**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**
**«МЭИ»**

**УДК:**

**Институт** Автоматики и вычислительной техники
**Кафедра** Вычислительных машин, систем и сетей
**Направление** 09.04.01 Информатика и вычислительная техника

# МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

**Программа:** Вычислительные машины, комплексы, системы и сети

**Тема:** Реализация статистического кодирования числовых данных для усиления их стойкости к атакам перебором

**Студент** А-07м-11 Зубов И.А.
*группа* *подпись* *фамилия, и., о.,*

**Научный руководитель** ст. преп. Филатов А.В.
*должность* *звание* *подпись* *фамилия, и., о.,*

**Консультант** _____
*должность* *звание* *подпись* *фамилия, и., о.,*

**Консультант** _____
*должность* *звание* *подпись* *фамилия, и., о.,*

**Магистерская диссертация допущена к защите**

**Зав.кафедрой** доцент, к.т.н. Вишняков С.В.
*звание* *подпись* *фамилия, и., о.,*

**Дата**_____

**МОСКВА** **2017 г.**

# АННОТАЦИЯ

В работе рассмотрена разработка и реализация на языке Си алгоритмов статистического кодирования числовых данных, применяемых вместе с шифрованием для усиления стойкости шифротекстов к атакам перебором (другое название – медовое шифрование). Кодирование применено к данным типов uint8_t, int8_t, uint16_t, int16_t, uint32_t, int32_t, uint64_t, int64_t с равномерным и произвольным законами распределения; float и double с равномерным законом распределения. Объём – 135 страниц; содержит 8 рисунков и 3 приложения.

## СОДЕРЖАНИЕ

Обозначения и сокращения ……...5

Введение ……..6

1. Анализ существующих работ ……..10

2. Постановка задачи ……14

3. Обработка равномерно распределённых целых чисел ……17

3.1. Целые числа в языке Си ……..17

3.2. Алгоритмы для типа *uint8_t* ……...19

3.3. Программная реализация для типа *uint8_t* ……24

3.4. Реализации для некоторых других целочисленных типов ……..27

3.5. Реализации для 64-битных целочисленных типов ……...31

3.6. Замечание, касающееся защищённости ……33

4. Обработка произвольно распределённых целых чисел ……36

4.1. Общий подход ……..36

4.2. Вопросы практической реализации ……...39

5. Обработка равномерно распределённых чисел с плавающей запятой …...43

5.1. Формат чисел с плавающей запятой ……..43

5.2. Наивное отображение чисел с плавающей запятой на целые числа ……44

5.3. Наброски усложнённого алгоритма ……..49

5.4. Программная реализация наивного алгоритма ……54

6. Общие вопросы ……57

6.1. Статистические функции ……57

6.2. Детали реализации ……..58

Заключение ……61

Список использованных источников ……..62

Приложение А. Техническое задание ……68
Приложение Б. Листинг программ ……73
Приложение В. Доказательство утверждения из подраздела 5.2 ……..131

# ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

| | |
|---|---|
| Медовое шифрование | Подход к кодированию, применяемый для усиления стойкости зашифрованных данных к атакам перебором |
| Шифрование | Совокупность операций зашифрования и расшифрования [55] |
| Зашифрование | Получение из пары (открытый текст, секретный ключ) шифротекста [55] |
| Расшифрование | Операция, обратная зашифрованию; получение из пары (шифротекст, секретный ключ) открытого текста [54, 55], не обязательно соответствующего изначально зашифрованному |
| Имитовставка | Специальный набор символов, который добавляется к сообщению; предназначен для обеспечения целостности сообщения и аутентификации источника данных [53] |
| *NaN*-значения | Одно из особых состояний числа с плавающей запятой, означающее «not a number» («не число»); может возникнуть в различных случаях, например, когда предыдущая математическая операция завершилась с неопределённым результатом [24] |
| МТИ | Массачусетский технологический институт |

# ВВЕДЕНИЕ

В начале 2014 года американские учёные Ари Джулс и Томас Ристенпарт выдвинули новую идею в криптографии, которая потенциально может иметь заметное практическое приложение. Эта идея носит название «медовое шифрование» («*honey encryption*») и заключается в таком шифровании, которое при неверно введённом ключе выдаёт ложные, но правдоподобно выглядящие данные. В своей работе [1] они приводят леммы и теоремы, доказывающие, что предлагаемая ими идея с математической точки зрения вполне имеет право на жизнь. Для быстрого знакомства с работой учёных хорошо подходит их презентация для конференции *Eurocrypt 2014* [2]. На русском языке об этой интересной концепции писал портал *ThreatPost* [3].

При распространённом сегодня подходе полный перебор всех ключей шифрования обязательно даёт результат. Даже если не используются имитовставки или электронные подписи, говорящие о правильности или неправильности расшифрования, при использовании неверного ключа осмысленные данные расшифровываются в последовательность псевдослучайных байт, легко отличимых от реальных данных компьютером или человеком. Длина ключей многих алгоритмов такова, что провести полный перебор просто невозможно, однако нередко из-за проблем криптографического алгоритма или его реализации можно заменить полный перебор сокращённым, сильно ускорив процесс [4].

В своей статье учёные предлагают подход, при котором не только частичный, но даже полный перебор всех возможных ключей может оказаться бесполезным за счёт хитрого трюка: при использовании любого ключа (правильного или неправильного) для расшифрования данных всегда получается правдоподобный результат, и из-за этого крайне сложно отличить правильно расшифрованные данные от неправильных. Это может быть очень

полезно в системах, где трудно обеспечить стойкое шифрование, например при генерации ключа шифрования из пароля, а также во многих других приложениях.

Медовое шифрование реализуется двумя компонентами – во-первых, подходящим обычным алгоритмом шифрования, а во-вторых, кодировщиком и декодировщиком, меняющими распределение. Именно в кодировщике и декодировщике заключена логика генерации правдоподобных ложных данных. Всё это показано на рис. В.1.

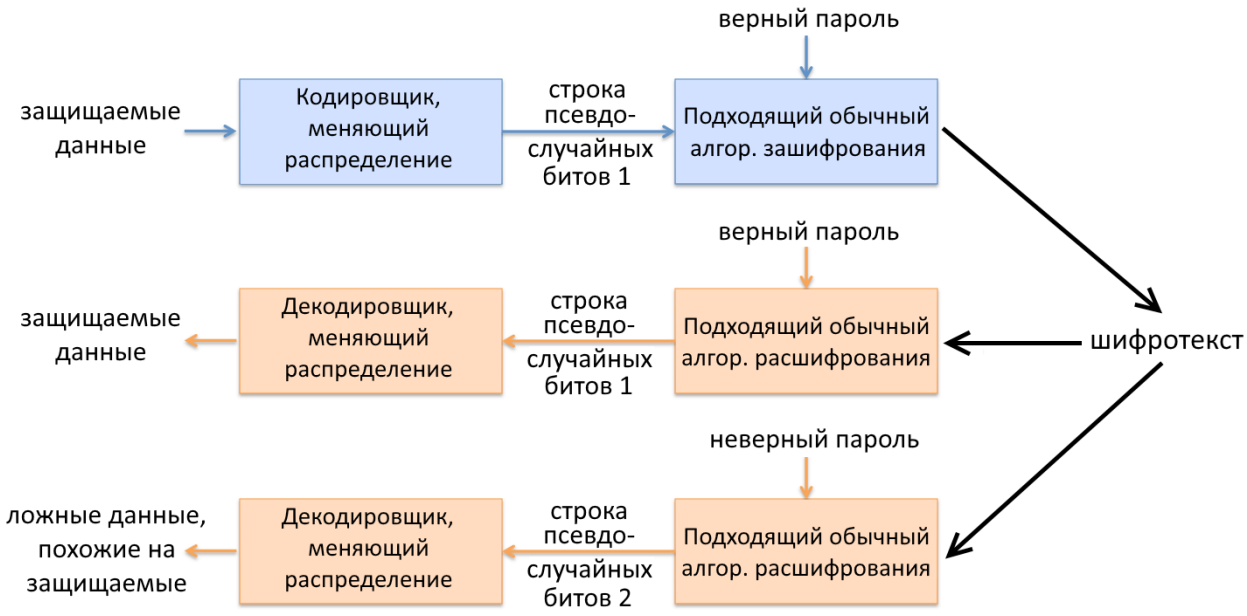


Рис. В.1. Схема медового шифрования (на основе [5])

Рассмотрим проблему и её решение на примере. Предположим, что вы хотите зашифровать какое-то имя с помощью четырёхзначного *PIN*-кода (возможные значения – от 0000 до 9999). В качестве шифруемого имени вы выбрали «Алина», в качестве *PIN*-кода – 3764. Если злоумышленник выкрадет шифротекст и если он знает, что открытый текст – это имя, то шифрование не спасёт вас. Злоумышленник может вручную перебрать все 10 000 комбинаций *PIN*-кода, и почти наверняка 9999 раз он получит что-то,

вообще не похожее на имя, и лишь один раз – «Алина». Хуже того, он может с минимальными усилиями поручить перебор и оценку результатов расшифрования компьютеру! Если же мы внедрим в систему медовое шифрование, то перебрав 10 000 возможных комбинаций *PIN*-кода на компьютере или вручную, злоумышленник 10 000 раз получит имена. Чтобы найти среди них то, которое вы зашифровали изначально, ему придётся произвести дополнительную работу.

Как же реализовать кодировщик и декодировщик? В самом простом случае вы можете, например, завести таблицу с двумя полями – «Имя» и «Код». Теперь при зашифровании вы будете заменять имя на соответствующий код, а потом зашифровывать этот код; при расшифровании вы сначала будете расшифровывать код, а затем переводить его в имя. Модулярная арифметика предоставляет удобный аппарат для того, чтобы каждая попытка расшифрования приводила к допустимому коду.

В дальнейшем мы будем для краткости использовать именно словосочетание «медовое шифрование» вместо намного более длинного «статистическое кодирование для усиления стойкости зашифрованных данных к атакам перебором», вынесенного в название диссертации. Термин «медовое шифрование» был предложен непосредственно авторами оригинальной научной работы и является устоявшимся в области криптографии.

Необходимо сделать ещё несколько замечаний, касающихся терминов. В соответствии с традициями отечественной литературы по криптографии будем называть «шифрованием» совокупность операций зашифрования и расшифрования, «зашифрованием» – операцию перевода открытого текста в шифротекст с помощью секретного ключа [55]. В русскоязычной литературе часто принято под «расшифрованием» понимать легитимное получение открытого текста (результат легитимного владения секретным ключом), а

под «дешифрованием» – нелегитимное получение открытого текста (результат взлома), причём под открытым текстом понимается изначально зашифрованный текст. Такая терминология аргументированно критикуется за внесение путаницы в рассуждения [54]. В этой работе для сохранения ясности используется исключительно термин «расшифрование» вне зависимости от того, каким способом добывается открытый текст и идентичен ли он изначально зашифрованному открытому тексту. Такой подход обусловлен тем, что различение в этой работе «расшифрования» и «дешифрования» с получающимися в результате «правильными» и «неправильными» открытыми текстами потребовало бы ненужных усилий по уточнению и пониманию этих терминов безо всякой видимой пользы. Текст был составлен так, чтобы читателю всегда было ясно из контекста, что конкретно имеется в виду.

# 1. АНАЛИЗ СУЩЕСТВУЮЩИХ РАБОТ

Медовое шифрование является новой концепцией в криптографии, поэтому число статей о нём, а также число его реализаций на практике пока сравнительно невелико. Ниже будут рассмотрены исключительно работы, связанные с практической реализацией медового шифрования.

В своей статье [1] авторы строят теоретические модели для таких реальных приложений, как секретные ключи криптографического алгоритма *RSA*, *CVV*-коды кредитных карт и выбранные пользователями *PIN*-коды. К сожалению, эти концепции не были программно реализованы авторами. Группа студентов Массачусетского технологического института (МТИ) описала [11] и осуществила реализацию медового шифрования применительно к номерам кредитных карт и простым текстовым сообщениям, исходные коды опубликованы на портале *GitHub* [12]. В этой же работе обосновывается расширение схемы медового шифрования: для его реализации может использоваться не только симметричная криптография (например, блочные шифры в режимах *CTR* и *CBC*, рекомендованные авторами оригинальной работы), но и асимметричная криптография, например алгоритм *RSA*.

Авторы оригинальной статьи вместе с двумя другими учёными написали обстоятельную статью [8] о применении медового шифрования к сводам паролей. Исходный код разработанной ими системы *NoCrack* опубликован на хостинге *GitHub* [9]; помимо этого, независимый разработчик Ширис Кумар сделал свою реализацию того же функционала и опубликовал её на этом же портале [10]. В работе [31] эта система подвергается аргументированной критике: описывается мощная атака на неё и предлагаются меры для защиты от такой атаки.

Ари Джулс вместе с группой других учёных описал теоретически и реализовал практически систему *GenoGuard*, реализующую концепцию медового шифрования применительно к генетическим данным человека [26].

Оригинальную работу в области медового шифрования сделал Филипп Тьювэн: он использовал парсер английского языка *Link Grammar* для того, чтобы зашифровывать предложения на английском языке в грамматически верные предложения на английском языке. Как он верно замечает в своей статье [7], статистический анализ может поспособствовать успешному взлому такой необычной криптосистемы (ведь средний человек использует довольно небольшую долю от всех слов языка), но в перспективе эту работу можно взять за основу для чего-то более серьёзного и основательного. Скрипты для реализации криптосистемы поверх программного пакета *Link Grammar* приложены к журналу, в котором была опубликована статья. Пример бесплодной атаки полного перебора против этой криптосистемы показан на рис. 1.1. На нём мы можем видеть, как при расшифровании шифротекста различными ключами получаются грамматически правильные, пусть и совершенно бессмысленные предложения с редко используемыми словами.

```
$ echo "your search cements a tannery to escort a unrelieved clause exuberantly"|./bruteforce
...
22366:their presentiment reprehends a saxophone to irk a topless mind perennially
22367:your cry compounds a examiner to shoulder a massive bootlegger unconsciously
22368:our handcart renounces a lamplighter to imprint a outbound doorcase weakly
22369:my neurologist fascinates a plenipotentiary to butcher a psychedelic imprint automatically
22370:their safecracker vents a spoonerism to refurnish a shaggy parodist complacently
22371:your epicure extols a governor to belittle a indecorous clip heatedly
22372:our kilt usurps a monger to punish a loud foothold indirectly
22373:my piranha mugs a resistor to evict a obstetric malaise laconically
22374:its controller unsettles a duchess to ponder a diversionary beggar riotously
22375:your glen mollifies a interjection to embezzle a forgetful decibel speciously
22376:our misdeal countermands a pedant to typify a imperturbable heyday topically
22377:their bower misstates a colloquialism to disorientate a apoplectic warrantee courteously
22378:its downpour copies a frolic to sweeten a circumspect cavalcade dispiritedly
22379:your infidel resurrects a masseuse to manufacture a differential fairway famously
22380:my abstract contaminates a birthplace to squire a unaltered subsection lukewarmly
22381:their co-op resents a deuce to inveigle a unsubtle attendant objectionably
```

Рис. 1.1. Пример бесплодной атаки полного перебора
против криптосистемы Филиппа Тьювэна [7]

Как уже было упомянуто выше, студенты МТИ в своей работе также коснулись темы медового шифрования сообщений. Они работали с сообщениями простой структуры, в которых используются самые распространённые существительные и глаголы в английском языке. Результаты этих студентов и Филиппа Тьювэна наглядно показывают, какой сложной может быть задача разработки схемы медового шифрования для некоторых данных. В работе [49] эта тема рассмотрена гораздо более серьёзно, описаны основные проблемы и предложен интересный подход к их решению, однако авторы не реализовывали и не анализировали его.

Группа южнокорейских учёных независимо от Джулса и Ристенпарта разработала концепцию статистической схемы кодирования, сходную с медовым шифрованием [28]. Им также очень интересен вопрос повышения защищённости текстовых сообщений, для чего они применяли марковские модели [28, 29]. Их результаты в этом направлении можно назвать значительно более зрелыми, чем результаты студентов МТИ и Филиппа Тьювена, однако и их можно значительно улучшить. Кроме того, южнокорейские учёные применили свою концепцию к изображениям, получив любопытные результаты в области стеганографии. Они назвали эту работу «визуальным медовым шифрованием» [30]. К сожалению, эти авторы не публиковали исходных кодов своих программ. Используя результаты работы по визуальному медовому шифрованию, группа французских учёных провела интересное исследование по применению компьютерного теста *CAPTCHA* в криптографии [27].

Профессор Уильям Бушанан опубликовал на своём сайте веб-приложение, демонстрирующее медовое шифрование номеров кредитных карт (сейчас не работает, ранее работало) [14]. Согласно информации с официального сайта, сотрудники Лаборатории криптографических исследований Миланского технического университета некоторое время работали над внедрением медового шифрования ключей криптоалгоритма

*RSA* в программные пакеты *OpenSSL/LibreSSL* и *GPG* [15], но затем эта работа была свёрнута.

Пользователь *yinweihappy168* опубликовал на портале *GitHub* простые программы для осуществления медового шифрования телефонных номеров, паролей, ключей криптоалгоритма *RSA* и некоторых других видов данных [48]. Пользователь *VIVEKHYPER* того же портала опубликовал свои наработки по теме медового шифрования баз данных пользователей некоторых ресурсов [52]. Простейшую, примитивную схему медового шифрования названий штатов США разработал Виктор Нгуен [32]. Эти работы могут рассматриваться только как основа для чего-то более серьёзного и строго обоснованного, но определённо показывают интерес к теме со стороны криптографического сообщества.

Автором не было обнаружено работ, в которых рассматривалось бы применение медового шифрования к числовым данным различных встроенных в языки программирования типов, хотя такие данные широко используются в реальной жизни и являются сравнительно простыми. Работы, связанные с телефонными номерами, номерами и *CVV*-кодами кредитных карт, выбранными пользователями *PIN*-кодами могут считаться частными случаями общей задачи медового шифрования числовых данных различных типов. Работы, связанные с секретными ключами криптоалгоритма *RSA*, стоят особняком, поскольку в этих ключах хранятся большие числа (до нескольких тысяч бит) очень специфического вида. Более конкретно, секретный ключ *RSA* состоит из числа, мультипликативно обратного к открытой экспоненте по некоторому модулю, а также полупростого числа.

## 2. ПОСТАНОВКА ЗАДАЧИ

Далее в этой магистерской диссертации будет детально рассмотрено в теории и осуществлено на практике медовое шифрование применительно к числовым данным различных типов. Техническое задание на диссертацию вынесено в приложение А.

***Актуальность.*** В настоящее время в информационных технологиях широко используются достижения современной криптографии. Особенно ярко это видно на примере сайтов и сервисов в Интернете: соединение между клиентом и сервером обычно защищено с помощью криптографического протокола *TLS*, включающего в себя шифрование и электронную подпись, а пароли на серверах согласно общепринятой практике хешируются. Прогресс, однако, не стоит на месте: с годами вычислительные мощности, необходимые для осуществления практических атак на защищённые криптографией данные, становятся всё дешевле, а сами атаки – всё более серьёзными, так что вопрос повышения криптографической стойкости всё ещё остаётся острым. Медовое шифрование способно усилить защищённость шифруемых данных, сделав определённые атаки на криптографические алгоритмы бесплодными.

***Цель.*** Целью работы над диссертацией является исследование вопросов применения общей концепции медового шифрования к конкретным типам числовых данных (целым и с плавающей запятой, без знака и со знаком, различного размера), разработка алгоритмов для осуществления медового шифрования данных этих типов, а также практическая реализация этих алгоритмов. Предполагается, что результаты этой работы позволят заметно усилить защищённость рассматриваемых типов данных.

***Методы.*** Для достижения поставленной цели будут использованы аналитические, алгоритмические и экспериментальные методы. Аналитические методы потребуются для проведения анализа существующих

работ и конкретных аспектов криптографической защиты числовых данных, алгоритмические – для разработки безопасных и эффективных алгоритмов решения поставленных задач, экспериментальные – для проверки корректности разработанных алгоритмов и их практической реализации.

***Достоверность.*** Правильность реализации медового шифрования можно будет проверять при помощи попыток расшифрования зашифрованных данных в предположении, что ключи имеют короткую длину (чтобы полный перебор длился быстро). В идеальном случае при применении медового шифрования каждая попытка расшифрования (с каждым возможным ключом расшифрования) будет давать правдоподобные, осмысленные результаты, в отличие от обычного шифрования, дающего в случае неверного ключа псевдослучайные данные. Для оценки правдоподобности результаты работы тестов стоит проанализировать с помощью статистики.

***Новизна.*** Научная новизна работы заключается в том, что в ней впервые рассмотрены вопросы применения медового шифрования к числовым данным в целом, а не только к их небольшим подмножествам (таким, как номера кредитных карт).

***Апробация.*** В осеннем семестре 2015-2016 учебного года автором был успешно выполнен, презентован перед другими студентами и защищён реферат по дисциплине «Современные проблемы информатики и вычислительной техники», связанный с темой диссертации. В декабре 2015-го года прошло выступление автора с презентацией концепции этой магистерской диссертации перед преподавателями кафедры Вычислительных машин, систем и сетей (ВМСС) Московского энергетического института, был подготовлен реферат. В феврале 2016-го года прошло выступление автора о первых результатах работы над этой магистерской диссертацией на XXII международной научно-технической конференции студентов и аспирантов «Радиоэлектроника, электротехника и энергетика» в Московском

энергетическом институте. В весеннем семестре 2015-2016 учебного года автором была успешно выполнена и защищена курсовая работа по дисциплине «Проблемы организации вычислений», связанная с темой диссертации. В мае 2016-го года прошло повторное выступление автора с презентацией достигнутых за весенний семестр практических и теоретических результатов перед преподавателями кафедры ВМСС, был подготовлен реферат. По итогам конференции и по итогам курсовой работы были написаны 2 статьи, опубликованные в электронном журнале «Вычислительные сети. Теория и практика» № 28 за июнь 2016-го года [16, 17]. В марте 2017-го года прошло выступление автора о некоторых результатах работы над этой магистерской диссертацией на XXIII международной научно-технической конференции студентов и аспирантов «Радиоэлектроника, электротехника и энергетика» в Московском энергетическом институте. Тезисы доклада были опубликованы в сборнике тезисов докладов этой конференции [50]. По итогам конференции была написана статья, опубликованная в электронном журнале «Вычислительные сети. Теория и практика» № 30 за июнь 2017-го года [51].

# 3. ОБРАБОТКА РАВНОМЕРНО РАСПРЕДЕЛЁННЫХ ЦЕЛЫХ ЧИСЕЛ

## 3.1. Целые числа в языке Си

Встроенные в язык Си целочисленные типы, безусловно, являются самыми простыми числовыми типами данных – они куда проще, чем числа с фиксированной или плавающей запятой, рациональные, комплексные числа, числа для интервальных вычислений и длинной арифметики. Тем не менее, даже у этих простых типов есть свои подводные камни, так что рассмотрим эти типы подробно с опором на [33].

Целочисленные типы в Си отличаются друг от друга по двум важным свойствам: наличие/отсутствие знака и размер. Первое свойство позволяет программисту кодировать в числе одного и того же размера разные диапазоны значений: если тип беззнаковый (*unsigned*), то число может принимать значения от 0 до некоторого положительного максимального; если тип знаковый (*signed*), то число может принимать значения от некоторого отрицательного минимального до некоторого положительного максимального. Поскольку у двух типов, отличающихся лишь знаковостью (например, *unsigned int* и *signed int*), размер остаётся одним и тем же, то их кодовое пространство позволяет хранить одно и то же количество различных значений – вопрос лишь в том, в каком интервале будут эти значения. Если не указать знаковость типа, то для типов *short*, *int*, *long* и *long long* значением знаковости по умолчанию является *signed*, для типа *char* – *unsigned* или *singed* в зависимости от платформы. С этим свойством никаких проблем обычно нет.

Увы, но со вторым свойством дела обстоят гораздо сложнее. Наиболее распространёнными в языке Си являются целочисленные типы *char*, *short*, *int*, *long* и *long long*, и беда в том, что стандарт оговаривает лишь их

минимальный размер! Более конкретно, тип *char* должен иметь размер минимум 1 байт, *short* и *int* – минимум 2 байта, *long* – минимум 4 байта, *long long* – минимум 8 байт. Размеры типов на разных платформах могут сильно отличаться – например, тип *int* на 16-битных платформах обычно имеет размер 2 байта, на 32-битных – 4 байта, на 64-битных – 8 байт. Из этого прямо следует, что при использовании этих типов у программиста на языке Си есть несколько возможных вариантов действий:

1) Консервативно полагать, что целочисленные типы имеют минимальную длину, оговоренную в стандарте. Очень часто это бывает удовлетворительным решением, но оно с большой долей вероятности ведёт к неэффективному использованию ресурсов.

2) Полагать, что целочисленные типы имеют некую заранее известную длину, превышающую минимальную. Такой вариант возможен, когда разработка ведётся под заранее известные программно-аппаратные платформы. Это ведёт к дополнительным трудностям при портировании программного обеспечения на платформы, отличные от выбранных изначально.

3) Писать программы с учётом любых имеющихся на целевой платформе целочисленных типов. Хороший с точки зрения производительности способ, требующий заметных дополнительных трудозатрат.

Как мы видим, каждый из вариантов имеет серьёзные минусы. Проблема оказалась настолько серьёзной, что в стандарте *C99* были введены целочисленные типы фиксированной длины: *uint8_t* и *int8_t* (соответственно беззнаковый и знаковый типы длиной ровно 1 байт на любой платформе), *uint16_t* и *int16_t* (то же самое, но длиной 2 байта), *uint32_t* и *int32_t* (длиной 4 байта), *uint64_t* и *int64_t* (длиной 8 байт). Ряд авторов (например, [34, 37]) рекомендует использовать эти типы во всех новых разработках вместо

традиционных *char*/*short*/*int*/*long*/*long long*, и автор этой диссертации во многом солидарен с таким мнением. Действительно, ценой использования непривычных поначалу обозначений мы получаем гораздо более портабельный, надёжный и понятный код! По этой причине в описываемом проекте было решено по возможности использовать именно целочисленные типы с фиксированным размером.

Напоследок следует заметить, что в языке Си целочисленные типы размером больше одного байта (в том числе с фиксированным размером) представляются в памяти по-разному в зависимости от платформы. Более конкретно, байты могут записываться в порядке от старшего к младшему (*big-endian*), от младшего к старшему (*little-endian*) или даже более запутанно (*middle-endian*) [36]. Для программиста это означает следующее: если нужно передать числовые данные с одного компьютера на другой и нет уверенности, что компьютер-получатель имеет точно такую же платформу, как и компьютер-отправитель, то данные следует перекодировать для одинаковой интерпретации их обеими устройствами. Такая перекодировка не была учтена в разрабатываемом проекте и при необходимости должна быть реализована прикладным программистом самостоятельно.

### 3.2. Алгоритмы для типа *uint8_t*

Начать рассмотрение медового шифрования числовых данных стоит с самого простого типа — равномерно распределённых 8-битных целых чисел без знака, чтобы затем развить его на более сложные типы и на случай произвольного распределения чисел.

Итак, мы имеем самый миниатюрный и простой тип числовых данных, который позволяет хранить в одном байте целые числа от 0 до 255. Предположим, что в шифруемом массиве по логике возможны лишь некоторые значения чисел — например, от 100 до 200, причём вероятность

их появления примерно одинакова (либо мы просто не хотим давать злоумышленнику более точных метаданных об этих числах в силу конфиденциальности этих метаданных). Если злоумышленник будет расшифровывать массив таких чисел путём атаки по словарю или полного перебора, то получив в любом элементе массива значение от 0 до 99 или от 201 до 255, он будет точно знать, что текущий ключ расшифрования неверен, и продолжит поиск. Поскольку операция расшифрования при неверном ключе даёт псевдослучайные данные, то вероятность появления любого из чисел от 0 до 255 равна 1/256. Для массива из одного элемента вероятность правдоподобной расшифрования произвольным ключом в этом случае равна (200 – 100 + 1) / 256 = 101 / 256 = 0.39453125, для массива из пяти элементов — 0.39453125 ^ 5 = 0.00955888072, для массива из пятидесяти — 0.39453125 ^ 50 = 6.3689879 * 10 ^ (–21). Для запутывания злоумышленника нам хотелось бы, чтобы попытка расшифрования любым ключом давала бы правдоподобный результат.

Первый наивный подход к достижению этой цели следующий (рис. 3.1):

1. На основе минимального и максимального возможных значений в массиве находим размер полной группы элементов (*group_size* = *max* – *min* + 1), а также число групп, которые поместятся в кодовое пространство от 0 до 255 (*group_num* = округление_с_избытком(256 / *group_size*), деление осуществляется в формате с плавающей запятой). Если 256 нацело делится на размер группы, то последняя группа будет полной (т. е. будет иметь ровно *group_size* элементов), однако скорее всего она будет неполной (т. е. будет иметь от 1 до (*group_size* – 1) элементов). Это можно выразить такой формулой: *last_group_size* = 256 % *group_size*, причём если *last_group_size* = 0, то последняя группа полная, иначе неполная.

2. При кодировании элемента массива мы сначала нормализуем его — вычитаем из него минимальное значение *min*, получая значение в диапазоне от 0 до (*max* – *min*), после чего помещаем это значение в псевдослучайно выбранную группу. Это можно сделать прибавлением к нормализованному элементу псевдослучайного неотрицательного числа, умноженного на размер группы (*rand* * *group_size*), при условии что результат не превышает 255.

3. При декодировании элемента сначала найдём его значение по модулю *group_size*, получая значение в диапазоне от 0 до (*max* – *min*). После этого денормализуем его, прибавив к нему минимальное значение *min*, получая значение в диапазоне от *min* до *max*.


Рис. 3.1. Наивный подход к медовому шифрованию 8-битных целых чисел без знака

К сожалению, такой подход имеет существенный минус: результирующее распределение скорее всего будет иметь повышенные вероятности в левой части и пониженные в правой, так как последняя группа почти наверняка будет неполной. Скажем, если мы кодируем числа от 0 до

254, то вероятность расшифрования числа 0 будет в два раза больше, чем чисел от 1 до 254, поскольку число 0 в отличие от всех остальных чисел кодируется двумя кодами: 0 и 255. Это существенно расходится с принятой нами моделью данных, в которой допустимые числа имеют равномерное распределение, то есть встречаются с одинаковой вероятностью. Такое расхождение крайне нежелательно для нас, поскольку оно может серьёзно помочь злоумышленнику в отделении изначально зашифрованных данных от сгенерированных алгоритмом ложных данных.

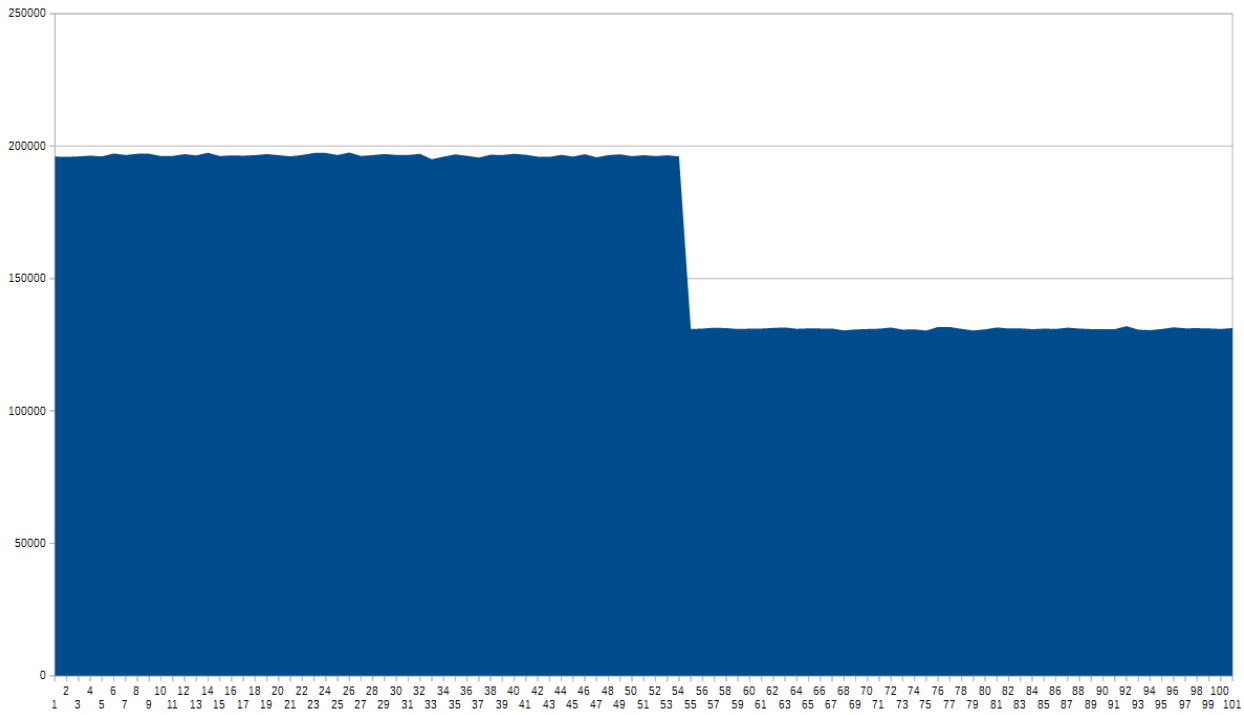


Рис. 3.2. Результат работы наивного алгоритма – явно неравномерное распределение

На рисунке 3.2 изображён возможный результат работы наивного алгоритма при расшифровании зашифрованного целочисленного массива различными ключами – одним верным и множеством неверных. Легко увидеть, что данные в целом имеют явно неравномерное распределение. Из этого следует, что после попытки расшифрования конкретным ключом

злоумышленник сможет сделать вывод, являлся ли этот ключ правильным. Это разрушает свойство медовости, которое мы стремимся обеспечить.

Для решения этой проблемы Джулс и Ристенпарт предлагают использовать для кодирования большее кодовое пространство, чем для изначальных чисел [1] — например, кодировать однобайтовые числа двумя байтами. Именно этим путём стоит пойти для решения возникшей проблемы. Будем кодировать беззнаковые однобайтовые числа в беззнаковые двухбайтовые. Почему беззнаковые? Такие числа проще знаковых в обработке, отладке и восприятии человеком; знаковые числа не имеют преимуществ перед беззнаковыми в рассматриваемом нами контексте. Разумеется, такой подход потребует в два раза больших затрат памяти, но существенное повышение защищённости оправдывает такие затраты. Несложно увидеть, что при использовании кодового пространства от 0 до 255 в худшем случае (размер группы от 129 до 255) есть 1 полная и 1 неполная группа, в результате чего вероятности появления одной части чисел будут на 50% выше, чем другой части чисел (вероятность значения 0 – 2 / 256, вероятность наименее вероятного значения — 1 / 256). При использовании кодового пространства от 0 до 65 535 в худшем случае (размер группы, равный 255) есть 257 полных и 1 неполная группа, в результате чего вероятности появления одной части чисел будут не более чем на 0,0016% выше, чем других (вероятность значения 0 – 258 / 65 536, вероятность наименее вероятного значения – 257 / 65 536). Если пользователя не устроит даже такое небольшое отклонение, он можем использовать ещё большее кодовое пространство, но для большинства приложений этого должно хватить. Более подробно этот вопрос рассмотрен в подразделе 3.5.

Возможны также несколько особых ситуаций, которые необходимо рассмотреть отдельно: это ситуации, когда *group_size* = 256 (т. е. возможны любые значения чисел от 0 до 255) и когда *group_size* = 1 (т. е. возможно лишь одно значение чисел). В первом случае мы копируем массив входных

данных в массив выходных данных, а затем дополняем его массивом псевдослучайных байт той же длины. Раз уж все значения уже допустимы, то опция медового шифрования таких данных фактически уже активирована, осталось лишь скрыть факт появления этой особой ситуации. Во втором случае мы вместо описанных выше операций просто записываем в выходной массив псевдослучайные числа, поскольку все они при расшифровании будут переведены в это единственное допустимое значение. Для скрытия факта появления этой особой ситуации размерность выходного массива в байтах должна быть в 2 раза больше оригинального. Таким образом получим, что при кодировании N однобайтовых чисел (то есть N байт) преобразуются в N двухбайтовых чисел (то есть 2*N байт), и наоборот, при декодировании N двухбайтовых чисел преобразуются в N однобайтовых чисел.

Схемы итоговых алгоритмов в общем виде представлены на рис. 3.3 и 3.4.

### 3.3. Программная реализация для типа *uint8_t*

Вторая, усложнённая версия алгоритма, использующая кодовое пространство от 0 до 65 535 (тип данных *uint16_t*), была реализована программно. При разработке этой программы особое внимание было уделено тестированию. Была написана тестовая программа, которая:

1) кодирует и декодирует псевдослучайные данные в определённых поддиапазонах типа *uint8_t*, проверяя их совпадение до и после этих операций;

2) кодирует, зашифровывает, расшифровывает, декодирует псевдослучайные данные в определённом поддиапазоне типа *uint8_t*, проверяя их совпадение до и после этих операций;

3) проводит атаку методом «грубой силы» [4] со сложностью 2^16

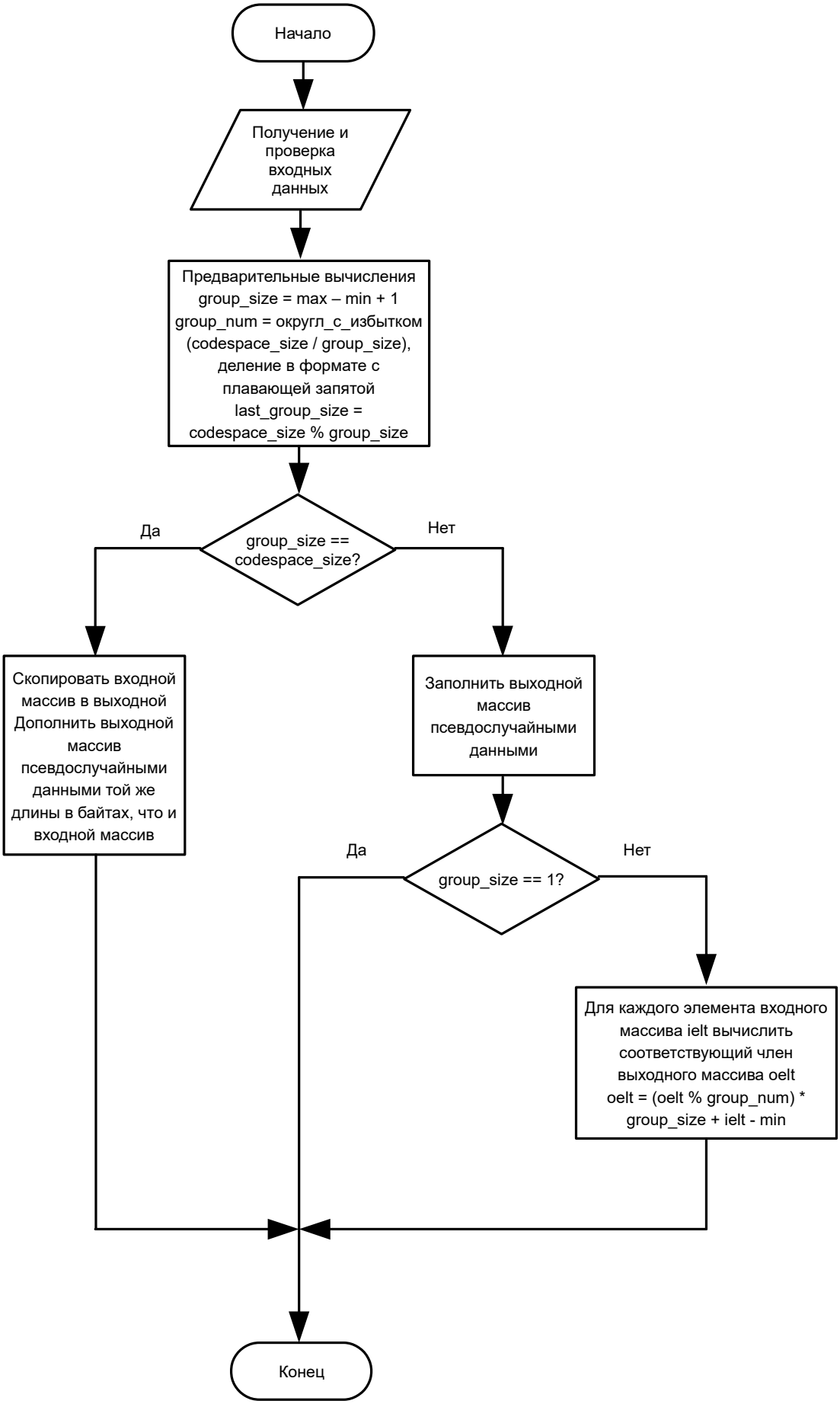


Рис. 3.3. Схема алгоритма кодирования равномерно распределённых целых чисел

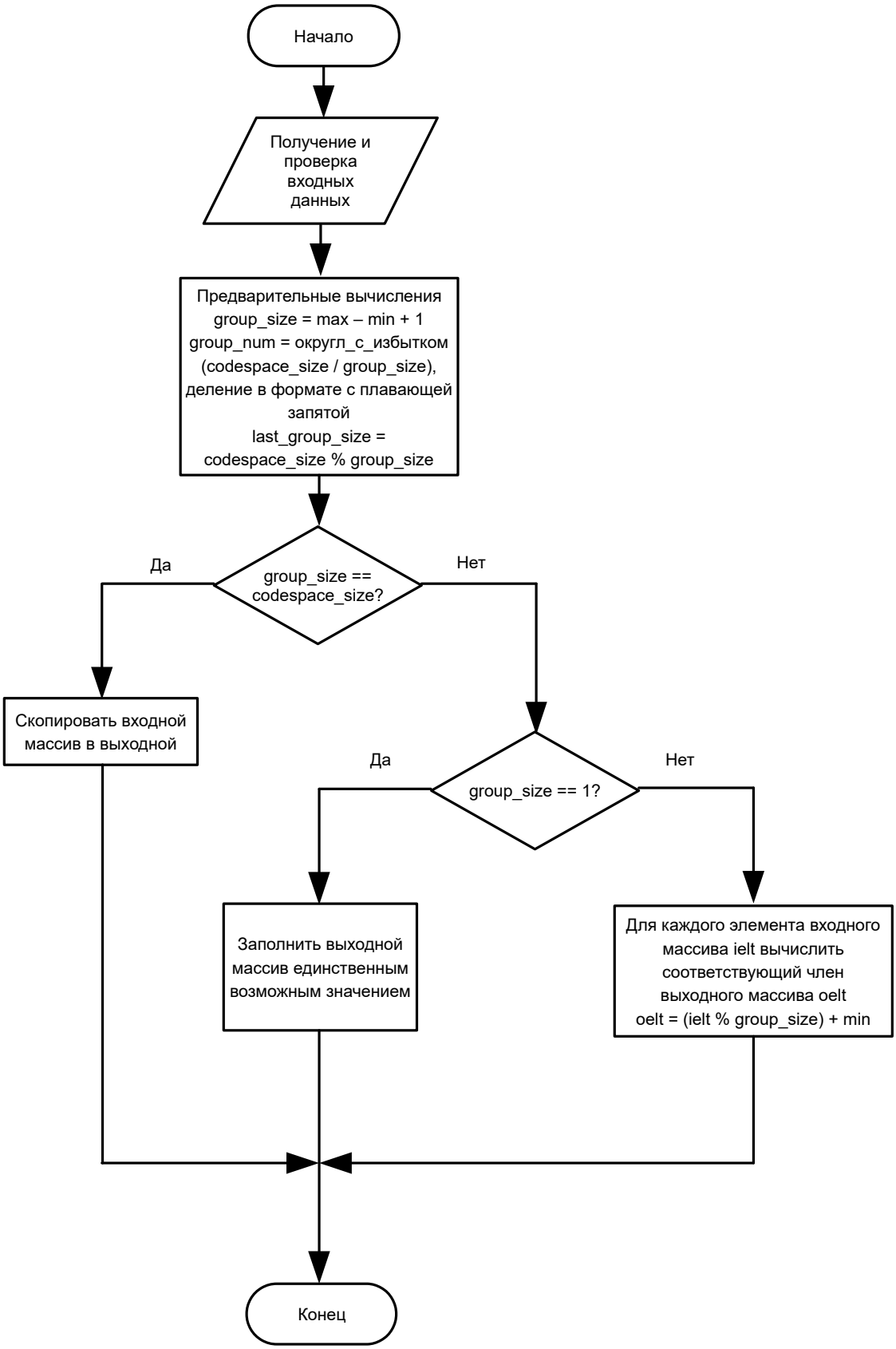


Рис. 3.4. Схема алгоритма декодирования равномерно распределённых целых чисел

вариантов на данные, зашифрованные с применением медового шифрования для демонстрации преимуществ последнего;

4) проверяет работу функций в фиксированных обычных ситуациях;

5) проверяет работу функций в особых ситуациях — когда возможны все значения исходных данных (*group_size* = 256) и когда возможно только одно значение исходных данных (*group_size* = 1);

6) проверяет работу функций в случае неправильных данных на входах процедур (нулевые указатели вместо действительных, нулевые длины массивов, неверные значения минимумов или максимумов в массивах);

7) собирает статистические данные по результатам пунктов 1 и 3 для оценки эффективности разработанной методики и её практической реализации.

Для использования медового шифрования требуется использование вполне определённых криптографических примитивов: авторы оригинальной работы [1] рекомендуют использовать блочный шифр в режимах *CBC* (режим сцепления блоков шифротекста) или *CTR* (режим счётчика) без дополнений. В этой работе для шифрования использовался блочный шифр *AES-256* в режиме *CBC* без дополнений. Этот шифр является действующим американским стандартом шифрования [20], широко применяющимся во всём мире. Длина ключа выбрана равной 256 битам, поскольку это минимум, обеспечивающий защиту от атак на квантовом компьютере, который может стать реальностью в ближайшие десятилетия [19].

### 3.4. Реализации для некоторых других целочисленных типов

Описанные выше алгоритмы для типа *uint8_t* можно распространить и на другие целочисленные типы с некоторыми изменениями. Как правило, нам остаётся только изменить типы и константы для успешного кодирования

N чисел, каждое размером в K байт (общий размер – N*K байт) в N чисел, каждое размером в 2*K байт (общий размер – 2*N*K байт), и обратного декодирования.

Для адаптации алгоритма к типу *int8_t* (однобайтовые целые числа со знаком, диапазон значений – от –128 до 127) нам действительно достаточно лишь изменения типов и констант, а также изменения тестов. Стоит заметить, что типом выходных данных остаётся uint16_t.

Работать со следующей парой типов, *uint16_t* и *int16_t* (двухбайтовые целые числа без знака и со знаком соответственно), оказывается уже гораздо сложнее. В двух прошлых случаях пространство незакодированных данных составляло 256 элементов (типы *uint8_t* и *int8_t*), а закодированных – 65 536 (тип *uint16_t*), так что для сбора статистики предоставлялись широчайшие возможности. Здесь же пространство незакодированных данных составляет 65 536 элементов (типы *uint16_t* и *int16_t*), а закодированных – 4 294 967 296 (тип *uint32_t*)! Другими словами, статистически исследовать закодированные данные в разумные сроки так же подробно, как и прежде, у нас уже нет возможности. Остаётся надеяться на обычные тесты, а также менее подробную статистику по закодированным и раскодированным данным. Поскольку используется тот же алгоритм, что и для ранее оттестированных и тщательно проанализированных типов данных *uint8_t* и *int8_t*, то всё должно работать правильно. В пользу этого утверждения также говорят полученные результаты тестов для типов *uint16_t* и *int16_t*.

С переходом к следующей паре типов, *uint32_t* и *int32_t* (четырёхбайтовые целые числа без знака и со знаком соответственно), проблема усугубляется. Пространство незакодированных данных составляет 4 294 967 296 элементов (типы *uint32_t* и *int32_t*), а закодированных – 18 446 744 073 709 551 616 (тип *uint64_t*). Сбор статистики ещё сильнее усложняется, но некоторые статистические данные всё равно удаётся

собрать. Кроме того, мы по-прежнему можем с успехом использовать обычные тесты. Результаты, полученные при тестировании типов *uint32_t* и *int32_t*, а также более подробные результаты работы для рассмотренных выше типов данных внушают уверенность, что всё работает правильно.

Реализация этого алгоритма для типов данных *uint64_t* и *int64_t* (восьмибайтовые целые числа без знака и со знаком соответственно) затруднена отсутствием в два раза более длинного целочисленного типа в текущем стандарте языка Си, так что её рассмотрение вынесено в подраздел 3.5.

Здесь возникает закономерный вопрос: «Что может пойти не так при использовании ранее рассмотренного алгоритма с другими типами данных?». В процессе тестирования оказалось, что автоматическое преобразование типов и целочисленное переполнение иногда мешают правильному на бумаге алгоритму работать на реальном оборудовании. С первой проблемой можно легко побороться благодаря явному, заданному программистом преобразованию типов, со второй же дело обстоит сложнее.

В подразделе 3.2 были рассмотрены алгоритмы кодирования и декодирования данных типа *uint8_t*. В алгоритме кодирования дважды использовался размер пространства выходных данных (256 элементов = 255 + 1 = *UINT8_MAX* + 1):

1) *last_group_size* = 256 % *group_size*;

2) *group_num* = округление_с_избытком(256 / *group_size*), деление осуществляется в формате с плавающей запятой.

Рассмотрим первую формулу. Она отлично работает в том виде, в котором она записана выше, но при использовании *uint64_t* в качестве выходного типа (то есть при кодировании данных типов *uint32_t* и *int32_t*) она перестаёт работать. Проблема в том, что из-за целочисленного переполнения результат сложения (*UINT64_MAX* + 1) равен 0, а большего

целочисленного типа в текущей редакции языка Си нет. К счастью, на помощь приходит модулярная арифметика: так как (*a* + *b*) mod *c* = ( (*a* mod *c*) + (*b* mod *c*) ) mod *c*, то (*UINT64_MAX* + 1) % *group_size* = ( (*UINT64_MAX* % *group_size*) + (1 % *group_size*) ) % *group_size*. Поскольку *group_size* >= 2 везде, где используется переменная *last_group_size*, то мы можем использовать упрощение (1 % *group_size*) = 1, тогда формула упрощается ещё сильнее и принимает вид ( (*UINT64_MAX* % *group_size*) + 1 ) % *group_size*. Поскольку *group_size* может принимать значения только от 1 до (*UINT32_MAX* + 1), то здесь уже больше не будет целочисленного переполнения.

Использование арифметики с плавающей запятой во второй формуле могло бы избавить нас от заботы о целочисленном переполнении: если мы конвертируем *UINT64_MAX* и/или 1 в тип с плавающей запятой, то сложение будет выполняться над числами в формате с плавающей запятой и уже не даст 0. С другой стороны, использование такой арифметики замедляет программу, делает её менее портабельной (реализация этой арифметики может быть весьма различной на различных устройствах, а где-то может не поддерживаться вовсе), а также делает её потенциально подверженной ошибкам округления и потери точности при вычислениях. По этим причинам было бы очень хорошо перевести вторую формулу в целые числа. Оказалось, что её целочисленный эквивалент выглядит как *group_num* = *UINT64_MAX* / *group_size* + 1 (деление нацело), что было вычислительно доказано путём написания и запуска специальной программы, перебиравшей все возможные значения выходного кодового пространства и размера группы в программе. Поскольку *group_size* >= 2 везде, где используется *group_num*, то целочисленного переполнения здесь не будет.

Математически это можно объяснить следующим образом: представим операцию *c* = округление_с_избытком(*a* / *b*), где *a* и *b* – целые числа, как следующую последовательность операций:

1. $c = a / b$, деление нацело;

2. если ($a$ % $b > 0$), то $c = c + 1$.

В нашем случае $a$ = *UINT64_MAX* + 1, $b$ = *group_size*, $c$ = *group_num*. Если мы сразу перенесём эту единицу в остаток от деления, то получим:

1. *group_num* = *UINT64_MAX* / *group_size*, деление нацело;

2. если ( (*UINT64_MAX* % *group_size* + 1) > 0), то *group_num* = group_num + 1.

Поскольку мы перенесли единицу в остаток от деления, то всегда будет выполняться условие (*UINT64_MAX* % *group_size* + 1) > 0, а следовательно, всегда нужно будет выполнить последнее действие *group_num* = *group_num* + 1. Получаем итоговую формулу *group_num* = *UINT64_MAX* / *group_size* + 1.

Таким образом, алгоритм был успешно применён к 6 из 8 целочисленных типов языка Си с использованием стандартных средств. Доказательством правильности алгоритма и его реализации служат пройденные тесты, а также собранная в ходе тестирования статистика.

### 3.5. Реализации для 64-битных целочисленных типов

Как уже упоминалось ранее, реализация обсуждаемого алгоритма для типов данных *uint64_t* и *int64_t* (восьмибайтовые целые числа без знака и со знаком соответственно) затруднена отсутствием в два раза более длинного целочисленного типа в текущем стандарте языка Си. Тем не менее, автор этой диссертации посчитал такую реализацию крайне желательной и осуществил её с использованием библиотеки длинной арифметики *GNU MP* [35]. Эта библиотека является широко распространённой, работает на большом количестве платформ и установлена во многих дистрибутивах *Linux* по умолчанию. Общая идея осталось прежней: кодировать числа длины N в числа длины 2*N.

Изменения коснулись в первую очередь кода самих алгоритмов, а тесты и сбор статистики по содержанию почти не отличались от тестов и сбора статистики для четырёхбайтовых целочисленных типов. Использование длинной арифметики заставило описывать алгоритмы более подробно: там, где раньше требовалась одна строка, теперь обычно требовалось уже несколько. Кроме того, для уменьшения накладных расходов пришлось задуматься над оптимизацией: как можно больше вычислений проводилось без перехода к длинной арифметике; значения вычислялись по возможности не всегда, а только в том случае, если они действительно требовались; число используемых переменных было минимизировано; параметры импорта и экспорта чисел были выбраны с целью достижения максимальной эффективности. В местах кода, где возможна более тонкая оптимизация, были оставлены соответствующие подробные комментарии.

Заметным отличием стало то, что закодированные данные сразу после применения арифметических операций кодирования экспортируются с помощью соответствующей функции библиотеки *GNU MP*. Соответственно, перед проведением арифметических операций декодирования данные импортируются с помощью соответствующей функции библиотеки *GNU MP*. Зачем нам потребовались эти дополнительные манипуляции? Дело в том, что числа, получаемые в результате применения арифметических операций кодирования, в этом случае хранятся не в обычном целочисленном типе, а в структурах типа *mpz_t* с несколькими полями. Поскольку мы не хотим хранить, анализировать, обрабатывать и шифровать служебные данные *GNU MP*, от них нужно избавиться. Выходным типом решено было сделать тип *unsigned char*, лучше всего подходящий для хранения произвольных бинарных данных; таким образом, каждое восьмибайтовое целое число кодировалось в массив из 16 байт (16 беззнаковых целых типа *char*). Благодаря такому подходу вся работа с библиотекой длинной арифметики

сосредоточена исключительно в модулях разрабатываемого проекта, и для прикладного программиста все операции происходят абсолютно прозрачно.

К сожалению, встроенные в *GNU MP* алгоритмы генерации псевдослучайных чисел не являются криптографически стойкими без сложной обработки в прикладной программе. По этой причине для генерации этих чисел использовался другой подход: байты, полученные с помощью системного криптографически стойкого генератора псевдослучайных чисел, записывались в буфер, откуда они уже импортировались в *GNU MP*.

Таким образом, в ходе работы над диссертацией алгоритм был успешно применён ко всем 8 целочисленным типам языка Си. Доказательством правильности алгоритма и его реализаций служат пройденные тесты, а также собранная в ходе тестирования статистика.

### 3.6. Замечание, касающееся защищённости

Как уже упоминалось выше, пространство закодированных данных явным образом влияет на правдоподобность результатов расшифрования неверным ключом. Чем больше оно будет, тем более равномерно (в рассматриваемом случае это эквивалентно фразе «тем более правдоподобно») будут распределены результаты расшифрования.

В подразделе 3.2 было приведено сравнение кодирования однобайтовых целых чисел в однобайтовые и двухбайтовые целые числа. Как мы выяснили, последняя неполная группа вызывает некоторое отклонение в распределении: несколько первых значений более вероятны, чем все остальные. Чем большим будет пространство закодированных данных, тем больше в нём поместится групп; чем больше в нём поместится групп, тем меньший эффект будет производить последняя неполная группа на общее распределение вероятностей; чем меньший эффект она будет производить,

тем более равномерным будет распределение; чем более равномерным оно будет, тем выше будет защищённость, ведь реальные данные также имеют равномерное распределение в рассматриваемой нами модели.

При написании этой магистерской диссертации автор сделал простой в реализации ход: кодировал N-байтовые целые числа в 2N-байтовые. Применительно к однобайтовым числам это дало максимальное отклонение в распределении вероятностей 0,0016% (см. подраздел 3.2). Применительно к двухбайтовым числам это отклонение ещё меньше, так как в пространство закодированных данных помещается значительно больше групп (наихудший случай – размер группы 65 535 элементов даёт минимум 65 538 групп против минимум 258 в предыдущем случае). Применительно к четырёхбайтовым числам это отклонение ещё меньше, чем в предыдущем случае, так как в пространство закодированных данных помещается ещё больше групп (наихудший случай – размер группы 4 294 967 295 элементов даёт минимум 4 294 967 298 групп против минимум 65 538 в предыдущем случае). Аналогично дело обстоит и применительно к восьмибайтовым числам: наихудший случай – размер группы 18 446 744 073 709 551 615 элементов даёт минимум 18 446 744 073 709 551 618 групп против минимум 4 294 967 298 в предыдущем случае.

Из этого ясно следует вывод: если пользователя не устраивает отклонение, предоставляемое этим проектом для однобайтовых типов данных, он может предварительно перекодировать свои данные в двух-/четырёх-/восьмибайтовые и применить к последним функции медового шифрования. Таким образом, защищённость данных будет повышена ценой увеличения ёмкости данных и замедления их обработки. То же самое касается двухбайтовых типов данных, которые можно без особых проблем перекодировать в четырёх-/восьмибайтовые и применить к последним функции медового шифрования; четырёхбайтовых типов данных, которые можно перекодировать в восьмибайтовые с той же целью.

Возникает, разумеется, и другой вопрос – нужна ли такая мощная защищённость ценой двукратного увеличения объёма данных? В случае с однобайтовыми типами такое увеличение видится более чем оправданным. В случае с другими целочисленными типами, очевидно, можно было бы обойтись и меньшим объёмом без заметных жертв для безопасности, хотя это и потребовало бы усложнения анализа, исходного кода и вычислений. Описанная выше простая реализация вполне может послужить в качестве основы для более эффективных работ, если в них возникнет необходимость.

# 4. ОБРАБОТКА ПРОИЗВОЛЬНО РАСПРЕДЕЛЁННЫХ ЦЕЛЫХ ЧИСЕЛ

## 4.1. Общий подход

Решив поставленную задачу для равномерно распределённых целых чисел, перейдём к более общей задаче медового шифрования произвольно распределённых целых чисел. Равномерное распределение нередко встречается на практике, но куда чаще мы имеем дело с неравномерным распределением. В теории вероятностей подробно рассмотрены такие понятия, как распределения Пуассона, Бернулли, биномиальное, геометрическое, гипергеометрическое, логарифмическое, отрицательное биномиальное. С практической точки зрения нам хотелось бы не привязываться к каким-то конкретным моделям, а получить универсальный инструментарий для медового шифрования целых чисел с любым законом распределения вообще.

У нас уже есть разработанные и протестированные алгоритмы и код для равномерно распределённых целых чисел, так что идея их повторного использования очень соблазнительна. Задачу медового шифрования произвольно распределённых целых чисел можно заменить на совокупность двух задач: 1) конверсия между целыми числами с произвольным распределением и целыми числами с равномерным распределением (пока не решена); 2) медовое шифрование равномерно распределённых целых чисел (уже решена). Как нам подступиться к первой задаче?

Рассмотрим сначала пример. Выше уже упоминалось, что расшифрование шифротекста неверным ключом даёт псевдослучайные байты вместо открытого текста. Вот что из этого следует: если мы договоримся, что значения байта от 0 до 100 (101 значение) хранят значение 0, а значения байта от 101 до 255 (155 значений) хранят значение 1, то при

расшифровании неверным ключом мы будем получать значение 0 с вероятностью 101/256 = 0.39453125, а значение 1 – с вероятностью 155/256 = 0.60546875. Другими словами, мы можем кодировать значения 0 и 1 в однобайтовые беззнаковые целые числа, причём контролируя вероятности значений 0 и 1 при расшифровании неверным ключом.

Изменяя объём долей кодового пространства, мы можем соответственно изменять и вероятности получаемых значений. Важно отметить, что при принятом кодировании мы можем выразить не любое соотношение вероятностей вообще, но лишь некоторое количество различных соотношений. Это является неизбежным следствием дискретности используемого нами кодового пространства. При переводе желаемого распределения в реальное нам скорее всего потребуется определённое округление. Например, мы не можем точно представить вероятности 0.5001 и 0.4999 для 0 и 1 соответственно. Если значениям 0 и 1 выделить по 128 чисел, то вероятностью каждого из этих значений будет 128/256 = 0.5; если значению 0 выделить 129 чисел, а значению 1 – 127 чисел, то их вероятностями будут 129/256 = 0.50390625 и 127/256 = 0.49609375; первое приближение более точно, так что лучше использовать именно его.

Проблема ещё сильнее проявляет себя в примере, когда мы хотим представить вероятности 0.9999 и 0.0001. Если значению 0 выделить 255 чисел, а значению 1 – 1 число, то их вероятностями будут 255/256 = 0.99609375 и 1/256 = 0.00390625; если значению 0 выделить 256 чисел, а значению 1 – 0 чисел, то их вероятностями будут 256/256 = 1 и 0/256 = 0. Второе приближение более точно, но мы не можем его использовать! Да, значение 1 весьма маловероятно, но мы не можем сказать, что оно невозможно – в этом случае кодирование такого маловероятного значения оказывается попросту невыполнимым. Увы, нам придётся использовать менее точное первое приближение.

Итак, шаг изменения представимых вероятностей равен 1/256 = 0.00390625, этому же значению равна минимально представимая вероятность, которую придётся использовать вместо всех ненулевых изначальных вероятностей меньше минимально представимой. Разумеется, если мы перейдём к кодированию значений в числовые типы с большим кодовым пространством, то проблемы заметно смягчатся, но тем не менее никуда не уйдут. Скажем, если в качестве контейнера будут использоваться двухбайтовые беззнаковые целые числа, то шаг изменения представимых вероятностей и минимально представимая вероятность станут равны 1/65536 = 0.00001525878, в то время как в идеальном случае оба этих значения должны быть бесконечно малы.

Подход, подобный рассмотренному выше, может быть с успехом применён и для более общего случая. Ещё раз напомним, что у нас уже есть функции медового шифрования равномерно распределённых целых чисел, так что наша задача – перейти от произвольно распределённых целых чисел к равномерно распределённым целым числам.

Пусть у нас есть такое изначальное распределение: возможными значениями являются 10 и 12, причём значение 10 является в 1.5 раза более вероятным, чем значение 12. Минимальным среди всех возможных значений является 10, максимальным – 12; получается, что мы имеем дело с произвольным распределением на интервале [10; 12], так как значения за его пределами всё равно невозможны. Мы можем присвоить каждому из значений на интервале [10; 12] числовой параметр, называемый далее весом. Вес значения равен количеству чисел в кодовом пространстве, которые будут выделены для его кодирования. Для получения желаемых вероятностей вес значения 10 будет равен 3, значения 11 – 0, значения 12 – 2. Пусть в соответствии с весами числа от 0 до 2 хранят значение 10, а числа от 3 до 4 – значение 12. Получим, что возможны лишь значения 10 и 12, причём первое из них будет в 1.5 раза более вероятным, чем второе. Для обеспечения

максимальной правдоподобности всех результатов расшифрования выбор конкретного значения внутри доли при кодировании должен осуществляться псевдослучайно, в противном случае результаты расшифрования верным ключом будут заметно отличаться от результатов расшифрования неверным ключом. Результаты такого кодирования передадим функции медового зашифрования равномерно распределённых целых чисел.

Как будет происходить обратный процесс? Сначала применим функцию медового расшифрования равномерно распределённых целых чисел. Если результатом расшифрования является число от 0 до 2, то переводим его в значение 10, а если число от 3 до 4 – в значение 12. *Voila*, мы сконвертировали произвольно распределённые целые числа в равномерно распределённые целые числа, а затем осуществили обратную конверсию!

### 4.2. Вопросы практической реализации

Поскольку мы занимаемся не только теоретическим рассмотрением, но и практической реализацией решения, стоит задуматься и о более мелких деталях, непосредственно связанных с программированием.

Все данные будут храниться в числах фиксированной разрядности, что налагает специфические ограничения. Для упрощения анализа и программирования будет логично использовать беззнаковые типы в качестве контейнера, как и прежде. Поскольку в этом случае кодовое пространство представляет собой числа от 0 до некоторого максимального значения, то для максимально эффективного использования этого пространства логично будет начинать кодирование данных с 0. Действительно, если мы начнём кодировать данные, например, с 100, то числа от 0 до 99 будут либо просто неиспользованными, либо для их использования потребуется

дополнительное усложнение алгоритма; если мы начнём с 0, то мы избежим и того, и другого.

Кстати, в примере выше вместо весов 3, 0, 2 мы могли бы использовать бесконечное количество других комбинаций, получающихся из исходных путём умножения каждого члена на фиксированное положительное целое число – например, 6, 0, 4; 9, 0, 6 и так далее. С точки зрения практической реализации нам выгодно использовать комбинацию с наименьшими весами, поскольку в этом случае использование выходного кодового пространства будет минимальным. По той же причине в мы не рассматривали более широкие исходные интервалы, например, [9; 12], где значения 9 и 12 являются невозможными – это было бы попросту бесполезной тратой ресурсов.

Более тонким вопросом является выбор конкретного беззнакового типа для хранения результатов кодировки. Поскольку из-за наличия весов разброс использования кодового пространства крайне велик, а хранить бесполезные значения в памяти и замедлять вычисления без особой на то нужды у нас нет желания, то пусть наше решение будет гибким и использует минимально возможный беззнаковый тип в качестве выходного (назовём его типом-контейнером). Увы, но за такой выбор придётся платить усложнением программирования как при разработке библиотеки, так и при её прикладном использовании: потребуется функция для вычисления минимально возможного беззнакового типа по весам, а также возможность кодирования, например, однобайтовых беззнаковых чисел в однобайтовые, двухбайтовые, четырёхбайтовые и восьмибайтовые беззнаковые целые.

Для упрощения кодирования и декодирования удобно перейти от ранее упомянутых весов к другому понятию – кумулятивным весам (название выбрано по аналогии с функцией распределения из теории вероятностей, называемой иногда кумулятивной). Если вес значения равен количеству

чисел в кодовом пространстве, которые будут выделены для его кодирования, то кумулятивный вес равен количеству чисел в кодовом пространстве, которые будут выделены для кодирования его и всех предыдущих значений. Вернёмся к предыдущему примеру: если веса равны комбинации 3, 0, 2, то соответствующие им кумулятивные веса равны 3, 3 (получается как 3+0), 5 (3+0+2). Кстати, последний член массива кумулятивных весов позволяет нам определить общий объём используемого кодового пространства, что удобно использовать для определения выходного типа.

Для упрощения программирования при разработке библиотеки было решено объединить некоторые проверки входных данных, переход от весов к кумулятивным весам и определение выходного типа в один макрос. Два других макроса предназначены непосредственно для кодирования и декодирования. Первый содержит перевод целочисленного массива с произвольным распределением в целочисленный массив с равномерным распределением определённого типа, а также применение к последнему соответствующей функции медового зашифрования равномерно распределённых целых чисел. Второй макрос, соответственно, содержит обратные преобразования: применение функции медового расшифрования равномерно распределённых целых чисел, а также перевод целочисленного массива с равномерным распределением в целочисленный массив с произвольным распределением. Для обеспечения максимального удобства пользователя все операции, включая выделение памяти, осуществляются внутри библиотеки; ему остаётся лишь очистить память после того, как данные перестанут быть ему нужны.

Тестовые программы для различных типов кодируют и декодируют массивы числовых данных с определёнными весами, проверяя побайтовое соответствие исходных и результирующих массивов. Веса в тестах меняются таким образом, чтобы охватить все допустимые типы-контейнеры, а также

случай, когда кодового пространства даже самого большого типа-контейнера будет недостаточно.

# 5. ОБРАБОТКА РАВНОМЕРНО РАСПРЕДЕЛЁННЫХ ЧИСЕЛ С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

## 5.1. Формат чисел с плавающей запятой

Прежде чем говорить об обработке чисел с плавающей запятой, нужно рассмотреть особенности таких чисел.

В соответствии со стандартом [23] числа с плавающей запятой типов *binary32* (обычно соответствует типу *float* языка Си) и *binary64* (обычно соответствует типу *double* языка Си) состоят из трёх частей: знакового бита, битов порядка и битов мантиссы. Если знаковый бит равен 0, то число положительно; если 1, то отрицательно.

Если все биты порядка это нули, то:

– если все биты мантиссы это нули, то в зависимости от знака число представляет положительный или отрицательный ноль (+0 или -0);

– если не все биты мантиссы это нули, то это денормализованное число, значение которого вычисляется как ( (–1) ^ знак ) * (2 ^ минимально возможный порядок) * 0.мантисса.

Если все биты порядка это единицы, то:

– если все биты мантиссы это нули, то в зависимости от знака число представляет положительную или отрицательную бесконечность (+*inf* или -*inf*);

– если не все биты мантиссы это нули, то это значение типа *NaN* («*not a number*», «не число»), в старшем бите мантиссы которого указано, является ли это значение сигнальным *NaN* или тихим *NaN*, в остальных битах мантиссы может передаваться полезная нагрузка. Знак может учитываться или не учитываться в зависимости от реализации. [24]

Если же биты порядка отличны от случаев «все нули» и «все единицы», то мы имеем дело с нормализованным числом, значение которого вычисляется как ( (–1) ^ знак ) * ( 2 ^ (порядок – минимально возможный порядок – 1) ) * 1.мантисса. [25]

## 5.2. Наивное отображение чисел с плавающей запятой на целые числа

Итак, вот что мы имеем на текущий момент:

1) Мы поставили себе цель реализации функций медового шифрования чисел с плавающей запятой, поскольку многие реальные данные хранятся именно в этом формате.

2) В предыдущем подразделе было показано, что формат чисел с плавающей запятой является достаточно сложным.

3) С другой стороны, раздел «Предыдущие работы» этой магистерской диссертации ясно показывает, что медовое шифрование можно успешно применить и к более сложным типам данных.

Как же нам поступить, как найти компромисс в этом противоречивом положении? Очевидно, что реализация «в лоб» медового шифрования для таких чисел будет сложной в теоретическом рассмотрении, разработке, отладке и поддержке. Есть ли какой-нибудь обходной путь?

Мы можем рассмотреть перевод чисел с плавающей запятой в целые числа и обратно. Такой перевод будет очень полезен, поскольку в нашем распоряжении уже есть написанные и отлаженные примитивы медового шифрования для равномерно и произвольно распределённых целых чисел, а разработка с нуля эквивалентных по функциональности примитивов медового шифрования для чисел с плавающей запятой требует больших затрат. При нашем подходе для зашифрования чисел с плавающей запятой

мы переведём их в целые числа, а затем применим к последним медовое зашифрование целых чисел; для получения исходных значений сначала применим медовое расшифрование целых чисел, а затем переведём результирующие целые числа в исходные числа с плавающей запятой.

Все эти размышления приводят к выводу, что конверсия должна осуществляться не произвольным, а каким-то рационально обоснованным образом. Очевидно, для хранения конвертированных чисел будет удобно использовать беззнаковые целые, как и прежде. Каким должен быть размер этих целых? Минимально достаточным для того, чтобы сохранить содержимое числа с плавающей запятой (к счастью, тип *float* языка Си обычно имеет тот же размер, что и тип *uint32_t* – 4 байта; размер типа *double* обычно равен размеру типа *uint64_t* – 8 байт). Если у нас есть значения с плавающей запятой *fp1* и *fp2*, то после перевода их соответственно в значения *int1* и *int2* должны выполняться следующие три импликации:

– если $fp1 > fp2$, то $int1 > int2$;

– если $fp1 < fp2$, то $int1 < int2$;

– если $fp1 = fp2$, то $int1 = int2$.

Для выполнения этих условий было решено переводить каждое число с плавающей запятой в его номер в последовательности всех возможных чисел с плавающей запятой этого типа. Таким образом, отрицательное число с плавающей запятой, имеющее максимальный порядок и максимальную мантиссу, будет представлено нулём; положительное число с плавающей запятой, имеющее максимальный порядок и максимальную мантиссу, будет представлено максимальным значением соответствующего целочисленного типа. Поскольку обычно размеры типов *float* и *uint32_t*, *double* и *uint64_t* соответственно равны между собой, то получается, что мы используем всё кодовое пространство целочисленных типов, исключая нежелательные для свойства медовости невозможные значения.

Доказательство того, что перевод каждого числа с плавающей запятой в его номер в последовательности всех возможных чисел с плавающей запятой этого типа при некоторых условиях влечёт за собой выполнение трёх импликаций, описанных выше, вынесено в приложение В. Важное замечание: в доказательстве эти импликации выполняются, если мы не учитываем знаки чисел и мы соглашаемся с тем, что после положительной бесконечности идут упорядоченные между собой сигнальные *NaN*-значения, а после них – упорядоченные между собой тихие *NaN*-значения.

Мы можем смириться с условием, касающимся редко используемых на практике *NaN*-значений, однако знаки чисел просто необходимо учесть. Напомним, если знаковый бит числа с плавающей запятой равен 0, то оно положительно, иначе отрицательно. Использование кодового пространства чисел с плавающей запятой представлено в верхней части рис. 5.1.

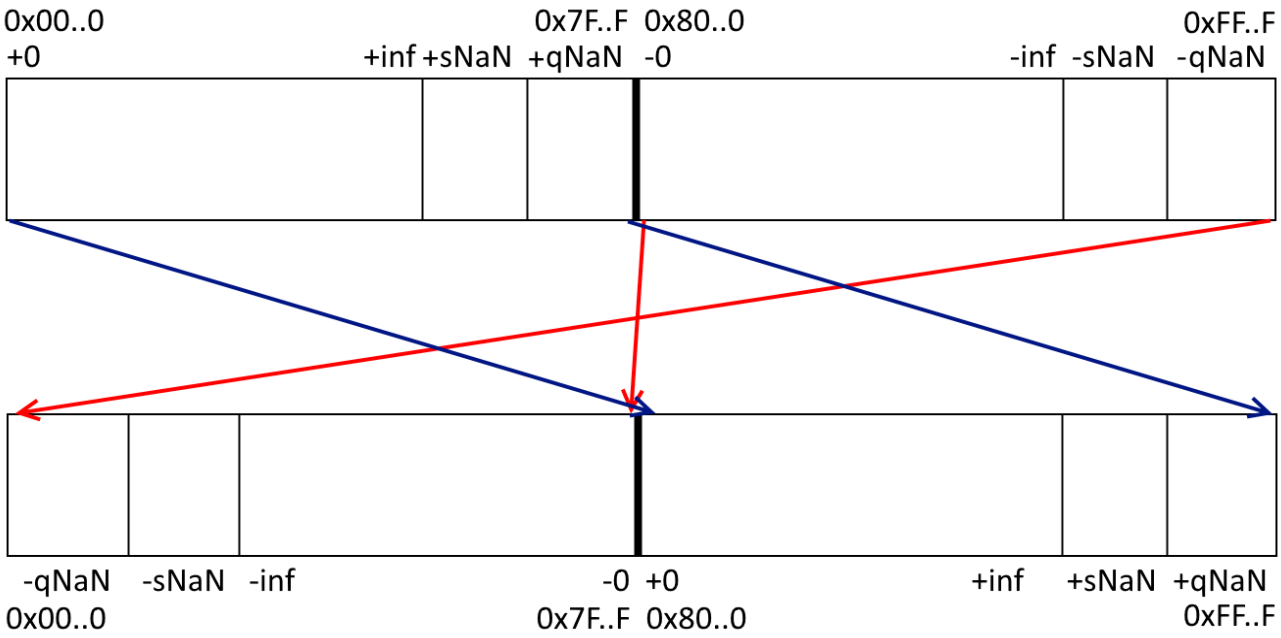


Рис. 5.1. Перевод чисел с плавающей запятой в целые числа

Итак, как же нам учесть знаки чисел с плавающей запятой, чтобы добиться выполнения трёх импликаций с учётом знаков чисел? Все необходимые для этого преобразования описаны ниже и наглядно показаны на рис. 5.1: в верхней его части показано кодовое пространство чисел с плавающей запятой, в нижней – пространство целых чисел того же размера, хранящих номера чисел с плавающей запятой в последовательности всех чисел с плавающей запятой этого типа.

Поскольку положительные числа уже отсортированы в нужном порядке, то их нужно просто переместить из первой половины кодового пространства во вторую для получения их номера: если *ielt* < 0x80..0, то *oelt* = *ielt* + 0x80..0. Для обратного преобразования перемещаем их обратно из второй половины в первую: если *oelt* >= 0x80..0, то *ielt* = *oelt* – 0x80..0. Это показано синими стрелками на рис. 5.1.

Отрицательные числа отсортированы в порядке, обратном нужному, поэтому их нужно пересортировать; кроме того, для получения их номеров их нужно переместить из второй половины кодового пространства в первую. Вот как это можно сделать: если *ielt* >= 0x80..0, то *oelt* = 0xFF..F – *ielt* = ~*ielt*, где символ «~» означает побитовую инверсию. Для обратного преобразования перемещаем их обратно из первой половины во вторую, попутно пересортировав их в обратном порядке: если *oelt* < 0x80..0, то *ielt* = 0xFF..F – *oelt* = ~*oelt*. Это показано красными стрелками на рис. 5.1.

Что нам даёт такое кодирование? К сожалению, совсем немного. Дело в том, что числа с плавающей запятой имеют довольно сложный закон распределения. Они распределены равномерно внутри интервалов, имеющих одинаковый порядок. Более конкретно, они распределены равномерно внутри интервалов, входящих в:

1) любой из интервалов вида [-2^(e+1); -2^e] или [2^e; 2^(e+1)], где e = (порядок + минимально возможный порядок – 1), порядок имеет допустимое

значение для этого типа, но не состоит в двоичном представлении из одних нулей (денормализованные числа, положительный и отрицательный нули) или из одних единиц (*NaN*-значения, положительная и отрицательная бесконечности);

2) интервал [-2^(e+1); -0] или интервал [+0; 2^(e+1)], где e – минимально возможный порядок;

3) интервал, содержащий все отрицательные сигнальные *NaN*-значения;

4) интервал, содержащий все положительные сигнальные *NaN*-значения;

5) интервал, содержащий все отрицательные тихие *NaN*-значения;

6) интервал, содержащий все положительные тихие *NaN*-значения.

При этом перечисленные выше интервалы в общем случае имеют различную длину (рис. 5.2), в чём и заключается главная проблема для нас.

Получается, что после применения алгоритма конверсии мы можем эффективно использовать примитивы медового шифрования равномерно распределённых целых чисел только в том случае, если исходные равномерно распределённые числа с плавающей запятой целиком лежат в одном из интервалов, перечисленных выше! Теоретически мы могли бы использовать более широкие интервалы, вплоть до [-*inf*; +*inf*], если бы исходные числа с плавающей запятой были бы распределены равномерно по множеству представимых чисел с плавающей запятой, но этот случай представляется автору крайне маловероятным. Вывод таков: для решения задачи необходимо разработать более сложный и практичный алгоритм.

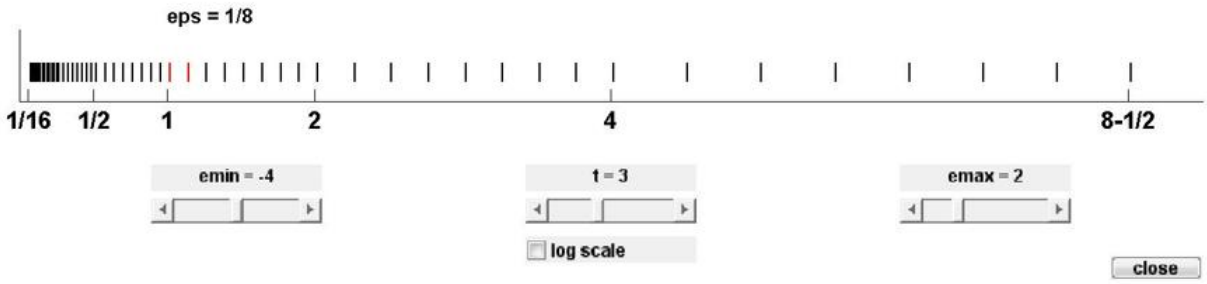


Рис. 5.2. Распределение чисел с плавающей запятой [43]

## 5.3. Наброски усложнённого алгоритма

Для большей наглядности дальнейших рассуждений перейдём от рассмотрения конкретных типов с плавающей запятой к упрощённому примеру. Пусть у нас имеется некоторый тип с плавающей запятой, который может хранить следующие значения:

– тихие *NaN*-значения с полезной нагрузкой -3, -2, -1, -0;

– сигнальные *NaN*-значения с полезной нагрузкой -3, -2, -1;

– отрицательная бесконечность -*inf*;

– нормализованные числа -7, -6, -5, -4, -3.5, -3, -2.5, -2, -1.75, -1.5, -1.25, -1;

– денормализованные числа -0.75, -0.5, -0.25;

– отрицательный и положительный нули -0, +0;

– денормализованные числа 0.25, 0.5, 0.75;

– нормализованные числа 1, 1.25, 1.5, 1.75, 2, 2.5, 3, 3.5, 4, 5, 6, 7;

– положительная бесконечность +*inf*;

– сигнальные *NaN*-значения с полезной нагрузкой 1, 2, 3;

– тихие *NaN*-значения с полезной нагрузкой +0, 1, 2, 3.

Подобным образом распределены и допустимые значения реальных типов с плавающей запятой *float* и *double*, что и делает такое упрощение достаточно реалистичным для нас.

Что мы можем увидеть на этом примере? В множестве представимых нормализованных чисел можно выделить интервалы, на которых шаг между двумя соседними значениями остаётся постоянным: [-7; -4] (шаг 1), [-4; -2] (шаг 0.5), [-2; -1] (шаг 0.25), [1; 2] (шаг 0.25), [2; 4] (шаг 0.5), [4; 7] (шаг 1). То же самое касается и денормализованных чисел, в которых можно выделить следующие интервалы: [-0.75; -0] (шаг 0.25) и [+0; 0.75] (шаг 0.25). Обратите внимание, что шаг в денормализованных интервалах равен наименьшему шагу, встречающемуся в нормализованных числах: в общем и целом шаг остаётся постоянным на интервалах [-7; -4] (шаг 1), [-4; -2] (шаг 0.5), [-2; -0] (шаг 0.25), [+0; 2] (шаг 0.25), [2; 4] (шаг 0.5), [4; 7] (шаг 1). Постоянным является и шаг полезной нагрузки *NaN*-значений: он равен 1 на всём множестве возможных значений полезной нагрузки, то есть на интервалах [-3; -1] и [1; 3] для сигнальных *NaN*-значений, [-3; -0] и [+0; 3] для тихих.

Как уже отмечалось ранее, на интервалах с равным шагом распределение значений будет фактически равномерным, так что описанное выше отображение чисел с плавающей запятой на целые числа будет отлично работать. Округление будет вносить небольшое отклонение от по-настоящему равномерного распределения, но его значением можно пренебречь. Более конкретно, для двоичных чисел с плавающей запятой, которые мы рассматриваем, вариантом округления по умолчанию в стандарте *IEEE 754* является так называемое «банковское» [23]. Оно подразумевает, что если число находится между двумя представимыми значениями, то оно округляется к тому из них, которое является чётным. Простой и чисто иллюстративный пример такого округления для десятичных чисел: значения +23.5 и +24.5 округляются до +24, в значения −23.5 и −24.5 округляются до −24 [44].

А что, если мы хотим отобразить массив чисел с плавающей запятой из другого интервала, например, [1.5; 2.5]? Представимыми значениями на этом интервале являются числа 1.5, 1.75, 2, 2.5. Если мы закодируем их по наивному алгоритму выше, то эти четыре значения будут встречаться с одинаковой вероятностью. Это звучит парадоксально, но такое распределение не будет соответствовать равномерному распределению чисел на интервале [1.5; 2.5]! Легко увидеть, что если бы точность чисел с плавающей запятой была бы равной шагу 0.25, наименьшему на этом интервале, то следующие 5 значений встречались бы с одинаковой вероятностью: 1.5, 1.75, 2, 2.25, 2.5. Проблема заключается в том, что значение 2.25 не представимо в явном виде, так что примерно половина чисел, которые могли быть округлены в это значение, будет округлены в 2, а ещё примерно половина – в 2.5. Слово «примерно» в предыдущем предложении вызвано тем, что точное значение 2.25 будет всегда округляться в 2 в соответствии с банковским округлением, что вызовет небольшую неравномерность в распределении. Таким образом, каждое из чисел 1.5 и 1.75 встречается с вероятностью $x$, каждое из чисел 2 и 2.5 – с вероятностью $1.5x$.

А как насчёт отображения чисел из интервала [1.5; 3]? Непредставимыми здесь будут значения 2.25 и 2.75, вероятности которых поделятся примерно пополам между представимыми значениями 2 и 2.5, 2.5 и 3 соответственно. Таким образом, каждое из чисел 1.5 и 1.75 встречается с вероятностью $x$, каждое из чисел 2 и 3 – с вероятностью $1.5x$, число 2.5 – с вероятностью $2x$.

С расширением интервала всё становится ещё сложнее. На интервале [1.5; 5] непредставимыми являются значения 2.25, 2.75, 3.25, 3.75, 4.25, 4.5, 4.75. С распределением вероятностей первых четырёх значений ситуация будет такой же, как и раньше. Все значения, которые могли бы округляться в 4.25 при точности чисел с плавающей запятой, равной нашему шагу 0.25,

теперь будут округляться в 4; все значения, которые могли бы округляться в 4.75, теперь будут округляться в 5. Вероятность числа 4.5 поделится примерно пополам между представимыми значениями 4 и 5. Итого: вероятность каждого из чисел 1.5 и 1.75 – $x$, числа 2 – $1.5x$ (вероятность значения 2, половина вероятности значения 2.25), каждого из чисел 2.5, 3, 3.5 – $2x$, числа 4 – $3x$ (вероятность значений 4 и 4.25, половины вероятностей значений 3.75 и 4.5), числа 5 – $2.5x$ (вероятность значений 5 и 4.75, половина вероятности значения 4.5).

Отдельно стоит рассмотреть ситуацию с положительным и отрицательным нулями. Для обеспечения равномерности распределения после перевода в целые числа следует поступить так: если интервал содержит только отрицательный или только положительный ноль, то этому значению следует присвоить ту же вероятность, что и любому денормализованному числу; если же интервал содержит оба нуля, то каждому из них следует присвоить вероятность, равную половине вероятности денормализованного числа. Такой подход имеет следующее обоснование. Шаг между двумя соседними денормализованными числами одного знака всегда остаётся постоянным (см. формулу в подразделе 5.1). Если мы попробуем найти число, которое будет на расстояние шага ближе к нулю, чем наибольшее представимое отрицательное денормализованное число или наименьшее представимое положительное денормализованное число, то мы в обоих случаях получим ноль. Кроме того, в стандарте *IEEE 754* сказано, что численное сравнение отрицательного и положительного нулей должно выдавать результат «числа равны» [45], а для большинства задач можно считать два этих значения равными друг другу. Таким образом, если внутри интервала содержатся два разных нуля, то будем считать их просто двумя разными кодовыми значениями, представляющих одно и то же число. Каждому *NaN*-значению (если такие значения входят в интервал) можно присвоить минимальную представимую вероятность, так как такие значения

встречаются сравнительно редко. Обрабатывать *NaN*-значения нужно очень аккуратно, поскольку результатом любых сравнений с ними является «ложь» [24].

А как нам поступить с положительной и отрицательной бесконечностями? Логика авторов стандарта *IEEE* вполне понятна при рассмотрении нашего простого типа: вообще-то следующим представимым числом после 7 должно было стать число 8, но поскольку мы не можем представить в нашем типе все последующие числа, то давайте объединим в понятии «положительная бесконечность» число 8 и все последующие. Отсюда вытекает прагматичное решение: в наших расчётах мы можем принять, что никаких бесконечностей нет, есть лишь заменяющие их представимые значения (-8 вместо *-inf* и 8 вместо *+inf*). Это позволит нам избежать учёта этой ситуации как особой при программировании, поскольку тогда случай интервала, например, [6; +inf] концептуально не отличается от случая интервала [1; 2]. Насколько обоснованным будет такое допущение? Значения *-inf* и *+inf* должны сравнительно редко встречаться на практике, что несколько смягчает остроту вопроса. Тем не менее, если такое значение появляется где-то в исходных, промежуточных или результирующих данных, то оно с большой вероятностью может появиться и где-то ещё в этом же наборе данных. Это вызвано тем, что значительное число математических операций с бесконечностями (скажем, сложение *+inf* и 1) приведёт к бесконечностям. Другими словами, это редкие значения, но если они всё же появляются, то целыми группами. В таких условиях наше допущение можно назвать достаточно правдоподобным, поскольку предыдущие рассуждения показывают – вес бесконечностей при таком рассмотрении окажется сравнительно крупным, но не чрезмерно. Если же бесконечности вообще не фигурировали в допустимых значениях исходных данных, их не будет и при расшифровании неверным ключом. Словом, такой подход к проблеме видится вполне удовлетворительным. Возможно, что такой же логикой

стоило бы руководствоваться и по отношению к NaN-значениям, однако это тонкий вопрос, требующий дополнительного исследования.

Отобразить описанные выше различия вероятностей в кодовом пространстве будет несложно, поскольку в разделе 4 этой магистерской диссертации была решена ровно эта задача: кодирование произвольно распределённых целых чисел при наличии их весов. С другой стороны, выше было показано, что нахождение нужных весов для конкретных значений является весьма непростой задачей. Очевидно, что задача медового шифрования произвольно распределённых чисел с плавающей запятой является обобщением только что рассмотренной и вполне сопоставима с последней по сложности. Автор этой магистерской диссертации оставляет эти вопросы открытыми для будущих исследований.

### 5.4. Программная реализация наивного алгоритма

Несмотря на огромные ограничения наивного алгоритма, автор диссертации всё же осуществил его программную реализацию. Проделанная по этому направлению работа может быть весьма полезна при реализации усложнённого алгоритма.

К сожалению, представление чисел с плавающей запятой является платформозависимым. По этой причине автор диссертации при написании кода ориентировался на свой компьютер, написал тест на совместимость с обработкой чисел с плавающей запятой в проекте, а также старался комментировать все платформозависимые участки кода для упрощения портирования. При возникновении каких-либо проблем совместимости их решение не должно стать трудной задачей.

Во время тестирования на компьютере автора диссертации обнаружилось неожиданное свойство алгоритма, вызванное низкоуровневой

обработкой чисел с плавающей запятой. Как оказалось, практически при любой обработке сигнальных *NaN*-значений они автоматически переводились в тихие *NaN*-значения. Было решено поступить следующим образом:

– принять условие, что алгоритмы кодирования и декодирования считают сигнальные *NaN*-значения эквивалентными тихим *NaN*-значениям;

– намеренно конвертировать все имеющиеся во входных данных *NaN*-значения в тихие для упрощения тестирования, для чего были написаны специальные функции;

– для обеспечения свойства медовости, то есть максимальной правдоподобности результатов любой попытки расшифрования, при кодировании *NaN*-значений устанавливать сигнальный бит псевдослучайным образом, при декодировании – устанавливать бит в единицу, что на компьютере автора диссертации соответствует тихим *NaN*-значениям.

Этот алгоритм был реализован для типов *float* и *double*, после чего была составлены две тестовые программы, выполняющие следующее:

1) псевдослучайно сгенерированные данные кодируются в целые и обратно, после чего проверяется их побитовое соответствие начальным данным;

2) 10 первых псевдослучайно сгенерированных чисел сравниваются между собой только на основе их сконвертированных значений, при этом на экран выводятся их значения до конвертации, чтобы пользователь мог самостоятельно убедиться в справедливости трёх импликаций, упомянутых выше;

3) проверяется работа функций в случае неправильных данных на входах процедур (нулевые указатели вместо действительных, нулевые длины массивов).

В конце стоит заметить, что этот алгоритм не был реализован для других типов чисел с плавающей запятой (например, [38-42]). Это было сделано по нескольким причинам:

1) они используются куда реже ранее рассмотренных выше *float* и *double* по причине ограниченной полезности этих других типов;

2) они гораздо хуже стандартизованы, их реализация может отличаться от стандартов *IEEE* [23];

3) они могут быть очень тесно привязаны к конкретным программно-аппаратным платформам, которых может не быть в наличии у автора диссертации (например, формат [42] почти не реализован аппаратно), или работа с которыми заметно усложнила бы проект (например, форматы [38-39] в основном реализованы на видеокартах, которые требуют особого программирования).

4) стандарт языка Си не описывает типы целых чисел длиной более 8 байт, которые могли бы пригодиться нам для кодирования некоторых чисел в формате с плавающей запятой.

Если пользователю этого проекта понадобятся функции для работы с этими более экзотическими типами данных, то он наверняка сможет реализовать их самостоятельно на основе ранее разработанных нами алгоритмов и исходных кодов.

# 6. ОБЩИЕ ВОПРОСЫ

## 6.1. Статистические функции

Для анализа разработанных в магистерской диссертации функций потребовалось применение средств математической статистики. Предположим, что при расшифровании данных неверным ключом мы хотели добиться равномерного распределения результатов на некотором интервале. Разумеется, можно сделать некоторые выводы на основании визуального анализа графиков, но автоматизация процесса анализа заметно упростит работу и повысит достоверность выводов. Как же решить эту задачу?

В этой работе мы имеем дело с задачей проверки статистических гипотез: у нас есть некоторые экспериментальные данные, и нужно сказать, насколько они похожи на те, которые мы хотели бы получить в идеале. Для решения такой задачи специально предназначен критерий хи-квадрат [46]. В такие популярные программы для работы с таблицами, как *Microsoft Excel*, *OpenOffice Calc* и *LibreOffice Calc*, встроена функция этого семейства, реализующая критерий согласия Пирсона [47]. К сожалению, эта функция в ряде случаев даёт недостаточно точные результаты [46]. Тем не менее, в начале работы для анализа использовалась именно она, поскольку она была знакома автору диссертации и её было легко применить к данным. На вход этой функции подаются два вектора с реальными и теоретическими данными, а на выходе она выдаёт вероятность того, что реальные данные соответствуют теоретическим, то есть реальные и теоретические данные имеют один и тот же закон распределения.

К сожалению, боевое применение критерия согласия Пирсона дало скромные результаты. Несмотря на вполне приличное количество данных для анализа (статистика собиралась по десяткам тысяч чисел), функция

выдавала совсем разные вероятности при повторении одних и тех же тестов с разными псевдослучайно сгенерированными исходными данными. В худшем случае результатом были числа порядка 0.05, в лучшем – порядка 0.95, обычно же результаты находились в пределах от 0.2 до 0.8, то есть разброс значений был колоссальным. Похожие результаты получались и при анализе псевдослучайных данных, что подтверждало: проблема не в наших данных, а в алгоритме статистической оценки. В сухом остатке получалось, что если значения вероятности почти в каждом опыте превышают 0.2, то можно говорить о наличии устойчивого сходства между теоретическим и практическим распределениями, в противном случае такого сходства нет.

Автор вынужден признать, что использованный им математический аппарат для статистической оценки является лишь удовлетворительным, но не максимально подходящим к ситуации, и оставляет этот вопрос открытым для будущих исследований.

### 6.2. Детали реализации

При написании программного кода активно использовались подходы, близкие к парадигме обобщённого программирования [22]. Для реализации многих функций со сходными алгоритмами, но разными мелкими деталями применялись макросы с параметрами. Это позволило свети количество повторяющегося кода к минимуму, а соответственно сильно упростить изменение и дополнение исходного кода разработанных программных модулей. Увы, за это пришлось заплатить некоторым усложнением изучения исходного кода и заметным усложнением отладки.

Тем не менее, крайности вредны в любом деле. При реализации алгоритма кодирования равномерно распределённых целых чисел можно было бы для всех типов чисел использовать библиотеку длинной

арифметики, однако результат получился бы очень неэффективным, так что такой подход не был применён. Можно было бы не предоставлять конечным пользователям функций для медового шифрования равномерно распределённых чисел, пользуясь простой логикой «равномерное распределение – это частный случай произвольного», но это опять же было бы неэффективно и просто очень неудобно для реального применения. Макросы с параметрами могли бы сильно сократить объём исходного кода тестов, но это привело бы к заметному усложнению исходного кода при крайне низкой практической пользе от такого обобщения.

Новый функционал в этом проекте всегда рассматривался как надстройка над старым. Это касается как теоретических рассуждений, так и практической реализации в алгоритмах и функциях. Такой подход от простого к сложному, снизу вверх имеет несомненные преимущества. Во-первых, и теория, и код делаются максимально понятными для читателя, в них проще вносить изменения. Во-вторых, новые разработанные функции фактически дополнительно тестируют старые путём использования последних; тесты для новых функций в косвенной мере являются также тестами для старых функций. Всё это увеличивает надёжность функций путём расширения тестового покрытия кода. В-третьих, практическое использование разработанных функций предоставляет дополнительные примеры обращения с этими функциями для других программистов. Это и есть повторное использование кода в действии, крайне желательное свойство для любого программного продукта.

Данные в проекте обрабатываются не по одному элементу за вызов функции, а целыми массивами сходных данных. Учитывая тот факт, что реальные данные часто хранятся в массивах, такой подход призван максимизировать производительность программных модулей проекта и удобство работы с этими модулями. Данные могут обрабатываться и поэлементно, для этого нужно просто указать размер массива, равный 1.

Исходные коды описываемого в диссертации проекта опубликованы на хостинге *GitHub* [21] под лицензией *BSD* из 2-х пунктов [18]. Открытая публикация проекта в виде исходных кодов важна для того, чтобы другие разработчики могли как-либо использовать её в составе своих проектов. Именно лицензия *BSD* из 2-х пунктов была выбрана среди множества лицензий для проектов с открытым исходным кодом за её понятность и за возможность использования кода в проприетарном программном обеспечении, что представляется автору очень важным для подобных проектов. Кроме того, исходные коды представлены в приложении Б.

Для того, чтобы исходными кодами проекта могли пользоваться люди по всему миру, все файлы с исходными кодами (включая комментарии в них) были написаны с ориентацией на читателя, владеющего английским языком.

# ЗАКЛЮЧЕНИЕ

В магистерской диссертации была рассмотрена реализация медового шифрования применительно к различным типам числовых данных (то есть хранимых в различных типах языка Си и имеющих различные законы распределения). Более конкретно, поддерживаются целочисленные типы uint8_t, int8_t, uint16_t, int16_t, uint32_t, int32_t, uint64_t, int64_t с равномерным и произвольным законами распределения, числовые типы в формате с плавающей запятой float и double с равномерным законом распределения. Применение программных модулей, разработанных на основе описанных в диссертации алгоритмов, позволяет заметно усилить защищённость рассмотренных типов числовых данных. Более того, на основе разработанных и протестированных функций можно без особого труда реализовать аналогичные функции для других типов числовых данных. В пояснительной записке к работе были рассмотрены теоретические и практические аспекты обсуждаемых вопросов, касающихся безопасности, производительности, надёжности, удобства использования и изменения. К сожалению, работе откровенно недостаёт принятых в криптографии формальных доказательств безопасности, и этот вопрос остаётся открытым для будущих исследований.

Автор диссертации приглашает других специалистов проанализировать и улучшить его решения. Эта работа может быть полезна для развития новаторской концепции медового шифрования и криптографии в целом.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. A. Juels, T. Ristenpart. Honey Encryption: Security Beyond the Brute-Force Bound. EUROCRYPT, pp. 293-310, 2014.
https://eprint.iacr.org/2014/155.pdf

2. A. Juels, T. Ristenpart. Honey Encryption: Security Beyond the Brute-Force Bound, презентация для конференции Eurocrypt 2014.
http://ec14.compute.dtu.dk/talks/19.pdf

3. Медовое шифрование обманывает хакеров ложной дешифровкой
https://threatpost.ru/medovoe-shifrovanie-obmany-vaet-hakerov-lozhnoj-deshifrovkoj

4. Полный перебор
https://ru.wikipedia.org/wiki/Полный_перебор

5. T. Ristenpart. New Encryption Primitives for Uncertain Times. Презентация для конференции FCE 2014.
http://fse2014.isg.rhul.ac.uk/slides/slides-10_2.pdf

7. P. Teuwen. Weird cryptography; or, How to resist brute-force attacks. Журнал PoC || GTFO, выпуск 08, страницы 60-63, 2015.
https://www.alchemistowl.org/pocorgtfo/pocorgtfo08.pdf

8. R. Chatterjee, J. Bonneau, A. Juels, T. Ristenpart. Cracking-Resistant Password Vaults using Natural Language Encoders. IEEE Symposium on Security and Privacy (SP), 2015.
https://pages.cs.wisc.edu/~rchat/papers/NoCrack.pdf

9. NoCrack
https://pages.cs.wisc.edu/~rchat/projects/NoCrack.html

10. Honey Encryption Implementation
https://github.com/shiriskumar/HoneyEncryption

11. N. Tyagi, J. Wang, K. Wen, D. Zuo. Honey Encryption Applications: Implementation of an encryption scheme resilient to brute-force attacks. http://www.mit.edu/~ntyagi/papers/honey-encryption-cc.pdf

12. Honey Encryption Implementation
https://github.com/danielzuot/honeyencryption

14. Honey Encryption of Credit Card Number Online
http://asecuritysite.com/encryption/honey

15. PoliMi Cryptography Group: Projects
http://crypto.dei.polimi.it/doku.php?id=projects:main#honeykeys_and_rsa

16. Зубов И.А. Медовое шифрование равномерно распределённых 8-битных целых чисел без знака. Электронный журнал «Вычислительные сети. Теория и практика», номер 28, июнь 2016.
http://network-journal.mpei.ac.ru/cgi-bin/main.pl?l=ru&n=28&pa=11&ar=1

17. Зубов И.А. Исследование совместного применения медового шифрования и шифрования RSA. электронный журнал «Вычислительные сети. Теория и практика», номер 28, июнь 2016.
http://network-journal.mpei.ac.ru/cgi-bin/main.pl?l=ru&n=28&pa=11&ar=2

18. Лицензия *BSD*
https://ru.wikipedia.org/wiki/Лицензия_BSD

19. Daniel J. Bernstein. Introduction to post-quantum cryptography. Post-Quantum Cryptography, pp. 1-14, Springer-Verlag Berlin Heidelberg, 2009.
http://pqcrypto.org/www.springer.com/cda/content/document/cda_downloaddocument/9783540887010-c1.pdf

20. AES – Advanced Encryption Standard (FIPS 197)
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

21. Honeydata — honey encryption library
https://github.com/postboy/honeydata

22. Обобщённое программирование
https://ru.wikipedia.org/wiki/Обобщённое_программирование

23. IEEE floating point
https://en.wikipedia.org/wiki/IEEE_floating_point

24. NaN
https://en.wikipedia.org/wiki/NaN

25. Single-precision floating-point format
https://en.wikipedia.org/wiki/Single-precision_floating-point_format

26. Zhicong Huang, Erman Ayday, Jacques Fellay, Jean-Pierre Hubaux, Ari Juels. GenoGuard: Protecting Genomic Data against Brute-Force Attacks. IEEE Symposium on Security and Privacy (SP), pp. 447-462, 2015.
https://infoscience.epfl.ch/record/205068/files/main.pdf

27. Houda Ferradi, Rémi Géraud, David Naccache. Human Public-Key Encryption. Cryptology ePrint Archive, доклад 2016/763.
https://eprint.iacr.org/2016/763.pdf

28. Hyun-Ju Jo, Ji Won Yoon. A New Countermeasure against Brute-Force Attacks That Use High Performance Computers for Big Data Analysis. International Journal of Distributed Sensor Networks, June 2015, vol. 11, no. 6.
http://dsn.sagepub.com/content/11/6/406915.full

29. Joo-Im Kim, Ji Won Yoon. Honey Chatting: a novel Instant Messaging system robust to eavesdropping over communication. 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).
http://ieeexplore.ieee.org/document/7472064/

30. Ji Won Yoon, Hyoungshick Kim, Hyun-Ju Jo, Hyelim Lee, Kwangsu Lee. Visual Honey Encryption: Application to Steganography. IH&MMSec '15: Proceedings of the 3rd ACM Workshop on Information Hiding and Multimedia Security, pp. 65-74.

http://dl.acm.org/citation.cfm?id=2756606

31. Maximilian Golla, Benedict Beuscher, Markus Dürmuth. On the Security of Cracking-Resistant Password Vaults. The ACM Conference on Computer and Communications Security 2016 (CCS '16), Vienna, Austria, October 24-28, 2016.

https://www.ei.rub.de/media/mobsec/veroeffentlichungen/2016/08/18/cracking-resistant_password_vaults.pdf

32. Honey Encryption

https://github.com/victornguyen75/honey-encryption

33. Типы данных в C

https://ru.wikipedia.org/wiki/Типы_данных_в_C

34. Matt Stancliff. How to C in 2016

https://matt.sh/howto-c

35. The GNU MP Bignum Library

https://gmplib.org

36. Порядок байтов

https://ru.wikipedia.org/wiki/Порядок_байтов

37. 10 правил программирования на Си, предотвращающих ошибки

http://chipenable.ru/index.php/programming-avr/item/62-10-pravil-programmirovaniya-na-si-predotvraschayuschih-oshibki.html

38. Число половинной точности

https://ru.wikipedia.org/wiki/Число_половинной_точности

39. Minifloat

https://en.wikipedia.org/wiki/Minifloat

40. Extended precision

https://en.wikipedia.org/wiki/Extended_precision

41. Число четверной точности

https://ru.wikipedia.org/wiki/Число_четверной_точности

42. Octuple-precision floating-point format

https://en.wikipedia.org/wiki/Octuple-precision_floating-point_format

43. Cleve Moler. Floating Point Numbers

http://blogs.mathworks.com/cleve/2014/07/07/floating-point-numbers

44. Rounding

https://en.wikipedia.org/wiki/Rounding

45. Signed zero

https://en.wikipedia.org/wiki/Signed_zero

46. Критерий хи-квадрат

https://ru.wikipedia.org/wiki/Критерий_хи-квадрат

47. Критерий согласия Пирсона

https://ru.wikipedia.org/wiki/Критерий_согласия_Пирсона

48. Honey Encryption

https://github.com/yinweihappy168/honeyencryption

49. Marc Beunardeau, Houda Ferradi, Rémi Géraud, David Naccache. Honey Encryption for Language. IACR Cryptology ePrint Archive 2017: 31 (2017).

https://eprint.iacr.org/2017/031.pdf

50. Зубов И.А. Применение медового шифрования к числам с плавающей точкой. Сборник тезисов XXIII международной научно-технической конференции студентов и аспирантов «Радиоэлектроника, электротехника и энергетика», том 1. Москва, Издательский дом МЭИ, 2017. http://reepe.mpei.ru/abstracts/Documents/ree-2017-book-1.pdf

51. Зубов И.А. Применение медового шифрования к числам с плавающей запятой. Электронный журнал «Вычислительные сети. Теория и практика», номер 30, июнь 2017.

52. Honey Encryption CLI https://github.com/VIVEKHYPER/Honey_Encryption_cli

53. Имитовставка https://ru.wikipedia.org/wiki/Имитовставка

54. Дешифрование http://cryptography.ru/docs/Дешифрование

55. Шифрование https://ru.wikipedia.org/wiki/Шифрование

# Приложение А.

# Техническое задание

# НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ «МЭИ»

**Институт** АВТ **Кафедра** Вычислительных машин, систем и сетей
**Направление** 09.04.01 – Информатика и вычислительная техника

# ЗАДАНИЕ НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ

**по программе подготовки магистров** Вычислительные машины, комплексы, системы и сети
**Тема** Реализация статистического кодирования числовых данных для усиления их стойкости к атакам перебором
**Время выполнения работы с** 6 февраля 2017 г. **по** 30 июня 2017 г.
**Студент** Зубов И.А. А-07м-11 .

*Фамилия, и., о.* *группа* *подпись*

**Научный руководитель** ст. преп. Филатов А.В. .
*должность, звание, фамилия, и., о.*

**Консультант** _____
*должность, звание, фамилия, и., о.*

**Консультант** _____
*должность, звание, фамилия, и., о.*

**Зав. кафедрой** к.т.н., доц. Вишняков С.В. .02.2017 г.
*звание, фамилия, и., о., подпись, дата утверждения задания*

**Место выполнения научной работы** _____
_____

**Москва** **«_____»_____20____г.**

**1.Обоснование выбора темы диссертационной работы**

В 2014 году американские учёные Ари Джулс и Томас Ристенпарт выдвинули новаторскую идею медового шифрования, которая потенциально может иметь заметный практический эффект. При обычном шифровании, если злоумышленник может осуществить атаку частичного или полного перебора ключа, то он имеет большие шансы на взлом криптосистемы. Медовое шифрование представляет собой дополнительный слой защищённости в этих сценариях: помимо перебора, злоумышленнику придётся также решить задачу отделения реальных данных от неверных, но правдоподобно выглядящих. Данная магистерская диссертация посвящена реализации медового шифрования для усиления защиты числовых данных различных типов (целых и с плавающей запятой, без знака и со знаком, различного размера). Это имеет смысл, так как именно в числовом формате хранится большое количество конфиденциальной информации, например, номера и CVV-коды банковских карт, выбранные пользователями PIN-коды, финансовая информация, некоторые персональные данные и многое другое.

**Научный руководитель _____ дата _____**
**Студент _____ дата _____**

**2.Консультации по разделу**

_____
_____
**Подпись консультанта_____ дата _____**

**3.Консультации по разделу**

_____
_____
**Подпись консультанта_____ дата _____**

## 4. План работы над магистерской диссертацией

| № п\п | Содержание разделов | Срок выполнения | Трудоёмкость в % |
|---|---|---|---|
| **I.** | **Теоретическая часть**<br>Исследование методов медового шифрования для равномерно и произвольно распределённых целых чисел, равномерно распределённых чисел с плавающей точкой. | **6 февраля 2017 г. – 17 марта 2017 г.** | **40** |
| **II.** | **Экспериментальная часть**<br>Программная реализация алгоритмов медового шифрования для равномерно и произвольно распределённых целых чисел, равномерно распределённых чисел с плавающей точкой, тестирование и анализ этих реализаций. | **18 марта 2017 г. – 26 апреля 2017 г.** | **40** |
| **III.** | **Публикации**<br>Статья в электронном журнале «Вычислительные сети. Теория и практика», номер 30, июнь 2017. | **июнь 2017 г.** | **5** |
| **IV.** | **Оформление диссертации**<br>Оформление результатов и защита диссертации на кафедре ВМСС. | **27 апреля 2017 г. – 30 июня 2017 г.** | **15** |

**5. Рекомендуемая литература**

1. Шнайер, Б. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си [Текст] / Брюс Шнайер ; перевод с англ. под ред. П. В. Семьянова — М. : Издательство ТРИУМФ, 2013. — 816 с. — Перевод изд.: Applied Cryptography / Bruce Schneier. New York, 1996. — 4000 экз. — ISBN 978-5-89392-527-2 (в пер.).

2. Керниган, Б. Язык программирования Си [Текст] / Брайан Керниган, Деннис Ритчи ; перевод с англ. Вт. С. Штаркмана — Спб. : Невский диалект, 2001. — 352 с. — Перевод изд.: The C Programming Language / Brian W. Kernighan, Dennis M. Ritchie. Englewood Cliffs, 1988. — 4000 экз. — ISBN 5-7940-0045-7 (в пер.).

**6. Краткие сведения о студенте:**

**Домашний адрес** 143954, Московская область, г. Балашиха, мкр. Северный, д. 52, кв. 13

**Телефон служебный** нет **домашний** +7-916-975-05-54

***Примечание:*** **задание брошюруется вместе с диссертацией и с отзывами руководителя и рецензентов.**

# Приложение Б.

# Листинг программ

**hdata/hd_common.c**

```
#include "hd_common.h"

//parameters in following generic functions:
//itype - type of input elements

//generic function for finding minimum and maximum in array------------------

#define GET_ARRAY_MINMAX(itype) \
(const itype *array, const size_t size, itype *min, itype *max) \
{ \
      \
      /*check the arguments*/ \
      if (array == NULL) { \
            error("array = NULL"); \
            return 1; \
            } \
      if (size == 0) { \
            error("size = 0"); \
            return 2; \
            } \
      if (min == NULL) { \
            error("min = NULL"); \
            return 3; \
            } \
      if (max == NULL) { \
            error("max = NULL"); \
            return 4; \
            } \
      \
      /*variables for storing temporary minimum and maximum values*/ \
      itype tmpmin, tmpmax; \
      size_t i; \
      \
      /*initialize minimum and maximum values*/ \
      tmpmin = array[0]; \
      tmpmax = array[0]; \
      /*let's try to find smaller minimum and bigger maximum*/ \
      for (i = 1; i < size; i++) { \
            if (array[i] < tmpmin)        /*then it's the new minimum*/ \
```

```
                tmpmin = array[i]; \
            if (array[i] > tmpmax)        /*then it's the new maximum*/ \
                tmpmax = array[i]; \
            /*we don't break the cycle if ((tmpmin == itype_MIN) && (tmpmax ==
itype_MAX)) because it can be used for timing attack; same with not
proceeding to next iteration if we found a new minimum (same element can't be
both minimum and maximum if it is not the first)*/ \
            } \
      \
      /*finally copy results to output buffers*/ \
      *min = tmpmin; \
      *max = tmpmax; \
      \
      return 0; \
}

extern int get_uint8_minmax
      GET_ARRAY_MINMAX(uint8_t)
extern int get_int8_minmax
      GET_ARRAY_MINMAX(int8_t)
extern int get_uint16_minmax
      GET_ARRAY_MINMAX(uint16_t)
extern int get_int16_minmax
      GET_ARRAY_MINMAX(int16_t)
extern int get_uint32_minmax
      GET_ARRAY_MINMAX(uint32_t)
extern int get_int32_minmax
      GET_ARRAY_MINMAX(int32_t)
extern int get_uint64_minmax
      GET_ARRAY_MINMAX(uint64_t)
extern int get_int64_minmax
      GET_ARRAY_MINMAX(int64_t)
extern int get_float_minmax
      GET_ARRAY_MINMAX(float)
extern int get_double_minmax
      GET_ARRAY_MINMAX(double)
extern int get_longd_minmax
      GET_ARRAY_MINMAX(long double)

#undef GET_ARRAY_MINMAX
```

```
//random data generation-------------------------------------------------------

//Windows-only
#ifdef WIN32

extern void randombytes(unsigned char *x, unsigned long long xlen)
{
      //initialize CryptoAPI only once
      if (hCryptProv != -1)
            if (CryptAcquireContext(&hCryptProv, NULL, NULL, PROV_RSA_FULL,
0))
                  printf("CryptAcquireContext succeeded.\n");
            else {
                  printf("CryptAcquireContext() error: %x\n", GetLastError());
                  return;
                  }

      if (CryptGenRandom(hCryptProv, xlen, *x) == 0) {
            printf("CryptGenRandom() error: %x\n", GetLastError());
            return;
            };

      }

//*nix-only
#else

/*
Original: http://hyperelliptic.org/nacl/nacl-20110221.tar.bz2, file
/randombytes/devurandom.c from
NaCl library version 20080713
*/

static int fd = -1;

extern void randombytes(unsigned char *x, unsigned long long xlen)
{
      int i;

      if (fd == -1) {
            for (;;) {
```

```
                  fd = open("/dev/urandom", O_RDONLY);
                  if (fd != -1) break;
                  sleep(1);
                  }
            }

      while (xlen > 0) {
            if (xlen < 1048576) i = xlen; else i = 1048576;

            i = read(fd,x,i);
            if (i < 1) {
                  sleep(1);
                  continue;
                  }

            x += i;
            xlen -= i;
      }
}

#endif
```

**hdata/hd_int_uniform.c**

```
#include "hd_int_uniform.h"

//parameters in following generic functions:
//itype - type of input elements
//utype - unsigned type of same size as input type
//otype - type of output elements (always unsigned, size is twice larger than
size of itype)
//UTYPE_MAX, OTYPE_MAX - maximum possible values of utype and otype,
respectively
//ISPACE - size of itype and utype code space (equals UTYPE_MAX + 1)
//OSPACE - size of otype code space (equals OTYPE_MAX + 1)

//generic DTE function for encoding integer arrays in integer arrays---------

#define ENCODE_IN_INT_UNIFORM(itype, utype, otype, UTYPE_MAX, OTYPE_MAX) \
(const itype *in_array, otype *out_array, const size_t size, const itype min,
const itype max) \
```

```
{ \
      /*check the arguments*/ \
      if (in_array == NULL) { \
            error("in_array = NULL"); \
            return -1; \
            } \
      if (out_array == NULL) { \
            error("out_array = NULL"); \
            return -1; \
            } \
      if (size == 0) { \
            error("size = 0"); \
            return -1; \
            } \
      if (min > max) { \
            error("min > max"); \
            return -1; \
            } \
      \
      /*current processing element after type promotion*/ \
      otype oelt; \
      /*size of full group in elements, from 1 to (itype_MAX-itype_MIN+1)*/ \
      const otype group_size = (otype)max - min + 1; \
      /*number of elements in the last group or 0 if the last group is full,
from 0 to ISPACE-1. original formula was OSPACE % group_size, but it didn't
work for uint64_t: UINT64_MAX + 1 = 0 because of integer overflow. modular
arithmetic used here: (a + b) mod c = ( (a mod c) + (b mod c) ) mod c, but
since c >= 2 where last_group_size is used, then 1 mod c = 1.*/ \
      const utype last_group_size = ( (OTYPE_MAX) % group_size + 1 ) %
group_size; \
      size_t i; \
      \
      /*if every value is possible*/ \
      /*note the type promotion here: (max_type_number+1) can become 0 without
it because of integer overflow!*/ \
      if (group_size == ( (otype)(UTYPE_MAX)+1 ) ) { \
      /*then just copy input array to output array to create a first part*/ \
            memcpy(out_array, in_array, size*sizeof(itype)); \
            /*follow it by random numbers to create a second part*/ \
            randombytes( (unsigned char *)out_array + size*sizeof(itype), \
                        size*(sizeof(otype) - sizeof(itype)) ); \
```

```
            return 0; \
            } \
      \
      /*write the random numbers to output array*/ \
      randombytes( (unsigned char *)out_array, size*sizeof(otype) ); \
      \
      /*if only one value is possible then use a random number for encoding
each number*/ \
      if (group_size == 1) { \
            /*we're already done with random numbers, but we should check an
input array*/ \
            for (i = 0; i < size; i++) { \
                  if (in_array[i] != min) { \
                        error("wrong min or max value"); \
                        return -1; \
                        } \
                  } \
            return 0; \
            } \
      \
      /*total number of groups (from ISPACE+1 to OSPACE/2), so they will have
indexes in interval [0; group_num-1]. original formula was ceill( (long
double)OSPACE / group_size), but this formula is faster, more portable and
reliable. see math.c for equivalence proof.*/ \
      const otype group_num = (OTYPE_MAX) / group_size + 1; \
      \
      /*else encode each number using random numbers from out_array for group
selection*/ \
      for (i = 0; i < size; i++) { \
            if (in_array[i] < min) { \
                  error("wrong min value"); \
                  return -1; \
                  } \
            else if (in_array[i] > max) { \
                  error("wrong max value"); \
                  return -1; \
                  } \
            /*note type promotion here: algorithm don't work right without it
on e.g. int32 tests*/ \
            /*normalize current element and make type promotion*/ \
            oelt = in_array[i] - (otype)min;
```

```
            \
            /*if we can place the current element in any group (including the
last one) then do it*/ \
            if ( (oelt < last_group_size) || (last_group_size == 0) ) \
                  oelt += (out_array[i] % group_num) * group_size; \
            /*else place it in any group excluding the last one*/ \
            else \
                  oelt += ( out_array[i] % (group_num-1) ) * group_size; \
            \
            out_array[i] = oelt;    /*finally write it to buffer*/ \
            } \
      \
      return 0; \
}

extern int encode_uint8_uniform
      ENCODE_IN_INT_UNIFORM(uint8_t, uint8_t, uint16_t, UINT8_MAX, UINT16_MAX)
extern int encode_int8_uniform
      ENCODE_IN_INT_UNIFORM(int8_t, uint8_t, uint16_t, UINT8_MAX, UINT16_MAX)
extern int encode_uint16_uniform
      ENCODE_IN_INT_UNIFORM(uint16_t, uint16_t, uint32_t, UINT16_MAX,
      UINT32_MAX)
extern int encode_int16_uniform
      ENCODE_IN_INT_UNIFORM(int16_t, uint16_t, uint32_t, UINT16_MAX,
      UINT32_MAX)
extern int encode_uint32_uniform
      ENCODE_IN_INT_UNIFORM(uint32_t, uint32_t, uint64_t, UINT32_MAX,
      UINT64_MAX)
extern int encode_int32_uniform
      ENCODE_IN_INT_UNIFORM(int32_t, uint32_t, uint64_t, UINT32_MAX,
      UINT64_MAX)

#undef ENCODE_IN_INT_UNIFORM

//generic DTE function for encoding integer arrays in mpz_t arrays-----------

#define ENCODE_IN_MPZ_UNIFORM(itype, TYPE_MIN, TYPE_MAX) \
(const itype *in_array, unsigned char *out_array, const size_t size, const
itype min, const itype max) \
{ \
      /*check the arguments*/ \
```

```
	if (in_array == NULL) { \
		error("in_array = NULL"); \
		return -1; \
		} \
	if (out_array == NULL) { \
		error("out_array = NULL"); \
		return -1; \
		} \
	if (size == 0) { \
		error("size = 0"); \
		return -1; \
		} \
	if (min > max) { \
		error("min > max"); \
		return -1; \
		} \
	\
	/*current processing element after type promotion*/ \
	mpz_t oelt; \
	/*size of full group in elements, from 1 to (itype_MAX-itype_MIN+1)*/ \
	mpz_t group_size; \
	/*total number of groups (from ISPACE+1 to OSPACE/2), so they will have
indexes in interval [0; group_num-1]. original formula was ceil(OSPACE /
group_size), but following formula is faster, more portable and reliable. see
math.c for equivalence proof for smaller types.*/ \
	mpz_t group_num, group_num_minus_1; \
	/*temporary variable for different computations*/ \
	mpz_t tmp; \
	/*number of elements in the last group or 0    if the last group is
full, from  0 to ISPACE-1.*/ \
	uint64_t last_group_size; \
	/*normalized value of current element*/ \
	uint64_t normalized; \
	size_t i; \
	\
	/*if every value is possible*/ \
	if ( (min == TYPE_MIN) && (max == TYPE_MAX) ) { \
	/*then just copy input array to output array to create a first part*/ \
		memcpy(out_array, in_array, size*sizeof(itype)); \
		/*follow it by random numbers to create a second part*/ \
		randombytes( out_array + size*sizeof(itype), size*(16 -
```

```
sizeof(itype)) ); \
            return 0; \
            } \
      \
      /*write the random numbers to output array*/ \
      randombytes(out_array, 16*size); \
      \
      /*if only one value is possible then use a random number for encoding
each number*/ \
      if (min == max) { \
            /*we're already done with random numbers, but we should check an
input array*/ \
            for (i = 0; i < size; i++) { \
                  if (in_array[i] != min) { \
                        error("wrong min or max value"); \
                        return -1; \
                        } \
                  } \
            return 0; \
            } \
      \
      mpz_inits(oelt, group_size, group_num, group_num_minus_1, tmp, NULL); \
      \
      /*group_size = max - min + 1*/ \
      /*load second half, then first half: oelt = second_half << 32 +
first_half. we do this because long int type can have size of 4 bytes, while
our itype has size of 8 bytes.*/ \
      normalized = max - min; \
      mpz_set_ui(group_size, normalized >> 32); \
      mpz_mul_2exp(group_size, group_size, 32); \
      mpz_add_ui(group_size, group_size, normalized & 0xFFFFFFFF); \
      mpz_add_ui(group_size, group_size, 1); \
      \
      /*last_group_size = OSPACE % group_size, where OSPACE = ISPACE^2, ISPACE
= UINT64_MAX + 1 = (UINT32_MAX + 1)^2, then last_group_size = (UINT32_MAX +
1)^4 % group_size*/ \
      mpz_set_ui(tmp, UINT32_MAX); \
      mpz_add_ui(tmp, tmp, 1); \
      mpz_pow_ui(tmp, tmp, 4); \
      /*save this intermediate result for group_num computation*/ \
      mpz_set(group_num, tmp); \
```

```
      /*note: mpz_tdiv and mpz_fdiv function families will return the same
results in  this algorithm, since n >= 0. it means that we can use fdiv here,
if it will be beneficial for some reason. however, usage of tdiv should be
more obvious for reader.*/ \
      mpz_tdiv_r(tmp, tmp, group_size); \
      last_group_size = mpz_get_ui(tmp); \
      \
      /*group_num = OTYPE_MAX / group_size + 1, where OTYPE_MAX = OSPACE - 1 =
(UINT32_MAX + 1)^4 - 1*/ \
      mpz_sub_ui(group_num, group_num, 1); \
      mpz_tdiv_q(group_num, group_num, group_size); \
      /*save this intermediate result for future computations*/ \
      mpz_set(group_num_minus_1, group_num); \
      mpz_add_ui(group_num, group_num, 1); \
      \
      /*else encode each number using random numbers from out_array for group
selection*/ \
      for (i = 0; i < size; i++) { \
            if (in_array[i] < min) { \
                  error("wrong min value"); \
                  return -1; \
                  } \
            else if (in_array[i] > max) { \
                  error("wrong max value"); \
                  return -1; \
                  } \
            \
            /*normalize current element and make type promotion: oelt =
in_array[i] - min*/ \
            normalized = in_array[i] - min; \
            mpz_set_ui(oelt, normalized >> 32); \
            mpz_mul_2exp(oelt, oelt, 32); \
            mpz_add_ui(oelt, oelt, normalized & 0xFFFFFFFF); \
            \
            /*if we can place the current element in any group (including the
last one) then do it: oelt += (out_array[i] % group_num) * group_size*/ \
            mpz_import(tmp, 16/sizeof(int), -1, sizeof(int), 0, 0,
out_array+16*i); \
            if ( (normalized < last_group_size) || (last_group_size == 0) ) \
                  mpz_tdiv_r(tmp, tmp, group_num); \
            /*else place it in any group excluding the last one: oelt +=
```

```
( out_array[i] % (group_num-1) ) * group_size*/ \
            else \
                  mpz_tdiv_r(tmp, tmp, group_num_minus_1); \
            mpz_mul(tmp, tmp, group_size); \
            mpz_add(oelt, oelt, tmp); \
            \
            /*we must clear destination memory because garbage there may not
be overwritten by next call: e.g., if current variable fits in 3 words then
4th word won't be overwritten*/ \
            memset(out_array+16*i, 0, 16); \
            mpz_export(out_array+16*i, NULL, -1, sizeof(int), 0, 0, oelt); \
            } \
      \
      mpz_clears(oelt, group_size, group_num, group_num_minus_1, tmp, NULL); \
      return 0; \
}

extern int encode_uint64_uniform
      ENCODE_IN_MPZ_UNIFORM(uint64_t, 0, UINT64_MAX)
extern int encode_int64_uniform
      ENCODE_IN_MPZ_UNIFORM(int64_t, INT64_MIN, INT64_MAX)

#undef ENCODE_IN_MPZ_UNIFORM

//generic DTE function for extracting integer arrays from integer arrays-----

#define DECODE_IN_INT_UNIFORM(itype, otype, UTYPE_MAX) \
(const otype *in_array, itype *out_array, const size_t size, const itype min,
const itype max) \
{ \
      /*check the arguments*/ \
      if (in_array == NULL) { \
            error("in_array = NULL"); \
            return -1; \
            } \
      if (out_array == NULL) { \
            error("out_array = NULL"); \
            return -1; \
            } \
      if (size == 0) { \
            error("size = 0"); \
```

```
            return -1; \
            } \
      if (min > max) { \
            error("min > max"); \
            return -1; \
            } \
      \
      /*size of full group in elements, from 1 to (itype_MAX-itype_MIN+1)*/ \
      const otype group_size = (otype)max - min + 1; \
      size_t i; \
      \
      /*if every value is possible then just copy first part of input array to
output array*/ \
      /*note the type promotion here: (max_type_number+1) can become 0 without
it because of integer overflow!*/ \
      if (group_size == ( (otype)(UTYPE_MAX)+1 ) ) { \
            memcpy(out_array, in_array, size*sizeof(itype)); \
            return 0; \
            } \
      \
/*if only one value is possible then fill output array with this value*/ \
      if (group_size == 1) { \
            for (i = 0; i < size; i++) \
                  out_array[i] = min; \
            return 0; \
            } \
      \
      /*else decode each number*/ \
      for (i = 0; i < size; i++) { \
      /*get its value in first group, denormalize it, do a type regression*/ \
            out_array[i] = (in_array[i] % group_size) + min; \
            \
            /*if algorithm works right, this errors should never be thrown*/ \
            if (out_array[i] < min) { \
                  error("algorithm error: wrong value < min"); \
                  return -1; \
                  } \
            else if (out_array[i] > max) { \
                  error("algorithm error: wrong value > max"); \
                  return -1; \
                  } \
```

```
            } \
      \
      return 0; \
}

extern int decode_uint8_uniform
      DECODE_IN_INT_UNIFORM(uint8_t, uint16_t, UINT8_MAX)
extern int decode_int8_uniform
      DECODE_IN_INT_UNIFORM(int8_t, uint16_t, UINT8_MAX)
extern int decode_uint16_uniform
      DECODE_IN_INT_UNIFORM(uint16_t, uint32_t, UINT16_MAX)
extern int decode_int16_uniform
      DECODE_IN_INT_UNIFORM(int16_t, uint32_t, UINT16_MAX)
extern int decode_uint32_uniform
      DECODE_IN_INT_UNIFORM(uint32_t, uint64_t, UINT32_MAX)
extern int decode_int32_uniform
      DECODE_IN_INT_UNIFORM(int32_t, uint64_t, UINT32_MAX)

#undef DECODE_IN_INT_UNIFORM

//generic DTE function for extracting integer arrays from mpz_t arrays-------

#define DECODE_IN_MPZ_UNIFORM(itype, TYPE_MIN, TYPE_MAX) \
(const unsigned char *in_array, itype *out_array, const size_t size, const
itype min, const itype max) \
{ \
      /*check the arguments*/ \
      if (in_array == NULL) { \
            error("in_array = NULL"); \
            return -1; \
            } \
      if (out_array == NULL) { \
            error("out_array = NULL"); \
            return -1; \
            } \
      if (size == 0) { \
            error("size = 0"); \
            return -1; \
            } \
      if (min > max) { \
            error("min > max"); \
```

```
            return -1; \
            } \
      \
      /*current processing element before type promotion*/ \
      mpz_t ielt; \
      /*size of full group in elements, from 1 to (itype_MAX-itype_MIN+1)*/ \
      mpz_t group_size; \
      /*temporary variable for different computations*/ \
      mpz_t tmp; \
      /*normalized value of current element*/ \
      uint64_t normalized; \
      size_t i; \
      \
      /*if every value is possible then just copy first part of input array to
output array*/ \
      if ( (min == TYPE_MIN) && (max == TYPE_MAX) ) { \
            memcpy(out_array, in_array, size*sizeof(itype)); \
            return 0; \
            } \
      \
/*if only one value is possible then fill output array with this value*/ \
      if (min == max) { \
            for (i = 0; i < size; i++) \
                  out_array[i] = min; \
            return 0; \
            } \
      \
      mpz_inits(ielt, group_size, tmp, NULL); \
      \
      /*group_size = max - min + 1*/ \
      normalized = max - min; \
      mpz_set_ui(group_size, normalized >> 32); \
      mpz_mul_2exp(group_size, group_size, 32); \
      mpz_add_ui(group_size, group_size, normalized & 0xFFFFFFFF); \
      mpz_add_ui(group_size, group_size, 1); \
      \
      /*else decode each number*/ \
      for (i = 0; i < size; i++) { \
      /*get its value in first group, denormalize it, do a type regression*/ \
            /*out_array[i] = (in_array[i] % group_size) + min*/ \
            mpz_import(ielt, 16/sizeof(int), -1, sizeof(int), 0, 0,
```

```
in_array+16*i); \
            mpz_tdiv_r(ielt, ielt, group_size); \
            /*save second half (i.e. its most significant 4 bytes) of ielt in
tmp, first half (i.e. its least significant 4 bytes) in ielt*/ \
            mpz_tdiv_q_2exp(tmp, ielt, 32); \
            mpz_tdiv_r_2exp(ielt, ielt, 32); \
            \
            normalized = mpz_get_ui(tmp); \
            normalized <<= 32; \
            normalized += mpz_get_ui(ielt); \
            out_array[i] = normalized + min; \
            \
            /*if algorithm works right, this errors should never be thrown*/ \
            if (out_array[i] < min) { \
                  error("algorithm error: wrong value < min"); \
                  return -1; \
                  } \
            else if (out_array[i] > max) { \
                  error("algorithm error: wrong value > max"); \
                  return -1; \
                  } \
            } \
      \
      mpz_clears(ielt, group_size, tmp, NULL); \
      return 0; \
}

extern int decode_uint64_uniform
      DECODE_IN_MPZ_UNIFORM(uint64_t, 0, UINT64_MAX)
extern int decode_int64_uniform
      DECODE_IN_MPZ_UNIFORM(int64_t, INT64_MIN, INT64_MAX)

#undef DECODE_IN_MPZ_UNIFORM
```

**hdata/hd_int_arbitrary.c**

```
#include "hd_int_arbitrary.h"

//first check if we can handle such array, then convert weights to cumulative
weights
/*optimization note: cumulative weights can be saved after computation in
```

```
encoding stage and just loaded (rather then re-computed) in decoding stage*/
#define CHECK_ARRAYS_GET_CUMULS() \
do { \
      /*get size of weights and cumuls arrays*/ \
      wsize_check = max - min + 1; \
      wsize = wsize_check; \
      \
      /*make a check for an overflow. it occurs if user wants to use too big
weights and cumuls arrays (i.e., size_t type can't hold their size). to catch
this situation, we make computations in biggest type available, then convert
it to size_t. however, overflow can occur even in biggest type, so we should
catch for this situation too.*/ \
      /*optimization note: we can skip such check(s) if overflow(s) in
question aren't possible, for example when we're handling (u)int8_t
arrays*/ \
      if ( (wsize != wsize_check) || ((max == UINT64_MAX) && (min == 0)) ) { \
            error("can't handle such big supplementary arrays"); \
            return -1; \
      } \
      \
      if ( (cumuls = malloc(wsize*sizeof(uint64_t))) == NULL ) { \
            error("couldn't allocate memory for cumuls"); \
            return -1; \
            } \
      \
      /*convert weights to cumulative weights: cumuls[i] = sum of weights[j],
where j = 0..i*/ \
      current = 0; \
      for (i = 0; i < wsize; i++) { \
            prev = current; \
            current += weights[i]; \
            if (current < prev) { \
                  error("integer overflow during cumuls computation"); \
                  free(cumuls); \
                  return -1; \
                  } \
            cumuls[i] = current; \
            } \
} while(0)

#define ENCODE_IN_TYPE_ARBITRARY(ctype_t, ctype, otype) \
```

```
do { \
      /*index and weight of current element*/ \
      size_t index; \
      uint32_t weight; \
      /*cumulative weight of previous element*/ \
      uint64_t cumul_prev; \
      /*arrays for temporary and random data (actually this is the same array
in memory)*/ \
      ctype_t *temp_array; \
      otype *rand_array; \
      /*return value*/ \
      int rv; \
      \
      if ( (temp_array = malloc(size*sizeof(otype))) == NULL ) { \
            error("couldn't allocate memory for temp_array"); \
            free(cumuls); \
            return -1; \
            } \
      randombytes( (unsigned char *)temp_array, size*sizeof(otype) ); \
      rand_array = (otype *)temp_array; \
      \
      if ( (*out_array = malloc(size*sizeof(otype))) == NULL ) { \
            error("couldn't allocate memory for out_array"); \
            free(cumuls); \
            free(temp_array); \
            return -1; \
            } \
      \
      for (i = 0; i < size; i++) { \
            if (in_array[i] < min) { \
                  error("wrong min value"); \
                  free(cumuls); \
                  free(temp_array); \
                  free(*out_array); \
                  *out_array = NULL; \
                  return -1; \
                  } \
            else if (in_array[i] > max) { \
                  error("wrong max value"); \
                  free(cumuls); \
                  free(temp_array); \
```

```
                free(*out_array); \
                *out_array = NULL; \
                return -1; \
                } \
          index = in_array[i] - min; \
          \
          if (index == 0) \
                cumul_prev = 0; \
          else \
                cumul_prev = cumuls[index-1]; \
/*weight is the difference between two consecutive cumulative weights*/ \
          weight = cumuls[index] - cumul_prev; \
          \
          /*check if current value is impossible according to cumuls*/ \
          if (weight == 0) { \
                error("value in array is impossible according to cumuls"); \
                free(cumuls); \
                free(temp_array); \
                free(*out_array); \
                *out_array = NULL; \
                return -1; \
                } \
          \
/*each temp value is pseudorandom value in [cumul_prev; cumuls[index]-1]*/ \
          /*we're very neat here: first we read rand_array member, which is
two times bigger then temp_array member, then we write temp_array member,
possibly overwriting rand_array member, which we don't need anymore*/ \
          temp_array[i] = (rand_array[i] % weight) + cumul_prev; \
          } \
    \
    /*finally encode uniformly distributed temporary values*/ \
    rv = encode_##ctype##_uniform(temp_array, (otype *)*out_array, size, 0,
cumuls[wsize-1]); \
    \
    free(cumuls); \
    free(temp_array); \
    \
    /*if error happened, then return -1, else return size of output type in
bytes*/ \
    if (rv) { \
          free(*out_array); \
```

```
            *out_array = NULL; \
            return -1; \
            } \
      else { \
            return sizeof(otype); \
            } \
} while(0)

#define ENCODE_ARBITRARY(itype) \
(const itype *in_array, void **out_array, const size_t size, const itype min,
const itype max, const uint32_t *weights) \
{ \
      /*check the arguments*/ \
      if (in_array == NULL) { \
            error("in_array = NULL"); \
            return -1; \
            } \
      if (out_array == NULL) { \
            error("out_array = NULL"); \
            return -1; \
            } \
      if (size == 0) { \
            error("size = 0"); \
            return -1; \
            } \
      if (min > max) { \
            error("min > max"); \
            return -1; \
            } \
      if (weights == NULL) { \
            error("weights = NULL"); \
            return -1; \
            } \
      \
      /*current and previous weights*/ \
      uint64_t current, prev; \
      /*cumulative weights*/ \
      uint64_t *cumuls; \
      size_t i, wsize; \
      uint64_t wsize_check; \
      \
```

```
	CHECK_ARRAYS_GET_CUMULS(); \
	\
	/*check maximum value after encoding (i.e., maximum cumulative weight
value)*/ \
	if (current < 256)                  /*2^8*/ \
		ENCODE_IN_TYPE_ARBITRARY(uint8_t, uint8, uint16_t); \
	else if (current < 65536)           /*2^16*/ \
		ENCODE_IN_TYPE_ARBITRARY(uint16_t, uint16, uint32_t); \
	else if (current < 4294967296)      /*2^32*/ \
		ENCODE_IN_TYPE_ARBITRARY(uint32_t, uint32, uint64_t); \
	else { \
		error("too many values for any supported output type"); \
		free(cumuls); \
		return -1; \
		} \
}

extern int encode_uint8_arbitrary
	ENCODE_ARBITRARY(uint8_t)
extern int encode_int8_arbitrary
	ENCODE_ARBITRARY(int8_t)
extern int encode_uint16_arbitrary
	ENCODE_ARBITRARY(uint16_t)
extern int encode_int16_arbitrary
	ENCODE_ARBITRARY(int16_t)
extern int encode_uint32_arbitrary
	ENCODE_ARBITRARY(uint32_t)
extern int encode_int32_arbitrary
	ENCODE_ARBITRARY(int32_t)
extern int encode_uint64_arbitrary
	ENCODE_ARBITRARY(uint64_t)
extern int encode_int64_arbitrary
	ENCODE_ARBITRARY(int64_t)

#undef ENCODE_ARBITRARY
#undef ENCODE_IN_TYPE_ARBITRARY

#define DECODE_IN_TYPE_ARBITRARY(ctype_t, ctype) \
do { \
	/*index and weight of current element*/ \
	size_t index; \
```

```
	/*uint32_t weight;*/ \
	/*temporary array*/ \
	ctype_t *temp_array; \
	/*return value*/ \
	int rv; \
	\
	if ( (temp_array = malloc(size*sizeof(ctype_t))) == NULL ) { \
		error("couldn't allocate memory for temp_array"); \
		free(cumuls); \
		return -1; \
		} \
	\
	/*firstly decode uniformly distributed temporary values*/ \
	/*optimization note: here we can use out_array instead of temp_array*/ \
	rv = decode_##ctype##_uniform(in_array, temp_array, size, 0,
cumuls[wsize-1]); \
	if (rv < 0) { \
		free(cumuls); \
		free(temp_array); \
		return rv; \
		} \
	\
	for (i = 0; i < size; i++) { \
		/*optimization note: binary search or some other clever algorithm
would be much faster*/ \
		for (index = 0; index < (max - min + 1); index++) { \
		/*in this simple algorithm, (weight > 0) condition always holds,
but generally we should check this*/ \
			/*if (index == 0) \
				weight = cumuls[index]; \
			else \
				weight = cumuls[index] - cumuls[index-1];*/ \
			if ((temp_array[i] < cumuls[index]) /*&& (weight > 0)*/) \
				break; \
			} \
		\
		if (index == (max - min + 1)) { \
			error("can't find corresponding cumuls element"); \
			free(cumuls); \
			free(temp_array); \
			return -1; \
```

```
                  } \
            \
            out_array[i] = index + min; \
            \
            /*if algorithm works right, this errors should never be thrown*/ \
            if (out_array[i] < min) { \
                  error("algorithm error: wrong value < min"); \
                  free(cumuls); \
                  free(temp_array); \
                  return -1; \
                  } \
            else if (out_array[i] > max) { \
                  error("algorithm error: wrong value > max"); \
                  free(cumuls); \
                  free(temp_array); \
                  return -1; \
                  } \
            } \
      \
      free(cumuls); \
      free(temp_array); \
      return 0; \
} while (0)

#define DECODE_ARBITRARY(itype) \
(const void *in_array, itype *out_array, const size_t size, const itype min,
const itype max, const uint32_t *weights) \
{ \
      /*check the arguments*/ \
      /*optimization note: some of this checks can safely be delegated to
decode_ctype_uniform()*/ \
      if (in_array == NULL) { \
            error("in_array = NULL"); \
            return -1; \
            } \
      if (out_array == NULL) { \
            error("out_array = NULL"); \
            return -1; \
            } \
      if (size == 0) { \
            error("size = 0"); \
```

```
            return -1; \
            } \
      if (min > max) { \
            error("min > max"); \
            return -1; \
            } \
      if (weights == NULL) { \
            error("weights = NULL"); \
            return -1; \
            } \
      \
      /*current and previous weights*/ \
      uint64_t current, prev; \
      /*cumulative weights*/ \
      uint64_t *cumuls; \
      size_t i, wsize; \
      uint64_t wsize_check; \
      \
      CHECK_ARRAYS_GET_CUMULS(); \
      \
      /*check maximum value after encoding (i.e., maximum cumulative weight
value)*/ \
      if (current < 256)                        /*2^8*/ \
            DECODE_IN_TYPE_ARBITRARY(uint8_t, uint8); \
      else if (current < 65536)           /*2^16*/ \
            DECODE_IN_TYPE_ARBITRARY(uint16_t, uint16); \
      else if (current < 4294967296)      /*2^32*/ \
            DECODE_IN_TYPE_ARBITRARY(uint32_t, uint32); \
      else { \
            error("too many values for any supported output type"); \
            free(cumuls); \
            return -1; \
            } \
}

extern int decode_uint8_arbitrary
      DECODE_ARBITRARY(uint8_t)
extern int decode_int8_arbitrary
      DECODE_ARBITRARY(int8_t)
extern int decode_uint16_arbitrary
      DECODE_ARBITRARY(uint16_t)
```

```
extern int decode_int16_arbitrary
      DECODE_ARBITRARY(int16_t)
extern int decode_uint32_arbitrary
      DECODE_ARBITRARY(uint32_t)
extern int decode_int32_arbitrary
      DECODE_ARBITRARY(int32_t)
extern int decode_uint64_arbitrary
      DECODE_ARBITRARY(uint64_t)
extern int decode_int64_arbitrary
      DECODE_ARBITRARY(int64_t)

#undef DECODE_ARBITRARY
#undef DECODE_IN_TYPE_ARBITRARY
#undef CHECK_ARRAYS_GET_CUMULS
```

**hdata/hd_fp_uniform.c**

```
#include "hd_fp_uniform.h"

//parameters in following generic functions:
//itype - type of input elements (floating-point type)
//otype - type of output elements (integer type of the same size as itype)
//MIDDLE - middle value of otype, i.e. first value of its code space's second
half
//MASK - bitmask for setting is_quiet bit of NaNs

//generic function for converting fp arrays to integer arrays----------------

/*convert fp number to integer and back such way that each integer value will
contain a number of corresponding fp number in sequence of all possible fp
numbers of this type. it means that negative fp number with highest possible
exponent and fraction will become 0 integer, and positive fp number with
highest possible exponent and fraction will become maximum value for
corresponding integer type.

is_quiet bit of all NaNs in array is assigned to random value during
encoding, and is assigned to 1 during decoding.

optimization note: turns out that just running UINT_TO_FP_UNIFORM without
explicit setting of is_quite bit may be enough to convert all signaling NaNs
to quiet NaNs, but it may be not portable.*/
```

```
#define FP_TO_UINT_UNIFORM(itype, otype, MIDDLE, MASK) \
(const itype *in_array, otype *out_array, const size_t size) \
{ \
      /*check the arguments*/ \
      if (in_array == NULL) { \
            error("in_array = NULL"); \
            return -1; \
            } \
      if (out_array == NULL) { \
            error("out_array = NULL"); \
            return -1; \
            } \
      if (size == 0) { \
            error("size = 0"); \
            return -1; \
            } \
      \
      /*current processing element before conversion. we use union notation
here for access to bitwise and integer arithmetic operations.*/ \
      union { \
            itype fp; \
            otype i; \
            } ielt; \
      size_t i; \
      \
      /*write a random numbers to output array*/ \
      randombytes( (unsigned char *)out_array, size*sizeof(itype) ); \
      \
      for (i = 0; i < size; i++) { \
            ielt.fp = in_array[i];        /*read the current element*/ \
            \
      /*if we process NaN then assign random value to its is_quiet bit*/ \
            if (isnan(ielt.fp)) { \
                  /*if random value is in first half of code space then set
bit, else clear bit*/ \
                  if ( out_array[i] < (MIDDLE) ) \
                        ielt.i = ielt.i | (MASK); \
                  else \
                        ielt.i = ielt.i & ~(MASK); \
                  } \
```

```
            \
            if (ielt.i < MIDDLE) /*if it has a positive value as fp number*/ \
                  out_array[i] = ielt.i + (MIDDLE); \
            else \
                  out_array[i] = ~ielt.i; \
            } \
      \
      return 0; \
}

extern int float_to_uint32_uniform
      FP_TO_UINT_UNIFORM(float, uint32_t, 0x80000000, 0x00400000)
extern int double_to_uint64_uniform
      FP_TO_UINT_UNIFORM(double, uint64_t, 0x8000000000000000,
0x0008000000000000)

#undef FP_TO_UINT_UNIFORM

//generic function for converting integer arrays back to fp arrays-----------

#define UINT_TO_FP_UNIFORM(itype, otype, MIDDLE, MASK) \
(const otype *in_array, itype *out_array, const size_t size) \
{ \
      /*check the arguments*/ \
      if (in_array == NULL) { \
            error("in_array = NULL"); \
            return -1; \
            } \
      if (out_array == NULL) { \
            error("out_array = NULL"); \
            return -1; \
            } \
      if (size == 0) { \
            error("size = 0"); \
            return -1; \
            } \
      \
      /*current processing element after conversion. we use union notation
here for access to bitwise and integer arithmetic operations.*/ \
      union { \
            itype fp; \
```

```
            otype i; \
            } ielt; \
      size_t i; \
      \
      for (i = 0; i < size; i++) { \
            /*if it has a positive value as fp number*/ \
            if (in_array[i] >= (MIDDLE) ) \
                  ielt.i = in_array[i] - (MIDDLE); \
            else \
                  ielt.i = ~in_array[i]; \
            \
            /*if we process NaN then assign 1 to its is_quiet bit*/ \
            if (isnan(ielt.fp)) \
                  ielt.i = ielt.i | (MASK); \
            \
            out_array[i] = ielt.fp; \
            } \
      \
      return 0; \
}

extern int uint32_to_float_uniform
      UINT_TO_FP_UNIFORM(float, uint32_t, 0x80000000, 0x00400000)
extern int uint64_to_double_uniform
      UINT_TO_FP_UNIFORM(double, uint64_t, 0x8000000000000000,
0x0008000000000000)

#undef UINT_TO_FP_UNIFORM

/*optimization notes:
1. 2^0 + ... + 2^n = 2^(n+1) - 1
2. logical operations are faster than arithmetical ones, so we should use
former when we can*/

extern int container_float_uniform(const float min, const float max)
{
      /*check the arguments*/
      if (min > max) {
            error("min > max");
            return -1;
            }
```

```
	/*components of minimum and maximum: sign, exponent and fraction*/
	/*true - positive number, false - negative*/
	bool minsign, maxsign;
	uint16_t minexp, maxexp;
	uint32_t minfrac, maxfrac;

	/*number of used groups, weight of current group, number of additional
elements*/
	uint64_t used_groups, weight, remainder, i;

	/*we use this union for logical operations on fp numbers*/
	union {
		float f;
		uint32_t i;
		} fpint;

	/*get components of minimum and maximum*/
	fpint.f = min;
	minsign = (fpint.i >> 31 ) ? false : true;
	minexp = (fpint.i >> 23) & 0xFF;
	minfrac = fpint.i & 0x007FFFFF;

	fpint.f = max;
	maxsign = (fpint.i >> 31 ) ? false : true;
	maxexp = (fpint.i >> 23) & 0xFF;
	maxfrac = fpint.i & 0x007FFFFF;

	/*if both minimum and maximum have same sign*/
	if (minsign == maxsign) {
		/*if both of them are negative then swap their components*/
		if (!maxsign) {
			uint16_t tmpexp;
			uint32_t tmpfrac;

			tmpexp = minexp;
			minexp = maxexp;
			maxexp = tmpexp;

			tmpfrac = minfrac;
			minfrac = maxfrac;
			maxfrac = tmpfrac;
			}
```

```
            /*get number of fully used groups*/
            used_groups = 0;
            /*if minimum belongs to denormal numbers then second group will
have same weight, else it's weight will be two times larger*/
            if (minexp == 0)
                  weight = 1;
            else
                  weight = 2;
            for (i = minexp + 1; i < maxexp; i++) {
                  used_groups += weight;
                  weight <<= 1;
                  /*check if overflow happened*/
                  if (weight == 0)
                        return -1;
                  }

            /*count all additional elements*/
            if (minexp == maxexp) {
                  remainder = maxfrac - minfrac + 1;
                  /*maybe there is some full groups in remainder now?*/
                  used_groups += remainder >> 23;
                  remainder &= 0x007FFFFF;
                  }
            else {
      /*if maximum belongs to NaNs then last group will have smallest weight*/
                  if (maxexp == 0xFF)
                        weight = 1;
                  /*first part of remainder is contained in maxfrac*/
                  remainder = (maxfrac + 1)*weight;
                  /*maybe there is some full groups in remainder now?*/
                  used_groups += remainder >> 23;
                  remainder &= 0x007FFFFF;
                  /*second part of remainder is contained in minfrac*/
                  remainder += 0x007FFFFF + 1 - minfrac;
                  /*maybe there is some full groups in remainder now?*/
                  used_groups += remainder >> 23;
                  remainder &= 0x007FFFFF;
                  }

            /*if we need to use one more group then use it*/
            if (remainder != 0)
```

```
                used_groups++;
            }
        /*if minimum is negative and maximum is positive*/
        else {

            /*get number of fully used groups*/
            used_groups = 0;
            weight = 1;
            for (i = 0; i < minexp; i++) {
                used_groups += weight;
/*denormal numbers has the same weight as first group of normalized ones*/
                if (i != 0)
                    weight <<= 1;
                /*check if overflow happened*/
                if (weight == 0)
                    return -1;
                }
            /*count additional negative elements*/
        /*if maximum belongs to NaNs then last group will have smallest weight*/
            if (minexp == 0xFF)
                weight = 1;
            remainder = (minfrac + 1)*weight;
            /*maybe there is some full groups in remainder now?*/
            used_groups += remainder >> 23;
            remainder &= 0x007FFFFF;

            weight = 1;
            for (i = 0; i < maxexp; i++) {
                used_groups += weight;
/*denormal numbers has the same weight as first group of normalized ones*/
                if (i != 0)
                    weight <<= 1;
                /*check if overflow happened*/
                if (weight == 0)
                    return -1;
                }
            /*count additional positive elements*/
        /*if maximum belongs to NaNs then last group will have smallest weight*/
            if (maxexp == 0xFF)
                weight = 1;
            remainder += (maxfrac + 1)*weight;
            /*maybe there is some full groups in remainder now?*/
```

```
            used_groups += remainder >> 23;
            remainder &= 0x007FFFFF;

            /*if we need to use one more group then use it*/
            if (remainder != 0)
                  used_groups++;
            }

      if (used_groups < 512)                    /*2^9*/
            return 32;
      else if (used_groups < 2199023255552)     /*2^41*/
            return 64;
      else
            return -1;
}
```

**tests/t_common.c**

```
#include "t_common.h"

//parameters in following generic functions:
//itype - type of input elements
//atype - type of auxiliary elements
//MIN - minimum value of current type
//FORMAT - format string for printf()
//MASK - bitmask for setting is_quiet bit of NaNs

//generic function for getting a number of occurences of different elements
in integer array

#define STATS_INT_ARRAY(itype, MIN) \
(const itype *in_array, const size_t size, uint64_t *stats) \
{ \
      /*check the arguments*/ \
      if (in_array == NULL) { \
            error("in_array = NULL"); \
            return 1; \
            } \
      if (size == 0) { \
            error("size = 0"); \
            return 2; \
            } \
```

```
	if (stats == NULL) { \
		error("stats = NULL"); \
		return 3; \
		} \
	\
	itype elt;  /*current processing element*/ \
	size_t i; \
	\
	for (i = 0; i < size; i++) { \
		elt = in_array[i];	/*read the current element*/ \
		/*increment the corresponding number in output array*/ \
		++stats[elt - (MIN)]; \
		} \
	\
	return 0; \
}

extern int stats_uint8_array
	STATS_INT_ARRAY(uint8_t, 0)
extern int stats_int8_array
	STATS_INT_ARRAY(int8_t, INT8_MIN)
extern int stats_uint16_array
	STATS_INT_ARRAY(uint16_t, 0)
extern int stats_int16_array
	STATS_INT_ARRAY(int16_t, INT16_MIN)

#undef STATS_INT_ARRAY

//generic function for printing a numeric array------------------------------

#define PRINT_ARRAY(itype, COMMAND) \
(const itype *array, const size_t size) \
{ \
	/*check the arguments*/ \
	if (array == NULL) { \
		error("array = NULL"); \
		return 1; \
		} \
	if (size == 0) { \
		error("size = 0"); \
		return 2; \
```

```
            } \
      \
      size_t i; \
      \
      for (i = 0; i < size; i++) {\
            COMMAND; \
            printf(" "); \
            } \
      printf("\n"); \
      return 0; \
}

extern int print_uint8_array
      PRINT_ARRAY(uint8_t, printf("%"PRIu8, array[i]) )
extern int print_int8_array
      PRINT_ARRAY(int8_t, printf("%"PRIi8, array[i]) )
extern int print_uint16_array
      PRINT_ARRAY(uint16_t, printf("%"PRIu16, array[i]) )
extern int print_int16_array
      PRINT_ARRAY(int16_t, printf("%"PRIi16, array[i]) )
extern int print_uint32_array
      PRINT_ARRAY(uint32_t, printf("%"PRIu32, array[i]) )
extern int print_int32_array
      PRINT_ARRAY(int32_t, printf("%"PRIi32, array[i]) )
extern int print_uint64_array
      PRINT_ARRAY(uint64_t, printf("%"PRIu64, array[i]) )
extern int print_int64_array
      PRINT_ARRAY(int64_t, printf("%"PRIi64, array[i]) )
extern int print_mpz_array
      PRINT_ARRAY(mpz_t, mpz_out_str(stdout, 16, array[i]) )
extern int print_16_bytes_array
      PRINT_ARRAY(unsigned char, int j; for (j = 0; j < 16; j++)
printf("%02x", array[16*i+j]) )
extern int print_float_array
      PRINT_ARRAY(float, printf("%e", array[i]) )
extern int print_double_array
      PRINT_ARRAY(double, printf("%e", array[i]) )
extern int print_longd_array
      PRINT_ARRAY(long double, printf("%Le", array[i]) )

#undef PRINT_ARRAY
```

```
//generic function for conversion from signaling NaNs to quiet NaNs----------

/*during testing, it was found that conversion from signaling NaN to integer
and back gives quiet NaN, so this functions were written for ease of testing.
here we work with binary format, most significand exponent bit determines
does NaN is quiet (bit = 1) or signaling (bit = 0).

optimization note: turns out that just "for (i = 0; i < size; i++)
out_array[i] = in_array[i]" may be enough to convert all signaling NaNs to
quiet NaNs, but it may be not portable.

quote https://en.wikipedia.org/wiki/NaN:
In practice, the most significant bit of the significand field determined
whether a NaN is signaling or quiet. Two different implementations, with
reversed meanings, resulted:
- most processors (including those of the Intel and AMD's x86 family, the
Motorola 68000 family, the AIM PowerPC family, the ARM family, and the Sun
SPARC family) set the signaling/quiet bit to non-zero if the NaN is quiet,
and to zero if the NaN is signaling. Thus, on these processors, the bit
represents an 'is_quiet' flag;
- in NaNs generated by the PA-RISC and MIPS processors, the signaling/quiet
bit is zero if the NaN is quiet, and non-zero if the NaN is signaling. Thus,
on these processors, the bit represents an 'is_signaling' flag.

The 2008 revision of the IEEE 754 standard (IEEE 754-2008) makes formal
recommendations for the encoding of the signaling/quiet state.
- For binary formats, the most significant bit of the significand field
should be an 'is_quiet' flag. I.e. this bit is non-zero if the NaN is quiet,
and zero if the NaN is signaling.
- For decimal formats, whether binary or decimal encoded, a NaN is identified
by having the top five bits of the combination field after the sign bit set
to ones. The sixth bit of the field is the 'is_quiet' flag. The standard
follows the interpretation as an 'is_signaling' flag. I.e. the
signaling/quiet bit is zero if the NaN is quiet, and non-zero if the NaN is
signaling. A signaling NaN is quieted by clearing this sixth bit.*/

#define TO_QUIET_NANS(itype, atype, MASK) \
(const itype *in_array, itype *out_array, const size_t size) \
{ \
      /*check the arguments*/ \
```

```
	if (in_array == NULL) { \
		error("in_array = NULL"); \
		return 1; \
		} \
	if (out_array == NULL) { \
		error("out_array = NULL"); \
		return 2; \
		} \
	if (size == 0) { \
		error("size = 0"); \
		return 3; \
		} \
	\
	size_t i; \
	/*current processing element after conversion. we use union notation
here for access to bitwise operations.*/ \
	union { \
		itype fp; \
		atype i; \
		} ielt; \
	\
	for (i = 0; i < size; i++) \
		/*if current element is NaN then set is_quiet bit and copy it,
else just copy it*/ \
		if (isnan(in_array[i])) { \
			ielt.fp = in_array[i]; \
			ielt.i = ielt.i | (MASK); \
			out_array[i] = ielt.fp; \
			} \
		else \
			out_array[i] = in_array[i]; \
	\
	return 0; \
}

extern int to_quiet_nans_float
	TO_QUIET_NANS(float, uint32_t, 0x00400000)
extern int to_quiet_nans_double
	TO_QUIET_NANS(double, uint64_t, 0x0008000000000000)

#undef TO_QUIET_NANS
```

```
extern void test_init(void)
{
	//initialise the crypto library
	ERR_load_crypto_strings();
	OpenSSL_add_all_algorithms();
	//OPENSSL_config(NULL);
}

extern void test_deinit(void)
{
	//clean up
	EVP_cleanup();
	ERR_free_strings();
}

extern void OpenSSL_error(void)
{
	ERR_print_errors_fp(stderr);
	test_error();
}

extern void test_error(void)
{
	test_deinit();
	exit(1);
}

//encrypt a message
extern int encrypt(const unsigned char *plaintext, const size_t
plaintext_len, const unsigned char *key, const unsigned char *iv, unsigned
char *ciphertext)
{
	EVP_CIPHER_CTX *ctx;
	int len, ciphertext_len;

	//create and initialise the context
	if (!(ctx = EVP_CIPHER_CTX_new()))
		OpenSSL_error();
	//EVP_CIPHER_CTX_set_padding(ctx, 0);
```

```
	/*Initialise the encryption operation. IMPORTANT - ensure you use a key
and IV size appropriate for your cipher. In this example we are using 256 bit
AES (i.e. a 256 bit key). The IV size for *most* modes is the same as the
block size. For AES this is 128 bits.*/
	if (EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv) != 1)
		OpenSSL_error();
	EVP_CIPHER_CTX_set_padding(ctx, 0);
	/*We should disable the padding for plausible decryption with any
decryption key. The total amount of data encrypted or decrypted must then be
a multiple of the block size or an error will occur.*/

	/*Provide the message to be encrypted, and obtain the encrypted output.
EVP_EncryptUpdate can be called multiple times if necessary.*/
	if (EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len) !
= 1)
		OpenSSL_error();
	ciphertext_len = len;

	//Finalise the encryption. Further ciphertext bytes may be written at
this stage.
	if (EVP_EncryptFinal_ex(ctx, ciphertext + len, &len) != 1)
		OpenSSL_error();
	ciphertext_len += len;

	//clean up
	EVP_CIPHER_CTX_free(ctx);

	return ciphertext_len;
}

//decrypt a message
extern int decrypt(const unsigned char *ciphertext, const size_t
ciphertext_len, const unsigned char *key, const unsigned char *iv, unsigned
char *plaintext)
{
	EVP_CIPHER_CTX *ctx;
	int len, plaintext_len;

	//create and initialise the context
	if (!(ctx = EVP_CIPHER_CTX_new()))
```

```
            OpenSSL_error();

      /*Initialise the decryption operation. IMPORTANT - ensure you use a key
and IV size appropriate for your cipher. In this example we are using 256 bit
AES (i.e. a 256 bit key). The IV size for *most* modes is the same as the
block size. For AES this is 128 bits.*/
      if (EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv) != 1)
            OpenSSL_error();
      EVP_CIPHER_CTX_set_padding(ctx, 0); //disable padding

      /*Provide the message to be decrypted, and obtain the plaintext output.
EVP_DecryptUpdate can be called multiple times if necessary.*/
      if (EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext,
ciphertext_len) != 1)
            OpenSSL_error();
      plaintext_len = len;

      //Finalise the decryption. Further plaintext bytes may be written at
this stage.
      if (EVP_DecryptFinal_ex(ctx, plaintext + len, &len) != 1)
            OpenSSL_error();
      plaintext_len += len;

      //clean up
      EVP_CIPHER_CTX_free(ctx);

      return plaintext_len;
}
```

**tests/int_uniform/uint8.c (файлы int8.c, uint16.c, int16.c, uint32.c, int32.c, uint64.c, int64.c аналогичны)**

```
#include "../t_common.h"
#include "../../hdata/hd_int_uniform.h"

extern int main(void)
{
      //Set up the key and IV. Do I need to say to not hard code these in a
real application? :-)
//a 256 bit key
unsigned char *key = (unsigned char *)"01234567890123456789012345678901";
unsigned char *iv = (unsigned char *)"01234567890123456"; //a 128 bit IV
```

```
	int32_t i, j;
	size_t size;
	//current array size
	#define ITYPE uint8_t		//type for testing in this test unit
	#define PRI PRIu8		//macro for printing it
	#define OTYPE uint16_t		//type of container in this test unit
	#define BYTESIZE (size*sizeof(ITYPE))//current input array size in bytes
	const size_t maxsize = 1048576/sizeof(ITYPE); //maximum array size (1MB)
	//statistics on pseudorandom and output arrays
uint64_t in_stats[256] = {0}, out_stats[256] = {0}, long_stats[65536] = {0};
	ITYPE min, max, orig_array[maxsize], decoded_array[maxsize];
	//minimum and maximim in array
	OTYPE encoded_array[maxsize];
	FILE *fp;

	/*Buffer for ciphertext. Ensure the buffer is long enough for the
ciphertext which may be longer than the plaintext, dependant on the algorithm
and mode (for AES-256 in CBC mode we need one extra block)*/
	unsigned char ciphertext[2*256*sizeof(ITYPE)];
	size_t decryptedtext_len, ciphertext_len;	//their lengths

	//variables for bruteforce test
	uint16_t bfkey;			//current iteration number
	unsigned char big_key[32];	//current key

	test_init();


	//random data encoding, encryption, decryption, decoding----------------

	size = 256;
	//write a random numbers to original array
	randombytes((unsigned char *)orig_array, BYTESIZE);

	//let orig_array contain numbers from 100 to 200
	for (i = 0; i < size; i++) {
		//write a fresh random element to this position until it will be
between 0 and 201
		while (orig_array[i] > 201)
			randombytes((unsigned char *)(orig_array+i), sizeof(ITYPE));
		orig_array[i] = (orig_array[i] % 101) + 100;
		}
```

```
	get_uint8_minmax(orig_array, size, &min, &max);
	encode_uint8_uniform(orig_array, encoded_array, size, min, max);
	ciphertext_len = encrypt((unsigned char *)encoded_array, 2*BYTESIZE,
key, iv, ciphertext);
	decryptedtext_len = decrypt(ciphertext, ciphertext_len, key, iv,
(unsigned char *)encoded_array);
	decode_uint8_uniform(encoded_array, decoded_array, size, min, max);

	//compare result of decryption and original array
	if ( memcmp(orig_array, decoded_array, BYTESIZE) || (decryptedtext_len !
= 2*BYTESIZE) ) {
		error("orig_array and decoded_array are not the same");
		printf("size = %zu, decryptedtext_len = %zu\n", size,
decryptedtext_len);
		print_uint8_array(orig_array, 10);
		print_uint8_array(decoded_array, 10);
		test_error();
		}


	//bruteforce test (complexity equals 2^16) with statistics collection---

	size = 256;

	//encode and encrypt data
	encode_uint8_uniform(orig_array, encoded_array, size, 100, 200);
	ciphertext_len = encrypt((unsigned char *)encoded_array, 2*BYTESIZE,
key, iv, ciphertext);

	//let's try to bruteforce last 2 bytes of a key
	bfkey = 0;
	memcpy(big_key, key, 30); //suppose that we know 30 first bytes of key
	memset(out_stats, 0, sizeof(out_stats)); //initialize statistics arrays

	for (i = 0; i < 65536; i++) {
		//get current key for decryption
		memcpy((void *)(big_key+30), &bfkey, sizeof(bfkey));
		bfkey++;
		//try next key on next iteration
		decryptedtext_len = decrypt(ciphertext, ciphertext_len, big_key,
iv, (unsigned char *)encoded_array);
		decode_uint8_uniform(encoded_array, decoded_array, size, 100,
```

```
200);
            if (decryptedtext_len != 2*BYTESIZE) {
                  error("wrong decryptedtext_len value");
                  printf("size = %zu, decryptedtext_len = %zu\n", size,
decryptedtext_len);
                  test_error();
                  }
            //get a statistics on current bruteforce iteration
            stats_uint8_array(decoded_array, size, out_stats);
            /*
            for (j=0; j < 32; j++) { //print current key
                  printf("%02x", big_key[j]);   //output format is HEX-code
                  //place a space every 2 bytes
                  if ( (j+1) % 4 == 0) printf(" ");
                  }
            printf("\n");
            */
            }

      //write overall bruteforce statistics to file
      //try to open file for writing
      if ((fp = fopen("uint8_bruteforce.ods", "w")) == NULL) {
            error("can't open file 'uint8_bruteforce.ods' for writing");
            test_error();
            }

      //compare actual vs. ideal distributions of output array
      if (fprintf(fp, "\t=CHITEST(B102:B202;C102:C202)\n") < 0) {
            error("cannot write to 'uint8_bruteforce.ods' file");
            if (fclose(fp) == EOF)
                  perror("test: fclose error");
            test_error();
            }
      //write two columns to file: actual and ideal distribution for CHITEST
      for (i = 0; i <= UINT8_MAX; i++) {
            if ( (i < 100) || (i > 200) ) {
                  if (fprintf(fp, "%i\t%"PRIu64"\t%i\n", i, out_stats[i], 0) <
0) {
                        error("cannot write to 'uint8_bruteforce.ods' file");
                        if (fclose(fp) == EOF)
                              perror("test: fclose error");
                        test_error();
```

```
                        }
                  }
            else
                  /*166 111 = 65 536 (number of keys in brutforce) * 256 (size
of each decrypted text in elements) / 101 (number of possible array values
from 100 to 200) = 16 777 216 (total amount of numbers) / 101 (their possible
values) - expected result in out_stats*/
                  if (fprintf(fp, "%i\t%"PRIu64"\t%i\n", i, out_stats[i],
166111) < 0) {
                        error("cannot write to 'uint8_bruteforce.ods' file");
                        if (fclose(fp) == EOF)
                              perror("test: fclose error");
                        test_error();
                        }
            }
      //close file
      if (fclose(fp) == EOF) {
            perror("test: fclose error");
            test_error();
            }


      //random data encoding and decoding with statistics collection----------

      size = maxsize;
      //write a random numbers to original array
      randombytes((unsigned char *)orig_array, BYTESIZE);
      memset(in_stats, 0, sizeof(in_stats));   //initialize statistics arrays
      memset(out_stats, 0, sizeof(out_stats));
      //get a statistics on a pseudorandom numbers
      stats_uint8_array((uint8_t *)orig_array, BYTESIZE, in_stats);
      /*write a fresh random numbers to original array and get a statistics on
them again for fair comparison with 2*BYTESIZE encoded bytes below*/
      randombytes((unsigned char *)orig_array, BYTESIZE);
      stats_uint8_array((uint8_t *)orig_array, BYTESIZE, in_stats);

      //let orig_array contain uniformly distributed numbers from 100 to 219
      for (i = 0; i < size; i++) {
            //write a fresh random element to this position until it will be
between 0 and 239
            while (orig_array[i] > 239)
                  randombytes((unsigned char *)(orig_array+i), sizeof(ITYPE));
```

```
            orig_array[i] = (orig_array[i] % 120) + 100;
            }

      encode_uint8_uniform(orig_array, encoded_array, size, 100, 219);
      //get a statistics on an encoded array
      stats_uint8_array((uint8_t *)encoded_array, 2*BYTESIZE, out_stats);
      decode_uint8_uniform(encoded_array, decoded_array, size, 100, 219);
      if (memcmp(orig_array, decoded_array, BYTESIZE)) {
            error("orig_array and decoded_array are not the same");
            print_uint8_array(orig_array, 10);
            print_uint8_array(decoded_array, 10);
            test_error();
            }

      //write statistics to file
      //try to open file for writing
      if ((fp = fopen("uint8_encoding.ods", "w")) == NULL) {
            error("can't open file 'uint8_encoding.ods' for writing");
            test_error();
            }

      //compare pseudorandom vs. ideal, actual vs. ideal distributions
      if (fprintf(fp,
"\t=CHITEST(B2:B257;C2:C257)\t\t=CHITEST(D2:D257;E2:E257)\n") < 0) {
            error("cannot write to 'uint8_encoding.ods' file");
            if (fclose(fp) == EOF)
                  perror("test: fclose error");
            test_error();
            }
      //write four columns to file: pseudorandom and ideal, actual and ideal
distributions for CHITEST
      for (i = 0; i <= UINT8_MAX; i++) {
            if (fprintf(fp, "%i\t%"PRIu64"\t%i\t%"PRIu64"\t%i\n", i,
in_stats[i], BYTESIZE/128, out_stats[i], BYTESIZE/128) < 0) {
                  error("cannot write to 'uint8_encoding.ods' file");
                  if (fclose(fp) == EOF)
                        perror("test: fclose error");
                  test_error();
                  }
            }
      //close file
      if (fclose(fp) == EOF) {
```

```
            perror("test: fclose error");
            test_error();
            }


      //random data encoding with advanced statistics collection (slow!)------

      size = 256;
      memset(long_stats, 0, sizeof(long_stats));//initialize statistics array
      for (i = 0; i < 65536; i++) {
            //write a random numbers to original array
            randombytes((unsigned char *)orig_array, BYTESIZE);

      //let orig_array contain uniformly distributed numbers from 100 to 199
            for (j = 0; j < size; j++) {
                  //write a fresh random byte to this position until it will
be between 0 and 199
                  while (orig_array[j] > 199)
                        randombytes((unsigned char *)(orig_array+j),
sizeof(ITYPE));
                  orig_array[j] = (orig_array[j] % 100) + 100;
                  }

            encode_uint8_uniform(orig_array, encoded_array, size, 100, 199);
            stats_uint16_array(encoded_array, size, long_stats);
            }

      //write overall statistics to file
      //try to open file for writing
      if ((fp = fopen("uint8_encoding2.ods", "w")) == NULL) {
            error("can't open file 'uint8_encoding2.ods' for writing");
            test_error();
            }

      //compare actual vs. ideal distributions of output array
      if (fprintf(fp, "\t=CHITEST(B2:B65537;C2:C65537)\n") < 0) {
            error("cannot write to 'uint8_encoding2.ods' file");
            if (fclose(fp) == EOF)
                  perror("test: fclose error");
            test_error();
            }
      //write two columns to file: actual and ideal distribution for CHITEST
      for (i = 0; i <= UINT16_MAX; i++) {
```

```
            /*256 = 65 536 (number of encodings) * 256 (size of each array for
encoding) / 65536 (number of possible array values from 0 to 65 535) = 16 777
216 (total amount of numbers) / 65536 (their possible values) - expected
result in long_stats*/
            if (fprintf(fp, "%i\t%"PRIu64"\t%i\n", i, long_stats[i], 256) < 0)
{
                  error("cannot write to 'uint8_encoding2.ods' file");
                  if (fclose(fp) == EOF)
                        perror("test: fclose error");
                  test_error();
                  }
            }
      //close file
      if (fclose(fp) == EOF) {
            perror("test: fclose error");
            test_error();
            }


      //random encoding and decoding----------------------------------------

      for (size = 1; size < 256; size++) {
            //write a random numbers to original array
            randombytes((unsigned char *)orig_array, BYTESIZE);
            get_uint8_minmax(orig_array, size, &min, &max);
            encode_uint8_uniform(orig_array, encoded_array, size, min, max);
            decode_uint8_uniform(encoded_array, decoded_array, size, min,
max);
            if (memcmp(orig_array, decoded_array, BYTESIZE)) {
                  error("orig_array and decoded_array are not the same");
                  print_uint8_array(orig_array, 10);
                  print_uint8_array(decoded_array, 10);
                  test_error();
                  }
            }


      //fixed general cases-------------------------------------------------

      size = 5;
      orig_array[0] = 20;
      orig_array[1] = 25;
```

```
	orig_array[2] = 30;
	orig_array[3] = 35;
	orig_array[4] = 40;

	printf("Original array:\n");  //print it
	print_uint8_array(orig_array, size);
	get_uint8_minmax(orig_array, size, &min, &max);

	printf("min = %"PRI", max = %"PRI":\n", min, max);
	encode_uint8_uniform(orig_array, encoded_array, size, min, max);
	print_uint16_array(encoded_array, size);
	decode_uint8_uniform(encoded_array, decoded_array, size, min, max);
//if original and decoded arrays are not equal then print a decoded array too
	if (memcmp(orig_array, decoded_array, BYTESIZE))
		print_uint8_array(decoded_array, size);

	printf("min = 15, max = 45:\n");
	encode_uint8_uniform(orig_array, encoded_array, size, 15, 45);
	print_uint16_array(encoded_array, size);
	decode_uint8_uniform(encoded_array, decoded_array, size, 15, 45);
	if (memcmp(orig_array, decoded_array, BYTESIZE))
		print_uint8_array(decoded_array, size);

	printf("min = 0, max = %"PRI":\n", UINT8_MAX);
	encode_uint8_uniform(orig_array, encoded_array, size, 0, UINT8_MAX);
	print_uint16_array(encoded_array, size);
	decode_uint8_uniform(encoded_array, decoded_array, size, 0, UINT8_MAX);
	if (memcmp(orig_array, decoded_array, BYTESIZE))
		print_uint8_array(decoded_array, size);
	printf("\n");


	//fixed special cases-------------------------------------------------

	size = 5;
	orig_array[0] = 20;
	orig_array[1] = 20;
	orig_array[2] = 20;
	orig_array[3] = 20;
	orig_array[4] = 20;

	printf("Original array:\n");
	print_uint8_array(orig_array, size);
```

```
get_uint8_minmax(orig_array, size, &min, &max);

printf("min = %"PRI", max = %"PRI":\n", min, max);
encode_uint8_uniform(orig_array, encoded_array, size, min, max);
print_uint16_array(encoded_array, size);
decode_uint8_uniform(encoded_array, decoded_array, size, min, max);
if (memcmp(orig_array, decoded_array, BYTESIZE))
      print_uint8_array(decoded_array, size);

printf("min = 0, max = %"PRI":\n", UINT8_MAX);
encode_uint8_uniform(orig_array, encoded_array, size, 0, UINT8_MAX);
print_uint16_array(encoded_array, size);
decode_uint8_uniform(encoded_array, decoded_array, size, 0, UINT8_MAX);
if (memcmp(orig_array, decoded_array, BYTESIZE))
      print_uint8_array(decoded_array, size);
printf("\n");


//wrong parameters-----------------------------------------------------

get_uint8_minmax(NULL, 0, NULL, NULL);
get_uint8_minmax(orig_array, 0, NULL, NULL);
get_uint8_minmax(orig_array, 1, NULL, NULL);
get_uint8_minmax(orig_array, 1, &min, NULL);
printf("\n");

encode_uint8_uniform(NULL, NULL, 0, 0, 0);
encode_uint8_uniform(orig_array, NULL, 0, 0, 0);
encode_uint8_uniform(orig_array, encoded_array, 0, 0, 0);
encode_uint8_uniform(orig_array, encoded_array, 1, 2, 1);
encode_uint8_uniform(orig_array, encoded_array, size, 21, 100);
encode_uint8_uniform(orig_array, encoded_array, size, 0, 19);
printf("\n");

decode_uint8_uniform(NULL, NULL, 0, 0, 0);
decode_uint8_uniform(encoded_array, NULL, 0, 0, 0);
decode_uint8_uniform(encoded_array, orig_array, 0, 0, 0);
decode_uint8_uniform(encoded_array, orig_array, 1, 2, 1);
printf("\n");

stats_uint8_array(NULL, 0, NULL);
stats_uint8_array(orig_array, 0, NULL);
stats_uint8_array(orig_array, 1, NULL);
```

```
	printf("\n");

	print_uint8_array(NULL, 0);
	print_uint8_array(orig_array, 0);


	#undef ITYPE
	#undef PRI
	#undef OTYPE
	#undef BYTESIZE
	test_deinit();

	return 0;
}
```

**tests/int_uniform/math.c**

```
#include <stdio.h>
#include <inttypes.h>
#include <math.h>

/*this test computationally proofs a following equation: group_num =
ceill( (long double)(OTYPE_MAX+1) / group_size) = OTYPE_MAX / group_size + 1.
in this test, OTYPE_MAX is UINT16_MAX, UINT32_MAX, and UINT64_MAX,
respectively. group_size is a value from 2 to (UINT8_MAX+1), (UINT16_MAX+1),
and (UINT32_MAX+1), respectively. group_size can also be 1 in library, but in
this case we don't use group_num variable at all, so we don't care about
integer overflow there. when we use floating point numbers, we use maximum
accuracy available in C: conversion to long double and ceill() function
exactly for that type.*/

#define FASTGROUPNUM(a, b, res) \
	do { \
		res = a / b + 1; \
		}\
	while (0)

extern int main(void)
{
	uint64_t i;

	for (i = 2; i <= (uint16_t)UINT8_MAX+1; i++) {
```

```
        uint16_t v1, v2;
        v1 = ceill( ((long double)UINT16_MAX + 1) / i);
        FASTGROUPNUM(UINT16_MAX, i, v2);
        if (v1 != v2) {
            printf("%"PRIu64": %"PRIu16" %"PRIu16"\n", i, v1, v2);
            goto end;
            }
        }
    printf("Tests for 8-bit integers successfully passed\n");

    for (i = 2; i <= (uint32_t)UINT16_MAX+1; i++) {
        uint32_t v1, v2;
        v1 = ceill( ((long double)UINT32_MAX + 1) / i);
        FASTGROUPNUM(UINT32_MAX, i, v2);
        if (v1 != v2) {
            printf("%"PRIu64": %"PRIu32" %"PRIu32"\n", i, v1, v2);
            goto end;
            }
        }
    printf("Tests for 16-bit integers successfully passed\n");

    //warning: slow!
    for (i = 2; i <= (uint64_t)UINT32_MAX+1; i++) {
        uint64_t v1, v2;
        v1 = ceill( ((long double)UINT64_MAX + 1) / i);
        FASTGROUPNUM(UINT64_MAX, i, v2);
        if (v1 != v2) {
            printf("%"PRIu64": %"PRIu64" %"PRIu64"\n", i, v1, v2);
            goto end;
            }
        //show progress
        /*else
            if (i % 1000000 == 0)
                printf("%"PRIu64"\n", i);*/
        }
    printf("Tests for 32-bit integers successfully passed\n");

    end:
    return 0;
}
```

**tests/int_arbitrary/uint8.c (файлы int8.c, uint16.c, int16.c, uint32.c, int32.c, uint64.c, int64.c аналогичны)**

```
#include "../t_common.h"
#include "../../hdata/hd_int_arbitrary.h"

extern int main(void)
{
	#define ITYPE uint8_t			//type for testing in this test unit
	#define BYTESIZE (size*sizeof(ITYPE))//current input array size in bytes

	const ITYPE orig_array[] = {210, 210, 210, 212, 212, 210, 210, 210, 212,
212};
	const size_t size = 10;
	ITYPE decoded_array[size], min, max;
	void *encoded_array;

	uint32_t weights[] = {3, 0, 2};
	int rv;

	//fixed general cases-----------------------------------------------------

	get_uint8_minmax(orig_array, size, &min, &max);

	if ((rv = encode_uint8_arbitrary(orig_array, &encoded_array, size, min,
max, weights)) != 2) {
		error("unexpected output type");
		printf("%d\n", rv);
		test_error();
		}
	decode_uint8_arbitrary(encoded_array, decoded_array, size, min, max,
weights);

	free(encoded_array);

	if (memcmp(orig_array, decoded_array, BYTESIZE)) {
		error("orig_array and decoded_array are not the same");
		print_uint8_array(orig_array, size);
		print_uint8_array(decoded_array, size);
		test_error();
		}
```

```
      weights[0] = 256;

      if ((rv = encode_uint8_arbitrary(orig_array, &encoded_array, size, min,
max, weights)) != 4) {
            error("unexpected output type");
            printf("%d\n", rv);
            test_error();
            }
      decode_uint8_arbitrary(encoded_array, decoded_array, size, min, max,
weights);

      free(encoded_array);

      if (memcmp(orig_array, decoded_array, BYTESIZE)) {
            error("orig_array and decoded_array are not the same");
            print_uint8_array(orig_array, size);
            print_uint8_array(decoded_array, size);
            test_error();
            }


      weights[0] = 65536;

      if ((rv = encode_uint8_arbitrary(orig_array, &encoded_array, size, min,
max, weights)) != 8) {
            error("unexpected output type");
            printf("%d\n", rv);
            test_error();
            }
      decode_uint8_arbitrary(encoded_array, decoded_array, size, min, max,
weights);

      free(encoded_array);

      if (memcmp(orig_array, decoded_array, BYTESIZE)) {
            error("orig_array and decoded_array are not the same");
            print_uint8_array(orig_array, size);
            print_uint8_array(decoded_array, size);
            test_error();
            }

      weights[0] = 4294967295;
```

```
      if ((rv = encode_uint8_arbitrary(orig_array, &encoded_array, size, min,
max, weights)) != -1) {
            error("unexpected output type");
            printf("%d\n", rv);
            test_error();
            }

      #undef ITYPE
      #undef BYTESIZE

      return 0;
}
```

**tests/fp_uniform/float.c (файлы double.c аналогичен)**

```
#include "../t_common.h"
#include "../../hdata/hd_fp_uniform.h"

extern int main(void)
{

      int32_t i;
      size_t size;                        //current array size
      #define TYPE float                  //type for testing in this test unit
      #define PRI "%g"                    //macro for printing it
      #define BYTESIZE (size*sizeof(TYPE))//current input array size in bytes
      const size_t maxsize = 5000;        //maximum array size
      //minimum and maximim in array
      TYPE orig_array[maxsize], decoded_array[maxsize];
      uint32_t encoded_array[maxsize];
      //uint8_t bad[] = {1, 0, 128, 255}; //signaling NaN

      test_init();


      //random encoding and decoding-----------------------------------------

      for (size = 1; size < maxsize; size++) {
            //write a random numbers to original array
            randombytes((unsigned char *)orig_array, BYTESIZE);
            /*memcpy(orig_array, bad, sizeof(TYPE));
            print_uint8_array((uint8_t *)orig_array, sizeof(TYPE));*/
```

```
            //convert signaling NaNs (if any) to quiet NaNs to avoid errors
with comparsion
            to_quiet_nans_float(orig_array, orig_array, size);
            //print_uint8_array((uint8_t *)orig_array, sizeof(TYPE));
            float_to_uint32_uniform(orig_array, encoded_array, size);
            uint32_to_float_uniform(encoded_array, decoded_array, size);
            if (memcmp(orig_array, decoded_array, BYTESIZE)) {
                  error("orig_array and decoded_array are not the same");
                  print_float_array(orig_array, size);
                  print_float_array(decoded_array, size);
                  print_uint8_array((uint8_t *)orig_array, BYTESIZE);
                  print_uint8_array((uint8_t *)decoded_array, BYTESIZE);
                  test_error();
                  }
            }


      //comparison test-----------------------------------------------------

      /*we tried to encode data such way that if (fl1 < fl2) then (int1 <
int2), if (fl1 = fl2) then (int1 = int2), if (fl1 > fl2) then (int1 > int2),
i.e. each integer value will contain a number of corresponding fp number in
sequence of all possible fp numbers. let's check do we succeeded */

      size = 10;

      for (i = 0; i < size-1; i++)
            if (encoded_array[i] > encoded_array[i+1])
                  printf(PRI" > "PRI"\n", orig_array[i], orig_array[i+1]);
            else if (encoded_array[i] < encoded_array[i+1])
                  printf(PRI" < "PRI"\n", orig_array[i], orig_array[i+1]);
            else
                  printf(PRI" = "PRI"\n", orig_array[i], orig_array[i+1]);

      printf("\n");


      //wrong parameters----------------------------------------------------

      float_to_uint32_uniform(NULL, NULL, 0);
      float_to_uint32_uniform(orig_array, NULL, 0);
      float_to_uint32_uniform(orig_array, encoded_array, 0);
      printf("\n");
```

```
	uint32_to_float_uniform(NULL, NULL, 0);
	uint32_to_float_uniform(encoded_array, NULL, 0);
	uint32_to_float_uniform(encoded_array, orig_array, 0);
	printf("\n");

	to_quiet_nans_float(NULL, NULL, 0);
	to_quiet_nans_float(orig_array, NULL, 0);
	to_quiet_nans_float(orig_array, decoded_array, 0);
	printf("\n");

	print_float_array(NULL, 0);
	print_float_array(orig_array, 0);



	#undef TYPE
	#undef PRI
	#undef BYTESIZE
	test_deinit();

	return 0;

}
```

**tests/fp_uniform/compatibility.c**

```
#include <stdio.h>
#include <inttypes.h>
#include <float.h>

/*unfortunately, implementation of floating-point arithmetic is platform-
dependent. Floating-point related code in this library based on some
assumptions about floating-point arithmetic implementation on target
platform. this file tests are this assumptions actually true on target
platform. if they are false then floating-point related code should be
appropriately edited.

in particular we check that:
- float is IEEE 754 single-precision binary floating-point format (binary32)
sign bit: 1 bit, exponent width: 8 bits, significand precision: 24 bits (23
explicitly stored)
- double is IEEE 754 double-precision binary floating-point format (binary64)
```

```
sign bit: 1 bit, exponent width: 11 bits, significand precision: 53 bits (52
explicitly stored)
- long double is IEEE 754 extended precision format, esp. x86 Extended
Precision Format
sign bit: 1 bit, exponent width: 15 bits, significand precision: 64 bits (64
explicitly stored)
as wikipedia notes, this type is sometimes stored as 12 or 16 bytes to
maintain data structure alignment author's PC uses little-endian encoding*/

#define CHECK(CONDITION) \
      do { \
            if (CONDITION) { \
                  printf("Error: " #CONDITION "\n"); \
                  return 1; \
                  } \
            } while (0)

extern int main(void)
{

      int i;

      CHECK(FLT_RADIX != 2);

      CHECK(FLT_MANT_DIG != 24);
      CHECK(sizeof(float) != 4);

      CHECK(DBL_MANT_DIG != 53);
      CHECK(sizeof(double) != 8);

      /*
      CHECK(LDBL_MANT_DIG != 64);
      CHECK(sizeof(long double) != 12);
      */

      union {
            float fl;
            uint32_t i;
            unsigned char ch[4];
            } hfloat;
```

```
hfloat.fl = -0.15625;

uint16_t sexp1 = hfloat.i >> 23;
uint32_t frac1 = hfloat.i & 0x007FFFFF;

CHECK(sexp1 != 380);
CHECK(frac1 != 2097152);
for (i = 0; i < sizeof(float); i++)
      printf("%02x", hfloat.ch[i]); //output format is HEX-code
//my output: 000020be
printf("\n");

union {
      double fl;
      uint64_t i;
      unsigned char ch[8];
      } hdouble;

hdouble.fl = -0.15625;

uint16_t sexp2 = hdouble.i >> 52;
uint64_t frac2 = hdouble.i & 0x000FFFFFFFFFFFF;

CHECK(sexp2 != 3068);
CHECK(frac2 != 1125899906842624);
for (i = 0; i < sizeof(double); i++)
      printf("%02x", hdouble.ch[i]);
//my output: 000000000000c4bf
printf("\n");

/*
union {
      long double fl;
      uint64_t i[2];
      unsigned char ch[12];
      } hlongd;

hlongd.fl = -0.15625;

uint16_t sexp3 = hlongd.i[0] & 0x000000000000FFFF;
uint64_t frac3 = hlongd.i[1];
```

```
	//CHECKs go here
	for (i = 0; i < sizeof(long double); i++)
		printf("%02x", hlongd.ch[i]);
	//my output: 00000000000000a0fcbf61b7
	printf("\n%"PRIu64" %"PRIu64"\n", hlongd.i[0], hlongd.i[1]);
	*/

	/*
	__fp16
	__float80/128
	_Decimal32/64/128
	_Complex int/float/double/long double
	*/

	//else all tests above succeeded
	printf("All tests passed\n");
	return 0;

}

#undef CHECK
```

# Приложение В.
# Доказательство утверждения из подраздела 5.2

Для понимания последующих рассуждений от читателя требуется знакомство с представлением чисел с плавающей запятой, которое было описано в подразделе 5.1.

Числа с плавающей запятой типов *binary32* и *binary64* состоят из трёх частей: знакового бита, битов порядка и битов мантиссы. На время забудем о знаковом бите, то есть будем рассматривать числа по модулю.

Пусть все биты порядка некоторого числа это нули. Если все биты мантиссы это нули, то это число – положительный ноль, то есть минимально возможное по модулю число. Представим себе последовательность чисел с плавающей запятой, у которых порядок по-прежнему состоит только из нулей, а мантисса последовательно проходит все значения от 1 до максимально возможного. Несложно увидеть, что эта последовательность денормализованных чисел будет упорядочена по возрастанию, поскольку будет иметь вид (2 ^ минимально возможный порядок) * 0.мантисса = константа * 0.мантисса, где мантисса будет возрастать с возрастанием номера элемента. Последовательность, состоящая из положительного нуля и денормализованных чисел, будет упорядочена по возрастанию, так как наименьшее денормализованное число, равное (2 ^ минимально возможный порядок) * 0.000..001, больше положительного нуля, а последовательность денормализованных чисел упорядочена.

Представим себе последовательность чисел с плавающей запятой, у которых имеется фиксированный порядок (отличный от случаев «все нули» и «все единицы»), а мантисса последовательно проходит все значения от 0 до максимально возможного. Аналогично со случаем выше, несложно увидеть, что эта последовательность нормализованных чисел будет упорядочена по возрастанию, поскольку будет иметь вид ( 2 ^ (порядок + минимально возможный порядок – 1) ) * 1.мантисса = константа * 1.мантисса, где мантисса будет возрастать с возрастанием номера элемента.

Составим новую последовательность (во избежание путаницы, назовём её суперпоследовательностью) из таких последовательностей нормализованных чисел: пусть порядок последовательно проходит все значения от 1 до (максимально возможного – 1). Такая суперпоследовательность также будет упорядочена по возрастанию, что несложно доказать. Рассмотрим две соседние последовательности с номерами n и (n + 1) в суперпоследовательности. Максимальное значение последовательности n (её последний член) будет равен (2 ^ порядок1) * 1.мантисса, минимальное значение последовательности (n + 1) (её первый член) будет равен ( 2 ^ (порядок1 + 1) ) * 1.0 = (2 ^ порядок1) * 2 > (2 ^ порядок1) * 1.мантисса, то есть две этих последовательности упорядоченны между собой. Поскольку все последовательности внутри себя упорядочены по возрастанию, а каждые две соседние последовательности в суперпоследовательности упорядочены по возрастанию между собой, то суперпоследовательность будет упорядочена по возрастанию.

Составим последовательность, состоящую из двух ранее рассмотренных: во-первых, из последовательности положительного нуля и денормализованных чисел, а во-вторых, из суперпоследовательности всех нормализованных чисел. Результат также будет упорядочен по возрастанию, поскольку самое больше число первой последовательности будет иметь вид (2 ^ минимально возможный порядок) * 0.мантисса, а минимальное число второй последовательности ( 2 ^ (1 + минимально возможный порядок – 1) ) * 1.0 = (2 ^ минимально возможный порядок) * 1.0 > (2 ^ минимально возможный порядок) * 0.мантисса, а каждая из двух начальных последовательностей упорядочена по возрастанию.

Пусть все биты порядка некоторого числа это единицы. Если все биты мантиссы это нули, то это число – положительная бесконечность, то есть максимально возможное по модулю число. Последовательность, состоящая во-первых из последовательности положительного нуля, денормализованных

чисел, суперпоследовательности всех нормализованных чисел, а во-вторых из положительной бесконечности, будет упорядочена по возрастанию, так как наибольшее нормализованное число, равное (2 ^ максимально возможный порядок) * 1.мантисса, меньше положительной бесконечности, а последовательность, состоящая из положительного нуля, денормализованных чисел и суперпоследовательности всех нормализованных чисел, упорядочена по возрастанию.

Представим себе последовательность чисел с плавающей запятой, у которых порядок по-прежнему состоит только из единиц, а мантисса последовательно проходит все значения от 1 до максимально возможного. Несложно увидеть, что эта последовательность *NaN*-значений будет упорядочена по возрастанию, поскольку изменяются лишь биты мантиссы, которая будет возрастать с возрастанием номера элемента. В большинстве компьютеров первая половина этой последовательности будет отвечать за сигнальные *NaN*-значения с возрастающими от 1 до максимально возможного значениями полезной нагрузки, а вторая половина – за тихие *NaN*-значения с возрастающими от 0 до максимально возможного значениями полезной нагрузки [24].

*NaN*-значения не являются числами, поэтому сравнивать их с другими ранее рассмотренными числами и даже между собой некорректно. Это оправдывает тот факт, что эти значения находятся в кодовом пространстве после положительной бесконечности и именно в таком порядке (впрочем, вполне логичном).

Что мы хотели доказать? Если у нас есть значения с плавающей запятой *fp1* и *fp2*, то после перевода их соответственно в номера *int1* и *int2* этих чисел в последовательности всех возможных чисел с плавающей запятой этого типа выполняются следующие условия:

– если *fp1* > *fp2*, то *int1* > *int2*;

– если *fp1* < *fp2*, то *int1* < *int2*;

– если *fp1* = *fp2*, то *int1* = *int2*.

Эти условия выполняются благодаря доказанной выше упорядоченности по возрастанию, если мы не учитываем знаки и соглашаемся с тем, что после положительной бесконечности идут упорядоченные между собой сигнальные *NaN*-значения, а после них – упорядоченные между собой тихие *NaN*-значения. Для завершения работы осталось лишь учесть знаки, что и было сделано в подразделе 5.2 магистерской диссертации.