические системы, как SMT-решатели, могут не справляться с моделированием работы с памятью в C-программах [44]. Применяемый нами подход, основанный на реализации сепарационной логики в системе VST, требует ручной работы при доказательстве корректности программ

с указателями, но при этом позволяет вручную приводить доказательство к успешному завершению.

Кроме того, в качестве родственного исследования рассмотрим проект по созданию системы C-lightVer для дедуктивной верификации C-программ [40, 41, 46]. Данная система позволяет упростить доказательство с помощью генерации вспомогательных лемм по определенным шаблонам [38, 39]. Но данная система, в отличие от применяемой нами системы VST, применима к ограниченному подмножеству языка программирования C [47, 48].

Родственным нашему проекту мероприятием является серия российских соревнований по формальной верификации программ VeHa. В рамках данной серии на текущий момент состоялись два соревнования, VeHa-2023 [16] и VeHa-2024 [10]. На данных соревнованиях команды исследователей также решают сложные для формальной верификации задачи. Но на нашем проекте, в отличие от соревнований серии VeHa, временные рамки намного более мягкие (две недели вместо трех дней), и, самое главное, на нашем проекте есть нацеленность на продолжение работы и после завершения самого мероприятия (Большой Математической Мастерской 2025 года).

**Структура статьи** Данная статья имеет следующую структуру: в главе 2 описаны применяемый нами метод дедуктивной верификации программ, применяемая нами система дедуктивной верификации VST и верифицируемая нами хэш-функция «Стрибог», в главе 3 описана проводимая нами дедуктивная верификация реализации хэш-функции «Стрибог», в заключении приведен список наших результатов и описаны наши планы на будущее, в Приложении A приведена предназначенная для упрощения доказательства корректности реализации XLPS лемма и схема ее доказательства.

**Благодарности** Авторы статьи выражают благодарность Группе Астра, а также лично Тимофею Юрьевичу Черганову, за запуск проекта «Формальная верификация реализации хэш-функции «Стрибог» с «Группой Астра»». Также авторы статьи выражают благодарность организаторам Большой Математической Мастерской 2025 года за организацию и поддержку проекта.

## 2. Основы дедуктивной верификации в системе VST и основы работы хэш-функции «Стрибог»

В данном разделе приводится описание используемых нами функций, методов и инструментария, а именно рассмотрены применяемый нами метод дедуктивной верификации программ, применяемая нами система дедуктивной верификации VST и верифицируемая нами хэш-функция «Стрибог».

## 2.1. Дедуктивная верификация программ

В 1969 году Ч. Хоар ввел способ задания аксиоматической семантики, ставший основой метода дедуктивной верификации программ. Подход Хоара заключается в том, чтобы представлять текст программы как особое отношение между утверждениями. Базовыми формулами в рассматриваемом подходе являются тройки Хоара $\{P\}\ S\ \{Q\}$, где $P$ — предусловие (логическая формула), $S$ — программа, $Q$ — постусловие (логическая формула). Частичная корректность тройки Хоара $\{P\}\ S\ \{Q\}$ означает, что "если предусловие $P$ истинно перед исполнением фрагмента программы $S$, и, если исполнение $S$ завершилось, тогда постусловие $Q$ выполняется после его завершения". Правила вывода задаются в виде

$$\frac{\psi_1, ..., \psi_n}{\varphi}$$

где $\psi_1$,...,$\psi_n$ – посылки правила вывода (набор троек Хоара и логических формул) и $\varphi$ — заключение правила вывода (тройка Хоара). Данная нотация означает, что $\varphi$ выводимо при гипотезе $\psi_1$,...,$\psi_n$. Семантика простых операторов (например, присваивания) задается, как правило, с помощью набора аксиом, а любого сложного оператора (например, оператора последовательного исполнения) – с помощью правила вывода. Логическая система, содержащая аксиомы и правила вывода для всех синтаксических форм языка программирования, называется логикой Хоара или аксиоматической семантикой языка.

Генератор условий корректности осуществляет вывод в автоматическом режиме по аксиоматической семантике и сводит частичную корректность тройки Хоара $\{P\}\ S\ \{Q\}$ к истинности некоторого числа лемм, называемых условиями корректности, в предметной области. Доказуемости этих лемм достаточно для частичной корректности исходной аннотированной программы. Реализацией генератора условий корректности является система дедуктивной верификации. В нашем проекте в качестве системы дедуктивной верификации используется VST.

## 2.2. Система дедуктивной верификации VST

Рассмотрим схему дедуктивной верификации программ в системе VST, изображенную на рисунке 2.

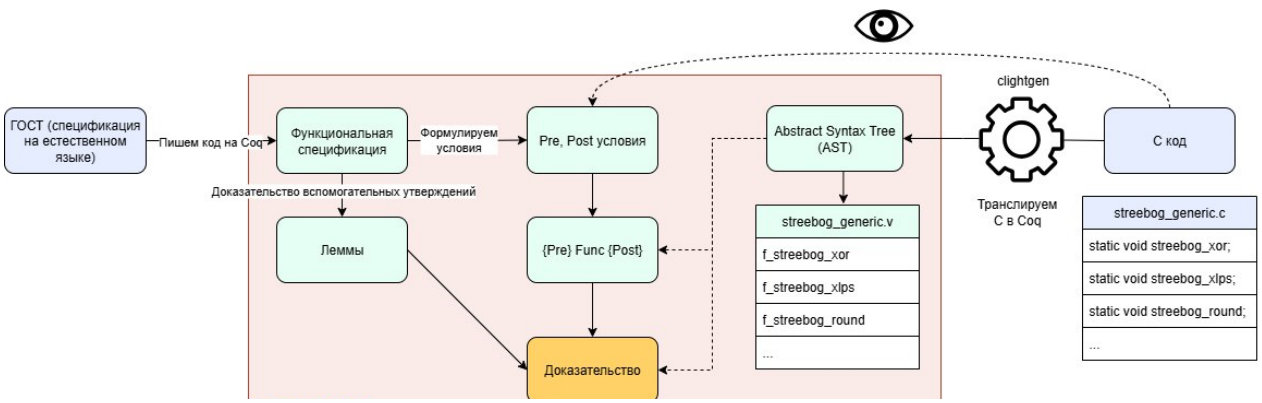


*Рис. 2.* Схема дедуктивной верификации программ в системе VST

Рассмотрим этапы верификации программ в системе VST в соответствии с данной схемой:

1. Задание спецификации программы в формальном виде на языке Gallina, входном языке системы доказательства Rocq. В случае нашего проекта функциональной спецификацией является ГОСТ Р 34.11-2012, то есть спецификация задана на естественном языке, и на данном этапе происходит задание стандарта ГОСТ Р 34.11-2012 в системе доказательства Rocq.
2. Задание для верифицируемых функций предусловий и постусловий. При этом для записи формул, выражающих предусловие и постусловие, активно используются заданные ранее определения из функциональной спецификации, а также имена переменных из C-программы.
3. Трансляция C-программы с помощью утилиты clightgen в абстрактное синтаксическое дерево (AST) в системе Rocq.
4. Связывание предусловий и постусловий с программой. Такое связывание происходит с помощью подключения модуля с AST-представлением программы, а затем заданием с помощью специальной функциональности VST леммы о корректности программы относительно предусловия и постусловия.
5. Задание дополнительных лемм для упрощения доказательства корректности программы и их доказательство. Такие леммы вместе с функциональной спецификацией формируют теорию предметной области.

6. Доказательство корректности программы относительно предусловия и постусловия. Такое доказательство происходит с помощью прямого прослеживания программы и доказательства полученных условий корректности прямо в момент их генерации при прямом прослеживании программы. Для упрощения доказательства возникающих таким образом условий корректности полезно использовать леммы из теории предметной области. Если при таком прямом прослеживании удается дойти до конца программы и доказать при этом все условия корректности, то программа корректна относительно предусловия и постусловия.

Отметим, что некоторые из данных этапов можно выполнять одновременно, например, задание спецификаций программ в формальном виде и трансляцию C-программы в AST-представление, или, например, задание лемм для упрощения доказательства корректности и само доказательство корректности. В нашем проекте данная схема применяется для дедуктивной верификации реализации хэш-функции «Стрибог» из ядра Linux.

## 2.3. Хэш-функция «Стрибог»

Рассмотрим схему работы хэш-функции «Стрибог», изображенную на рисунке 3.

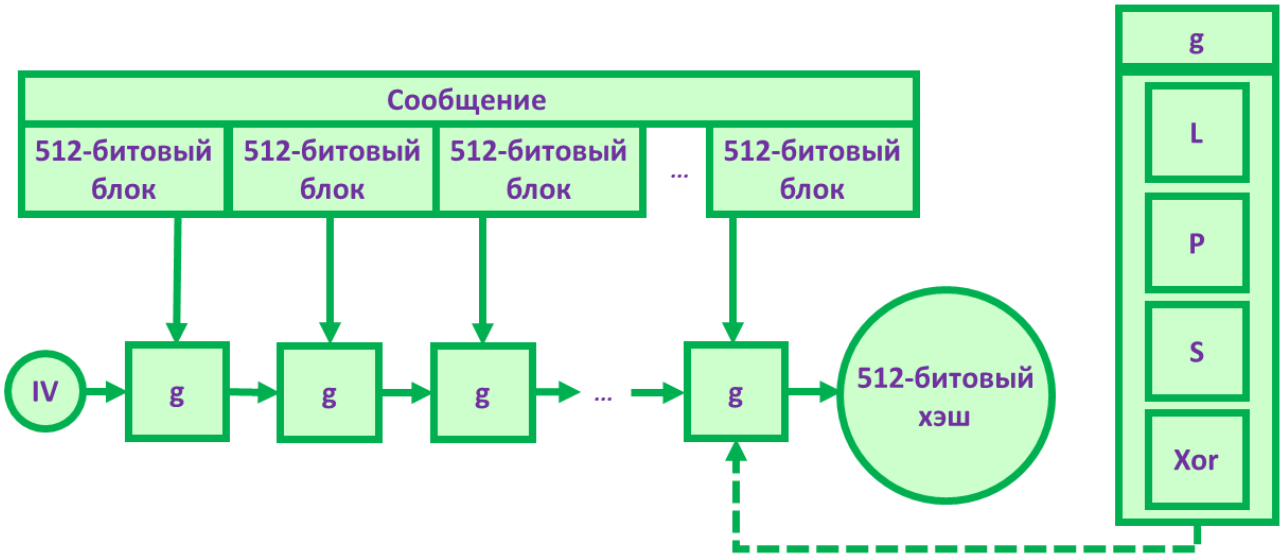


*Рис. 3.* Схема работы хэш-функции «Стрибог»

Данная хэш-функция принимает на вход инициализирующий вектор IV, а также сообщение, для которого нужно сгенерировать хэш. В качестве инициализирующего вектора в верифицируемом нами случае 512-битовой хэш-функции «Стрибог» используется 512-битовый нулевой вектор. Сообщение разделяется на части, каждая длиной 512 бит, а если

длина сообщения в битах не кратна 512, то последняя часть дополняется до 512-битового вектора нулями и одной единицей перед последней частью. Таким образом получается набор 512-битовых векторов. Хэш-функция «Стрибог» выполняет итерации над данным набором векторов, применяя на каждой итерации функцию сжатия g к текущей 512-битовой части сообщения и к результату предыдущей итерации. Функция сжатия основана на преобразовании XLPS, однако реализация данного преобразования в ядре Linux значительно отличается от описания из ГОСТ Р 34.11-2012 тем, что вместо описанных в стандарте преобразований (линейное преобразование L, подстановка P и нелинейное преобразование S) в рассматриваемой реализации используется преобразование со специальной матрицей. На первой итерации в качестве результата предыдущей итерации используется инициализирующий вектор. Результатом последней итерации является хэш всего сообщения.

## 3. Дедуктивная верификация реализации хэш-функции «Стрибог»

В данной главе мы презентуем такие результаты нашей работы, как формализацию ГОСТ Р 34.11-2012 в системе Rocq, задание для базовых функций из реализации хэш-функции «Стрибог» из ядра Linux предусловий и постусловий, доказательство корректности осуществляющей xor над 512-битовыми векторами функции "streebog_xor", расширение теории предметной области леммами об используемых в рассматриваемой реализации структурах данных и задание функциональной спецификации оптимизированной реализации преобразования XLPS из ядра Linux.

### 3.1. Формализация ГОСТ Р 34.11-2012 в системе интерактивного доказательства теорем Rocq

Так как ГОСТ Р 34.11-2012 задан в алгоритмическом стиле, то формализация стандарта в системе Rocq представляет собой кодирование алгоритмов из стандарта на языке Gallina. В результате получился файл "functional_spec.v", общедоступный в репозитории нашего проекта [55]. Отметим, что при задании алгоритмов из ГОСТ Р 34.11-2012 мы пытались придерживаться обозначений из стандарта, чтобы наша формализация была понятнее и другим исследователям, которые тоже могут использовать нашу формализацию в качестве спецификации. Также для упрощения понимания нашей формализации

другими исследователями мы по-возможности снабдили код комментариями.

ГОСТ Р 34.11-2012 основан на использовании таких типов данных, как 512-битовые вектора и разбиение 512-битовых векторов побайтово (на 8-битовые части). Для задания такого типа данных удобно иметь тип битового вектора параметрической длины, который можно в нашем случае параметризовать длиной 512, и длиной 8. Такой подход более гибок, чем подход из известной работы по формальной верификации SHA-256, где для представления 256-битовых векторов не использовались последовательности бит, а использовался список байтов. Мы тоже используем списки байтов для хранения разбиения 512-битовых векторов побайтово, однако мы кроме того используем тип данных для представления 512-битовых векторов в виде последовательности бит.

В качестве типа данных для представления последовательности бит параметрической длины мы использовали тип данных int из библиотеки compcert.lib.Integers. Данный тип представляет собой последовательность бит параметрической длины. Для данного типа уже предопределены параметризации по длине 8 (тип Byte), по длине 32 (тип Int) и по длине 64 (тип Int64). Тип Byte пригодился нам для хранения в виде списка байтов разбиение 512-битовых векторов на 8-битовые части. Для представления 512-битовых векторов мы параметризовали тип int длиной 512 и задали для такого типа в соответствии со стандартом названия block512 и Vec512.

Так как используемые в ГОСТ Р 34.11-2012 преобразования требуют конвертации из 512-битовых блоков в список байт и обратно, то мы задали функции для таких преобразований, которые названы block512_to_bytes и bytes_to_block512 соответственно. Вместе с другими вспомогательными функциями для данных типов мы сделали важные шаги на пути к задании теории для 512-битовых векторов и их разбиений на части, которая может быть использована для формальной верификации таких функций, где применяются 512-битовые вектора.

Использование библиотеки compcert.lib.Integers позволило нам использовать встроенные функции над машинными представлениями целых чисел. Отметим, что нам не пришлось задавить играющую важную роль в преобразовании XLPS функцию xor для 512-битовых векторов, так как после параметризации типа int мы автоматически получили функцию Vec512.xor для операции исключающего или над 512-битовыми векторами.

С помощью заданных нами типов данных мы определили алгоритмы из ГОСТ Р 34.11-2012 в виде функций на языке Gallina. Для базовых алгоритмов, выполняющих преобра-

зование LPS, мы задали функции с именами l, p и s соответственно. Для всего преобразования LPSX мы задали функцию LPSX как композицию функций l, p, s и Vec512.xor. Использование функция LPSX позволило нам определить функцию сжатия g под названием g_N. Итеративное исполнение хэш-функции «Стрибог», где применяется определенная нами функция сжатия g_N, реализована в функции stage2.

Отдельно отметим вопрос доверия к заданной нами формализации ГОСТ Р 34.11-2012. К сожалению, пока что нет индустриально применимых методов формальной верификации соответствия текста на естественном языке и кода в системе интерактивного доказательства. В стандарте предлагаются тесты, поэтому мы провели тестирование нашей формализации, реализовав применение всех тестов из стандарта в файле "functional_spec_test.v", доступном в репозитории нашего проекта [55]. Отметим, что в стандарте предлагаются тесты не только для запуска всей реализации хэш-функции «Стрибог», но и по-отдельности для базовых преобразований. Все предложенные в ГОСТ Р 34.11-2012 тесты были успешно пройдены нашей формализацией стандарта на языке Gallina. Заданная нами функциональная спецификация предназначена для использования в предусловиях и постусловиях реализаций хэш-функции «Стрибог», чтобы доказать эквивалентность таких реализаций нашей функциональной спецификации.

## 3.2. Задание предусловий и постусловий базовых функций из реализации хэш-функции «Стрибог» из ядра Linux

В первую очередь важно верифицировать базовые функции реализации хэш-функции «Стрибог» из ядра Linux, поэтому мы задали спецификации для следующих функций:

- "streebog_xor" – функция, осуществляющая операцию xor над 512-битовыми векторами, представленными в виде 8-ми 64-битовых частей. Предусловие и постусловие данной функции определены в конструкции streebog_xor_spec в файле "spec_streebog_xor.v", доступном в репозитории нашего проекта [55]. В постусловии данной функции мы задали ее эквивалентность функции Vec512.xor из функциональной спецификации.
- "streebog_init" – функция, осуществляющая инициализацию начального вектора. Предусловие и постусловие данной функции определены в конструкции streebog_init_spec в файле "spec_streebog_init.v", доступном в репозитории нашего проекта [55]. В постусловии данной функции мы задали, что в результате должен

получиться инициализирующий вектор. Также в целях дальнейшей верификации данной функции мы задали в конструкции memset_spec предусловие и постусловие применяемой в данной функции стандартной функции memset. Отметим, что задание спецификаций для функций стандартной библиотеки языка C является важной задачей.

- "streebog_add512" – функция, осуществляющая прибавление 512-битового вектора. Предусловие и постусловие данной функции определены в конструкции streebog_add512_spec в файле "spec_streebog_add512.v", доступном в репозитории нашего проекта [55]. В постусловии данной функции мы задали, что результате должна получиться сумма векторов.
- "streebog_xlps" – функция, осуществляющая преобразование XLPS. Предусловие и постусловие данной функции определены в конструкции streebog_xlps_spec в файле "spec_streebog_xlps.v", доступном в репозитории нашего проекта [55]. В постусловии данной функции мы задали ее эквивалентность функции lpsx из функциональной спецификации.

После задания предусловий и постусловий стало возможным доказательство корректности функций.

## 3.3. Доказательство корректности функции "streebog_xor", осуществляющей xor над 512-битовыми векторами

Нами была дедуктивно верифицирована функция "streebog_xor" осуществляющей xor над 512-битовыми векторами. Мы задали корректность данной функции относительно ее предусловия и постусловия в лемме body_streebog_xor и доказали данную лемму. Лемма body_streebog_xor и ее доказательство приведены в файле "spec_streebog_xor.v", доступном в репозитории нашего проекта [55].

В доказательстве леммы body_streebog_xor 24 раза применяется тактика forward, что позволяет осуществить прямое прослеживание по всем 24 инструкциям данной функции, внося факты о структуре программы в предусловие. Отметим, что данный процесс доказательство можно рассматривать, как доказательство эквивалентности функции "streebog_xor" и функции Vec512.xor из функциональной спецификации. Но структура функций "streebog_xor" и функции Vec512.xor существенно различается, что приводит к сложности такого доказательства. Функция Vec512.xor осуществляет xor над представле-

ниями векторов в виде последовательности бит, тогда как функция "streebog_xor" для эффективности исполнения осуществляет xor над представлениями векторов в виде 8-ми 64-битовых частей. Эти различия отражаются во внесенных в предусловие в ходе прямого прослеживания фактов о программе. В конце доказательства для вывода того, что из накопленного предусловия следует постусловие, применяется тактика entailer!, но перед этим для упрощения вывода нужно сформулировать и применить леммы о связи операции xor над 64-битовыми векторами и операции xor над последовательностями битов. Для этого потребовалось расширять теорию предметной области леммами об используемых в программе структурах данных (леммы xor_repr_comm, xor_unsigned_comm, Z_to_chunks_xor) и применять эти леммы.

## 3.4. Задание лемм о свойствах структур данных, используемых в реализации хэш-функции «Стрибог» из ядра Linux

Для доказательства корректности функция "streebog_xor" мы расширили теорию предметной области леммами о свойствах структур данных, применяемых в реализации хэш-функции «Стрибог» из ядра Linux. Данные леммы доступны в нашем репозитории [55] в файле "spec_streebog_xor.v", но так как свойства 64-битовых векторов могут пригодиться при верификации и других реализаций хэш-функции «Стрибог», то мы планируем перенести эти леммы в файл с функциональной спецификацией "functional_spec.v", чтобы сформировать теорию предметной области в одном файле.

Леммы xor_repr_comm, xor_unsigned_comm и Z_to_chunks_xor применяются непосредственно в доказательстве корректности функции "streebog_xor". Данные леммы описывают связь между операцией xor над последовательностями битов и операцией xor над битовыми векторами определенной длины. В качестве операции xor над последовательностями битов имеется в виду операция Z.lxor, определенная над типом Z, представляющим целое число в двоичном представлении с помощью беззнаковой или со знаком последовательности битов любой длины. Отметим, что для доказательства данных лемм были введены другие леммы, которые с одной стороны могут рассматриваться как вспомогательные, но с другой стороны тоже описывают важные свойства о связях двух видов операции xor.

Лемма xor_repr_comm описывает, что применение операции xor над 64-битовыми векторами к результатам приведения к 64-битному вектору двух последовательностей бит эк-

вивалентно приведению к типу 64-битового вектора результата применения Z.lxor к этим последовательностями бит. Приведем лемму xor_repr_comm и ее доказательство:

```
Lemma xor_repr_comm : forall x y,
  Int64.xor (Int64.repr x) (Int64.repr y) = Int64.repr (Z.lxor x y).
Proof.
  intros x y.
  specialize (Int64.same_bits_eq (Int64.xor (Int64.repr x) (Int64.repr y))
             (Int64.repr (Z.lxor x y))) as H; lapply H; clear H.
  - intros T; exact T.
  - intros i R. rewrite Int64.bits_xor.
    -- rewrite 3!Int64.testbit_repr.
       --- rewrite <- Z.lxor_spec.
           reflexivity.
       --- exact R.
       --- exact R.
       --- exact R.
    -- exact R.
Qed.
```

Доказательство данной леммы основано на применении лемм Int64.same_bits_eq и Int64.testbit_repr из библиотеки compcert.lib.Integers. Лемма Int64.same_bits_eq для 64-битовых векторов утверждает, что если все биты данных векторов равны, то и сами вектора равны. Лемма Int64.testbit_repr утверждает, что если индекс бита находится в диапазоне от 0 до длины машинного слова из 64 бит, то бит с таким индексом от результата приведения к 64-битовому вектору равен биту с тем же индексом от последовательности бит. Отметим, что определениях этих двух лемм основаны на операции Z.testbit, которая принимает на вход последовательность бит (имеет тип Z) и индекс бита и возвращает значение бита как булево значение.

Лемма xor_unsigned_comm описывает, что результат применения Z.lxor к беззнаковым приведениям к 512-битовым векторам двух последовательностей бит равен результату беззнакового приведения к 512-битовому вектору результата применения Z.lxor к этим последовательностям бит. Приведем лемму xor_unsigned_comm и ее доказательство:

```
Lemma xor_unsigned_comm : forall x y,
```

```
  Z.lxor (Vec512.unsigned x) (Vec512.unsigned y) =
  Vec512.unsigned (Vec512.xor x y).
Proof.
  intros [x Hx] [y Hy]. simpl.
  rewrite Vec512.Z_mod_modulus_eq.
  unfold Vec512.modulus in *.
  rewrite two_power_nat_equiv in *.
  rewrite Zlxor_mod_pow2 by easy.
  now rewrite 2!Zmod_small by lia.
Qed.
```

Доказательство данной леммы основано на применении заданной нами вспомогательной леммы Zlxor_mod_pow2. Данная лемма утверждает, что результат применения Z.lxor к двум аргументам, взятый по модулю степени двойки, равен результату применения Z.lxor к тем же аргументам, но взятым по модулю той же самой степени двойки. Приведем вспомогательную лемму Zlxor_mod_pow2 и ее доказательство:

```
Lemma Zlxor_mod_pow2 : forall x y n, 0 <= n ->
  (Z.lxor x y) mod 2 ^ n = Z.lxor (x mod 2 ^ n) (y mod 2 ^ n).
Proof.
  intros x y n Hn. apply Z.bits_inj'. intros i Hi.
  rewrite Z.lxor_spec. destruct (Z_lt_le_dec i n).
  - rewrite 3!Z.mod_pow2_bits_low by assumption.
    apply Z.lxor_spec.
  - now rewrite 3!Z.mod_pow2_bits_high.
Qed.
```

Доказательство данной вспомогательной леммы основано на рассмотрении двух случаев: случай для младших по отношению к степени двойки бит и случай для старших по отношению к степени двойки бит. Отметим, что в данной вспомогательной лемме формулируется относительно общее утверждение, которое может иметь обширные сферы применения при доказательстве теорем.

Лемма Z_to_chunks_xor описывает, что результат попарного применения Z.lxor к результатам разбиения последовательностей бит на одинаковое количество частей одинаковой длины равен результату попарного разбиения на то же количество частей той же

длины результата применения Z.lxor к этим последовательностям бит. Приведем лемму Z_to_chunks_xor и ее доказательство:

```
Lemma Z_to_chunks_xor : forall n m x y,
  map (uncurry Z.lxor) (combine (Z_to_chunks m n x) (Z_to_chunks m n y)) =
  Z_to_chunks m n (Z.lxor x y).
Proof.
   induction n; intros m x y; simpl.
  - reflexivity.
  - now rewrite <- xor_LSB_comm, Z.shiftr_lxor, IHn.
Qed.
```

Доказательство данной леммы основано на использовании заданной нами вспомогательной леммы xor_LSB_com. Данная вспомогательная лемма утверждает, что результат применения Z.lxor к j наименьшим значащим битам обоих аргументов равен j наименьшим значащим битам от результата применения Z.lxor к этим же аргументам. Приведем вспомогательную лемму xor_LSB_com и ее доказательство:

```
Lemma xor_LSB_comm : forall (j : nat) (x y : Z),
  Z.lxor (LSB j x) (LSB j y) = LSB j (Z.lxor x y).
Proof.
  intros j x y.
  unfold LSB.
  rewrite 3!Zbits.Z_mod_two_p_eq.
  rewrite <- Z.bits_inj_iff.
  unfold Z.eqf.
  intros n.
  rewrite Z.lxor_spec.
  specialize (Z.lt_ge_cases n (Z.of_nat j)) as [Hnltj | Hngej].
  - rewrite two_power_nat_equiv.
    rewrite 3!Z.mod_pow2_bits_low.
    --- rewrite <- Z.lxor_spec.
        reflexivity.
    --- exact Hnltj.
    --- exact Hnltj.
```

```
    --- exact Hnltj.
  - rewrite 3!testbit_ge_k.
    -- reflexivity.
    -- apply Z.le_ge. exact Hngej.
    -- apply Z.le_ge. exact Hngej.
    -- apply Z.le_ge. exact Hngej.
Qed.
```

Доказательство данной вспомогательной леммы основано на рассмотрении случаев для бит младше j-того и для бит старше j-того. Отметим, что в данной вспомогательной лемме также формулируется относительно общее утверждение, которое может упростить доказательство многих теорем.

Мы планируем пополнять теорию предметной области новыми леммами, которые будут упрощать доказательство корректности функций и описывать новые свойства типов данных, используемых при реализации хэш-функции «Стрибог». В качестве первоочередных задачи мы планируем доказать лемму о том, что описанная в ГОСТ Р 34.11-2012 композиция преобразований LPSX эквивалента применяемому в реализации хэш-функции «Стрибог» преобразованию со специальной матрицей.

## 3.5. Функциональная спецификация оптимизированной реализации преобразования XLPS из ядра Linux

Для доказательства корректности эффективной реализации преобразования XLPS из ядра Linux мы предложили использовать подход, состоящий из следующих шагов:

1. Задание функциональной спецификации оптимизированной реализации преобразования XLPS из ядра Linux.
2. Задание леммы об эквивалентности двух функциональных спецификаций XLPS: соответствующей стандарту и соответствующей эффективной реализации.
3. Применение такой леммы об эквивалентности для упрощения доказательства корректности реализации преобразования XLPS из ядра Linux.

Таким образом, в данном подходе для упрощения доказательства корректности оптимизированной программы вводится промежуточный этап в виде задания функциональной спецификации оптимизированной реализации и доказательства ее эквивалентности стандартной реализации.

Мы задали функциональную спецификацию оптимизированной реализации в файле "functional_spec.v" из нашего репозитория [55], дополнив тем самым теорию предметной области. Оптимизированный алгоритм реализует заданная нами на языке Gallina функция LPS_opt. Также мы задали интересующую нас лемму alg_equiv об эквивалентности LPS_opt и композиции l, p и s, но данная лемма пока что не доказана. Мы свели доказательство леммы alg_equiv к доказательству набора лемм, часть из которых еще предстоит доказать, и, таким образом, мы получили схему доказательства этой леммы. Лемма alg_equiv и схема ее доказательства приведены в Приложении А. Данная схема доказательства основана на материале статьи о том, как из стандартной реализации можно получить рассматриваемую оптимизированную реализацию [3].

## 4. Заключение

В данной статье мы презентуем следующие результаты:

- Формализацию ГОСТ Р 34.11-2012 в системе интерактивного доказательства Rocq. Данный результат может быть использован в качестве функциональной спецификации при верификации и других реализаций хэш-функции «Стрибог». Кроме того, данный результат демонстрирует актуальность формализации стандартов.
- Задание предусловий и постусловий базовых функций из реализации хэш-функции «Стрибог» из ядра Linux. Отметим, что была также заданы предусловие и постусловие стандартной функции memset. Данный результат позволяет верифицировать те функции, где применяются данные базовые функции. Кроме того, данный результат полезен не только для формальной верификации, но и для тестирования данных функций, позволяя проверять результат исполнения тестов с помощью проверки истинности постусловий по итогам тестов.
- Доказательство корректности функции xor над 512-битовыми векторами (функция "streebog_xor"), осуществляемая над данными векторами поблочно (8-блоков по 64 бита в каждом блоке). Данное доказательство позволяет доверять реализации данной функции. Поэтому, данную функцию можно использовать во всех программных системах, где требуется эффективная операция xor над 512-битовыми векторами.
- Задание лемм о свойствах структур данных, используемых в реализации хэш-функции «Стрибог» из ядра Linux. Данный результат позволяет упростить верификацию входящих в рассматриваемую реализацию функций с помощью переис-

пользования лемм при доказательстве корректности таких функций. Кроме того, данный результат может быть применим и при верификации других криптографических функций, так как в их реализациях могут встречаться подобные структуры данных.

- Задание функциональной спецификации оптимизированной реализации преобразования XLPS из ядра Linux. Данная функциональная спецификация ориентирована на упрощение доказательства корректности реализации XLPS из ядра Linux с помощью задания леммы об эквивалентности двух функциональных спецификаций XLPS: соответствующей стандарту и соответствующей оптимизированной реализации. Такое задание функциональных спецификаций оптимизированных программ, ориентированное на доказательство леммы об эквивалентности стандартной и оптимизированной версии программы, может быть примером подхода к дедуктивной верификации оптимизированных программ.

Отметим, что данные результаты могут служить прототипом комплексного подхода к дедуктивной верификации криптографических функций.

Мы рассматриваем планы по формальной верификации криптографических свойств реализации хэш-функции «Стрибог» из ядра Linux. Мы планируем завершить формальную верификацию реализации хэш-функции «Стрибог» из ядра Linux и сформировать по итогам такой верификации комплексный подход к верификации реализаций криптографических функций.

## Список литературы

1. Васенин В.А., Кривчиков М.А. Формальные модели программ и языков программирования. Часть 1. Библиографический обзор 1930–1989 гг. // Программная инженерия. 2015. № 5. С. 10–19.
2. Васенин В.А., Кривчиков М.А. Формальные модели программ и языков программирования. Часть 2. Современное состояние исследований // Программная инженерия. 2015. № 6. С. 24–33.
3. Гафуров И.Р. Высокоскоростная программная реализация алгоритма хэширования «Стрибог» // Ученые записки УлГУ. Серия "Математика и информационные технологии". 2023. № 2. С. 19–27.
4. ГОСТ Р 34.11-2012. Информационная технология. Криптографическая защита информации. Функция хэширования.
5. Девянин П.Н., Кулямин В.В., Петренко А.К., Хорошилов А.В., Щепетков И.В. Интеграция

мандатного и ролевого управления доступом и мандатного контроля целостности в верифицированной иерархической модели безопасности операционной системы // Труды Института системного программирования РАН. 2020. Т. 32. № 1. С. 7–26.

6. Девянин П.Н. , Леонова М.А. Приёмы описания модели управления доступом ОССН Astra Linux Special Edition на формализованном языке метода Event-B для обеспечения её верификации инструментами Rodin и ProB // Прикладная дискретная математика. 2021. № 52. С. 83–96.
7. Девянин П.Н., Тележников В.Ю., Хорошилов А.В. Формирование методологии разработки безопасного системного программного обеспечения на примере операционных систем // Труды Института системного программирования РАН. 2021. Т. 33. № 5. С. 25–40.
8. Камкин А.С. Введение в формальные методы верификации программ. М.: ДМК Пресс, 2024. – 304 с.
9. Кокорин А.О., Тиевский С.Д., Девянин П.Н. Приемы дедуктивной верификации программного кода с использованием AstraVer Toolset // Прикладная дискретная математика. Приложение. 2022. № 15. С. 80–90.
10. Кондратьев Д.А., Старолетов С.М., Шошмина И.В., Красненкова А.В., Зиборов К.В., Шилов Н.В., Гаранина Н.О., Черганов Т.Ю. Соревнования по формальной верификации VeHa-2024: накопленный в течение двух лет опыт и перспективы // Труды Института системного программирования РАН. 2025. Т. 37. № 1. С. 159–184.
11. Лейно К.Р.М. Доказательство корректности программ. М.: ДМК Пресс, 2024. – 522 с.
12. Миронов А.М. Методы верификации программ. М.: ДМК Пресс. 2023 – 332 с.
13. Непомнящий В.А., Рякин О.М. Прикладные методы верификации программ. М.: Радио и связь, 1988. – 256 с.
14. Операционная система Astra Linux. Режим доступа: `https://astralinux.ru` , свободный (дата обращения: 25.07.2025)
15. Седов Г.К. Стойкость ГОСТ Р 34.11-2012 к атаке поиска прообраза и к атаке поиска коллизий // Математические вопросы криптографии. 2015. Т. 6. № 2. С. 79–98.
16. Старолетов С.М., Кондратьев Д.А., Гаранина Н.О., Шошмина И.В. Соревнования по формальной верификации VeHa-2023: опыт проведения // Труды Института системного программирования РАН. 2024. Т. 36. № 2. С. 141–168.
17. Шилов Н.В. Основы синтаксиса, семантики, трансляции и верификации программ: учебное пособие. Новосибирск: Издательство Новосибирского государственного университета, 2011. – 292 с.
18. AlTawy R., Youssef A.M. Integral distinguishers for reduced-round Stribog // Information Processing Letters. 2014. Volume 114. Issue 8. pp. 426–431.
19. Appel A.W. Verification of a Cryptographic Primitive: SHA-256 // ACM Transactions on Programming Languages and Systems (TOPLAS). Volume 37. Issue 2. Article No.: 7. pp. 1–31.
20. Appel A.W. Verified Software Toolchain // Lecture Notes in Computer Science. 2011. Volume 6602. pp. 1–17.

21. Appel A.W., Beringer L., Cao Q., Dodds J. Verifiable C: Applying the Verified Software Toolchain to C programs. 2023. URL: `https://github.com/PrincetonUniversity/VST/raw/master/doc/VC.pdf` (Accessed 25 Jul 2025)
22. Apt K.R., Olderog E.-R. Assessing the Success and Impact of Hoare's Logic // Theories of Programming: The Life and Works of Tony Hoare. New York: ACM, 2021. pp. 41–76.
23. Apt K.R., Olderog E.-R. Fifty years of Hoare's logic // Formal Aspects of Computing. 2019. Volume 31. Issue 6. pp. 751–807.
24. Biryukov A., Perrin L., Udovenko A. Reverse-Engineering the S-Box of Streebog, Kuznyechik and STRIBOBr1 // Lecture Notes in Computer Science. 2016. Volume 9665. pp. 372–402.
25. Brain M., Polgreen E. A Pyramid Of (Formal) Software Verification // Lecture Notes in Computer Science. 2025. Volume 14934. pp. 393–419.
26. Cao Q., Beringer L., Gruetter S., Dodds J., Appel A.W. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs // Journal of Automated Reasoning. 2018. Volume 61. Issue 1. pp. 367–422.
27. Degtyarev A., Chikunov V. Streebog Hash Function. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/crypto/streebog_generic.c` (Accessed 25 Jul 2025).
28. Dijkstra E.W., Schölten C.S. The strongest postcondition // Predicate Calculus and Program Semantics. New York: Springer, 1990. pp 209–215.
29. Efremov D., Mandrykin M., Khoroshilov A. Deductive verification of unmodified Linux kernel library functions // Lecture Notes in Computer Science. 2018. Volume 11245. pp. 216–234.
30. Filliâtre J.C. Deductive software verification // International Journal on Software Tools for Technology Transfer. 2011. Volume 13. Issue 5. Article ID: 397.
31. Floyd R.W. Assigning meanings to programs // Proc. Symposia in Applied Mathematics. Providence, 1967. Volume 19. pp. 19–32.
32. Guo J., Jean J., Leurent G., Peyrin T., Wang L. The Usage of Counter Revisited: Second-Preimage Attack on New Russian Standardized Hash Function // Lecture Notes in Computer Science. 2014. Volume 8781. pp. 195–211.
33. Hähnle R., Huisman M. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools // Lecture Notes in Computer Science. 2019. Volume 10000. pp. 345–373.
34. Harrison J. Theorem Proving for Verification (Invited Tutorial) // Lecture Notes in Computer Science. 2008. Volume 5123. pp. 11–18.
35. Hoare C.A.R. An axiomatic basis for computer programming // Communications of the ACM. 1969. Volume 12. Issue 10. pp. 576–580.
36. Ishtiaq S.S., O'Hearn P.W. BI as an assertion language for mutable data structures // Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '01). London, United Kingdom, January 17–19, 2001. New York: Association for Computing Machinery, 2001. pp. 14–26.

37. Kazymyrov O., Kazymyrova V. Algebraic aspects of the Russian hash standard GOST R 34.11-2012 // Proceedings of the CTCrypt'13. Ekaterinburg, Russia, June 23–24, 2013. IACR ePrint, URL: `http://eprint.iacr.org/2013/556` (Accessed 25 Jul 2025)
38. Kondratyev D.A., Maryasov I.V., Nepomniaschy V.A. The Automation of C Program Verification by the Symbolic Method of Loop Invariant Elimination // Automatic Control and Computer Sciences. 2019. Volume 53. Issue 7. pp. 653–662.
39. Kondratyev D., Maryasov I., Nepomniaschy V. Towards Automatic Deductive Verification of C Programs over Linear Arrays // Lecture Notes in Computer Science. 2019. Volume 11964. pp. 232–242.
40. Kondratyev D.A., Nepomniaschy V.A. Automation of C Program Deductive Verification without Using Loop Invariants // Programming and Computer Software. 2022. Volume 48. Issue 5. pp. 331–346.
41. Kondratyev D.A., Promsky A.V. Developing a self-applicable verification system. Theory and practice // Automatic Control and Computer Sciences. 2015. Volume 49. Issue 7. pp. 445–452.
42. Ma B., Li B., Hao R., Li X. Improved Cryptanalysis on Reduced-Round GOST and Whirlpool Hash Function // Lecture Notes in Computer Science. 2014. Volume 8479. pp. 289–307.
43. Mandrykin M.U., Khoroshilov A.V. High-level memory model with low-level pointer cast support for Jessie intermediate language // Programming and Computer Software. 2015. Volume 41. Issue 4. pp. 197–207.
44. Mandrykin M.U., Khoroshilov A.V. Region analysis for deductive verification of C programs // Programming and Computer Software. 2016. Volume 42. Issue 5. pp. 257–278.
45. Mandrykin M.U., Khoroshilov A.V. Towards deductive verification of C programs with shared data // Programming and Computer Software. 2016. Volume 42. Issue 5. pp. 324–332.
46. Maryasov I.V., Nepomniaschy V.A., Promsky A.V., Kondratyev D.A. Automatic C Program Verification Based on Mixed Axiomatic Semantics // Automatic Control and Computer Sciences. 2014. Volume 48. Issue 7. pp. 407–414.
47. Nepomniaschy V.A., Anureev I.S., Mikhailov I.N., Promsky A.V. Towards verification of C programs. C-light language and its formal semantics // Programming and Computer Software. 2002. Volume 28. Issue 6. pp. 314–323.
48. Nepomniaschy V.A., Anureev I.S., Promskii A.V. Towards Verification of C Programs: Axiomatic Semantics of the C-kernel Languages // Programming and Computer Software. 2003. Volume 29. Issue 6. pp. 338–350.
49. O'Hearn P. Separation logic // Communications of the ACM. 2019. Volume 62. Issue 2. pp. 86–95.
50. O'Hearn P. Separation Logic Tutorial // Lecture Notes in Computer Science. 2008. Volume 5366. pp. 15–21.
51. Paulin-Mohring C. Introduction to the Coq Proof-Assistant for Practical Software Verification // Lecture Notes in Computer Science. 2012. Volume 7682. pp. 45–95.
52. Reynolds J.C. An Overview of Separation Logic // Lecture Notes in Computer Science. 2008. Volume 4171. pp. 460–469.

53. Reynolds J.C. Separation logic: a logic for shared mutable data structures // Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. Copenhagen, Denmark, July 22–25, 2002. IEEE, 2002. pp. 55–74.
54. Saarinen M.-J. O. StriBob: authenticated encryption from GOST R 34.11-2012 LPS permutation // Mathematical Aspects of Cryptography. 2015. Volume 6. Issue 2. pp. 67–78.
55. Stribog Hash Formal Verification. URL: `https://github.com/MathAndLive/StribogHashFormalVerification` (Accessed 25 Jul 2025)
56. Volkov G., Mandrykin M. Efremov D. Lemma functions for Frama-C: C programs as proofs // Proceedings of the 2018 Ivannikov Ispras Open Conference (ISPRAS). Moscow, Russia, November 22–23, 2018. IEEE, 2018. pp. 31–38.
57. Wang Z., Yu H., Wang X. Cryptanalysis of GOST R hash function // Information Processing Letters. 2014. Volume 114. Issue 12. pp. 655–662.

## A. Лемма об эквивалентности двух функциональных спецификаций XLPS: соответствующей стандарту и соответствующей оптимизированной реализации

Лемма alg_equiv задана следующим образом:

```
Lemma alg_equiv : forall (b : block512),
  LPS_opt b = l (p (s b)).
Proof.
  intros b.
  unfold LPS_opt.
  unfold tableLPS_i_j.
  assert ((fun k : Z => fold_right Int64.xor (Int64.repr 0)
  (map (fun i : Z => fold_right Int64.xor (Int64.repr 0)
  (map (fun k0 : Z => nthi_int64 A (63 - 8 * i - k0))
  (bit (Byte.unsigned
  (pi_byte (nthi_bytes (tau_bytes (block512_to_bytes b)) (8 * k + i)))))))
  [0; 1; 2; 3; 4; 5; 6; 7]))
          =
          the_thing b) as H by reflexivity; rewrite H; clear H.
  assert (map (the_thing b) [0; 1; 2; 3; 4; 5; 6; 7]
```

```
        =
        map (fun k : Z => fold_right Int64.xor (Int64.repr 0)
        (map (fun i : Z => fold_right Int64.xor (Int64.repr 0)
        (map (fun k0 : Z => nthi_int64 A (63 - 8 * i - k0))
        (bit (Byte.unsigned
        (nthi_bytes (block512_to_bytes (p (s b))) (8 * k + i))))))
        [0; 1; 2; 3; 4; 5; 6; 7]))
        [0; 1; 2; 3; 4; 5; 6; 7]) as H.
{
 unfold the_thing.
 pose proof map_ext_in as M.
 (* обязательное действие, без него specialize не сработает *)
 specialize (M
             Z
             int64
             (fun k : Z => fold_right Int64.xor (Int64.repr 0)
             (map (fun i : Z => fold_right Int64.xor (Int64.repr 0)
             (map (fun k0 : Z => nthi_int64 A (63 - 8 * i - k0))
             (bit (Byte.unsigned (pi_byte
             (nthi_bytes (tau_bytes (block512_to_bytes b)) (8 * k + i)))))))
             [0; 1; 2; 3; 4; 5; 6; 7]))
             (fun k : Z => fold_right Int64.xor (Int64.repr 0)
             (map (fun i : Z => fold_right Int64.xor (Int64.repr 0)
             (map (fun k0 : Z => nthi_int64 A (63 - 8 * i - k0))
             (bit (Byte.unsigned
             (nthi_bytes (block512_to_bytes (p (s b))) (8 * k + i))))))
             [0; 1; 2; 3; 4; 5; 6; 7]))
             [0; 1; 2; 3; 4; 5; 6; 7]).
 apply M.
 clear M.
 intros i R1.
 apply remove_fold_right_xor.
```

```
   pose proof map_ext_in as M. (* то же самое *)
   specialize (M
               Z
               int64
               (fun i0 : Z => fold_right Int64.xor (Int64.repr 0)
               (map (fun k0 : Z => nthi_int64 A (63 - 8 * i0 - k0))
               (bit (Byte.unsigned (pi_byte (nthi_bytes
               (tau_bytes (block512_to_bytes b)) (8 * i + i0)))))))
               (fun i0 : Z => fold_right Int64.xor (Int64.repr 0)
               (map (fun k0 : Z => nthi_int64 A (63 - 8 * i0 - k0))
               (bit (Byte.unsigned
               (nthi_bytes (block512_to_bytes (p (s b))) (8 * i + i0))))))
               [0; 1; 2; 3; 4; 5; 6; 7] ).
   apply M.
   clear M.
   intros l R2.
   apply remove_fold_right_xor.
   apply remove_map.
   apply f_equal.
   apply f_equal.
   apply ith_of_pi_tau.
   unfold In in R1.
   unfold In in R2.
   assert ((Datatypes.length tau) = 64%nat) as obvious by
   reflexivity; rewrite obvious; clear obvious.
   lia.
  }
  rewrite H; clear H.
  unfold l.
  apply f_equal.
  apply l_equiv.
Qed.
```

Схема доказательства данной леммы задана между конструкциями `Proof.` и `Qed`.