

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет (институт)
информационных технологий
и электроники
(наименование)

Направление подготовки

Магистерская программа

Кафедра
«Нано- и микроэлектроника»
(наименование)

11.04.04. «Электроника и
наноэлектроника»
(код и наименование)

Нано- и микроэлектроника
(наименование)

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Мандатная защита на основе контроля состояния памяти с помощью
хеш-функции

Студент 23ПДм1
номер группы

Сухов Иван Викторович
подпись, дата 03.06.25 ФИО полностью

Научный руководитель
заведующий кафедрой, ПГУ
должность, наименование организации

И.А. Пронин
подпись, дата 9.06.25 инициалы, фамилия

Рецензент
начальник отдела,
АО «ПО «Электроприбор»
должность, наименование организации

А.В. Комиссаров
подпись, дата 06.06.25 инициалы, фамилия

Нормоконтролер
ст. преподаватель
должность

Т. Н. Рыжова
подпись, дата 08.06.25 инициалы, фамилия

Работа допущена к защите в ГЭК (протокол заседания кафедры от 09.06.25 № 10)

Заведующий кафедрой НимЭ

ПГУ
должность, наименование организации

И.А.Пронин
подпись, дата 9.06.25 инициалы, фамилия

Работа защищена с оценкой отл. (протокол ГЭК от 18.06.25 № 1)

Секретарь ГЭК

доцент
должность, наименование организации

А.А. Вольсков
подпись, дата 18.06.25 инициалы, фамилия

РЕФЕРАТ

Пояснительная записка состоит из 85 страниц, 6 рисунков, 4 таблиц, списка использованных источников из 45 наименований и 5 приложений.

Ключевые слова: параллельный расчет *CRC*, многоядерная архитектура, ПО, хеш-функции, высоконагруженный вычислитель, защита от *glitch*-атак, *LVA*, *SOVA*, *MAC*, *DAC*, *RBAC*, *ABAC*, контроль состояния памяти

Цель: Исследовать вопрос реализации мандатной защиты на основе контроля состояния памяти с помощью хеш-функции

Задачи:

1. Выполнить аналитический обзор научно-технической литературы.
2. Разработать методику контроля состояния памяти с помощью хеш-функции *CRC*.
3. Разработать алгоритм расчета хеш-функции *CRC* для многоядерного процессора.
4. Разработать консольное приложение для реализации различных методов расчета хеш-функции *CRC*.
5. Выполнить верификацию разработанного приложения на различной аппаратной платформе (минимум 3-х ПЭВМ) на рандомизированном наборе данных файлового типа.
6. Определить критерий эффективности и провести сравнительный анализ методов расчета хеш-функции *CRC* по этому критерию.
7. Выработать предложение по реализации мандатной защиты на основе контроля состояния памяти с помощью хеш-функции *CRC*.

В настоящее время вопрос обеспечения эффективной защиты является актуальной проблемой, большинство успешных взломов программного обеспечения (ПО) какого-либо оборудования или систем управления, реализуются с применением не декларированных возможностей, реализуемых как с помощью аппаратных, так и программных методов, им необходимо противодействовать.

В связи с вероятностью вредоносного вмешательства в программное обеспечение какого-либо оборудования или системы управления необходимо осуществлять контроль состояния долговременной и кратковременной памяти с помощью сторонних механизмов, в частности хеш- функции *CRC* (*CyclicRedundancyCheck*).

Основная идея *CRC* заключается в добавлении в конец сообщения значения контрольной суммы фиксированной длины. Эта контрольная сумма рассчитывается на основе содержимого сообщения с помощью математического алгоритма и передаётся вместе с сообщением.

Когда сообщение получено, получатель может повторно вычислить контрольную сумму с помощью того же алгоритма и сравнить её с контрольной суммой, передаваемой вместе с сообщением. Если два значения совпадают, получатель может быть уверен, что сообщение не было повреждено во время передачи.

Расчет хеш - функции предлагается вычислять в произвольный момент времени, тогда вероятность подмены данных будет низкой, в отличие от алгоритмов вычисления осуществляющий контроль через определенное время, что позволяет осуществлять подмену результата хеш - функции злоумышленниками.

В основе современных систем и оборудования лежит многоядерный процессор, для использования преимуществ многоядерных систем, необходимо использовать параллельный алгоритм для ускорения вычисления хеш-функции, с помощью которой осуществляется контроль состояния памяти.

Полученные результаты: консольное приложение и выработанные предложения по мандатной защите на основе контроля состояния памяти с помощью хеш- функции *CRC*.

СОДЕРЖАНИЕ

Введение.....	5
1 Аналитический обзор научно-технической литературы.....	8
1.1 Системы разграничения доступа.....	8
1.2. Роль памяти в обеспечении безопасности.....	25
1.3. Хеш-функции в задачах информационной безопасности.....	28
1.4. Обзор существующих подходов к контролю состояния памяти	43
1.5 Средства проверки целостности состояния памяти	47
2 Разработка методики контроля состояния памяти с помощью хеш-функции <i>CRC</i>	51
2.1 Разработка алгоритмов расчета хеш-функции <i>CRC</i>	51
2.2 Разработка консольного приложения для реализации различных методов расчета хеш-функции <i>CRC</i>	55
2.3 Выработка предложения по реализации мандатной защиты на основе контроля состояния памяти с помощью хеш- функции <i>CRC</i>	65
3 Техника безопасности и охрана труда офисного сотрудника	74
3.1 Идентификация потенциальных опасностей и вредных факторов. Микроклимат рабочей зоны.....	74
3.2 Освещение и шумовая нагрузка. Влияние электромагнитных и электростатических полей.....	75
3.3 Электробезопасность. Пожарная безопасность	76
Заключение	78
Список использованных источников	81
Приложение А.Задание на выпускную квалификационную работу	86
Приложение Б.Код консольного приложения для реализации различных методов расчета хеш-функции <i>CRC</i> на языке <i>C</i>	90
Приложение В.Список опубликованных статей.....	95
Приложение Г.Заявление и протокол проверки ВКР на оригинальность в системе«Антиплагиат.ВУЗ».....	97
Приложение Д.Презентация ВКР	101

ВВЕДЕНИЕ

В настоящее время вопрос обеспечения эффективной защиты является актуальной проблемой, большинство успешных взломов программного обеспечения (ПО) какого-либо оборудования или систем управления, реализуются с применением не декларированных возможностей, реализуемых как с помощью аппаратных, так и программных методов, им необходимо противодействовать.

Не декларированные возможности – это последовательность инструкций процессора, которые скрытно внедряются в атакуемую систему и предоставляют нарушителю скрытый доступ к ресурсам атакуемой системы, вносят уязвимость в её подсистему безопасности, противодействуют:

- антивирусному ПО;
- пакетным фильтрам;
- системам обнаружения атак.

Компьютерные вирусы и сетевые черви являются частными случаями не декларированных возможностей.

Задача обеспечения безопасности информации, раньше решалась при помощи использования межсетевых экранов и разграничения доступа. Сейчас этих технологий недостаточно, поскольку любая информация, которая имеет финансовую, посредническую, военную, политическую, научную ценность, хранится в компьютере и, следовательно, подвергается угрозе. Дополнительным риском становится возможность перехвата управления критическими объектами информационной инфраструктуры. Согласно статистическим данным, более 80% компаний несут финансовые убытки из-за нарушения целостности и конфиденциальности используемых данных

В связи с вероятностью вредоносного вмешательства в программное обеспечение какого-либо оборудования или системы управления необходимо

осуществлять контроль состояния долговременной и кратковременной памяти с помощью сторонних механизмов, в частности хеш- функции *CRC* (*CyclicRedundancyCheck*).

Циклическая проверка на избыточность (*CRC*) это не криптографическая хеш-функция, которая используется для обнаружения ошибок при передаче данных и вредоносного вмешательства. Она является быстрой и эффективной, но ее нельзя использовать с точки зрения криптографии. Вычисление криптографических функций требуют большого объема вычислительных ресурсов и не подразумевает распараллеливание данных. С целью уменьшения объема вычислений и обнаружения атак на ранней стадии применяется контроль состояния долговременной и кратковременной памяти с помощью, в хеш- функции *CRC*. В случае несовпадения контрольной суммы *CRC* инициируются криптографические процедуры.

Основная идея *CRC* заключается в добавлении в конец сообщения значения контрольной суммы фиксированной длины. Эта контрольная сумма рассчитывается на основе содержимого сообщения с помощью математического алгоритма и передаётся вместе с сообщением.

Когда сообщение получено, получатель может повторно вычислить контрольную сумму с помощью того же алгоритма и сравнить её с контрольной суммой, передаваемой вместе с сообщением. Если два значения совпадают, получатель может быть уверен, что сообщение не было повреждено во время передачи.

Расчет хеш - функции предлагается вычислять в произвольный момент времени, тогда вероятность подмены данных будет низкой, в отличие от алгоритмов вычисления осуществляющий контроль через определенное время, что позволяет осуществлять подмену результата хеш - функции злоумышленниками.

В основе современных систем и оборудования лежит многоядерный процессор, для использования преимуществ многоядерных систем, необходимо использовать параллельный алгоритм для ускорения вычисления хеш-функции, с помощью которой осуществляется контроль состояния памяти.

Цель: Исследовать вопрос реализации мандатной защиты на основе контроля состояния памяти с помощью хеш-функции

Задачи:

1. Выполнить аналитический обзор научно-технической литературы.
2. Разработать методику контроля состояния памяти с помощью хеш-функции *CRC*.
3. Разработать алгоритм расчета хеш-функции *CRC* для многоядерного процессора.
4. Разработать консольное приложение для реализации различных методов расчета хеш-функции *CRC*.
5. Выполнить верификацию разработанного приложения на различной аппаратной платформе (минимум 3-х ПЭВМ) на рандомизированном наборе данных файлового типа.
6. Определить критерий эффективности и провести сравнительный анализ методов расчета хеш-функции *CRC* по этому критерию.
7. Выработать предложение по реализации мандатной защиты на основе контроля состояния памяти с помощью хеш-функции *CRC*.

Объект исследования: вопрос реализации мандатной защиты на основе контроля состояния памяти с помощью хеш-функции *CRC*

Предмет исследования: разработка методики параллельного расчета хеш-функций *CRC*

1 Аналитический обзор научно-технической литературы

1.1 Системы разграничения доступа

В условиях цифровизации практически всех сфер жизни возрастает необходимость в защите информации от несанкционированного доступа. В организациях, работающих с конфиденциальными данными, важно точно контролировать, кто и какие ресурсы может использовать. Для этого применяются специальные механизмы, объединённые под названием «системы разграничения доступа». Они помогают ограничивать и регулировать доступ к информации в зависимости от уровня полномочий пользователя, что играет ключевую роль в обеспечении информационной безопасности.

Система разграничения доступа – это совокупность методов и технологий, позволяющих управлять доступом к данным и ресурсам информационной среды. Основная задача таких систем – защитить информацию от посторонних и предоставить к ней доступ только тем, кто действительно имеет на это право.

Такие системы применяются в бизнесе, государственных учреждениях, образовательных и медицинских организациях. Они защищают как внутреннюю информацию, так и персональные данные клиентов и сотрудников. Кроме того, наличие правильно настроенной системы доступа помогает компаниям соответствовать требованиям законодательства в области защиты информации [1].

Системы разграничения доступа строятся на ряде ключевых принципов:

- принцип минимального доступа. Пользователю предоставляются только те права, которые необходимы для выполнения его задач;
- идентификация и проверка личности. Каждый пользователь должен быть точно распознан системой перед получением доступа;

- разделение обязанностей. Разделение прав и полномочий между разными пользователями, чтобы снизить риск злоупотреблений;

- фиксация действий. Все действия в системе записываются в журнал для возможного последующего анализа.

В зависимости от подхода к управлению правами доступа различают несколько моделей:

1. Дискреционная модель (*Discretionary Access Control, DAC*). *DAC*, предоставляет гибкий и ориентированный на пользователя подход к управлению разрешениями в информационной системе. *DAC* характеризуется своей зависимостью от усмотрения владельцев ресурсов, которые определяют права доступа и привилегии для других пользователей. Эта модель контрастирует с более жесткими схемами управления доступом, такими как мандатная модель управления доступом, которая применяет строгие политики, устанавливаемые администраторами системы или политиками безопасности[2].

В современных вычислительных средах *DAC* играет ключевую роль в обеспечении удобства использования и совместного доступа к ресурсам при сохранении безопасности. Она широко применяется в операционных системах, базах данных, облачных вычислениях и корпоративных сетях. Данная модель особенно распространена в файловых системах, где пользователи могут самостоятельно определять, кто имеет право читать, записывать или выполнять их файлы. Однако, несмотря на свою гибкость, *DAC* сталкивается с определёнными проблемами безопасности, такими как возможность повышения привилегий и несанкционированное распространение данных, что делает необходимым её использование в сочетании с дополнительными мерами безопасности.

Значимость *DAC* выходит за рамки отдельных систем, оказывая влияние на более широкие архитектуры безопасности и соответствие норматив-

ным требованиям. Организации используют *DAC* для поддержания структурированной политики управления доступом, обеспечивая баланс между операционной эффективностью и безопасностью.

Дискреционная модель управления доступом основана на принципе, согласно которому правообладатели (владельцы данных или ресурсов) самостоятельно определяют, кто может получать доступ к их объектам и какие операции разрешены. Этот подход предоставляет значительную гибкость, но также создаёт определённые риски, связанные с децентрализованным управлением правами доступа.

Основные принципы *DAC*:

1) управление доступом на основе владельца – владельцы объектов (например, файлов, записей в базе данных) имеют полномочия назначать права другим субъектам (пользователям, процессам);

2) гибкость управления разрешениями – владельцы могут динамически изменять права доступа, добавляя или удаляя пользователей в списках доступа.

3) использование списков контроля доступа (*AccessControlList, ACL*) и матриц доступа, определяют, какие субъекты могут взаимодействовать с объектами и в какой форме (чтение, запись, исполнение)[3];

4) возможность наследования прав. В ряде систем права могут передаваться или дублироваться, что упрощает управление доступом, но может привести к ошибкам конфигурации.

Структурные компоненты *DAC*:

1) субъекты – пользователи, группы, процессы или службы, запрашивающие доступ к объектам системы;

2) объекты – файлы, каталоги, базы данных, процессы или иные ресурсы, требующие защиты;

3) права доступа – определённые разрешения, назначаемые субъектам (например, *read*, *write*, *execute*);

4) механизмы управления – использование *ACL*, матриц доступа или других средств контроля.

Реализация дискреционной модели управления доступом в различных информационных системах зависит от нескольких ключевых механизмов.

Основными средствами обеспечения *DAC* являются:

1) списки контроля доступа (*ACL*) – представляют собой структуры данных, хранящие информацию о разрешениях для каждого субъекта в отношении объекта [2];

2) матрицы доступа – двумерные структуры, в которых строки соответствуют субъектам, столбцы – объектам, а на пересечении указываются разрешённые операции.

3) таблицы прав пользователей – альтернативный метод управления доступом, применяемый в ряде операционных систем и баз данных.

4) наследование прав – встроенные механизмы, позволяющие передавать права внутри иерархии объектов.

Каждый из этих механизмов применяется в операционных системах, базах данных и облачных средах для гибкого и детального контроля доступа к ресурсам.

Преимущества *DAC*:

1) гибкость и адаптивность:

– контроль на уровне владельца. Пользователь или владелец ресурса самостоятельно решает, кому предоставить доступ, без необходимости согласования с администратором;

– быстрое реагирование. Идеально подходит для динамичных сред, где права доступа часто меняются (например, совместная работа над проектами) В *Linux* пользователь может мгновенно изменить права на файл с помощью *chmod*, открыв доступ коллегам;

2) простота использования:

– настройка прав через графический интерфейс (например, вкладка "Безопасность" в *Windows*) или команды (*chmod*, *chown* в *Linux*) [3];

– не требует сложных политик или меток, как в *MAC* (*Mandatory Access Control*), что снижает нагрузку на администраторов;

3) децентрализация управления:

– распределение ответственности: владельцы ресурсов сами управляют доступом, что эффективно в организациях с горизонтальной структурой;

– снижение нагрузки на ИТ (информационный отдел) – отдел. Пользователи решают локальные задачи без обращения к системным администраторам;

3) прозрачность и понятность:

– ясность настроек. Легко отследить, кто имеет доступ к ресурсу, через списки *ACL*;

– аудит. Владелец всегда знает, кому и какие права были выданы, что упрощает проверки;

5) совместимость с другими моделями:

– гибридные решения. *DAC* можно комбинировать с более строгими моделями (например, мандатная или ролевая модель) для повышения безопасности. В *Linux* базовые права (*DAC*) работают вместе с *SELinux* (*MAC*), обеспечивая многоуровневую защиту;

6) универсальность и поддержка:

- широкая распространенность, которая реализована в большинстве ОС (операционных систем) и (*Windows, Linux, macOS*) и файловых систем (*NTFS, ext4*);
- поддержка в облачных сервисах. Используется для управления доступом к файлам в *GoogleDrive, Dropbox* и других платформах;

7) эффективность для малых систем:

- минимум накладных расходов. Не требует сложной настройки ролей или меток, что идеально для небольших команд или личного использования.

Недостатки *DAC*:

- 1) отсутствие адаптивности. Статические правила не учитывают изменения контекста или состояния системы;
- 2) сложность анализа последствий. Автоматическая проверка безопасности действий пользователя является алгоритмически неразрешимой задачей в общем случае [4];
- 3) статичность модели создаёт риски несанкционированного распространения прав через цепочки делегирования, что требует дополнительных механизмов аудита и мониторинга в реальном времени.

DAC – отличный выбор для сред, где важны гибкость, простота и децентрализация. Она доминирует в персональных системах, малом бизнесе и ситуациях, где пользователям нужно оперативно управлять доступом. Однако в высокозащищенных средах (военные, госструктуры) её стоит дополнять мандатными или ролевыми механизмами.

2. Мандатная модель (*MAC*). *MAC* представляет собой строгий и централизованный подход к управлению доступом в информационных системах. В отличие от более гибкой дискреционной модели, в *MAC* права пользователей на доступ к ресурсам не определяются владельцами объектов, а жёстко задаются системной политикой безопасности, разработанной администраторами или соответствующими нормативами. Это делает *MAC* особенно актуальной для организаций, где требуется высокий уровень контроля над информацией, например, в государственных структурах, оборонной сфере и на критически важных объектах инфраструктуры.

Основная идея модели *MAC* заключается в том, что каждый объект и субъект в системе получает определённый уровень допуска (например, «секретно», «совершенно секретно», «общедоступно» и т. п.). Доступ разрешён только в том случае, если уровень допуска субъекта соответствует или превышает уровень секретности объекта. Такой подход исключает произвольное управление доступом со стороны пользователей, обеспечивая максимальную безопасность, особенно в средах, где защита данных от утечек или несанкционированного распространения критически важна.

Мандатная модель активно используется в операционных системах с усиленной защитой (например, *SELinux*, *Windows* с метками безопасности), системах управления документами с конфиденциальной информацией и защищённых базах данных. Несмотря на ограниченную гибкость, она обеспечивает высокий уровень защиты от внутренних и внешних угроз, а также соответствует множеству международных стандартов и регламентов по безопасности [5].

Ключевые принципы *MAC*:

1) строгая централизованная политика безопасности. Все права доступа устанавливаются не пользователями, а на уровне системы или организации. Пользователь не может самостоятельно изменять правила доступа;

2) метки безопасности. Каждому объекту (файлу, процессу, записи) и субъекту (пользователю, программе) присваивается определённый уровень секретности;

3) недопущение делегирования прав. Пользователи не могут передавать свои разрешения другим субъектам, что исключает несанкционированное распространение доступа;

4) контроль на уровне ядра системы. Все запросы на доступ проверяются в соответствии с политикой безопасности, встроенной в ядро или службы операционной системы.

Основные элементы *MAC*:

1) субъекты– пользователи, процессы, службы, запрашивающие доступ к ресурсам;

2) объекты– любые защищаемые данные: файлы, каталоги, базы данных, устройства;

3) уровни доступа– заранее заданные классы или категории секретности;

4) политика управления– заранее определённые правила, регулирующие, какие субъекты могут взаимодействовать с какими объектами в зависимости от их меток [6].

Способы реализации *MAC*:

1) метки безопасности. Информация о секретности прикрепляется к каждому объекту и субъекту;

2) политика доступа. Определяют правила сравнения уровней доступа (например, модель Белла-ЛаПадулы);

3) контроль потоков информации. Запрет на передачу данных от более высокого уровня к менее защищённому субъекту;

4) дополнительные механизмы контроля. Включают проверку категорий, классов и контекстов выполнения процессов.

Преимущества *MAC*:

1) высокий уровень защиты:

- строгий контроль доступа позволяет исключить утечку данных;
- невозможность манипулировать правами доступа обычным пользователем.

Пример: в системе *SELinux* каждый процесс и файл имеет контекст безопасности, который невозможно изменить без специальных прав;

2) соответствие стандартам:

- *MAC* часто используется в системах, соответствующих требованиям *FSTEC*, *ISO/IEC 15408* и *NIST*;
- подходит для реализации требований по защите государственной тайны и персональных данных;

3) устойчивость к внутренним угрозам:

- предотвращает распространение доступа даже в случае компрометации одного из пользователей;
- исключает делегирование прав, что снижает риск «разрастания» привилегий [7];

4) надежность контроля:

- все решения о доступе принимаются автоматически на основе политики, без участия пользователя;
- снижает риск человеческих ошибок при настройке доступа;

5) совместимость с другими моделями. Может использоваться в сочетании с *DAC* и *RBAC(Role-Based Access Control)* для создания многоуровневых

систем защиты. Пример: в *Linux* можно комбинировать *SELinux* (*MAC*) с обычными правами (*DAC*), создавая гибкую, но безопасную систему.

Недостатки *MAC*:

1) низкая гибкость:

- изменение политик требует участия администратора или внесения изменений в конфигурационные файлы;

- пользователи не могут самостоятельно делиться файлами или настраивать доступ;

2) сложность в администрировании:

- требует предварительной настройки политик, меток, уровней секретности;

- не всегда интуитивно понятна обычным пользователям;

3) неподходящая для динамичных сред:

- при частом изменении состава пользователей или задач настройка может быть затруднена;

4) высокие требования к ресурсам и обучению:

- требует обучения ИТ-специалистов, понимания ими модели безопасности;

- возможны проблемы совместимости с программами, не поддерживающими метки безопасности[8].

Мандатная модель управления доступом – это строгий, но надёжный способ защиты информации, основанный на централизованных правилах безопасности. Она идеально подходит для сред, где важна полная управляемость и защита критически важных данных. Несмотря на ограничения в плане гибкости и удобства, *MAC* обеспечивает высокий уровень доверия и

применяется в системах, где безопасность является приоритетом. Для достижения оптимального баланса её часто сочетают с другими моделями, такими как *DAC* и *RBAC*, создавая гибридные системы контроля доступа.

3. Ролевая модель (*RBAC*) – это широко применяемый и эффективный подход к управлению доступом, основанный на распределении прав доступа через роли, а не напрямую через пользователей. В отличие от дискреционной (*DAC*) или мандатной (*MAC*) моделей, где контроль осуществляется индивидуально или с помощью строгих политик, в *RBAC* права назначаются группам пользователей с одинаковыми обязанностями. Такая модель обеспечивает гибкий и масштабируемый контроль, особенно востребованный в крупных организациях, где число пользователей и ресурсов может быть весьма значительным.

Суть *RBAC* заключается в том, что каждому пользователю назначается одна или несколько ролей, каждая из которых имеет набор прав на доступ к ресурсам. Это упрощает администрирование и делает систему более управляемой. Например, роль «бухгалтер» может автоматически предоставлять доступ к финансовым документам, а роль «разработчик» — к исходному коду программ[9].

RBAC особенно эффективна в корпоративных информационных системах, базах данных, облачных инфраструктурах и системах документооборота. Она позволяет централизованно управлять доступом, контролировать изменения и проводить аудит.

Основные принципы *RBAC*:

1) роли как центральное звено. Доступ назначается не напрямую пользователю, а роли, которую он занимает. Это повышает гибкость и унифицирует права;

2) разделение обязанностей. Разные роли имеют чётко разграниченные полномочия, что помогает избежать конфликтов интересов;

3) политика на основе потребностей. Каждый пользователь получает только те права, которые необходимы ему для выполнения рабочих задач;

4) масштабируемость. Добавление новых пользователей и изменение их обязанностей не требует пересмотра всех прав – достаточно сменить роль.

Ключевые элементы *RBAC*:

1) субъекты– пользователи или сервисы, которым назначаются роли;

2) роли– это абстрактные группы прав, соответствующие должностям, функциям или обязанностям;

3) права доступа– конкретные разрешения (например, чтение, изменение, удаление);

4) объекты – ресурсы, к которым осуществляется доступ (файлы, базы данных, приложения).

Типы политик в *RBAC*:

1) *RBAC0* (основная модель)– базовая структура, в которой пользователи получают роли, а роли – права;

2) *RBAC1* (иерархическая модель)– роли формируют иерархии, наследуя права друг друга;

3) *RBAC2* (ограниченная модель)– поддерживает статические и динамические ограничения (например, максимальное количество активных ролей);

4) *RBAC3* (полная модель) — объединяет возможности *RBAC1* и *RBAC2* [10].

Преимущества *RBAC*:

1) простота администрирования:

- изменение роли сразу влияет на всех пользователей, что ускоряет управление доступом;

- позволяет централизованно вводить новые политики без пересмотра отдельных прав.

1) повышение безопасности:

- ограничивает доступ только нужными функциями, снижая риск утечек или ошибок;

- упрощает аудит и контроль – права логически сгруппированы и легко отслеживаются;

2) универсальность:

- применяется в различных системах, от операционных систем (*Windows, Unix*) до облачных платформ (*Azure, AWS*);

- легко адаптируется к структурам организаций;

3) интеграция с другими моделями:

- может использоваться совместно с *MAC* или *ABAC* для повышения точности управления. В системе можно настроить ролевой доступ, а также учитывать контекст (время, местоположение) через *ABAC(Attribute-BasedAccessControl)* [8].

Недостатки *RBAC*:

1) ограниченная гибкость:

- не учитывает динамические параметры, например местоположение пользователя или состояние системы;

- сложность настройки для непредсказуемых сценариев;

2) сложность управления в больших системах:

– со временем количество ролей может увеличиться, особенно при индивидуальных потребностях;

– необходимо периодически проводить ревизию и упрощать роли;

3) зависимость от структуры организации. При частых изменениях обязанностей пользователей может потребоваться переработка ролевой модели.

RBAC– это надёжный и широко применяемый подход к управлению доступом, который обеспечивает баланс между безопасностью и удобством. Он особенно полезен в системах с чёткой организационной структурой. При необходимости *RBAC* может быть дополнен другими моделями (например, *ABAC*) для учёта контекста, что повышает гибкость и адаптивность системы управления доступом.

4. Атрибутно-ориентированное управление доступом (*ABAC*)– это современный и высокоадаптивный подход к контролю доступа, основанный на использовании атрибутов, описывающих пользователей, ресурсы, действия и окружение. В отличие от ролевой модели (*RBAC*), где доступ определяется на основе роли, в *ABAC* решения принимаются динамически – система анализирует совокупность характеристик субъекта, объекта и условий, прежде чем предоставить доступ [11].

Атрибуты могут быть самыми разными:

- 1) имя пользователя;
- 2) отдел;
- 3) должность;
- 4) тип устройства;
- 5) *IP*-адрес;
- 6) текущее местоположение;
- 7) время суток;

8) уровень риска и другие параметры.

Это позволяет реализовать тонкую настройку доступа и учитывать контекст, в котором происходит запрос.

ABAC особенно полезна в облачных средах, распределённых системах и организациях с гибкой структурой. Она поддерживает автоматизацию, масштабируемость и соответствует современным требованиям кибербезопасности.

Ключевые принципы *ABAC*:

1) контекст решений. Доступ зависит не только от личности пользователя, но и от текущих условий (например, времени, устройства);

2) логическая политика. Правила доступа описываются в виде логических выражений, использующих атрибут;

3) динамическое принятие решений. Каждый запрос оценивается в режиме реального времени, что повышает точность и безопасность;

4) гибкость управления. Можно точно управлять доступом без создания новых ролей или групп[12];

Компоненты модели *ABAC*:

1) субъекты– пользователи, приложения или процессы, запрашивающие доступ;

2) объекты– данные, документы, сервисы и ресурсы;

3) атрибуты субъектов– имя, должность, роль, сертификация и т. д;

4) атрибуты объектов– тип ресурса, метка секретности, владелец;

5) атрибуты среды– время суток, местоположение, уровень угрозы;

6) политика доступа– правила в формате «если..., то...»;

Преимущества *ABAC*:

1) максимальная гибкость:

- доступ можно регулировать с учётом множества факторов без необходимости изменения ролей.

- легко адаптируется к любым организационным структурам и условиям;

2) динамичность:

- решение принимается в режиме реального времени с учётом текущих рисков и ситуации;

- особенно полезно при удалённой работе или использовании мобильных устройств;

3) масштабируемость:

- новые пользователи или ресурсы не требуют пересмотра ролей, достаточно задать атрибуты;

- упрощает управление доступом в быстрорастущих организациях;

4) интеграция с аналитикой и мониторингом:

- позволяет собирать данные о запросах, улучшая контроль и безопасность;

- может работать в тандеме с системами анализа поведения (*User and Entity Behavior Analytics, UEBA*)[13].

Недостатки *ABAC*:

1) сложность реализации:

- требует продуманной схемы атрибутов, политик и логики;

- возможна высокая нагрузка на вычислительные ресурсы;

2) зависимость от корректности атрибутов:

- ошибка в одном параметре может повлиять на доступность ресурса;

- необходимо регулярно проверять и обновлять данные;

3) необходимость в автоматизации:

- для больших систем желательно внедрение систем принятия решений на основе искусственного интеллекта или машинного обучения;

ABAC – самая гибкая и интеллектуальная модель управления доступом, способная учитывать множество факторов в момент принятия решения. Она подходит для организаций с распределённой инфраструктурой, гибкими бизнес-процессами и высокими требованиями к безопасности. Несмотря на сложность настройки, *ABAC* обеспечивает точный контроль, снижает риски и помогает соответствовать современным стандартам информационной безопасности [11].

Помимо перечисленных подходов, в теоретических исследованиях часто рассматриваются дополнительные методы: контроль передачи данных между объектами, и изоляция программных процессов для минимизации рисков утечек.

Такие системы применяются во всех сферах, где важна безопасность данных: от корпоративных сетей до облачных сервисов, от банковских приложений до медицинских информационных систем. Помимо программных решений, разграничение доступа используется и в физической безопасности – например, при входе в помещения по пропускам или отпечаткам пальцев.

Системы разграничения доступа – важный элемент комплексной защиты информации. Они позволяют грамотно управлять доступом к ресурсам, предотвращают утечку данных и помогают избежать злоупотреблений. Выбор модели зависит от целей, структуры организации и необходимого уровня

безопасности. Сегодня такие системы становятся стандартом для всех, кто заботится о надёжном хранении и использовании информации.

1.2. Роль памяти в обеспечении безопасности

Память является центральным элементом любой компьютерной системы, обеспечивая не только хранение данных, но и выполнение программного кода, организацию взаимодействия между программными модулями, а также поддержку различных уровней абстракции – от операционной системы до прикладного программного обеспечения. Её критическая роль делает память одной из основных целей атак, направленных на нарушение целостности, конфиденциальности или доступности системы. Успешные атаки на память могут позволить злоумышленнику получить несанкционированный доступ к информации, изменить поведение программ или даже полностью захватить управление системой.

Современные угрозы памяти включают широкий спектр уязвимостей, среди которых наиболее распространёнными являются:

- переполнение буфера. Эта уязвимость возникает, когда программа записывает данные за пределы выделенного буфера, что может привести к перезаписи соседних участков памяти, включая критически важные структуры, такие как указатели возврата, таблицы вызовов функций или переменные управления потоком. Классический пример – атака на стек, когда злоумышленник подменяет адрес возврата, заставляя процессор выполнить произвольный код. Разновидности переполнения включают как стековое, так и кучевое переполнение, каждое из которых имеет собственные методы эксплуатации и защиты [14];

- инъекции кода. Атаки, при которых вредоносный код внедряется в память целевого процесса, а затем управление передаётся на этот код. Такие

атаки часто используют уязвимости в обработке входных данных – например, недостаточную проверку границ массива, некорректную работу с указателями или уязвимости в интерпретаторах (например, *SQL*-инъекции или инъекции в скриптовых языках). Современные эксплойты могут комбинировать инъекцию с обходом *DEP* и *ASLR*, используя, например, цепочки *ROP* (*Return-Oriented Programming*) – метод, при котором злоумышленник не вводит новый код, а использует существующие фрагменты инструкций («гаджеты») в адресном пространстве;

– *use-after-free* и *double-free*. Уязвимости, возникающие при неправильном управлении динамической памятью. Первая связана с попыткой обращения к памяти после её освобождения, вторая — с повторным освобождением уже освобождённого блока. Обе могут привести к повреждению кучи и выполнению произвольного кода, особенно если атакующий может контролировать содержимое освобожденных блоков [15];

– *out-of-bounds* доступ (*OOB*). Эта уязвимость появляется, когда программа обращается к данным за пределами допустимого диапазона – как в массивах, так и при использовании указателей. Эти ошибки часто незаметны при тестировании, но могут быть фатальными в боевой среде [16].

Для защиты памяти применяются как программные, так и аппаратные методы. Среди них, стоит выделить:

– *ASLR* (*AddressSpaceLayoutRandomization*). Технология, которая случайным образом размещает ключевые области памяти (включая стек, кучу и разделяемые библиотеки) при каждом запуске процесса. Это значительно снижает предсказуемость адресов, необходимых для успешной атаки. Однако *ASLR* может быть неэффективен в условиях частичных утечек адресов (например, при наличии *side-channel* атак), а также в случае повторного запуска процессов без регенерации сида [17];

– *DEP (DataExecutionPrevention)*. Механизм, обеспечивающий разграничение между областями памяти, предназначенными для выполнения кода, и областями, предназначенными только для хранения данных. *DEP* реализуется аппаратно через *NX*-биты в современных процессорах, а также может быть дополнен средствами ОС, запрещающими выполнение кода в определённых сегментах. Однако *DEP* не препятствует атакам, использующим легитимный, но переформатированный код (например, *ROP– Return-OrientedProgramming* или *JOP –Jump-OrientedProgramming*) [18];

– контроль доступа к памяти (*MemoryAccessControl*). Он включает аппаратные механизмы защиты, такие как сегментация и страничная адресация в архитектуре *x86*, а также расширенные технологии защиты в современных процессорах — *Intel MPX*, *ARM MemoryTaggingExtension* и др. Эти средства позволяют отслеживать и ограничивать действия программ в отношении определённых участков памяти, устанавливая права на чтение, запись и исполнение [19];

– изоляция памяти (*Memoryisolation*). Она используется в современных ОС и гипервизорах для обеспечения разграничения между процессами и уровнями привилегий. Например, в архитектуре ядро/пользователь память ядра недоступна из пользовательского режима. Кроме того, в современных браузерах применяется изоляция процессов для вкладок, чтобы минимизировать последствия уязвимостей типа *Spectre* и *Meltdown*[20];

– аппаратные средства защиты от уязвимостей *side-channel*. В ответ на угрозы, выявленные в ряде процессоров (например, *Meltdown*, *Spectre*, *Foreshadow*), производители процессоров внедрили аппаратные и микрокодовые меры защиты от атак, использующих кэш-память, спекулятивное выполнение и прочие побочные эффекты архитектуры [21].

Тем не менее, даже в совокупности эти методы не обеспечивают абсолютной безопасности. Например, *ASLR* может быть обойдён при наличии

информационных утечек, *DEP* неэффективен против атак, не использующих инъекцию, а контроль доступа требует тщательной настройки и встраивания в архитектуру ПО и ОС. Это особенно критично в условиях мандатного управления доступом, где важно не только техническое исполнение механизмов защиты, но и формальное соблюдение политик безопасности, включая требования к неизменности конфигурации, контролю целостности и невозможности обхода правил.

Таким образом, защита памяти остаётся одной из ключевых проблем информационной безопасности, требующей комплексного подхода, включающего статический и динамический анализ кода, использование современных компиляторов с поддержкой защиты (*StackCanaries*, *ControlFlowGuard* и др.), а также постоянного мониторинга и обновления систем.

1.3. Хеш-функции в задачах информационной безопасности

Хеш-функции представляют собой математические алгоритмы, преобразующие произвольный набор данных в значение фиксированной длины, называемое хешем или контрольной суммой. В задачах информационной безопасности они используются для проверки целостности данных, идентификации объектов и обеспечения быстрого доступа к информации. Хеш-функции делятся на две категории: криптографические и некриптографические, каждая из которых имеет свои особенности и области применения.

Современные компоненты вычислительной системы:

- многоядерные процессоры;
- *SIMD*-инструкции;
- аппаратное ускорение [22].

Применение этих компонентов позволяет существенно ускорить выполнение хеш-функций, особенно при обработке больших объёмов данных в задачах анализа памяти, сетевого трафика или мониторинга целостности.

Однако возможности использования параллелизма зависят от типа хеш-функции.

Некриптографические хеш-функции изначально ориентированы на высокую производительность, поэтому легко масштабируются в многопоточной и *SIMD*-среде. Они допускают разбиение данных на блоки и независимую обработку, что делает их особенно подходящими для высоконагруженных систем.

Существует большое количество различных некриптографических хеш-функций, каждая из которых оптимизирована под конкретные задачи: от высокой производительности в системах реального времени до эффективного распределения данных в хеш-таблицах. Ниже представлены наиболее распространённые и эффективные алгоритмы, используемые в современных вычислительных системах:

1. *CRC(CyclicRedundancyCheck)* –циклический избыточный код. *CRC* был впервые предложен в 1960-х годах для обнаружения ошибок в каналах связи и цифровых сетях. Он стал широко использоваться в различных протоколах и механизмах, таких как:

- 1) сетевые протоколы;
- 2) протоколы передачи файлов;
- 3) обнаружения ошибок в данных.

Версии *CRC*:

1) *CRC8* – 8-битная контрольная сумма, используется в таких системах, как *Bluetooth* и небольшие микроконтроллеры, где необходима простая и быстрая проверка целостности данных;

2) *CRC16* – 16-битная контрольная сумма, распространена в протоколах *Modbus*, *USB*, *XMODEM*, *HDLC*, *GSM*-сетях [23];

3) *CRC32* – 32-битная контрольная сумма, наиболее популярна в *ZIP*, *Ethernet*, *PNG*, в файловых системах и других, требующих высокой надежности передачи данных;

4) *CRC64* – 64-битная контрольная сумма, применяется в системах хранения данных и телекоммуникационных стандартах, таких как сети высокой надежности и системы архивирования данных, где требуется высокая степень защиты от ошибок.

Для каждой версии контрольной суммы, для расчета используется свой полином, что и делает циклический избыточный код уникальным.

Особенности *CRC*:

1) *CRC* поддерживается на уровне аппаратных инструкций в современных процессорах (например, *CRC32* в *SSE4.2* и *AVX*), что позволяет значительно ускорить вычисления;

2) является идеальным выбором для применения в реальном времени в таких системах, как файловые системы, сетевые протоколы, а также в *RAID*-массивах для контроля целостности данных;

3) несмотря на свою простоту, *CRC32* эффективен для обнаружения случайных ошибок и фальсификаций данных.

2. *MurmurHash*– это быстрое, детерминированное хеширование, ориентированное на равномерное распределение выходных значений и минимизацию коллизий. Название не является аббревиатурой, происходит от «*murmur*» – «бурчание» (назван из-за звука коллизий), была предложена в 2008г.[24]. Широко используется в:

1) хеширование строк;

2) хеширование байтов и ключей;

2) хеш-таблицах;

- 4) структурах данных;
- 5) *JVM (Java Virtual Machine)*.

Версии *MurmurHash*:

- 1) *MurmurHash2* – старая версия, всё ещё используется в старых проектах;
- 2) *MurmurHash3* – под архитектуры x86 (32 бит) и x64 (128 бит).

Особенности *MurmurHash*:

- 1) *MurmurHash3* известен своей высокой производительностью, особенно при обработке больших объемов данных, что делает его популярным выбором в системах с высокими требованиями к скорости, таких как хеш-таблицы, системы поиска и базы данных;
- 2) Он широко используется в таких проектах, как *ApacheHadoop*, *ApacheCassandra*, *Elasticsearch*, и других распределённых системах, требующих быстрого хеширования.

3. *CityHash*– это высокопроизводительная некриптографическая хеш-функция от *Google*, оптимизированная для коротких и длинных строк. Название образовано как внутренняя *Google*-наработка, название не является аббревиатурой, была предложена в 2011 г. [25] Используется для:

- 1) увеличение эффективности в процессорах архитектуры x86 и *ARM*;
- 2) поддержки длинных строк;
- 3) минимизации коллизий для больших объемов текста.

Версии *CityHash*:

1) *CityHash64* – возвращает 64-битный хеш, оптимизированный для 64-разрядных процессоров. Он особенно эффективен для коротких строк, что делает его идеальным для хеш-таблиц;

2) *CityHash128* – возвращает 128-битный хеш, оптимизированный для длинных строк (не менее нескольких сотен байт). Он обеспечивает более высокую производительность, чем *CityHash64*, для больших объемов, данных.

Особенности *CityHash*:

1) *CityHash* обеспечивает отличное распределение хешей, минимизируя коллизии, и является очень быстрым для коротких данных;

2) Он широко используется в поисковых системах и веб-приложениях, таких как *GoogleSearch*, *GoogleChrome*, а также в системах логирования и кэширования.

4. *FarmHash* – семейство хеш-функций, пришедших на смену *CityHash*, обеспечивающее кросс-платформенную стабильность и улучшенную производительность. Название не расшифровывается, было предложено в 2011 г [26]. Применяется в тех структурах что и применялся *CityHash*, но появился ряд улучшений:

1) более стабильный результат при компиляции под разные архитектуры;

2) поддержка *SIMD*-инструкций;

3) улучшенное распределение хеша.

Версии *FarmHash*:

1) *FarmHash64* – это 64-битная хеш-функция, которая преобразует входные данные (строки или массивы байтов) в 64-битное хеш-значение. Она

оптимизирована для скорости и эффективности, особенно в задачах, где важна быстрая обработка;

2) *FarmHash*128 – 128-битная функция отпечатка (*fingerprint*), которая возвращает 128-битное значение, обычно представленное в виде двух 64-битных чисел. Она предназначена для задач, требующих стабильности и независимости от платформы.

Особенности *FarmHash*:

1) *FarmHash* подходит для задач, требующих универсальности и стабильности, таких как обработка больших данных, системы с распределёнными вычислениями, а также для веб-сервисов, таких как *GoogleBigQuery* и *LevelDB*;

2) он также отлично масштабируется на многопоточных системах, что позволяет эффективно обрабатывать большие объёмы данных.

5. *HighwayHash* – это быстрый, параллелизуемый хеш-алгоритм от *Google*, оптимизированный для современных *CPU* с *SIMD*. Название отражает высокую скорость выполнения, без аббревиатуры, предложен в 2017 г.[27]

Подходит для использования:

1) инструкций *AVX2/AVX-512*;

2) хеширования данных в потоках;

3) в системах требующих высокую устойчивость к коллизиям.

Особенности *HighwayHash*:

1) *HighwayHash* предназначен для работы в реальном времени, что делает его идеальным для систем, обрабатывающих потоки данных, например, в телеметрии, системах мониторинга и анализе больших данных;

2) он также используется для обеспечения высокой скорости в системах безопасности и анализе потоков данных.

Версии *HighwayHash*:

1) *HighwayHash64* – 64-битный хеш, оптимизированный для задач, в которых требуется компактное хеш-значение, например, в хеш-таблицах или для коротких сообщений;

2) *HighwayHash128* – 128-битный хеш, представленный в виде двух 64-битных значений. Подходит для задач, требующих большей устойчивости к коллизиям;

3) *HighwayHash256* – 256-битный хеш, обеспечивая максимальную устойчивость к коллизиям среди функций *HighwayHash*. Подходит для задач, где важна высокая надёжность.

Эти хеш-функции используются в таких областях, как поиск данных, распределённые системы, контроль целостности и потоковая обработка данных.

Преимущества некриптографических хеш-функций:

- высокая скорость вычислений, быстрее криптографических аналогов;
- реализуются аппаратно (например, *CRC* поддерживается на уровне инструкций *CPU*), что позволяет достигать миллионов хешей в секунду, что является критически важно в системах реального времени, базах данных, хеш-таблицах и телеметрии;

- поддержка параллельной обработки данных. Простые алгоритмы легко адаптируются для многопоточной или *SIMD*-обработки. Возможность параллельного хеширования больших блоков памяти делает их идеальными для задач контроля целостности в распределённых и высоконагруженных системах. Например, *CRC* может применяться на множестве ядер для анализа памяти без конфликта данных [28];

- низкие накладные расходы. Занимают минимум оперативной памяти и процессорного времени. Идеальны для:

1) встраиваемых систем;

2) *IoT(InternetofThings)*;

3) микроконтроллеров и других ограниченных по ресурсам сред;

– хорошее распределение значений (для задач безопасного хеширования не требуется). Несмотря на отсутствие криптостойкости, такие функции обеспечивают равномерное распределение хешей, что снижает количество коллизий в практических приложениях (например, при индексации или поиске);

– простота реализации и отладки. Алгоритмы просты для понимания, написания и тестирования. Не требуют сторонних библиотек или сложной математической подготовки;

– детектирование случайных ошибок. Несмотря на отсутствие защиты от умышленных изменений, многие функции (например, *CRC*) эффективно обнаруживают случайные ошибки (битовые сбои, сбои памяти, искажения при передаче данных).

Несмотря на высокую скорость и эффективность, некриптографические хеш-функции имеют ряд ограничений, которые важно учитывать при выборе алгоритма для конкретной задачи:

–отсутствие криптографической стойкости. Такие функции не защищены от преднамеренных атак, злоумышленник может подобрать входные данные, приводящие к определённому хешу или вызвать коллизию. Они не гарантируют однозначное соответствие между входом и хешем в условиях враждебной среды. Например, *CRC32* легко поддаётся подстановкам, когда известен желаемый контрольный хеш— это делает его непригодным для аутентификации или цифровых подписей [29];

–уязвимость к коллизиям. У некриптографических алгоритмов меньшая устойчивость к коллизиям, особенно при работе с малым объёмом данных или предсказуемыми шаблонами. Это может привести к снижению качества детектирования изменений и ложноположительным результатам контроля целостности;

–отсутствие гарантии односторонности. В отличие от криптографических функций, некриптографические хеши могут быть обратимы или легко подобраны при переборе, особенно если объём входных данных мал. Это исключает их применение в задачах хранения паролей или генерации токенов;

– недостаточная длина выходных значений. Многие некриптографические функции возвращают 32-битные или 64-битные хеши, что ограничивает количество возможных уникальных значений и увеличивает риск коллизий в больших выборках. При работе с миллионами объектов вероятность совпадения хешей (эффект дня рождения) возрастает.

– непригодность для длительного хранения или юридических доказательств. В системах, где требуется доказуемость неизменности данных (архивы, цифровые доказательства), некриптографические функции не удовлетворяют требованиям регуляторов и стандартов.

Криптографические хеш-функции проектируются в первую очередь с прицелом на стойкость к атакам, а не на производительность. Большинство таких функций имеют последовательную структуру вычислений, что затрудняет их эффективное распараллеливание:

1. *SHA(Secure Hash Algorithm)* – алгоритм безопасного хеширования. Представляет собой семейство криптографических хеш-функций, разработанных национальным институтом стандартов и технологий США для обеспечения целостности и аутентичности данных, предложен в 1993 г. [30]. Подходит для использования:

1) где необходима односторонность, чтобы нельзя было восстановить исходных данных по хешу;

2) детектирования малейших изменений данных (лавинοобразный эффект);

3) в системах требующих устойчивость к коллизиям, атаке по входу и выходу.

Особенности *SHA*:

1) выполняется последовательно, поскольку каждая итерация зависит от предыдущего состояния;

2) нельзя использовать для параллельной обработки данных.

Версии *SHA*:

1) *SHA-0* – первоначальный вариант, быстро отозван из-за обнаруженных уязвимостей;

2) *SHA-1* – хеш-функция с размером хеша 160 бит. Применялся в *SSL*-сертификатах и криптографических системах до начала массового отказа после доказанных коллизий (2005–2017 гг.);

3) *SHA-2* – представляет собой группу из нескольких функций:

– *SHA-224*;

– *SHA-256*;

– *SHA-384*;

– *SHA-512*;

– *SHA-512/224*;

– *SHA-512/256* [31];

Отличаются длиной хеш-суммы и объемом данных, обрабатываемых за раунд. *SHA-256* и *SHA-512* – наиболее популярные и используются в блокчейне и *VPN (Virtual Private Network)*. Поддерживаются большинством современных криптографических библиотек (*OpenSSL* и *BouncyCastle*). Уязвим к атакам типа collision; признан небезопасным.

3) *SHA-3 (Keccak)* – представляет собой принципиально иную структуру (основана на губчатой конструкции). Поддерживает хеши размером:

– 224 бит ;

– 256 бит;

– 384 бит;

– 512 бит.

Обеспечивает высокую стойкость к атакам, устойчив к коллизиям, и потенциально заменит *SHA-2* в будущем. Поддерживает расширения *SHAKE128* и *SHAKE256* – настраиваемые функции с переменной длиной выходного значения (*extendable-output functions*)[31].

– *BLAKE* – семейство криптографических хеш-функций, основанных на конструкции *HAIFA* и криптографическом примитиве *ChaCha* (модифицированной функции *Salsa20*). Разработана с целью высокой безопасности, скорости и устойчивости к атакам. Название *BLAKE* не является аббревиатурой, а выбрано в честь английского поэта и художника Уильяма Блейка, предложен в 2010 г. Подходит для использования:

1) в системах требующих высокую скорость хеширования (выше, чем у *SHA-2* и *SHA-3*);

2) в системах требующих потоковое хеширование;

3) в системах требующих высокую устойчивость к коллизиям.

4) в системах требующих устойчивость к дифференциальным атакам и атаке на длину сообщения;

5) в системах требующих гибкость в настройке:

– изменение выходной длины;

– ключевого хеширования.

Особенности *BLAKE*:

1) *BLAKE* унаследовал устойчивость от поточно-шифрующей функции *ChaCha* и конструкции *HAIFA*;

2) Он использует детерминированный вывод и сильное лавинообразное поведение.

Версии *BLAKE*:

1) *BLAKE* (1.0) – представляет собой первую версию. Варианты:

– *BLAKE-224*;

– *BLAKE-256*;

– *BLAKE-384*;

– *BLAKE-512*.

Применялась структура *HAIFA*, не опередил *SHA-3*, но получил высокие оценки;

2) *BLAKE2* – представляет собой вторую версию хеш-функции.

Варианты:

– *BLAKE2b*: ориентирован на 64-битные платформы (выход – 512 бит).

– *BLAKE2s*: оптимизирован для 8-32-битных систем (выход – 256 бит).

Обеспечивает лучшую производительность, чем *SHA-2* (при сохранении криптоустойчивости) [32];

3) *BLAKE3* – представляет собой третью версию хеш-функции. Поддерживает произвольную длину выходного значения. Один универсальный алгоритм вместо нескольких (в отличие от *BLAKE2*). Основные отличия от предыдущей версии:

- детерминированность;
- компактность;
- высокая масштабируемость и скорость;

Криптографические хеш-функции разработаны с акцентом на безопасность и стойкость к атакам. Их свойства делают их незаменимыми в системах, где важна гарантия подлинности, целостности и защиты от злоумышленников.

Преимущества криптографических хеш-функций:

–односторонность (невозможность обратного восстановления). По известному хешу невозможно восстановить исходные данные, даже если известна структура входа. Это свойство особенно важно для хранения паролей, токенов и биометрических шаблонов. Хеш пароля в базе данных не может быть напрямую обращён к самому паролю;

–устойчивость к коллизиям. Крайне сложно найти два разных входа с одинаковым хешем, особенно у современных алгоритмов (*SHA-2*, *SHA-3*). Это критично для обеспечения достоверности электронных документов, цифровых подписей и блокчейн-систем. Даже малейшее изменение в данных вызывает значительное изменение в хеше ("лавинный эффект") [29];

– устойчивость к подбору (*preimageresistance*). Подбор данных по заданному хешу невозможен на практике, даже с учётом всей вычислительной

мощности современных систем. Это обеспечивает устойчивость против атак на подделку данных;

- юридическая и регуляторная применимость. Криптографические хеш-функции признаются стандартами безопасности и могут использоваться в системах, где требуется подтверждение неизменности данных или цифровая аутентификация;

- используемость в криптографических протоколах. Являются неотъемлемой частью таких протоколов, как:

- 1) *SSL/TLS* (безопасность веб-соединений);
- 2) *IPsec* (безопасность сетевых соединений);
- 3) *Blockchain* и криптовалюты (обеспечение целостности блоков);
- 4) протоколы аутентификации (*OAuth, Kerberos*).

Несмотря на высокую стойкость и широкую применимость в задачах информационной безопасности, криптографические хеш-функции имеют ряд существенных недостатков, особенно в контексте высоконагруженных или ресурсозависимых систем:

- высокая вычислительная нагрузка. Криптографические функции требуют значительно больше ресурсов *CPU*, чем некриптографические аналоги. Это делает их неоптимальными для встраиваемых систем, реального времени, задач с частыми пересчётами хешей или большого объёма данных. В системах мониторинга памяти с тысячами проверок в секунду *SHA-256* может создавать нежелательные задержки [32];

- непригодность для параллельной обработки. Многие криптографические функции имеют последовательную структуру, где следующий шаг зависит от предыдущего. Это затрудняет или исключает их распараллеливание, в отличие от простых функций, таких как *CRC*;

– более сложная реализация и сопровождение. Алгоритмы содержат сложные математические конструкции, требуют точности в реализации и могут быть подвержены ошибкам при ручной разработке. Встраивание в ограниченные среды требует дополнительной оптимизации и может занимать больше памяти;

– возможные устаревания и уязвимости. Некоторые криптографические хеш-функции со временем теряют актуальность из-за открытых уязвимостей. Пример: *SHA-1* признан небезопасными из-за успешных атак на коллизии, но всё ещё встречаются в старых системах;

– избыточность для простых задач. Для задач, где не требуется криптоустойчивость (например, быстрое сравнение блоков памяти или индексация), использование криптофункций приводит к неоправданным затратам.

Криптографические хеш-функции – мощный инструмент безопасности, но их применение должно быть обосновано потребностью в стойкости к атакам. В сценариях, где ключевыми являются скорость, экономия ресурсов и массовая обработка, их недостатки становятся критичными. Поэтому важно отделять задачи защиты информации от задач структурной оптимизации и контроля целостности, чтобы применять хеш-функции адекватно поставленной цели.

В задачах контроля памяти, целостности и мониторинга в реальном времени, где критически важны высокая скорость обработки, минимальные задержки и возможность параллельного вычисления, наилучшим выбором являются некриптографические хеш-функции, а именно – *CRC32* [33].

CRC32 (циклический избыточный код) обладает рядом преимуществ:

– высокая скорость вычислений благодаря аппаратной поддержке на современных процессорах (например, через инструкции *CRC32* в наборах *SSE4.2* и *AVX* на *x86*, а также аналогах на *ARM*), вычисление контрольной суммы занимает считанные такты процессора;

– поддержка параллельной обработки. Память можно разбить на блоки и вычислять хеши для каждого блока независимо, что идеально подходит для многопоточной среды или *SIMD*-обработки;

– минимальные накладные расходы. В отличие от криптографических хеш-функций, *CRC32* требует меньше ресурсов и может эффективно работать в системах с ограниченными вычислительными возможностями (встраиваемые устройства и микроконтроллеры) [33];

– широкая поддержка на уровне оборудования и стандартов. Применяется в:

- 1) сетевых протоколах;
- 2) файловых системах;
- 3) *RAID*-массивах и других критичных к целостности данных системах.

Таким образом, *CRC32* – это оптимальное решение для высокопроизводительных и ресурсоэффективных задач, связанных с обнаружением изменений в памяти, контролем целостности и мониторингом систем в реальном времени. Его простота, совместимость с параллельными архитектурами, и высокая скорость делают его стандартом де-факто в этой области.

Криптографические хеш-функции не эффективны в параллельной среде из-за своей внутренней структуры и требований безопасности. Их использование оправдано только там, где важна защита от умышленных искажений.

При разработке высокопроизводительных систем контроля состояния памяти важно учитывать совместимость функции с архитектурой процессора, наличие *SIMD*-инструкций и поддержку многопоточности.

1.4. Обзор существующих подходов к контролю состояния памяти

Контроль состояния памяти направлен на обнаружение несанкционированных изменений или нарушений целостности данных в адресном про-

странстве процесса. Этот процесс критически важен для предотвращения атак, нарушающих работу программ или системы в целом. Среди современных технологий и инструментов можно выделить следующие:

– *canary* («стековые канарейки») это использование специальных значений («канареек»), размещаемых в стеке перед возвращаемыми адресами. Если «канарейка» изменяется, это указывает на попытку переполнения буфера.

Преимущества *canary*:

- 1) низкие накладные расходы;
- 2) простота реализации.

Главный недостаток *canary* это уязвимость к атакам, которые обходят проверку канарейки, например, через точечное изменение памяти или утечку значений канарейки [34];

– *memorytagging* (тегирование памяти) это современный подход, реализованный в аппаратных решениях, таких как *ARM MTE (MemoryTaggingExtension)*. Каждому блоку памяти присваивается тег, который проверяется при доступе. Это эффективно против ошибок типа *use-after-free* и переполнения буфера.

Преимущества *memorytagging*:

- 1) высокая точность;
- 2) минимальное влияние на производительность при аппаратной поддержке.

Главный недостаток *memorytagging* заключается в том что требуется совместимость с новым оборудованием и ОС, что ограничивает распространение [35];

– *sandboxing* (песочницы это изоляция процессов или частей программы в ограниченном адресном пространстве для минимизации последствий атак. Примеры включают *seccomp* в *Linux* или *AppContainer* в *Windows*.

Преимущества *sandboxing*:

- 1) высокая степень изоляции;
- 2) защита от эксплуатации уязвимостей.

Недостатки *sandboxing*:

- 1) сложность настройки;
- 2) потенциальное ограничение функциональности программы [36];

– *RASP (RuntimeApplicationSelf-Protection)* это технология, интегрирующая защитные механизмы непосредственно в приложение. *RASP* отслеживает выполнение программы в реальном времени, обнаруживая аномалии, такие как несанкционированные изменения памяти.

Преимущества:

- 1) высокая адаптивность;
- 2) возможность работы без изменения ОС.

Недостатки:

- 1) зависимость от качества реализации;
- 2) потенциальные накладные расходы;

Формальная верификация памяти это использование математических методов для доказательства корректности управления памятью в программе. Применяется в критически важных системах (например, в авионике). Главным преимуществом является максимальная надежность.

Недостатки:

- 1) высокая сложность;
- 2) высокая стоимость разработки;
- 3) ограниченная применимость для больших проектов [37];

– инструменты динамического анализа в реальном времени. Помимо *Valgrind* и *AddressSanitizer*, существуют решения, такие как *Intel MPX* (*MemoryProtectionExtensions*) или *MicrosoftControlFlowGuard*, которые обеспечивают мониторинг памяти на уровне аппаратуры или ОС. Однако их использование ограничено из-за устаревания (*MPX* снят с поддержки) или необходимости специфической платформы.

Ограничения:

– совместимость с мандатной моделью защиты остается проблемой, так как большинство подходов не учитывают уровни доступа и политики безопасности на уровне ядра. Например, *ASLR* и *DEP* не взаимодействуют с мандатными политиками, такими как *SELinux*;

– высокие вычислительные затраты инструментов вроде *CFI* и *AddressSanitizer* делают их менее подходящими для ресурсограниченных устройств, таких как *IoT*;

– рост атак через побочные каналы (например, *Spectre* и *Meltdown*) требует интеграции новых методов, таких как изоляция памяти на уровне ядра или использование криптографических методов для защиты адресов.

Перспективы:

- развитие аппаратных решений (*ARM MTE* и *Intel CET*) обещает повысить эффективность контроля памяти с минимальными затратами;
- интеграция машинного обучения для анализа аномалий в реальном времени может дополнить существующие подходы, но требует осторожного подхода из-за риска ложных срабатываний;
- стандартизация мандатных политик в сочетании с современными технологиями (например, *CFI* с *SELinux*) может устранить разрыв между защитой памяти и политиками безопасности[38].

Эти дополнения подчеркивают многогранность проблемы контроля состояния памяти и необходимость комплексного подхода, сочетающего аппаратные, программные и формальные методы для обеспечения устойчивой защиты.

1.5 Средства проверки целостности состояния памяти

Средство проверки целостности использует, свойство вредоносного ПО изменять файлы, для поиска и отслеживания несанкционированных изменений в файлах.

Проверка целостности должна начинаться с абсолютно чистой, на 100% свободной от вредоносного ПО, это невозможно переоценить. Программа проверки целостности изначально вычисляет и сохраняет контрольную сумму для каждого файла в системе, за которой она следит. Затем контрольная сумма файла пересчитывается и сравнивается с исходной сохранённой контрольной суммой. Если контрольные суммы отличаются, значит, в файле произошли изменения.

Существует несколько типов средств проверки целостности:

– в автономном режиме. Контрольные суммы проверяются только периодически, например, раз в неделю;

– самопроверка. Исполняемые файлы модифицируются таким образом, чтобы проверять себя при запуске. Модификация исполняемых файлов для самопроверки их целостности включает в себя механизмы, подобные вирусам. Самопроверку можно выполнить менее навязчивым способом, добавив код самопроверки в общие библиотеки;

– оболочки целостности. Контрольная сумма исполняемого файла проверяется непосредственно перед выполнением. Это может быть встроено в ядро операционной системы для двоичных исполняемых файлов; для других типов «исполняемых» файлов, таких как пакетные файлы, сценарии оболочки и программы на языке сценариев, идеальное расположение менее очевидно;

– контроль состояния памяти. С целью уменьшения объема вычислений и обнаружения атак на ранней стадии применяется контроль состояния долговременной и кратковременной памяти с помощью хеш- функции *CRC* [39].

Расчет хеш - функции предлагается вычислять в произвольный момент времени, тогда вероятность подмены данных будет низкой, в отличие от алгоритмов вычисления осуществляющий контроль через определенное время, что позволяет осуществлять подмену результата хеш - функции злоумышленниками.

1.6 Мандатная защита реализованная на архитектурном уровне

Для защиты контекста программы (представленного объектами) архитектура «Эльбрус» применяет аппаратный контроль состояния памяти, основанный на хеш-функциях. Основной механизм защиты: весь доступ к внутреннему представлению объекта (в памяти или регистрах) осуществляется строго через его дескриптор. Этот дескриптор (служебное слово) содержит

ссылку на объект и информацию о его типе. Аппаратная реализация операций создания и использования дескрипторов надежно блокирует любые попытки прямого воздействия на объект.

Фундаментальную роль в этой защите играет тегирование: добавление к данным специального поля (тега), которое определяет их тип.

Каждое 4-байтовое слово в памяти, регистрах и шинах сопровождается 2-разрядным тегом.

Тег слова кодирует следующие признаки:

- 0 – слово содержит числовую информацию либо само по себе, либо являясь фрагментом формата;
- 1 – слово содержит нечисловую информацию с форматом одинарного слова;
- 2 – слово содержит фрагмент нечисловой информации формата двойное слово;
- 3 – слово содержит фрагмент нечисловой информации формата квадр.

Ни длина, ни тип числовых данных архитектурно не различимы. Семантическое наполнение числовой переменной отслеживается компилятором и проявляется, когда она становится операндом какой-либо операции [40].

В отличие от числовых данных, нечисловые данные строго типизируются.

Прежде всего гарантируется целостность составных форматов (двойное слово и квадрат), что предполагает следующее:

- все фрагменты должны иметь тип, соответствующий формату;
- при любых манипуляциях фрагменты должны сохранять свой порядок.

Второе свойство обеспечивается строгим выравниванием данных в памяти и регистровом файле. Например, младшее слово 64-битной адресной переменной всегда размещается по адресу, кратному 8 байтам, а старшее — непосредственно следом. Любая пересылка фрагментов данных нарушает их типизацию, превращая в неспециализированные данные. Операции, требующие специализированных операндов (например, адресов), аппаратно контролируют их тип и целостность.

Конкретный тип специализированных данных внутри форматной группы определяется значением выделенных битов внутреннего тега.

Помимо стандартных дескрипторов для массивов, архитектура вводит специальные дескрипторы для объектно-ориентированных структур данных, что значительно упрощает доступ к ним. Дескриптор объекта содержит описание открытых, защищенных и приватных областей данных [41].

Архитектура дополняет регистры контекста программы специальными регистрами управления модулем: индекс текущего модуля компиляции, его дескриптор, дескриптор глобальных данных модуля, дескриптор типов и текущий тип. Для работы с памятью введены операции, использующие эти дескрипторы. Все дескрипторы снабжены внутренними тегами, которые аппаратно контролируют корректность их применения — разрешая использование только в специализированных операциях доступа к памяти и операциях понижения прав.

2 Разработка методики контроля состояния памяти с помощью хеш-функции *CRC*

2.1 Разработка алгоритмов расчета хеш-функции *CRC*

Современные вычислительные системы часто становятся мишенью для атак, целью которых является изменение содержимого оперативной памяти. Нарушения такого рода способны привести к серьёзным последствиям: от утечки конфиденциальной информации до изменения алгоритма работы приложений и перехвата управления системой. Эффективным способом обнаружения подобных вмешательств выступает контроль состояния памяти, реализуемый посредством хеширования.

Одна из наиболее простых и быстрых хеш-функций – *CRC*. Изначально этот механизм был разработан для выявления ошибок при передаче данных, однако со временем нашёл применение и внутри программ, включая задачи анализа содержимого памяти. *CRC* позволяет отслеживать изменения в массиве данных за счёт вычисления остатка от полиномиального деления двоичной последовательности.

Хеш-функция *CRC* отличается тем, что при повторной обработке одного и того же фрагмента памяти возвращает одинаковое значение. Даже малейшее изменение одного бита приводит к другому результату, что делает её эффективным инструментом для фиксации модификаций. Простота реализации, высокая скорость выполнения и минимальные требования к ресурсам делают *CRC* удобным решением для оперативного или периодического анализа состояния памяти.

Для надёжного мониторинга критически важна не только точность, но и скорость вычисления контрольной суммы. Существуют 3 базовых подхода к реализации *CRC*, отличающихся производительностью и сложностью:

1. Побитовый алгоритм. Наиболее простой и понятный способ вычисления – побитовая реализация, работающая следующим образом:

- 1) создаётся регистр, размер которого равен степени выбранного полинома, и он инициализируется нулями;
- 2) к исходному блоку данных добавляется дополнение – несколько нулей в конце, равное числу битов полинома;
- 3) один за другим, биты данных подаются в младший разряд регистра, при этом из старшего разряда удаляется по одному биту;
- 4) если выталкиваемый бит равен единице, происходит операция *XOR* между текущим содержимым регистра и полиномом;
- 5) процедура продолжается до тех пор, пока не будут обработаны все биты входной последовательности;
- 6) финальное значение регистра после завершения – это и есть *CRC*-хеш [42].

Важно, что результат зависит не только от данных, но и от порядка обработки байтов, может использоваться порядок от младших к старшим или наоборот. Также возможно дополнительное модифицирование результата, например, деление его на константу или инвертирование. В данной работе реализовывать его нет смысла, т.к. он является самым неэффективным способом расчета *CRC*;

2. Табличный метод (оптимизированный). Для ускорения вычислений *CRC* был разработан табличный метод. Его идея заключается в том, что часто повторяющиеся операции *XOR* и сдвигов можно заменить обращением к заранее вычисленным значениям.

Основные шаги алгоритма:

- 1) байт входных данных загружается в регистр, после чего сдвигается;
- 2) выполняется *XOR* между загруженным байтом и старшим байтом регистра;
- 3) полученный результат используется в качестве индекса в таблице предрасчитанных значений;

4) содержимое регистра обновляется с использованием найденного значения;

5) повторить, пока не будут обработаны все байты.

Такой подход значительно повышает производительность, особенно при больших объёмах данных. Более того, можно реализовать оптимизации с обработкой не по одному байту, а сразу по 2, 4 или даже 8 байт за итерацию. Однако слишком большие размеры обрабатываемого блока могут перегрузить системную шину или привести к потере эффективности из-за ограничений архитектуры.

– метод параллельного расчета *CRC*. Современные процессоры расширили возможности параллелизма на уровне команд и параллелизма на уровне потоков. Эти ресурсы используются в алгоритмах для ускорения циклической проверки избыточности (*CRC*).

Основная идея параллельного алгоритма заключается в полном использовании современных процессоров с точки зрения параллелизма как на уровне команд, так и на уровне потоков.

Во-первых, детально проработанный алгоритм выполняет вычисление *CRC* чередующимся образом, так что несколько независимых потоков данных могут обрабатываться одновременно. Этот алгоритм допускает параллелизм на уровне команд, который утраивает и удваивает производительность существующих алгоритмов.

Во-вторых, крупнозернистый алгоритм в идеале может параллельно обрабатывать данные путем распараллеливания семейства последовательных алгоритмов генерации *CRC*. Таким образом, этот алгоритм допускает параллелизм на уровне потоков, что позволяет в полной мере использовать возможности многоядерных вычислений. В результате достигается ускорение, почти равное количеству используемых потоков.

Метод параллельного расчета *CRC* заключается в одновременном чтении нескольких байтов. Однако размер справочной таблицы увеличивается экспоненциально с одновременным считыванием битов, что может привести

к тому, что справочная таблица не сможет поместиться в кэш. Записи требуются для достижения ускорения за счет чтения 16 бит за раз. Алгоритм с разделением на 4 [43] считывает 32 бита за раз, при этом требования к памяти оптимизированы для того, чтобы занимать разумные объемы кэша. На каждой итерации, 32-битный фрагмент считывается из буфера данных и разбивается на четыре фрагмента. Затем выполняются четыре поиска в таблице, которые используют эти четыре фрагмента в качестве индексов. Этот алгоритм параллельного вычисления может выполнять поиск по четырём таблицам одновременно. Также этот метод может быть улучшен и осуществлять поиск по 8, по 16 таблицам одновременно.

Этот алгоритм делает существующие табличные *CRC*-алгоритмы пригодными для одновременной обработки нескольких независимых потоков данных, что обеспечивает параллелизм на уровне инструкций. В результате это утраивает и удваивает производительность табличных методов. Этот алгоритм может идеально распараллеливать обычные *CRC*-алгоритмы, что обеспечивает параллелизм на уровне потоков.

Контроль состояния памяти с использованием *CRC* реализуется путём периодического сканирования заданных участков оперативной памяти. Результаты хеширования сохраняются и сравниваются при следующих проходах анализа. Любое несоответствие контрольной суммы фиксируется как потенциальная угроза – будь то атака или несанкционированное вмешательство извне.

Для повышения надёжности мониторинга следует:

- проводить проверки в случайные моменты времени, чтобы атаки было сложнее предсказать;
- контролировать как кратковременные, так и долговременные области памяти;
- хранить результат хеширования в изолированной памяти, исключая вмешательство злоумышленников.

CRC, несмотря на свою простоту, остаётся эффективным решением в задачах мониторинга состояния памяти. Её способность фиксировать малейшие изменения и быстро обрабатывать большие объёмы информации делает её ценной частью архитектуры защиты от атак, связанных с вмешательством в оперативную память. При правильной реализации *CRC* может стать надёжной основой для систем раннего предупреждения и контроля целостности данных в памяти.

2.2 Разработка консольного приложения для реализации различных методов расчета хеш-функции *CRC*

Для проверки производительности различных методов расчёта *CRC* применительно к задачам мониторинга памяти было разработано консольное приложение для реализации различных методов расчета хеш-функции *CRC* на языке *C* (Приложение Б). В качестве примера расчета выбрана хеш-функция *CRC16*.

Основная задача программы – вычислить контрольную сумму *CRC16* для каждого файла в директории и вывести результаты. *CRC16* – это способ генерации уникального 16-битного значения (контрольной суммы) для фрагмента данных с целью проверки его целостности. Представьте, что это отпечаток ваших данных. Программа измеряет время, затрачиваемое на вычисление *CRC* различными методами, и может использовать несколько ядер процессора для параллельного вычисления *CRC* одним из этих методов. Она представляет собой сочетание работы с файлами, многопоточность и сложные математические вычисления *CRC*.

Консольное приложение включает в себя следующие файлы:

– *main.c* – это точка входа. Этот файл настраивает всё, считывает каталог и организует вычисления *CRC*;

– *stdio.h* – библиотека, которая используется для работы с функциями ввода-вывода:

1) *printf*;

2) *fprintf*;

3) *stdout*;

4) *stderr*;

– *stdlib.h* – библиотека, которая используется для:

1) работы с памятью (*malloc*, *free*);

2) преобразования типов (*atoi*, *strtol*);

3) выхода из программы (*exit*, *abort*);

– *stdint.h* – библиотека, которая необходима для использования типов с фиксированной разрядностью:

1) *uint8_t*;

2) *uint16_t*;

3) *uint32_t*.

Это важно для *CRC* и многопоточной обработки;

– *locale.h* – библиотека, которая настраивает языковую среду программы. *Setlocale(0, "Russian")* устанавливает русский язык для вывода;

– *time.h* – библиотека, которая позволяет использовать функции для работы со временем:

1) *clock_gettime*;

2) *time*;

3) *difftime*.

В коде используется для измерения времени выполнения *CRC* (в *CRC_TEST_MODE*);

– *windows.h* – библиотека, которая используется для доступа к *APIWindows*, а именно использование функции *GetSystemInfo* чтобы динамически определить количество доступных логических ядер процессора и задать соответствующее количество потоков для параллельного вычисления *CRC*. Это позволяет максимально эффективно использовать ресурсы системы;

– *pthread.h* – библиотека, которая необходима для работы с потоками *POSIX*:

1) *pthread_create*;

2) *pthread_join*;

3) *pthread_exit*;

– *pthread_time.h* – библиотека, являющаяся расширением *POSIX*, которое дополняет *pthread* возможностью управления временем ожидания в многопоточной среде;

– *crc.h* – библиотека, используемая для объявления функций и структур для вычисления *CRC*:

1) *crc_make*;

2) *crc_make_table*;

3) *crc_shiftByZeros*;

4) *crc_тидр.* ;

– *filesList.h* – библиотека, используемая для обрабатывания чтения списка файлов из каталога:

1) *readFilesFromDir*;

2) *getFilesAmount*;

3) *printFilesNames*;

4) *getArrayByIndexтидр.*;

– *pthread_queue.h* – библиотека, в которой объявлены структуры и функции для работы с очередями, поддерживающими многопоточную работу:

1) *queue_init*;

2) *queue_push*;

3) *queue_pop*;

4) *queue_drop*;

– *filesList.c* – обрабатывает чтение файлов из каталога и сохраняет их имена, пути и размеры;

– *crc.c* – реализует логику вычисления *CRC16* различными способами:

1) по 8 бит;

2) по 16 бит;

3) по 32 бита;

4) параллельный;

– *pthread_queue.c* – реализует потокобезопасную очередь для управления задачами при выполнении параллельных вычислений *CRC*;

– *pthread_queue_def.h* – определяет тип элемента очереди (*qitem_t*).

Работа консольного приложения:

1. Осуществляется запуск программы. Можно запустить её без аргументов (она будет использовать текущий каталог) или передать в качестве аргумента путь к одному каталогу. Если передать более одного аргумента, программа выдаст ошибку и завершится. Используется *Setlocale(0, «Russian»)* чтобы программа могла обрабатывать русские символы в именах файлов или в выводе на консоль, поскольку это удобно для пользователя.

Происходит вызов *readFilesFromDir(путь)* из *filesList.c*, чтобы просканировать каталог и составить список файлов.

2. Чтение файлов. Модуль *filesList* предназначен для получения списка файлов из каталога. Процесс получения списка файлов из каталога:

1) запись информации о файлах в 3 глобальных массивах:

– *fileList_nameFile* (только имена файлов);

– *fileList_fullPath* (полные пути);

– *fileList_dataSize* (размеры файлов).

Также есть структура *fileList_t* для отслеживания дескриптора каталога и количества файлов (*fl.size*);

2) функция *init* обнуляет массивы, чтобы избежать появления мусорных данных;

3) *readFilesFromDir()* открывает каталог с помощью *tinydir_open*. Происходит перебой каждой записи, пропуская каталоги, и для каждого файла:

– выделяет память для хранения имени и пути;

– копирует имя файла и путь к нему в *fileList_nameFile* и *fileList_fullPath*;

– сохраняет размер файла в *fileList_dataSize*;

– увеличивает количество файлов (*fl.size*);

Если файлов слишком много (больше чем *MAX_FILES_AMOUNT* = 256), выдается ошибка (*FILE_LIST_AMOUNT*);

4) *getFilesAmount()* возвращает количество найденных файлов;

5) *printFilesNames()* выводит все имена файлов с необязательным заголовком;

6) *getArrayByIndex*(индекс, данные) считывает файл по заданному индексу в динамически выделяемый буфер и возвращает его содержимое с указанием размера в данных.

Этот модуль позволяет легко извлекать все файлы и их характеристики, которые нужны для последующего расчёта *CRC*.

3. Настройка расчёта *CRC*. В *main.c* была создана структура *crc_t* для хранения конфигурации *CRC*:

1) *flag*– определяет, будут ли биты ввода/вывода инвертированы;

2) *poly* – образующий многочлен для *CRC16*, равен 0x94ED;

3) *init*– начальное значение *CRC*, равное 0x0000;

4) *xor* – значение *XOR*, применяемое в конце, также 0x0000;

5) *table* – справочная таблица для более быстрого вычисления *CRC*.

Следующая строка *crc_make_table(&crc_set)* необходима для инициализации таблицы поиска. В режиме (*CRC_TEST_MODE*) она всегда генерирует таблицу динамически и подготавливает дополнительные таблицы для методов расчета *CRC*.

4. Режимы вычисления *CRC*. В консольном приложении реализовано несколько способов вычисления *CRC16* для сравнения их производительности:

1) вычисление по байту (8 бит) (*crc_byte*). Используется предварительно рассчитанная таблица поиска для обработки одного байта за раз. Гораздо быстрее, чем побитовая обработка, поскольку позволяет избежать повторных битовых манипуляций;

2) вычисление по 2 байта (16 бит) (*crc_half*). Обрабатывается два байта за раз с помощью двух справочных таблиц, предполагая, что размер входных данных в основном чётный. Оптимизирован для 16-битных операций;

3) вычисление по 4 байта (32 бит) (*crc_word*). Обрабатывается по четыре байта за раз с помощью четырёх таблиц поиска, ориентированных на 32-битную эффективность. Это самый быстрый однопоточный метод для больших файлов;

4) параллельное вычисление (*crc_parallel_make*). Получая информация о количестве ядер процессора с помощью функции *GetSystemInfo*, происходит сохранение значения в *cpuAmount*. Происходит инициализация двух очередей (*queue2processTasks* и *queue2showResults*) с помощью *queue_init* из *pthread_queue.c*.

Файл разбивается на примерно равные фрагменты, для простоты выровненные по 16-байтовым границам. Каждый фрагмент присваивается структуре *task_data_t* со следующими параметрами:

- 1) *offset* – указатель на начало фрагмента;
- 2) *size* – размер фрагмента;
- 3) *zeros* – количество байт, оставшихся после этого фрагмента (используется для комбинации CRC).

Далее происходит создание потоков количество потоков, каждый из которых выполняет задачу. А именно, каждый поток:

- 1) извлекает индекс из *queue2processTasks*;
- 2) вычисляет CRC для своего фрагмента с помощью *crc_make*;
- 3) корректирует CRC с помощью *crc_shiftByZeros* для учёта оставшихся байтов (нулей);
- 4) результат сохраняется в *crc_out[index]*.

Далее происходит объединение результатов. После завершения работы всех потоков (с помощью *pthread_join*) выполняется операция XOR для всех CRC фрагментов, чтобы получить окончательный CRC (*crc_set.init*).

Модуль *pthread_queue* используется для создания потокобезопасности, он реализует:

- 1) циклическую очередь (*queue_t*) с мьютексом (*mtx*) и переменной условия (*cond*) для синхронизации;
- 2) *queue_push* добавляет элемент (индекс) в очередь, ожидая, пока очередь не заполнится;
- 3) *queue_pop* удаляет элемент, ожидая, пока очередь не станет пустой и не закроется;
- 4) *queue_trypush* и *queue_trypop* – это неблокирующие версии.
- 5) *queue_close* помечает очередь как закрытую, поэтому потоки перестают ожидать;
- 6) *queue_drop* очищает память очереди.

Такая настройка позволяет потокам безопасно выполнять задачи, не мешая друг другу.

Функция *crc_make* в *crc.c* определяет, какой метод использовать.

5. Выполнение вычисления CRC. В консольном приложении для вычисления CRC для каждого файла, используется функция *calulate_crc_time* с аргументом соответствующий каждому методу:

- 1) *CRC_TEST_BIT_08* (8 бит);
- 2) *CRC_TEST_BIT_16* (16 бит);
- 3) *CRC_TEST_BIT_32* (32 бита);
- 4) *CRC_TEST_MULTI* (параллельный расчет).

Измеряет время, затраченное на каждый метод, с помощью *clock_gettime* и выводит значение CRC и время в миллисекундах в консоль. Конечный вывод в качестве примера, показан на рисунке 2.1:

```

C:\Soft\CODEBLOCK\20\bin\Debug\20.exe
Open directory TEST
Files in dir:
TEST_100.mp4
CRC16 - 0x5F73, Time - 726 ms
CRC16 - 0x5F73, Time - 259 ms
CRC16 - 0x5F73, Time - 175 ms
CRC16 - 0x5F73, Time - 34 ms

Process returned 0 (0x0)   execution time : 1.519 s
Press any key to continue.

```

Рисунок 2.1 – Вывод результата консольного приложения для файла размером 100Мб

Было выполнена верификация разработанного приложения на персональных электронно- вычислительных машинах (ПЭВМ) со следующими характеристиками, указанными в таблице 2.1:

Таблица 2.1 –Характеристики ПЭВМ

№ ПЭВМ	Процессор с тактовой частотой	Объем ОЗУ	Объем ПЗУ	Кол-во ядер/ потоков
№1	<i>Intel PentiumN3530</i> с 2.2ГГц	4 Гб	<i>SSD 128 Гб</i>	4/4
№2	<i>Intel Corei7-3770K</i> с 3.5ГГц	16 Гб	<i>SSD 512 Гб</i>	4/8
№3	<i>AMD Ryzen 7 7700X</i> с 4.5ГГц	32 Гб	<i>SSD 1 Тб</i>	8/16

Проверялись на заранее выделенные блоки памяти объёмом от 0,1 Гб до 1,5 Гб являющиеся рандомизированными наборами данных файлового типа. Результатыпредставлены в таблицах 2.2 -2.4:

Таблица 2.2 – Результаты времени расчета CRC16на ПЭВМ №1

Размер файла	Табличный метод (1 байт), мс	Табличный метод (2 байта), мс	Табличный метод (4 байта), мс	Параллельный метод ,мс
0,1 Гб	1109	594	500	124
0,2 Гб	2046	1109	922	235

0,3 Гб	3218	1735	1454	374
0,4 Гб	5107	2797	2344	625
0,5 Гб	5577	3015	2515	750
1,1 Гб	11828	6374	5360	1391
1,5 Гб	16686	8984	7485	2407

Таблица 2.3 – Результаты времени расчета CRC16 на ПВЭМ №2

Размер файла	Табличный метод (1 байт), мс	Табличный метод (2 байта), мс	Табличный метод (4 байта), мс	Параллельный метод ,мс
0,1 Гб	746	266	188	37
0,2 Гб	1359	488	326	80
0,3 Гб	2201	793	541	108
0,4 Гб	3439	1232	834	183
0,5 Гб	3702	1331	897	173
1,1 Гб	7942	2885	1910	420
1,5 Гб	10971	3966	2651	534

Таблица 2.4 – Результаты времени расчета CRC16 на ПВЭМ №3

Размер файла	Табличный метод (1 байт), мс	Табличный метод (2 байта), мс	Табличный метод (4 байта), мс	Параллельный метод ,мс
0,1 Гб	341	114	68	8
0,2 Гб	619	209	124	15
0,3 Гб	981	328	195	23

0,4 Гб	1579	527	320	32
0,5 Гб	1682	570	338	38
1,1 Гб	3572	1200	719	70
1,5 Гб	5009	1682	1040	98

Как видно из таблиц, параллельный метод вычисляет *CRC* быстрее табличных методов, соответственно, нужно меньшее количество тактов для расчета *CRC*. На эффективность параллельного вычисления оказывает влияние:

- кол-во ядер/ потоков;
- тактовая частота процессора;
- кол-во транзисторов;
- размер кеша процессора.

Контроль состояния памяти вычисленный с помощью параллельного метод показывает высокую эффективность для обнаружения несанкционированных изменений данных. Проведённые эксперименты доказывают, что параллельный метод расчёта *CRC* позволяют добиться скорости выше, чем табличные методы. Учитывая низкие вычислительные накладные расходы, *CRC* может быть успешно применена в реальном времени для контроля целостности памяти программ и системных процессов.

Таким образом, использование параллельного метода расчета *CRC* является практичным решением для построения надёжных систем мониторинга безопасности в информационной инфраструктуре. Это приведет к более быстрому расчету *CRC* за счет уменьшения количества тактов на выполнение операций, соответственно повысится надежность системы, а также понизится энергопотребление.

2.3 Выработка предложения по реализации мандатной защиты на основе контроля состояния памяти с помощью хеш- функции CRC

Используя параллельный расчет *CRC* в качестве контроля состояния памяти, для реализации мандатной защиты. Выработано несколько предложений, которые можно реализовать:

1. Высоконагруженный вычислитель с общими ресурсами. Этот подход интегрирует *CRC* в механизм мандатной защиты для мониторинга целостности критических данных в памяти. Каждому ресурсу (например, сегменту памяти) присваивается мандат (уровень доступа) и контрольная сумма *CRC*, которая проверяется при операциях чтения/записи. На рисунке 2.2 представлен схема реализации высоконагруженного вычислителя с общими ресурсами.

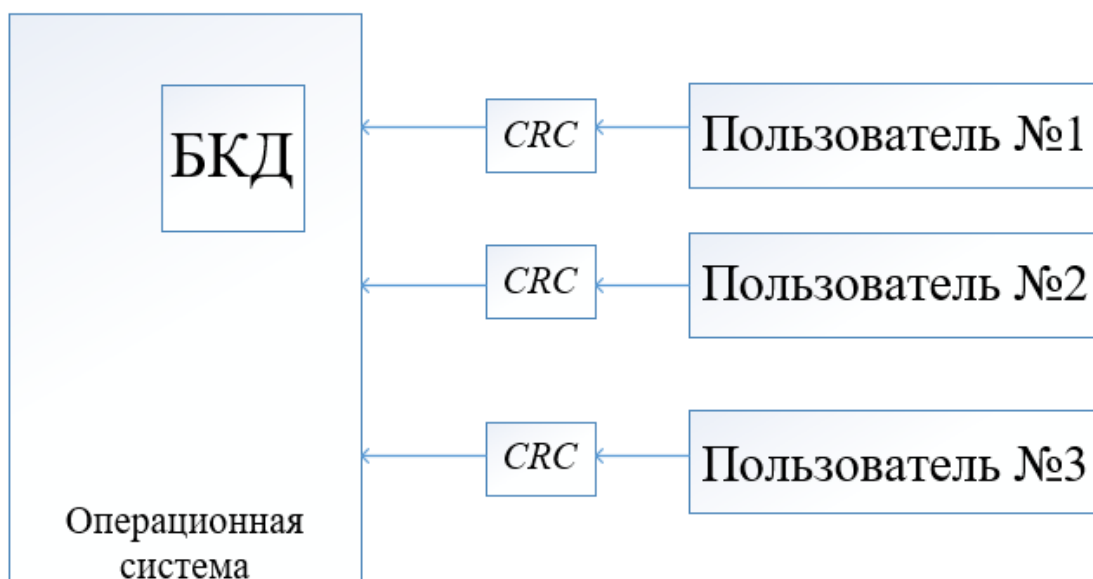


Рисунок 2.2 – Схема реализации высоконагруженного вычислителя с общими ресурсами

Каждому пользователю выделяется свой блок памяти и назначается:

– мандат (уровень доступа, например секретно или совершенно секретно);

- *CRC*, вычисленный при инициализации (хранится в защищенной области блока контроля доступа (БКД)).

Блок контроля доступа управляет двумя аспектами:

- проверка мандата (сравнение уровня доступа пользователя и ресурса);
- верификация *CRC* (контроль целостности данных перед доступом).

Принцип работы системы:

- запрос доступа. Пользователь с мандатом №2 запрашивает доступ к ресурсу №2;

- проверка мандата. БКД сравнивает мандат пользователя №2 и ресурса №2. Если уровни совпадают, происходит переход к проверке целостности;

- проверка *CRC*:

- 1) БКД пересчитывает *CRC* текущего состояния памяти ресурса №2;

- 2) сравнивает с эталонным *CRC*, хранящимся в защищенной таблице;

- 3) если *CRC* совпадает, значит доступ разрешен. Если *CRC* не совпадает, значит данные изменены несанкционированно и доступ блокируется, система генерирует тревогу.

Сценарий нарушения целостности:

- атака. Злоумышленник (пользователь №1) пытается изменить данные пользователя №2 через уязвимость;

- обнаружение. При попытке обращения к ресурсам пользователя №2 БКД пересчитывает *CRC*. Новый *CRC* не совпадет с эталонным, соответ-

ственно доступ будет запрещен. Система фиксирует попытку нарушения целостности.

Преимущества данного подхода:

—двойная защита:

1) мандатная защита ограничивает доступ на основе привилегий;

2) *CRC* детектирует изменения данных (случайные или злонамеренные);

– гарантия целостности. Ресурс не будет обработан, если его состояние не соответствует эталону;

– простота интеграции:

1) *CRC* вычисляется быстро, используя параллельное вычисление;

2) предлагаемая система менее требовательная к вычислительным ресурсам, в отличии от конкурентов.

Такой подход подходит для систем, где целостность данных не менее важна, чем конфиденциальность;

2. Система защиты от *glitch*-атак. *Glitch*-атаки – это класс физических атак на электронные системы, где злоумышленник целенаправленно создает кратковременные аномалии в работе устройства. Основные типы атак:

1) тактовые атаки (*ClockGlitching*)

– внедрение аномальных импульсов в тактовый сигнал. Например, удлинение или укорочение тактовых периодов во время выполнения критических инструкций с целью пропуска операций проверки;

2) атаки по питанию (*VoltageGlitching*):

– кратковременные скачки или просадки напряжения питания от 0.5 до 5В и длительностью 1-500 нс. Это может происходить во время опе-

рации сравнения пароля. Произойдет Искажение результатов вычислений в регистрах процессора;

3) электромагнитные атаки (*EM Glitching*):

– точечное электромагнитное воздействие на конкретные компоненты кристалла, для этого не требуется физического контакта с устройством [44].

Glitch-атака может пропустить условный переход, выполнив привилегированную команду без проверок.

Аппаратно-программные меры противодействия:

1) временная избыточность. Тройное исполнение критических проверок с временным разделением;

2) пространственное разделение:

– физическое размещение компонентов защиты.

– использование *MemoryProtectionUnit (MPU)* для блокировки доступа к зоне *CRC*;

3) использование аппаратных ускорителей:

– в *STM32* используется *CRC Calculation Unit (CRC-CU)*;

– *NXP* используется *Hardware CRC Engine*;

– в *ESP32* используется *Cryptographic Accelerator* с поддержкой *CRC*.

Система мониторинга в реальном времени показана на рисунке 2.3:

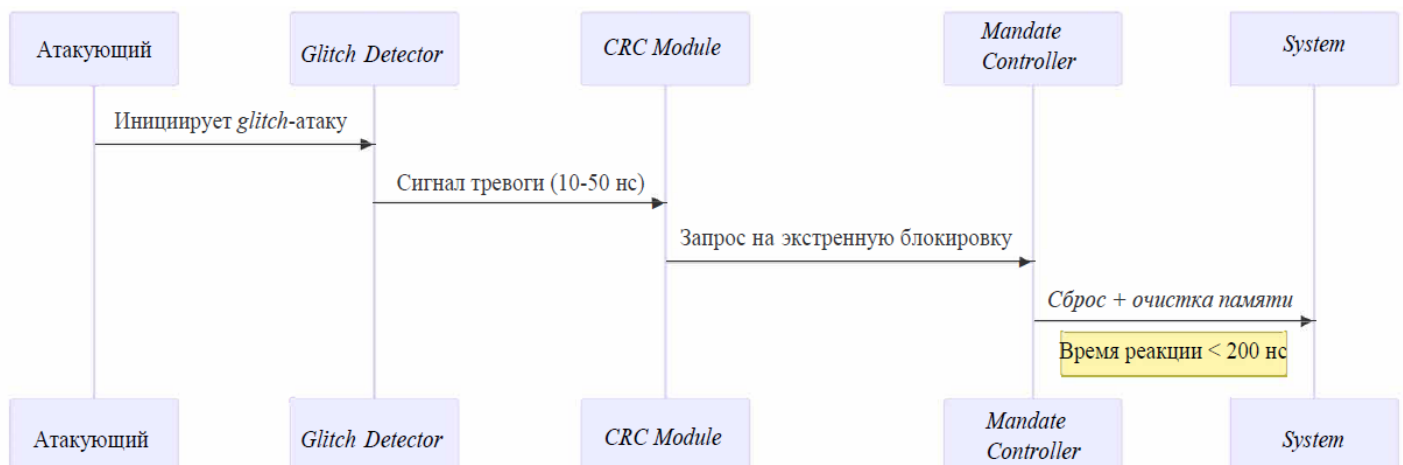


Рисунок 2.3 – Система мониторинга в реальном времени

Механизм работы:

1) инициирование атаки. Злоумышленник применяет кратковременные аномалии, которые были описаны ранее;

2) детектирование аномалии (10-50 нс). Аппаратные сенсоры непрерывно совершают мониторинг:

- аналоговые компараторы отслеживают отклонение напряжения $\pm 15\%$;

- счетчики тактов детектируют аномалии частоты;

- датчики электромагнитного поля детектируют аномалии;

3) эскалация сигнала в *CRC*-модуль:

- аппаратное прерывание с максимальным приоритетом;

- автоматическая проверка целостности критических областей.

Если выявлено что *CRC* совпадает переходит на следующий шаг ;

4) блокировка системы (реакция < 200 нс). Контроллер мандатной защиты выполняет:

- немедленную очистку чувствительных регистров;

- активацию аварийного сброса через *WatchdogTimer*;

- запись события в защищенный *EEPROM* с меткой времени.

Современные решения позволяют снизить успешность атак до $< 1\%$, но требуют 15-25% вычислительных ресурсов, использование *CRC* уменьшает требовательность к вычислительным ресурсам. Для критически важных систем такие затраты оправданы необходимостью обеспечения максимальной безопасности.

Кроме описанных ранее применений параллельного расчета *CRC*, в ходе выполнения ВКР было найдено еще одно эффективное применение – увеличения вероятности безошибочного приёма пакетных данных.

Списочный декодер Витерби (*LVA - ListViterbiAlgorithm*) используется в области высоких соотношений сигнал/шум (10^{-2} и выше) для увеличения вероятности безошибочного приёма пакетных данных.

Структура *LVA* представлена на рисунке 2.4:

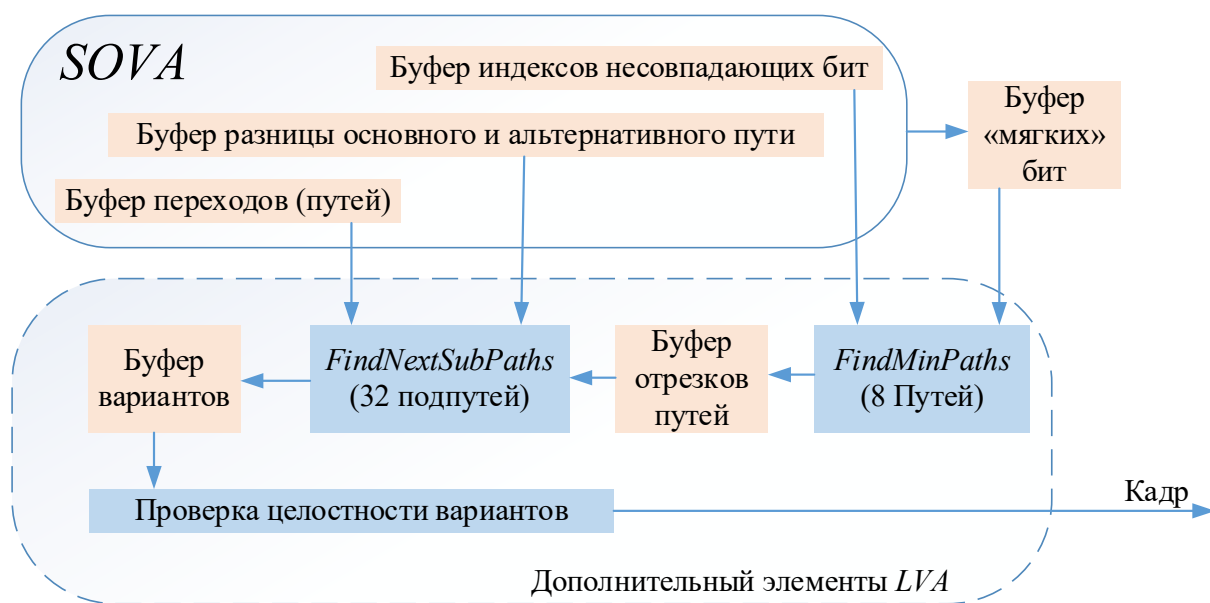


Рисунок 2.4 – Структура *LVA*

Алгоритм Витерби реализует декодер максимального правдоподобия который находит наиболее вероятную последовательность(путь).Декодер Витербиявляетсяоптимальным*ML(Maximum-likelihood)*-декодером. При использовании проверочной контрольной комбинации, например *CRC*, правильная исходная последовательность может быть найдена в альтернативных путях декодера. Классический алгоритм реализуется на основе стека (обхода дерева) что ограничивает кол во возможных вариантов, анализируемых в последовательности до нескольких десятков в реальном масштабе времени.Однако, достаточно часто в особенностях использования дифференциальной модуляции в декодере существуют биты (узлы) которые принадлежат одновременно как альтернативному, так и основному пути. Вы-

ражение «достаточно часто» означает что при реальном соотношении сигнал/шум (когда система синхронизации позволяет успешно демодулирует символы) такое происходит всегда. Что позволяет использовать алгоритм разработанной для расчета контроля целостности что повышает кол-во анализируемых вариантов, что в свою очередь повышает вероятность успешного декодирования на 3 дБ.

Пример работы алгоритма *LVA* представлена на рисунке 2.5:

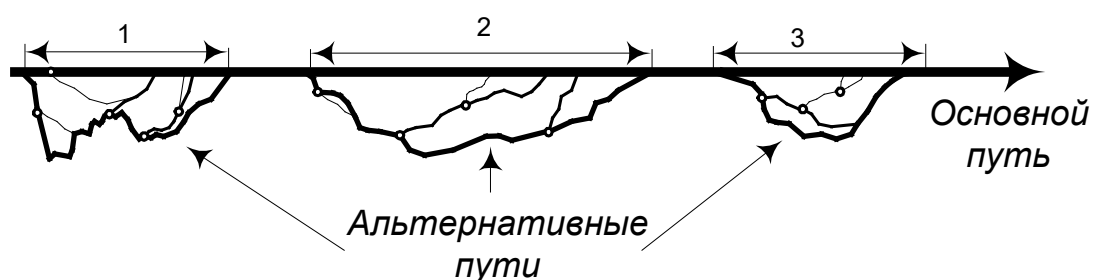


Рисунок 2.5 – Пример работы алгоритма *LVA*

Алгоритм *LVA* находит несколько альтернативных путей с минимальной надёжностью. Поиск выполняется в двух направлениях (прямом и обратном) поскольку пути могут сливаться с основным путем из альтернативного так же в обоих направлениях.

Анализ кода последовательного расчёта *CRC16* показывает, что один бит требует не менее 8 операций и при размере кадра 784 бит затратит около 6 тыс. операций. Вычисления можно существенно ускорить используя заранее рассчитанную таблицу решений (например, байтовую). Но и в этом случае количество операций для вычисления *CRC16* составляет около 600.

При количестве ошибок $5 \cdot 10^{-2}$ ошибок (5 %), количество анализируемых отрезков (в среднем) 6 при количестве путей в отрезке (максимально) 32 (2^5).

В итоге, необходимо рассчитывать *CRC16* кадра не менее чем 2^{30} раз, что при количестве операций 600 на кадр составит более 500 млрд. операций. Для увеличения эффективности предлагается применение параллельного вычисления, схема работы предлагаемого алгоритма показана на рисунке 2.6:

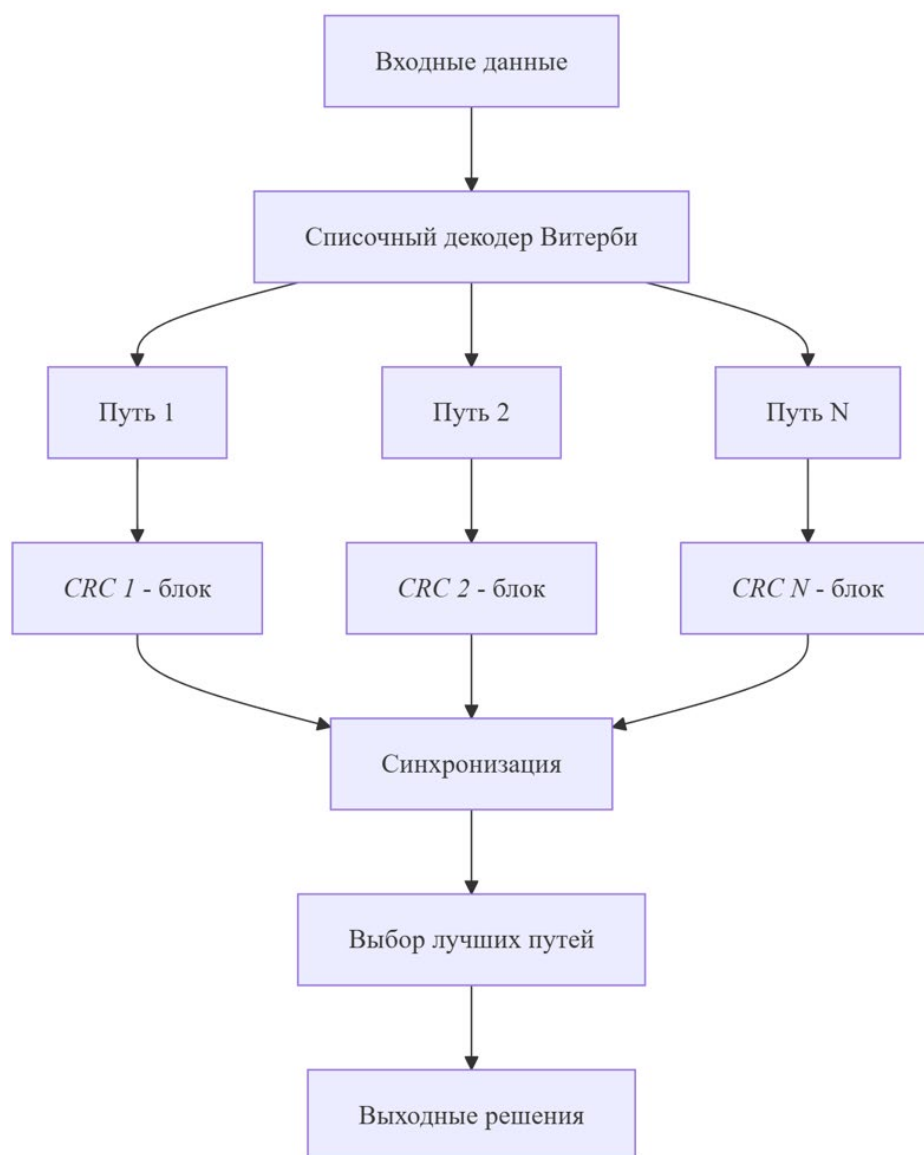


Рисунок 2.6 –Схема расчета альтернативных путей с применением параллельного вычисления *CRC*

Алгоритм схемы расчета путей с применением параллельного вычисления *CRC*:

- входные данные. Из канала связи идет прием символов, содержащих полезную информацию и шум;
- списочный декодер Витерби. Происходит обработка входных данных, осуществляется поиск не пересекающихся альтернативных путей;
- параллельные пути обработки. Каждый путь представляет возможную траекторию полученного по методу максимального правдоподобия(фильтра);

- *CRC*-блоки. Производят вычисление контрольной суммы для своего пути и осуществляют автоматическое отсечение некорректных путей;
- синхронизация. Объединяет результаты от всех *CRC*-блоков:
 - 1) сравнивает контрольные суммы с эталоном;
 - 2) формирует список наиболее вероятных путей;
- выбор лучших путей, на основе метрики вероятности (максимальное значение) и успешной проверки *CRC*. Отбирает наиболее достоверные пути для финального решения;
- выходные решения. Формирует единственно верный путь декодирования.

Данная схема обеспечивает эффективное декодирование в системах связи с высокой скоростью передачи данных и требованиями к надежности, обеспечивая увеличения вероятности безошибочного приёма пакетных данных.

При использовании *VLIW*-ядра *DelCore-30M* вычисление *CRC16* можно конвейеризировать для всех кадров пакета, тогда для кадра потребуется не более 120 тактов, что при частоте 500 МГц составит не более 2 % мощности ПЦОС на кадр.

3 Техника безопасности и охрана труда офисного сотрудника

3.1 Идентификация потенциальных опасностей и вредных факторов. Микроклимат рабочей зоны

В офисной среде, особенно при длительной работе за компьютером, сотрудники подвергаются воздействию различных негативных факторов. Условно все опасности делятся на следующие группы:

1. Физические воздействия, среди которых:

- несоответствующая температурная среда (перегрев или переохлаждение помещения);
- недостаточный или избыточный уровень освещения;
- наличие сквозняков;
- электромагнитные поля от электронного оборудования;
- статическое электричество;
- акустическое загрязнение (шум от техники, систем вентиляции, разговоров);
- длительное сидячее положение, вызывающее мышечное напряжение.

2. Химические факторы:

- испарения от пластиковых корпусов и деталей офисного оборудования;
- загрязнение воздуха тонером, особенно при использовании лазерных принтеров и копиров.

3. Психофизиологические и нервно-эмоциональные нагрузки:

- продолжительная концентрация внимания на экране;
- напряжение зрительного анализатора;

- однообразие задач, вызывающее переутомление;
- стрессовые ситуации, обусловленные дедлайнами и взаимодействием с клиентами или коллегами.

4. Биологические угрозы, как правило, минимальны, но не исключены:

- возможность заражения инфекциями при контакте с плохо дезинфицированными рабочими поверхностями или оборудованием.

Комфортные климатические условия на рабочем месте являются основой эффективной работы и профилактики заболеваний. В соответствии с гигиеническими нормативами, рекомендованы следующие параметры:

- температура воздуха в диапазоне от 21°C до 23°C;
- относительная влажность воздуха – от 40% до 60%;
- скорость движения воздуха не более 0,1 м/с.

Нарушения этих параметров могут привести к ухудшению самочувствия, снижению работоспособности и развитию хронических заболеваний. В офисах следует предусматривать возможность регулирования микроклимата с помощью отопительных приборов и систем кондиционирования воздуха. Регулярное проветривание помещений обязательно.

3.2 Освещение и шумовая нагрузка. Влияние электромагнитных и электростатических полей

Офис должен быть оснащён как естественным, так и искусственным освещением. Яркость и расположение источников света должны исключать появление бликов на экране монитора и создавать равномерную засветку рабочей поверхности. Освещённость рабочих зон – не менее 500 лк.

Недостаточное освещение приводит к быстрой утомляемости глаз и снижению концентрации. Для устранения бликов следует использовать антибликовые экраны и шторы.

Шум в офисе (от оргтехники, разговоров, вентиляции) должен не превышать 50 дБ по санитарным нормам. При превышении этого уровня применяются следующие меры:

- шумоизоляция помещений;
- размещение шумных устройств в отдельных помещениях;
- организация «тихих зон» для сосредоточенной работы.

Электронные устройства, особенно в больших количествах, создают электромагнитное поле, которое при длительном воздействии может негативно влиять на здоровье сотрудников. Основные симптомы — головные боли, быстрая утомляемость, нарушения сна.

Для снижения риска воздействия рекомендуется:

- использование современного оборудования с низким уровнем излучения (например, ЖК- и *LED*-мониторов);
- правильная организация рабочего пространства (расстояние до монитора не менее 50–70 см);
- заземление всех электроприборов; – соблюдение регламентированных перерывов.

Электростатическое напряжение также может быть источником дискомфорта. Использование антистатических ковров и кресел с антистатическим покрытием помогает минимизировать этот фактор.

3.3 Электробезопасность. Пожарная безопасность

Рабочее место должно быть оборудовано безопасной электропроводкой с обязательным заземлением и автоматическими выключателями. Электроприборы должны регулярно проходить проверку на целостность изоляции и исправность.

Офисный персонал обязан:

- не подключать устройства с повреждёнными кабелями;
- избегать перегрузки розеток; – немедленно сообщать о неисправности техники;
- не пользоваться электроприборами без соответствующего обучения.

Во всех помещениях должна быть размещена инструкция по электробезопасности, а также проводиться регулярный инструктаж.

Офисные помещения, в которых размещается большое количество электротехники и документации, должны быть оборудованы средствами пожаротушения: огнетушителями, системами пожарной сигнализации и аварийного оповещения.

Наиболее вероятные причины возгорания:

- короткое замыкание;
- перегрев оборудования;
- использование неисправных электроприборов;
- курение в непредназначенных для этого местах.

Профилактические мероприятия включают:

- регулярную проверку технического состояния электросетей;
- обучение персонала действиям при пожаре;
- поддержание свободных путей эвакуации;
- наличие планов эвакуации на видимых местах.

В случае обнаружения возгорания необходимо:

- немедленно отключить электричество;
- сообщить ответственным лицам и вызвать пожарную службу;
- при возможности – использовать огнетушитель;
- покинуть здание согласно маршрутам эвакуации.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы были достигнуты следующие результаты:

1. Выполнен аналитический обзор научно-технической литературы по теме исследования реализации мандатной защиты на основе контроля состояния памяти с помощью хеш-функции.

2. Разработана методика контроля состояния памяти с помощью хеш-функции *CRC*.

3. Разработан алгоритм расчета хеш-функции *CRC* для многоядерного процессора.

4. Разработано консольное приложение для реализации различных методов расчета хеш-функции *CRC*.

5. Выполнена верификация разработанного приложения на различной аппаратной платформе (3-х ПЭВМ) на рандомизированном наборе данных файлового типа.

6. Определен критерий эффективности, а именно время расчета *CRC* и проведен сравнительный анализ методов расчета хеш-функции *CRC* по этому критерию.

7. Выработаны предложения по мандатной защите на основе контроля состояния памяти с помощью хеш-функции *CRC*.

В работе исследован вопрос реализации мандатной защиты на основе контроля состояния памяти с помощью хеш-функции. Выработаны предложения используя параллельный расчет *CRC* в качестве контроля состояния памяти, для реализации мандатной защиты. Предложено 2 реализации, которые можно применить:

– высоконагруженный вычислитель с общими ресурсами. Данная реализации описана выше и подходит для систем, где целостность данных не менее важна, чем конфиденциальность;

– система защиты от *glitch*-атак. Предложены аппаратно-программные меры противодействия и описан механизм работы предложенной реализации. Благодаря использованию параллельного расчета *CRC* на уровне команд требовательность к ресурсам уменьшается.

Кроме предложенных реализаций применения параллельного расчета *CRC* в мандатной защите основанной на контроле состояния, в ходе выполнения ВКР было найдено еще одно эффективное применение – увеличения вероятности безошибочного приёма пакетных данных, для этого используется списочный декодер Витерби.

Он находит несколько альтернативных путей с минимальной надёжностью. Поиск выполняется в двух направлениях (прямом и обратном) поскольку пути могут сливаться с основным путем из альтернативного так же в обоих направлениях.

Анализ кода последовательного расчёта *CRC16* показывает, что один бит требует не менее 8 операций и при размере кадра 784 бит затратит около 6 тыс. операций. Вычисления можно существенно ускорить используя заранее рассчитанную таблицу решений (например, байтовую). Но и в этом случае количество операций для вычисления *CRC16* составляет около 600.

Предложенная реализация обеспечивает эффективное декодирование в системах связи с высокой скоростью передачи данных и требованиями к надежности, обеспечивая увеличения вероятности безошибочного приёма пакетных данных.

Предложенная система была применима используя *VLIW*-ядро *DelCore-30M*. Вычисление *CRC16* используя *VLIW*-ядер можно конвейеризировать для всех кадров пакета, тогда для кадра потребуется не более 120 тактов, что при частоте 500 МГц составит не более 2 % мощности ПЦОС на кадр.

Также в рамках ВКР были написаны статьи:

– Головяшкин А.Н., Козин А. С. , Сухов И.В. Алгоритмы вычисления циклического избыточного кода // Труды международного симпозиума "Надежность и качество". – 2024. – Т. 1. – С. 421-422;

– Головяшкин А.Н., Сухов И.В., Козин А. С., Баннов В.Я. Ускорение вычисления контрольной комбинации // Труды международного симпозиума "Надежность и качество". – 2024. – Т. 1. – С. 415-416;

– И.Б. Мещеряков, И.В Сухов, М.Н.Рогоулин, А.С.Козин , А.А. Фролов, В.Д.Гапоненко. Итеративная обработка кадровой последовательности, переданной *OFDM*-символами с канальным свёрточным кодированием и перемещение по спектру и времени//Радиолокация, навигация, связь : сборник трудов XXXI Международной научно-технической конференции (г. Воронеж, 15–17 апреля 2025 г.) : в 6 т. / Воронежский государственный университет ; АО «Концерн “Созвездие”». – Воронеж : Издательский дом ВГУ, 2025.–т.2. – с 313-322;

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Федеральный закон "О безопасности критической информационной инфраструктуры Российской Федерации" от 26.07.2017 N 187-ФЗ (последняя редакция)

2. Khan M. F. F., Sakamura K. A discretionary delegation framework for access control systems // *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*. – Cham : Springer International Publishing, 2016. – С. 865-882.

3. Lin T. Y. T. Y., Vachon P. Secure information flow and file movements: A topological theory of discretionary access controls // *2017 IEEE International Conference on Big Data (Big Data)*. – IEEE, 2017. – С. 1821-1829.

4. Bai Q., Zheng Y. Study on the access control model // *Proceedings of 2011 Cross Strait Quad-Regional Radio Science and Wireless Technology Conference*. – IEEE, 2011. – Т. 1. – С. 830-834.

5. Shan Z., Wang X., Chiueh T. Enforcing mandatory access control in commodity OS to disable malware // *IEEE Transactions on Dependable and Secure Computing*. – 2012. – Т. 9. – №. 4. – С. 541-555.

6. De Capitani di Vimercati S., Samarati P. Mandatory Access Control (MAC) Policies // *Encyclopedia of Cryptography, Security and Privacy*. – Berlin, Heidelberg : Springer Berlin Heidelberg, 2021. – С. 1-2.

7. Ferrari E. Mandatory Access Control // *Access Control in Data Management Systems*. – Cham : Springer International Publishing, 2010. – С. 49-60.

8. Kuhn R., Coyne E., Weil T. Adding attributes to role-based access control. – 2010.

9. de Carvalho Junior M. A., Bandiera-Paiva P. Health Information System Role-Based Access Control Current Security Trends and Challenges // *Journal of healthcare engineering*. – 2018. – Т. 2018. – №. 1. – С. 6510249.

10. Cruz J. P., Kaji Y., Yanai N. *RBAC-SC: Role-based access control using smart contract* // *Ieee Access*. – 2018. – T. 6. – C. 12240-12251.
11. Hu V. C. et al. *Attribute-based access control* // *Computer*. – 2015. – T. 48. – №. 2. – C. 85-88.
12. Servos D., Osborn S. L. *Current research and open problems in attribute-based access control* // *ACM Computing Surveys (CSUR)*. – 2017. – T. 49. – №. 4. – C. 1-45.
13. Ding S. et al. *A novel attribute-based access control scheme using block-chain for IoT* // *Ieee Access*. – 2019. – T. 7. – C. 38431-38441
14. Azevedo de Amorim A., Hrițcu C., Pierce B. C. *The meaning of memory safety* // *Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7*. – Springer International Publishing, 2018. – C. 79-105.
15. Shi W., Lu C., Lee H. H. S. *Memory-centric security architecture* // *High Performance Embedded Architectures and Compilers: First International Conference, HiPEAC 2005, Barcelona, Spain, November 17-18, 2005. Proceedings 1*. – Springer Berlin Heidelberg, 2005. – C. 153-168.
16. Crenne J. et al. *Configurable memory security in embedded systems* // *ACM Transactions on Embedded Computing Systems (TECS)*. – 2013. – T. 12. – №. 3. – C. 1-23.
17. Marco-Gisbert H., Ripoll Ripoll I. *Address space layout randomization next generation* // *Applied Sciences*. – 2019. – T. 9. – №. 14. – C. 2928.
18. Stojanovski N. et al. *Bypassing data execution prevention on Microsoft windows xp sp2* // *The Second International Conference on Availability, Reliability and Security (ARES'07)*. – IEEE, 2016. – C. 1222-1226.

19. Ahmed M. A., Aljumah A., Ahmad M. G. *Design and implementation of a direct memory access controller for embedded applications*//*International Journal of Technology*. – 2019. – T. 10. – №. 2. – C. 309-319.
20. Frassetto T. et al.:*Memory Isolation* //27th USENIX Security Symposium (USENIX Security 18). – 2018. – C. 83-97.
21. Devi M., Majumder A. *Side-channel attack in Internet of Things: A survey* // *Applications of Internet of Things: Proceedings of ICCCIOT 2020*. – Springer Singapore, 2021. – C. 213-222.
22. Nisan N., Schocken S. *The elements of computing systems: building a modern computer from first principles*. – MIT press, 2021.
23. Borrelli C. *IEEE 802.3 cyclic redundancy check* //application note: *Virtex Series and Virtex-II Family, XAPP209* (v1. 0). – 2001.
24. Yamaguchi F., Nishi H. *Hardware-based hash functions for network applications* //2013 19thIEEE International Conference on Networks (ICON). – IEEE, 2013. – C. 1-6.
25. Cheng M. et al. *Efficient Web Archive Searching*. – 2020.
26. Shamis A. et al. *Fast general distributed transactions with opacity* // *Proceedings of the 2019 International Conference on Management of Data*. – 2019. – C. 433-448.
27. Alakuijala J., Cox B., WassenbergJ. *Fast keyed hash/pseudo-random function using SIMD multiply and permute* //arXiv preprint arXiv:1612.06257. – 2016.
28. Dobai R., Korenek J. *Evolution of non-cryptographic hash function pairs for FPGA-based network applications* //2015 IEEE Symposium Series on Computational Intelligence. – IEEE, 2015. – C. 1214-1219.

29. Sharma A. K., Mittal S. K. *Cryptography & network security hash function applications, attacks and advances: A review* //2019 Third International Conference on Inventive Systems and Control (ICISC). – IEEE, 2019. – C. 177-188.
30. Khan B. U. I. et al. *Evolution and Analysis Of Secure Hash Algorithm (Sha) Family* //Malaysian Journal of Computer Science. – 2022. – T. 35. – №. 3. – C. 179-200.
31. Dolmeta A., Martina M., Masera G. *Comparative study of keccak sha-3 implementations* //Cryptography. – 2023. – T. 7. – №. 4. – C. 60.
32. Purohit R., Mishra U., Bansal A. *A survey on recent cryptographic hash function designs* //International Journal of Emerging Trends & Technology in Computer Science. – 2013.
33. Russell S. *Chorba: A novel CRC32 implementation* //arXiv preprint arXiv:2412.16398. – 2024.
34. Tan X. et al. *Is the Canary Dead? On the Effectiveness of Stack Canaries on Microcontroller Systems* //Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing. – 2024. – C. 1350-1357.
35. Serebryany K. et al. *Memory Tagging and how it improves C/C++ memory safety* //arXiv preprint arXiv:1802.09517. – 2018.
36. Maass M. et al. *A systematic analysis of the science of sandboxing* //PeerJ Computer Science. – 2016. – T. 2. – C. e43.
37. Seth A. et al. *Comparing effectiveness and efficiency of interactive application security testing (IAST) and runtime application self-protection (RASP) tools in a large java-based system* //Empirical Software Engineering. – 2025. – T. 30. – №. 3. – C. 67.
38. Vermeulen S. *SELinux System Administration: Implement mandatory access control to secure applications, users, and information flows on Linux*. – Packt Publishing Ltd, 2020.

39. Gruhn M., Freiling F. C. *Evaluating atomicity, and integrity of correct-memory acquisition methods* // *Digital Investigation*. – 2016. – Т. 16. – С. S1-S10.
40. Ким А. К. Микропроцессорные вычислительные комплексы с архитектурой "Эльбрус" и их программное обеспечение // *Вопросы радиоэлектроники*. – 2009. – Т. 4. – №. 3. – С. 5-37.
41. Ишин П. А., Логинов В. Е., Васильев П. П. Ускорение вычислений с использованием высокопроизводительных математических и мультимедийных библиотек для архитектуры Эльбрус // *Включён в Перечень ВАК*. – 2015.
42. Chi M., Liu J. *VACA: A high-performance variable-length adaptive CRC algorithm* // *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. – IEEE, 2017. – С. 1-6.
43. Sarwate D. V. *Computation of cyclic redundancy checks via table look-up* // *Communications of the ACM*. – 1988. – Т. 31. – №. 8. – С. 1008-1013.
44. Liu W., Chang C. H., Zhang F. *Stealthy and robust glitch injection attack on deep learning accelerator for target with variational viewpoint* // *IEEE Transactions on Information Forensics and Security*. – 2020. – Т. 16. – С. 1928-1942.
45. Seshadri N., Sundberg C. E. W. *List Viterbi decoding algorithms with applications* // *IEEE transactions on communications*. – 1994. – Т. 42. – №. 234. – С. 313-323.

ПРИЛОЖЕНИЕ А

(обязательное)

Задание на выпускную квалификационную работу

«Утверждаю»
Заведующий кафедрой
Пронин И.А
(фамилия, имя, отчество)
«22 » мая 2025 г.

З А Д А Н И Е

на выпускную квалификационную работу магистра

1. Студент гр. 23 ПДм1 факультета ФИТЭ
Направление подготовки 11.03.04 «Электроника и нанoeлектроника»
Магистерская программа
Сухов Иван Викторович
(фамилия, инициалы)

2. Руководитель Пронин Игорь Александрович
(фамилия, инициалы)

3. Тема выпускной работы Мандатная защита на основе контроля состояния памяти с помощью хеш-функции

Тема утверждена приказом ПГУ № 588/о от «28 »04 2025 г.

4. Техническое задание на выпускную работу (разработать, исследовать (назначение объекта разработки, исследования), исходные данные, граничные условия разрабатываемого или исследуемого объекта, условия эксплуатации и т.д.)

1. Выполнить аналитический обзор научно-технической литературы.

2. Разработать методику контроля состояния памяти с помощью хеш-функции CRC.

3. Разработать алгоритм расчета хеш-функции CRC для многоядерного процессора.

4. Разработать консольное приложение для реализации различных методов расчета хеш-функции CRC.

5. Выполнить верификацию разработанного приложения на различной аппаратной платформе (минимум 3-х ПЭВМ) на рандомизированном наборе данных файлового типа.

6. Определить критерий эффективности и провести сравнительный анализ методов расчета хеш-функции CRC по этому критерию.

7. Выработать предложение по мандатной защите на основе контроля состояния памяти с помощью хеш-функции CRC.

5. Объем и содержание основной части выпускной работы

5.1. Пояснительная записка (перечень вопросов, подлежащих разработке, обоснований, описаний, расчетов и т.д.)

1 Аналитический обзор научно-технической литературы

1.1 Системы разграничения доступа

1.2 Роль памяти в обеспечении безопасности

1.3 Хеш-функции в задачах информационной безопасности

1.4 Обзор существующих подходов к контролю состояния памяти

1.5 Средства проверки целостности памяти

1.6 Мандатная защита реализованная на архитектурном уровне

2 Разработка методики контроля состояния памяти с помощью хеш-функции CRC

2.1 Разработка алгоритмов расчета хеш-функции CRC

2.2 Разработка консольного приложения для реализации различных методов расчета хеш-функции CRC

2.3 Выработка предложения по мандатной защите на основе контроля состояния памяти с помощью хеш- функции CRC

3 Техника безопасности и охрана труда

3.1 Идентификация потенциальных опасностей и вредных факторов. Микроклимат рабочей зоны

3.2 Освещение и шумовая нагрузка. Влияние электромагнитных и электростатических полей

3.3 Электробезопасность. Пожарная безопасность

5.2. Иллюстративная часть (перечень и название слайдов презентации)

Титульный лист.Слайд 1.

Цель работы. Слайд 2.

Блок-схема консольного приложения для реализации различных методов расчета хеш-функции CRC. Слайд 3.

Результаты времени расчета хеш-функции CRC различными методами. Слайд 4.

Высоконагруженный вычислитель с общими ресурсами. Слайд 5.

Система защиты от *glitch*-атак. Слайд 6.

Списочный декодер Витерби. Слайд 7.

Расчет альтернативных путей с применением параллельного вычисления CRC. Слайд 8

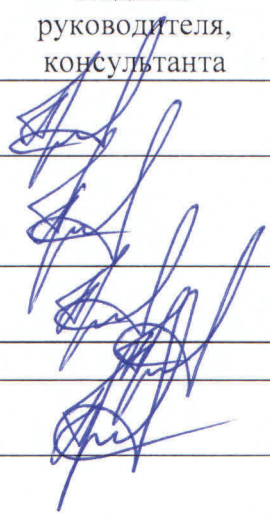
Список опубликованных статей. Слайд 9.

Спасибо за внимание! Слайд 10.

6. Консультанты и содержание дополнительных разделов

6.1.

7. Календарный план по выполнению выпускной работы магистра

Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя, консультанта
Аналитический обзор научно-технической литературы	30%	24.05.2025	
Разработка методики контроля состояния памяти с помощью хеш-функции CRC	45%	28.05.2025	
Техника безопасности и охрана труда офисного сотрудника	15%	31.05.2025	
Иллюстративная часть	5%	01.06.2025	
Оформление пояснительной записки	5%	03.06.2025	

Дата выдачи задания « 22 » мая 2025 г.

Руководитель выпускной работы _____ Пронин И.А.
подпись фамилия, инициалы

Задание к исполнению принял _____ Сухов И.В.
подпись фамилия, инициалы

« 22 » мая 2025 г.

Выпускную квалификационную работу к защите допустить

Декан факультета _____ Кревчик В.Д.
подпись фамилия, инициалы

« 06 » 06 2025 г.

ПРИЛОЖЕНИЕ Б

(обязательное)

Код консольного приложения для реализации различных методов
расчета хеш-функции *CRC* на языке *C*

```

#include<stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <locale.h>
#include <windows.h>
#include <pthread.h>
#include <pthread_time.h>
#include <time.h>
#include "crc.h"
#include "filesList.h"
#include "pthread_queue.h"

#ifndef CRC_TEST_MODE
#define TABLE_BUFFER_SIZE      256
#else
#define TABLE_BYTE_TABLE_SIZE  256
#define TABLE_CRC_DATA_SIZE    4
#define TABLE_CPU_AMOUNT       4
#define TABLE_BUFFER_SIZE      (TABLE_BYTE_TABLE_SIZE * TA-
TABLE_CRC_DATA_SIZE * TABLE_CPU_AMOUNT)

#define MAX_CPU_AMOUNT          64

#endif

uint16_t crc16_array[TABLE_BUFFER_SIZE];

uint16_t crc16;

crc_t crc_set = {
    .flag = 0,
    .poly = 0x94ED,
    .init = 0x0000,
    .xor = 0x0000,
    .table = crc16_array
};

typedef struct {
    uint8_t* offset;
    uint32_t size;
    uint32_t zeros;
} task_data_t;

uint8_t tcpuAmount;

```

```

queue_t*    writeToProcess;
pthread_t    process[MAX_CPU_AMOUNT];
task_data_t params [MAX_CPU_AMOUNT];
qitem_t crc_out[MAX_CPU_AMOUNT];

uint8_t      test_source[16] = {0};

void processTask(void) {
    qitem_t index;
    crc_t crc_task = crc_set;
    while (queue_pop(writeToProcess, &index)) {
        crc_task.init = 0;
        uint16_t crc = crc_make (&crc_task, params[index].offset, params[index].size,
CRC_TEST_BIT_32);
        crc_out[index] = crc_shiftByZeros(crc, params[index].zeros);
        pthread_exit(0);
    }
}

uint16_t crc_parallel_make(uint8_t* src, uint32_t size) {
    queue_t    queue2processTasks;
    queue_t    queue2showResults;

    writeToProcess =&queue2processTasks;
    queue_init(&queue2processTasks, cpuAmount);
    queue_init(&queue2showResults, cpuAmount);
    uint32_t chunk = size / cpuAmount;
    chunk&= 0xFFFFFFFF;

    uint32_t chunk_offset = 0;
    uint32_t chunk_size = size;
    for (inti = 0; i<cpuAmount - 1; i++) {
        chunk_size -= chunk;
        params[i].offset = &src[chunk_offset];
        params[i].zeros = chunk_size;
        params[i].size = chunk;
        chunk_offset += chunk;
    }
    params[cpuAmount-1].offset = &src[chunk_offset];
    params[cpuAmount-1].zeros = 0;
    params[cpuAmount-1].size = chunk_size;
    for (inti = 0; i<cpuAmount; i++) pthread_create(&process[i], NULL,
(void*)(*)(void*))processTask, NULL);

```

```

for (inti = 0; i<cpuAmount; i++) queue_push (&queue2processTasks,
(qitem_t*)&i);
for (inti = 0; i<cpuAmount; i++) pthread_join (process[i], NULL);
queue_drop(&queue2processTasks);

```

```

for (inti = 0; i<cpuAmount; i++) crc_set.init ^= crc_out[i];
return crc_set.init;
}

```

```

#ifdef CRC_TEST_MODE
struct timespec timeStart;
struct timespec timeStop;
void calculate_crc(uint16_t cnt, CRC_TEST_t crc_test_type) {
#else
void calculate_crc(uint16_t cnt) {
#endif
uint32_t size = 0;
uint8_t* ptr = NULL;
for (inti = 0; i<cnt; i++) {
ptr = getArrayByIndex (i, &size);
if (size) {
#ifdef CRC_TEST_MODE
crc16 = crc_make (&crc_set, ptr, size);
#else
clock_gettime(0, &timeStart);
if(CRC_TEST_MULTI != crc_test_type) {
crc16 = crc_make (&crc_set, ptr, size, crc_test_type);
} else {
crc16 = crc_parallel_make(ptr, size);
}
clock_gettime(0, &timeStop);
#endif
crc_set.init = crc16;
}
}
}
}

```

```

#ifdef CRC_TEST_MODE
void calculate_crc_time(uint16_t cnt, CRC_TEST_t crc_test_type) {
crc_set.init = 0x0000;

```

```

calculate_crc(cnt, crc_test_type);

```

```

int32_t second = (timeStop.tv_sec - timeStart.tv_sec) * 1000;

```

```

longnano  = (timeStop.tv_nsec - timeStart.tv_nsec) / 1000000;
nano  += second;

printf("CRC16 - 0x%04X, Time - %d mc\n", crc16, (int)nano);
}
#endif

int main(intargc, char *argv[]) {
setlocale(0,"Russian");
if (argc> 2) {
fprintf(stderr, "Usage - calculate cyclic redundancy check (CRC16) from specified
path: test4.exe <paths> \n");
return 1;
}
char* path = ".";
if (2 == argc) path = argv[1];
if (readFilesFromDir(path)) {
fprintf(stdout, "Can not open directory %s\n", path);
return 1;
}
printFilesNames ("Files in dir:\n");

crc_make_table (&crc_set);
uint16_tcnt = getFilesAmount ();
#ifdef CRC_TEST_MODE
calulcate_crc(cnt);
printf("CRC16 0x%04X\n", crc16);
#else
SYSTEM_INFO sysinfo;
GetSystemInfo(&sysinfo);
cpuAmount = sysinfo.dwNumberOfProcessors;
// cpuAmount = 2;

calulcate_crc_time(cnt, CRC_TEST_BIT_08);
calulcate_crc_time(cnt, CRC_TEST_BIT_16);
calulcate_crc_time(cnt, CRC_TEST_BIT_32);
calulcate_crc_time(cnt, CRC_TEST_MULTI);

#endif

return 0;

```

ПРИЛОЖЕНИЕ В

(обязательное)

Список опубликованных статей

1. Головяшкин А.Н., Козин А. С. , Сухов И.В. Алгоритмы вычисления циклического избыточного кода // Труды международного симпозиума "Надежность и качество". – 2024. – Т. 1. – С. 421-422;

2. Головяшкин А.Н., Сухов И.В., Козин А. С., Баннов В.Я. Ускорение вычисления контрольной комбинации // Труды международного симпозиума "Надежность и качество". – 2024. – Т. 1. – С. 415-416.

3. И.Б. Мещеряков, И.В Сухов, М.Н.Рогоулин, А.С.Козин , А.А. Фролов, В.Д.Гапоненко. Итеративная обработка кадровой последовательности, переданной *OFDM*-символами с канальным свёрточным кодированием и перемещение по спектру и времени // Радиолокация, навигация, связь : сборник трудов XXXI Международной научно-технической конференции (г. Воронеж, 15–17 апреля 2025 г.) : в 6 т. / Воронежский государственный университет ; АО «Концерн “Созвездие”». – Воронеж : Издательский дом ВГУ, 2025.–т.2. – с 313-322;

ПРИЛОЖЕНИЕ Г

(обязательное)

Заявление и протокол проверки ВКР на оригинальность в системе

«Антиплагиат.ВУЗ»

Ректору Пензенского государственного
университета Гулякову А.Д.
студента 2 курса ФИТЭ очной формы обуче-
ния
Сухова Ивана Викторовича

ЗАЯВЛЕНИЕ

Я, Сухов Иван Викторович, студента 2 курса очной формы обучения направления подготовки 11.04.04 Электроника и наноэлектроника, Нано- и микроэлектроника, прошу провести проверку с использованием системы «Антиплагиат.ВУЗ» магистерской выпускной квалификационной работы на тему «Мандатная защита на основе контроля состояния памяти с помощью хеш-функции», выполненной мной самостоятельно, на содержание элементов плагиата.

Все прямые заимствования из печатных и электронных источников, а также из защищенных ранее выпускных квалификационных работ, кандидатских и докторских диссертаций имеют соответствующие ссылки.

Я ознакомлен с действующим Стандартом университета «Выпускная квалификационная работа обучающихся по образовательным программам высшего образования - программам бакалавриата, программам специалитета и программам магистратуры», согласно которому обнаружение плагиата является основанием для недопуска ВКР к защите и отчисления из университета.

ВКР представлена на проверку «04.06.2025»

Руководитель ВКР Пронин И.А.

Протокол

проверки на оригинальность в системе «Антиплагиат.ВУЗ» выпускной бакалаврской работы на тему «Мандатная защита на основе контроля состояния памяти с помощью хеш-функции»

студента 2 курса Сухова И.В.

направления подготовки 11.04.04 Электроника и нанoeлектроника, профиль «нано- и микроэлектроника»

Руководитель ВКР Пронин И.А.



СПРАВКА

о результатах проверки текстового документа на наличие заимствований

Федеральное государственное бюджетное образовательное учреждение высшего образования «Пензенский государственный университет»

ПРОВЕРКА ВЫПОЛНЕНА В СИСТЕМЕ АНТИПЛАГИАТ.ВУЗ

Автор работы: Сухов Иван Викторович
Самодитирование: Сухов Иван Викторович
Название работы: Мандатная защита на основе контроля состояния памяти с помощью хеш-функции
Тип работы: Магистерская диссертация
Подразделение: Кафедра «Нано- и микроэлектроника»

РЕЗУЛЬТАТЫ

■ ОТЧЕТ О ПРОВЕРКЕ КОРРЕКТИРОВАЛСЯ: НИЖЕ ПРЕДСТАВЛЕНЫ РЕЗУЛЬТАТЫ ПРОВЕРКИ ДО КОРРЕКТИРОВКИ

СОВПАДЕНИЯ	7.6%	СОВПАДЕНИЯ	5.31%
ОРИГИНАЛЬНОСТЬ	92.4%	ОРИГИНАЛЬНОСТЬ	92.4%
ЦИТИРОВАНИЯ	0%	ЦИТИРОВАНИЯ	0%
САМОЦИТИРОВАНИЯ	0%	САМОЦИТИРОВАНИЯ	2.29%

ДАТА ПОСЛЕДНЕЙ ПРОВЕРКИ: 04.06.2025

ДАТА И ВРЕМЯ КОРРЕКТИРОВКИ: 04.06.2025 15:18

Структура документа: Проверенные разделы: основная часть с 2, 11-77, титульный лист с 1, содержание с 3, введение с 4-10, выводы с 77-79
Модули поиска: Цитирование; Шаблонные фразы; ИПС Адилет; Диссертации НББ; Переводные заимствования; Медицина; Коллекция НБУ; СМИ России и СНГ; Патенты СССР, РФ, СНГ; IEEE; Кольцо вузов; Публикации eLIBRARY; СПС ГАРАНТ: аналитика; Сводная коллекция ЗБС; Публикации РГБ (переводы и перефразирования); СПС ГАРАНТ: нормативно-правовая документация; Публикации РГБ; Публикации eLIBRARY (переводы и перефразирования); Перефразирования по коллекции IEEE; Переводные заимствования IEEE; Перефразированные заимствования по коллекции Интернет в английском сегменте; Перефразированные заимствования по коллекции Интернет в русском сегменте; Переводные заимствования по коллекции Интернет в английском сегменте; Переводные заимствования по коллекции Интернет в русском сегменте; Кольцо вузов

Работу проверил: Головяшкин Алексей Николаевич

ФИО проверяющего

Дата подписи: 04.06.2025



Чтобы убедиться в подлинности справки, используйте QR-код, который содержит ссылку на отчет

Ответ на вопрос: является ли обнаруженное заимствование корректным, система оставляет на усмотрение проверяющего. Предоставленная информация не подлежит использованию в коммерческих целях.

Руководитель ВКР

«04» 06 2025 год

подпись

Пронин И.А.

фамилия, инициалы



АНТИПЛАГИАТ
ОБНАРУЖЕНИЕ ЗАИМСТВОВАНИЙ

СПРАВКА

о результатах проверки текстового документа
на наличие заимствований

Федеральное государственное бюджетное
образовательное учреждение высшего
образования "Пензенский государственный
университет"

ПРОВЕРКА ВЫПОЛНЕНА В СИСТЕМЕ АНТИПЛАГИАТ.ВУЗ

Автор работы: Сухов Иван Викторович
Самоцитирование
рассчитано для: Сухов Иван Викторович
Название работы: Мандатная защита на основе контроля состояния памяти с помощью хеш-функции
Тип работы: Магистерская диссертация
Подразделение: Кафедра "Нано- и микроэлектроника"

РЕЗУЛЬТАТЫ

■ ОТЧЕТ О ПРОВЕРКЕ КОРРЕКТИРОВАЛСЯ: НИЖЕ ПРЕДСТАВЛЕНЫ РЕЗУЛЬТАТЫ ПРОВЕРКИ ДО КОРРЕКТИРОВКИ

СОВПАДЕНИЯ	7.6%	СОВПАДЕНИЯ	5.31%
ОРИГИНАЛЬНОСТЬ	92.4%	ОРИГИНАЛЬНОСТЬ	92.4%
ЦИТИРОВАНИЯ	0%	ЦИТИРОВАНИЯ	0%
САМОЦИТИРОВАНИЯ	0%	САМОЦИТИРОВАНИЯ	2.29%

ДАТА ПОСЛЕДНЕЙ ПРОВЕРКИ: 04.06.2025

ДАТА И ВРЕМЯ КОРРЕКТИРОВКИ: 04.06.2025 15:18

Структура документа: Проверенные разделы: основная часть с.2, 11-77, титульный лист с.1, содержание с.3, введение с.4-10, выводы с.77-79
Модули поиска: Цитирование; Шаблонные фразы; ИПС Адилет; Диссертации НББ; Переводные заимствования; Медицина; Коллекция НБУ; СМИ России и СНГ; Патенты СССР, РФ, СНГ; IEEE; Кольцо вузов; Публикации eLIBRARY; СПС ГАРАНТ: аналитика; Сводная коллекция ЭБС; Публикации РГБ (переводы и перефразирования); СПС ГАРАНТ: нормативно-правовая документация; Публикации РГБ; Публикации eLIBRARY (переводы и перефразирования); Перефразирования по коллекции IEEE; Переводные заимствования IEEE; Перефразированные заимствования по коллекции Интернет в английском сегменте; Перефразированные заимствования по коллекции Интернет в русском сегменте; Переводные заимствования по коллекции Интернет в английском сегменте; Переводные заимствования по коллекции Интернет в русском сегменте; Кольцо вузов...

Работу проверил: Головяшкин Алексей Николаевич

ФИО проверяющего

Дата подписи: 04.06.2025

Подпись проверяющего



Чтобы убедиться
в подлинности справки, используйте QR-код,
который содержит ссылку на отчет.

Ответ на вопрос, является ли обнаруженное заимствование
корректным, система оставляет на усмотрение проверяющего.
Предоставленная информация не подлежит использованию
в коммерческих целях.

ПРИЛОЖЕНИЕ Д

(обязательное)

Презентация ВКР



Федеральное государственное бюджетное образовательное учреждение высшего образования

«Пензенский государственный университет»

Факультет информационных технологий и электроники

Кафедра: «Нано- и микроэлектроника»

Мандатная защита на основе контроля состояния памяти с помощью хеш-функции

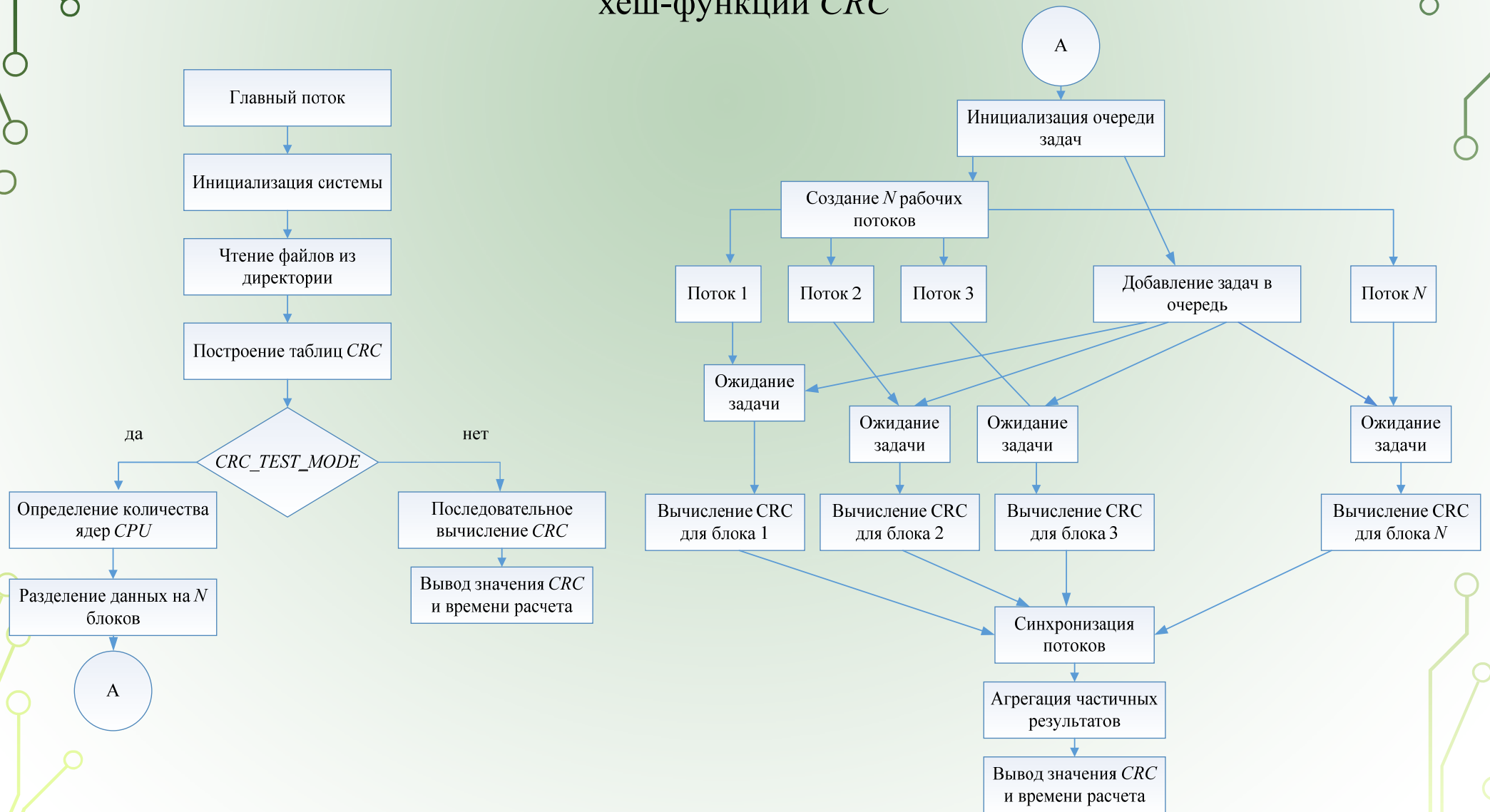
Выполнил: Сухов Иван Викторович

Научный руководитель: доктор технических наук, доцент,
заведующий кафедрой нано- и микроэлектроники Пензенского
государственного университета Пронин Игорь Александрович

Цель: Исследовать вопрос реализации мандатной защиты на основе контроля состояния памяти с помощью хеш-функции



Блок-схема консольного приложения для реализации различных методов расчета хеш-функции CRC



Результаты времени расчета хеш-функции CRC используя параллельный метод на разных платформах

Таблица 1- Характеристики платформ

№ ПЭВМ	Процессор с тактовой частотой	Объем ОЗУ	Объем ПЗУ	Кол-во ядер/ потоков
№1	<i>Intel Pentium N3530</i> с 2.2ГГц	4 Гб	<i>SSD 128 Гб</i>	4/4
№2	<i>Intel Core i7-3770K</i> с 3.5ГГц	16 Гб	<i>SSD 512 Гб</i>	4/8
№3	<i>AMD Ryzen 7 7700X</i> с 4.5 ГГц	32 Гб	<i>SSD 1 Тб</i>	8/16

Таблица 2- Результаты расчета

Размер файла	ПВЭМ №1 Параллельный метод , мс	ПВЭМ №2 Параллельный метод , мс	ПВЭМ №3 Параллельный метод , мс
0,1 Гб	124	37	8
0,2 Гб	235	80	15
0,3 Гб	374	108	23
0,4 Гб	625	183	32
0,5 Гб	750	173	38
1,1 Гб	1391	420	70
1,5 Гб	2407	534	98

Высоконагруженный вычислитель с общими ресурсами

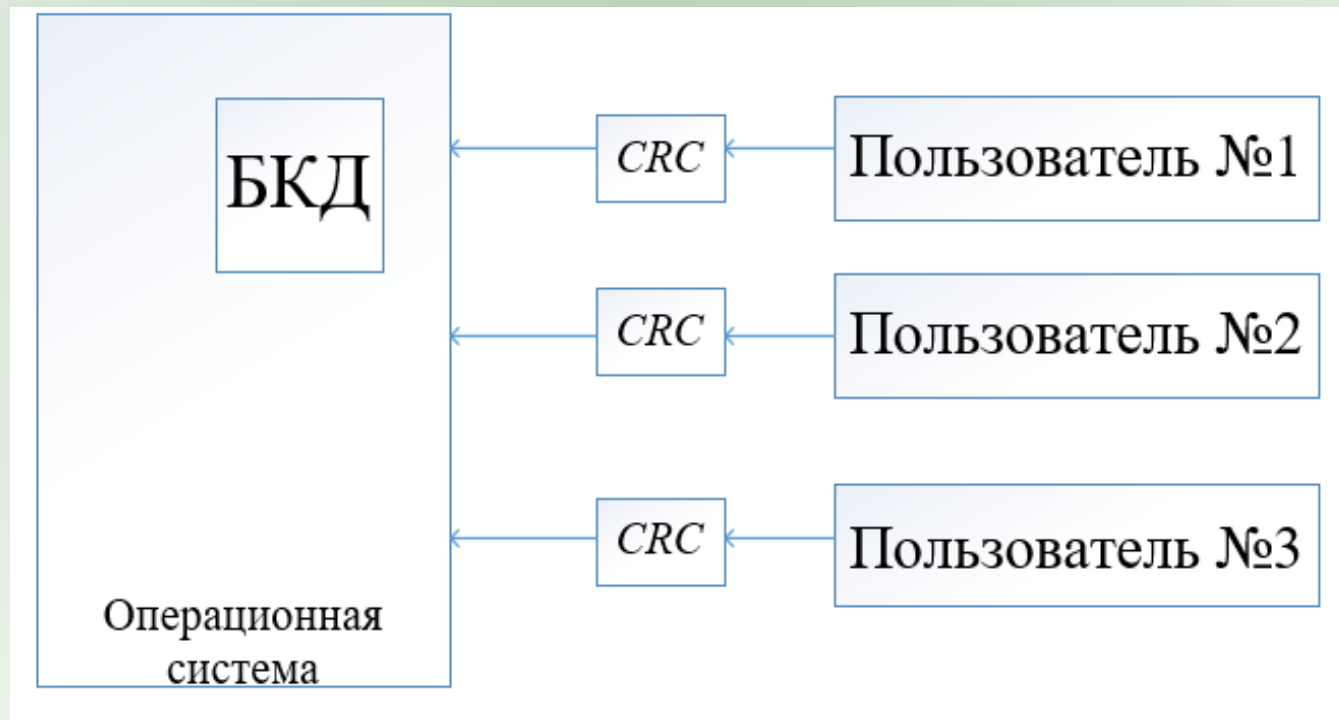


Рисунок 1 – Схема реализации высоконагруженного вычислителя с общими ресурсами

Система защиты от *glitch*-атак

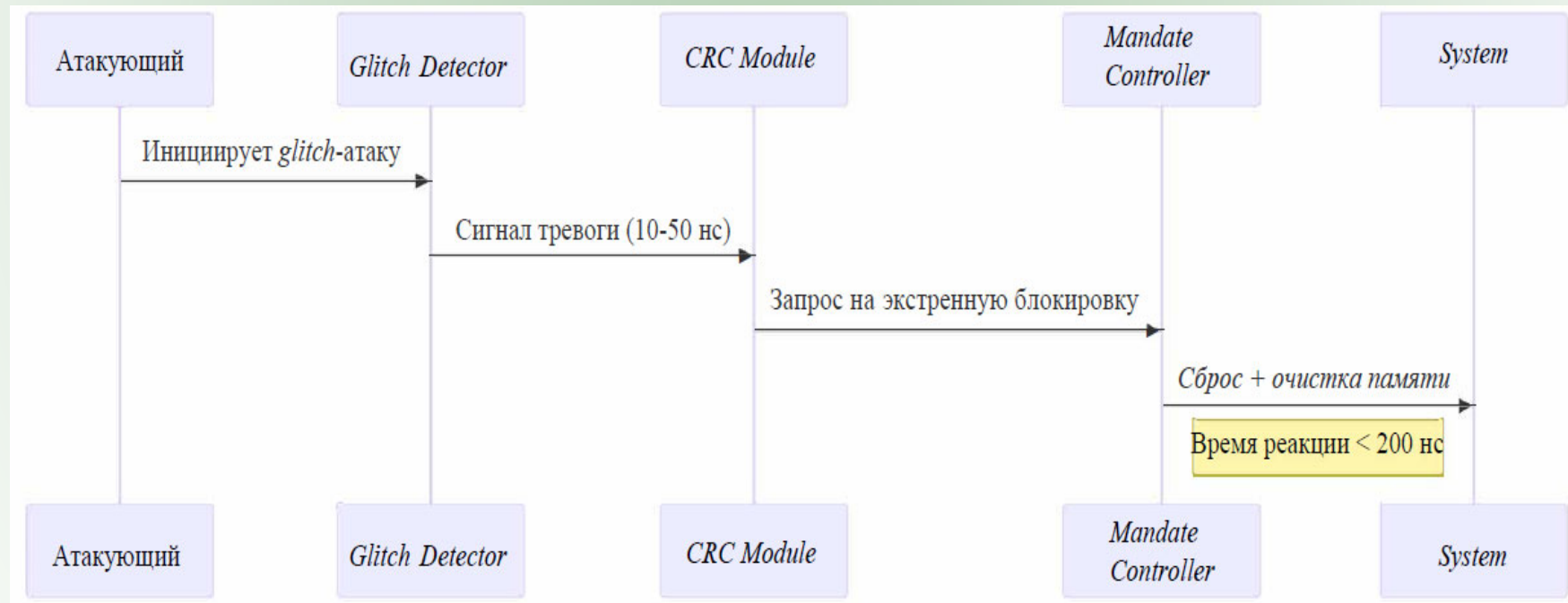


Рисунок 2 – Система мониторинга в реальном времени

Списочный декодер Витерби

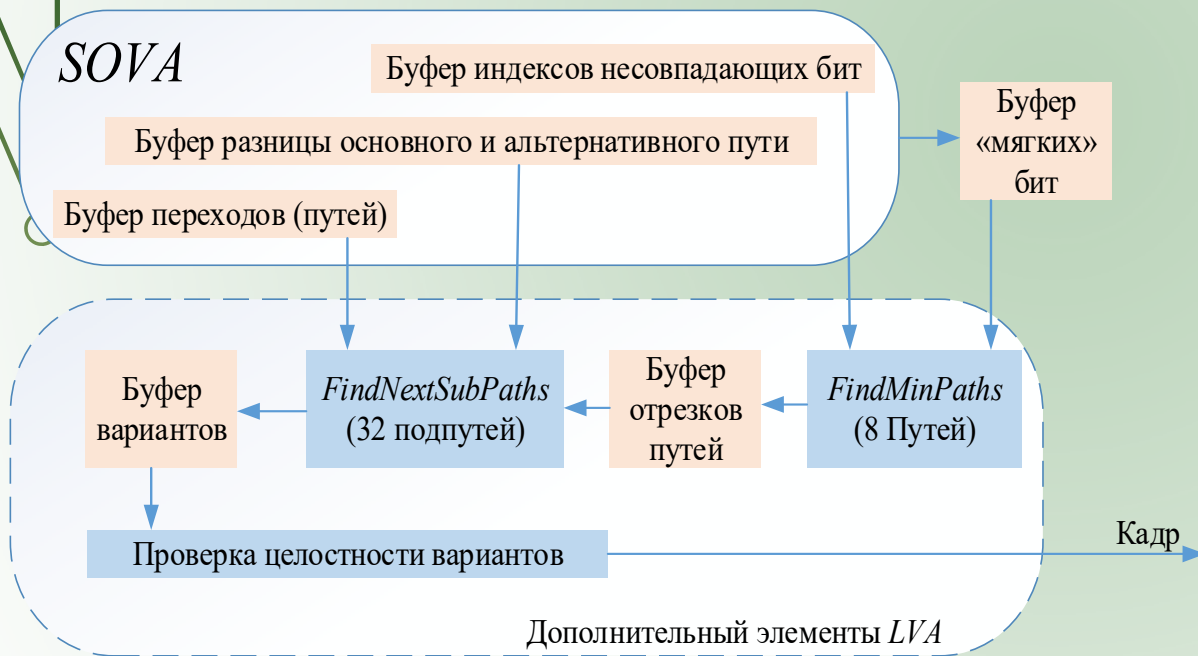


Рисунок 3 – Структура LVA

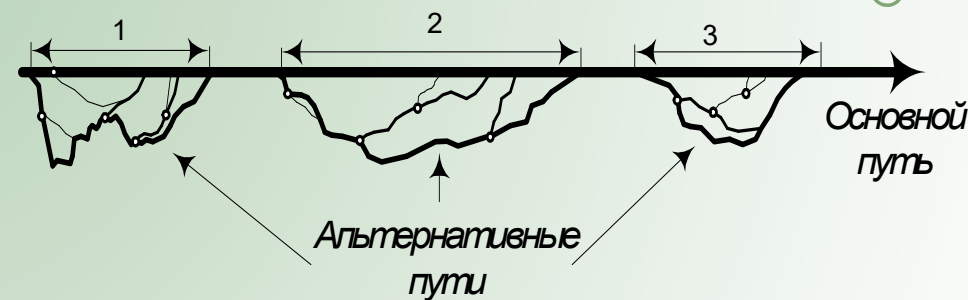


Рисунок 4 – Пример работы алгоритма LVA

Расчет альтернативных путей с применением параллельного вычисления *CRC*

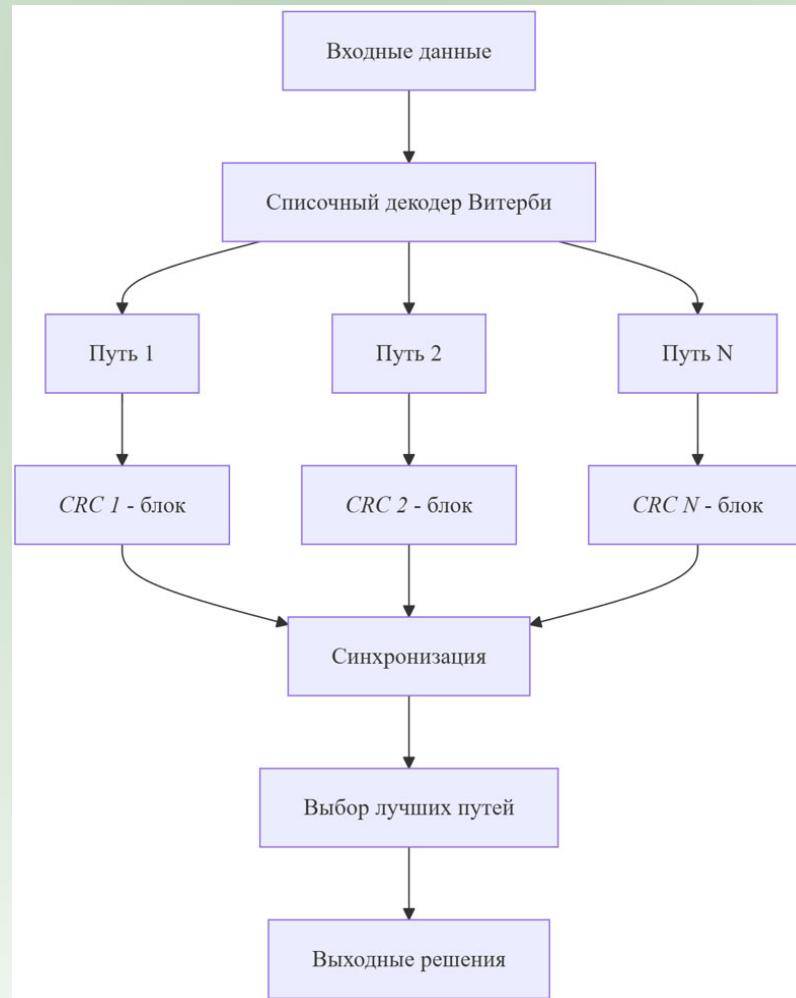


Рисунок 5 – Схема расчета альтернативных путей с применением параллельного вычисления *CRC*

Список опубликованных статей

1. Головяшкин А.Н., Козин А. С. , Сухов И.В. Алгоритмы вычисления циклического избыточного кода // Труды международного симпозиума "Надежность и качество". – 2024. – Т. 1. – С. 421-422;

2. Головяшкин А.Н., Сухов И.В., Козин А. С., Баннов В.Я. Ускорение вычисления контрольной комбинации // Труды международного симпозиума "Надежность и качество". – 2024. – Т. 1. – С. 415-416.

3. И.Б. Мещеряков, И.В Сухов, М.Н. Рогулин, А.С. Козин , А.А. Фролов, В.Д. Гапоненко. Итеративная обработка кадровой последовательности, переданной *OFDM*-символами с канальным свёрточным кодированием и перемежение по спектру и времени // Радиолокация, навигация, связь : сборник трудов XXXI Международной научно-технической конференции (г. Воронеж, 15–17 апреля 2025 г.) : в 6 т. / Воронежский государственный университет ; АО «Концерн “Созвездие”». – Воронеж : Издательский дом ВГУ, 2025.– т.2. – с 313-322;

The image features a light green background with a subtle gradient. In the four corners, there are decorative elements resembling circuit board traces or stylized trees, drawn in a dark green color. These elements consist of thin lines with small circles at the ends, creating a geometric, branching pattern.

СПАСИБО ЗА ВНИМАНИЕ!