Правительство Российской Федерации
Санкт-Петербургский государственный университет

Кафедра системного программирования

Бабанов Пётр Андреевич

Синхронизация в многопоточных MAK-обфусцированных программах

Выпускная квалификационная работа

Научный руководитель
доцент к. т. н. Т. А. Брыксин
Научный консультант:
М. В. Баклановский
Рецензент:
ст. преподаватель А. Е. Сибиряков

Санкт-Петербург
2020

SAINT PETERSBURG STATE UNIVERSITY

Software Engineering

Babanov Petr

Synchronization in multithreaded MAC-obfuscated programs

Graduation Thesis

Scientific supervisor:
Associate Professor,
Candidate of Engineering T. Bryksin
Scientific Advisor:
M. Baklanovsky
Reviewer:
A. Sibiryakov

Saint Petersburg
2020

Содержание

1 Введение

В последние десятилетия вместе с ростом применения компьютеров во всех сферах жизни все острее встает проблема защиты компьютерных программ. Под защитой программы обычно понимается ее устройство таким образом, чтобы сделать невозможным несанкционированное вмешательство в ее работу, а также получение информации о методах ее работы, алгоритмах, используемых в ней. Попытки такого рода воздействий называют атаками. Цели атаки могут быть различными: от простого любопытства, обхода блокировок, связанных с лицензионными ограничениями и до коммерческого шпионажа и поиска уязвимостей и их эксплуатации, в том числе в противозаконных целях.

Одним из наиболее распространенных методов атаки на программное обеспечение является метод обратной разработки (реверс-инжиниринг). Его суть заключается в анализе программы, ее машинного кода, взаимодействия с внешними объектами и построении выводов на основе полученных данных о применяемых алгоритмах, возможностях ее модификации и т.д. в зависимости от цели атаки.

Полностью предотвратить такую атаку невозможно, поскольку код программы, передаваемый потребителю, является описанием алгоритма работы программы на машинном языке. Однако можно усложнить проведение реверс-инжиниринга. Для этого используется обфускация - умышленное запутывание программы для усложнения ее анализа сторонним пользователем. Поскольку, как указывалось ранее, пресечь атаку путем реверс-инжиниринга программы невозможно, обфускация считается надежной, если для анализа программы требуется больше времени, чем проходит до выхода новой версии программы.

Обфускация компилируемых программ обычно происходит на языке высокого уровня. Такая обфускация, как правило, малоэффективна, так как не противодействует главному инструменту, применяемому при реверс-инжиниринге - дизассемблеру, восстанавливающему код программы на языке ассемблера из бинарного исполняемого файла. При этом применение обфускации на низком уровне крайне затруднительно из-за сложной организации низкоуровневого программного кода. Подробный обзор методов обфускации приведен в ста-

тье [2].

Проблему сложности применения низкоуровневой обфускации решает компилятор МАК, разрабатываемый на Кафедре системного программирования СПбГУ. В его основе лежит идея разбиения программы на линейные участки - участки программы, команды внутри которых выполняются строго последовательно. Это значит, что из середины линейного участка невозможен переход никуда кроме как к следующей команде, находящейся в этом участке; также невозможны переходы к середине такого участка, только к началу. Такая организация позволяет применять различные способы обфускации на низком уровне. В частности, это делает достаточно простым применение одного из наиболее эффективных методов обфускации - кодирования исполняемого кода программы. Он заключается в том, что фрагменты программы, которые необходимо защитить, находятся в закодированном виде. Во время работы программы перед исполнением они раскодируются, исполняются, а потом снова закодируются. Важно учитывать также то, что современные методы кодирования чаще всего работают в несколько этапов (раундов) и в случае применения к программному коду эти раунды в целях маскировки обычно разделяют участками исходной программы.

Говоря об обфускации, важно также отметить, что большинство промышленных программ сегодня являются многопоточными. Это связано с тем, что масштабы решаемых задач неуклонно растут. Еще в период 1965-1975 годов один из основателей компании Intel Гордон Мур сделал наблюдения и вывел эмпирический закон: число транзисторов в процессоре будет удваиваться каждые 24 месяца. Количество транзисторов напрямую влияет на производительность, однако в последнее десятилетие сохранить такую скорость становится все сложнее, поскольку размеры транзисторов приближаются к физическому пределу, а вместе с тем все острее встает вопрос охлаждения таких систем. Решением проблемы, связанной с необходимостью увеличения производительности на фоне сокращения скорости прироста производительности процессоров является создание многопоточных вычислительных систем. Теоретически рост производительности таких систем не ограничен, и все возникающие ограничения связаны с возможностью параллельного вычисления конкретных задач.

Однако возникает проблема, связанная с разделяемыми ресурсами - ресурсами, используемыми при работе программы, которые являются общими для нескольких потоков. Обычно такими ресурсами являются общие переменные, структуры данных, потоки данных. Задача синхронизации доступа к разделяемым ресурсам такого вида известна давно и существует большое количество методов ее решения. Синхронизация доступа к ним является частью логики работы программы и решается программистом для каждой конкретной задачи.

В случае применения обфускации к многопоточным программам возникает дополнительная проблема синхронизации: код программы в памяти находится в единственном экземпляре, общем для всех потоков. Это значит, что с точки зрения обфускации код программы становится разделяемым ресурсом. Однако логика работы программы не включает обфускацию и решение задачи синхронизации потоков при работе с кодом программы должно реализовываться в компиляторе.

Частично эта задача была решена ранее: в работе [10], для компилятора MAK были реализованы и апробированы различные механизмы синхронизации, которые позволяют синхронизировать доступ к коду программы как к разделяемому ресурсу. Однако встраивание их в код исходной программы производилось вручную.

При решении задач, связанных с автоматизированной модификацией программ, обычно работают с графом потока управления – множеством всех возможных путей исполнения программы, представленном в виде графа. При этом линейные участки будут вершинами графа, а команды перехода – дугами.

2 Цель работы

Целью данной работы является разработка программного модуля, реализующего автоматизированное встраивание механизмов синхронизации в MAK-компилированные программы с целью дальнейшей обфускации.

Для достижения данной цели были выделены следующие задачи.

- Сделать обзор существующих подходов к решению задач синхронизации разделяемых ресурсов.
- Разработать архитектуру требуемого модуля.
- Реализовать прототип требуемого модуля.
- Провести тестирование прототипа и оценить влияние синхронизации на работу программ.

3 Обзор методов синхронизации

Сегодня чаще всего применяется следующая классификация механизмов синхронизации без блокировок в зависимости от условий, выполняемых алгоритмами:

1. Obstruction-free
2. Lock-free
3. Wait-free

В данный момент существует большое количество различных шаблонов реализации многопоточных структур данных, часть из них подробно рассмотрена в [1]. В этой статье автор рассматривает основные шаблоны, используемые в каждом из видов, и приводит примеры реализации таких шаблонов.

Механизмы первого вида гарантируют отсутствие конфликтующих операций. Это достигается путем прерывания операции, во время выполнения которой возник конфликт, и попыткой повторить ее позднее. Такие алгоритмы просты в реализации, поскольку в них самая важная часть - обнаружение конфликта, для которого может применяться простая операция сравнения с обменом. Однако в таком случае нельзя гарантировать прогресс ни для каждого из потоков, ни для программы в целом.

Механизмы второго вида гарантируют, что как минимум один вызов операции завершит работу за конечное число шагов. Такое свойство наиболее часто встречается в алгоритмах с неблокирующей синхронизацией в силу того, что оно, с одной стороны, гарантирует "общесистемный"прогресс - то есть прогресс в выполнении задачи решаемой программой, а с другой стороны, реализуется проще по сравнению к алгоритмам без ожидания, рассматриваемым далее.

Механизмы третьего вида гарантируют, что каждый вызов операции завершится за конечное число шагов. Это очень сильное требование, означающее, что в процессе работы программы не могут возникать ситуации, при которых хотя бы один поток находится в состоянии ожидания или возвращается к предыдущему состоянию для попытки повторного совершения операции. Положительной стороной алгоритмов, обладающих таким свойством, является

возможность расчета времени работы даже в пессимистичном случае, однако в [5] отмечается, что обычно такие алгоритмы требуют широкой программной синхронизации, основанной на взаимодействии потоков через фрагменты памяти с одним записывающим потоком и остальными читающими. Таким образом один поток может сообщать другим о проводимой в данный момент операции. Тогда каждый поток, который успешно продвигается в вычислениях, должен периодически проверять состояние остальных потоков и "помогать"тем из них, которые отстают.

Очевидно, что эти требования следуют по возрастанию и каждое последующее включает в себя предыдущее.

Рассмотрим подробнее каждое из них.

3.1 Отсутствие препятствий

Пример алгоритма, удовлетворяющего такому требованию, представлен в статье [3]. В ней авторы на примере двунаправленной очереди и циклического буфера представляют идею синхронизации без препятствий.

Основная идея заключается в использовании системы версионирования для каждого элемента очереди. Операции добавления и удаления из очереди используют бесконечный цикл, в котором контролируют состояние версии текущего и соседнего элементов и таким образом поддерживают конгруэнтность данных. В случае изменения версии в процессе работы, операция начинается сначала. Очевидно, что для такой реализации, с одной стороны, отсутствуют какие-либо препятствия (как этого и требует условие), но с другой стороны, возможен сценарий, при котором операции не завершатся никогда.

3.2 Отсутствие блокировок

Реализация алгоритма, удовлетворяющего этому требованию, представлена в статье [4]. В ней авторы на примере односвязного списка и производных от него структурах данных рассматривают методы неблокирующей синхронизации и проблемы, связанные с обновлением и обходом такого списка. Несмотря на то, что в рассматриваемой в этой статье задаче мало аналогий с нашей,

общие принципы и способы решения задач поддержания конгруэнтности данных и сохранения данных, модифицируемых другими потоками, могут быть обобщены и применены к решению нашей задачи. Особенно важен момент сохранения данных в моменты удаления их другими потоками, так как алгоритмы модификации могут зависеть от "соседних"данных, не модифицируя, но используя их. Также важно отметить, что реализация алгоритма, используемого в этой работе, использует функцию сравнения с обменом одного слова, так как во многих архитектурах представлена только такая операция.

Несколько походов к реализации алгоритмов, удовлетворяющих данному условию рассмотрены в работе [5]. В частности, в ней рассмотрены алгоритмы, основанные на операции сравнения с обменом нескольких слов, а также способ реализации такой операции с использованием операции сравнения с обменом одного слова. Также описывается синхронизация работы с общими данными при помощи транзакций.

3.3 Отсутствие ожидания

Пример алгоритма, отвечающего данному требованию, представлен в статье [6]. В ней авторы отмечают сложность программирования многопоточных приложений и предлагают компромиссное решение в виде замены блокировок на транзакции. Суть предлагаемого решения в следующем: авторы предлагают выполнять критические секции без использования блокировок, а затем разрешать конфликты в случае их возникновения.

В статье [7] предлагается реализация алгоритма очереди без ожидания. Особенностью этого решения является реализация двух вариантов исполнения в каждой операции: быстрого и медленного, однако предлагаемый механизм использует вспомогательные структуры данных, размеры которых зависят от количества потоков, что накладывает целый ряд требований к программе и делает невозможным создание переменного числа потоков.

Также представляет интерес работа [8], в которой рассматривается подход к решению задачи неблокирующей синхронизации с точки зрения "малых объектов" – объектов, размеры которых позволяют их эффективно копировать. В

ней рассмотрены алгоритмы без блокирования и без ожидания, а также их модификации для случая "больших объектов" – объектов, копирование которых не целесообразно по соображениям производительности.

4 Архитектура прототипа

В начале определим сущности, которыми мы будем оперировать при решении данной задачи. Первое, с чем мы будем работать - это линейные участки.

Следующая сущность - обработчик раунда кодирования. Поскольку различные алгоритмы кодирования могут использовать разное число раундов - необходимо, чтобы была возможность создавать произвольное количество обработчиков раундов и связи между ними.

Кроме этого будем рассматривать синхронизатор как набор раундов. В соответствии с используемой архитектурой механизма синхронизации, есть часть данных, которые являются общими для всех раундов. Он также будет отдельной сущностью.

Вся рассматриваемая программа также является сущностью, имеющей собственную функциональность. Далее, называя некую сущность, будем иметь в виду экземпляр соответствующего класса.

Соответствующая диаграмма классов представлена на рис. 1

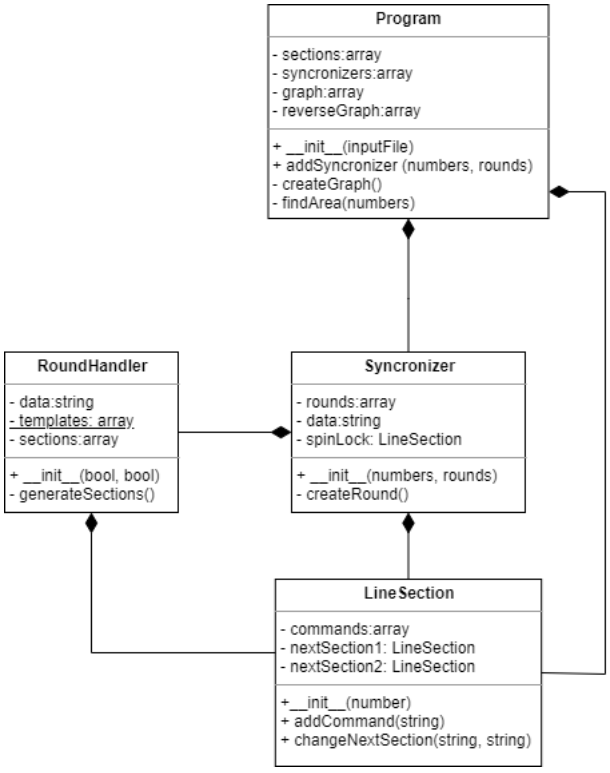

Рис. 1: Диаграмма классов

Рассмотрим каждую сущность по отдельности

4.1 Программа

Крупнейшей сущностью является программа. Внутри этой сущности находятся все общие данные об исходной программе и генерируются служебные данные для дальнейшей работы, такие как граф потока управления, номера существующих линейных участков для предотвращения совпадения с создаваемыми и т.д.

4.2 Синхронизатор

Синхронизатор полностью отвечает за работу с одним защищаемым участком. При создании он получает список номеров линейных участков, входящих в защищаемый участок. Внутри себя он содержит общие данные, которые требуются для работы на всех раундах кодирования, такие как общие счетчики, флаги, линейный участок, отвечающий за реализацию спинлока, блокирующего исполнение неподготовленного защищаемого участка и тд. Кроме этого он содержит в себе массив всех обработчиков раундов.

4.3 Обработчик раунда кодирования

Поскольку различные механизмы кодирования требуют разного числа раундов работы — удобнее вынести обработчики раундов кодирования в отдельную сущность и реализовать требуемую функциональность, которая включает в себя механизм синхронизации на данном раунде и данные, которые используются при его работе.

4.4 Линейный участок

Линейный участок — наименьшая из сущностей, с которыми мы будем работать. Она хранит в себе список команд, из которых состоит линейный участок, номера линейных участков, на которые передается управление из данного.

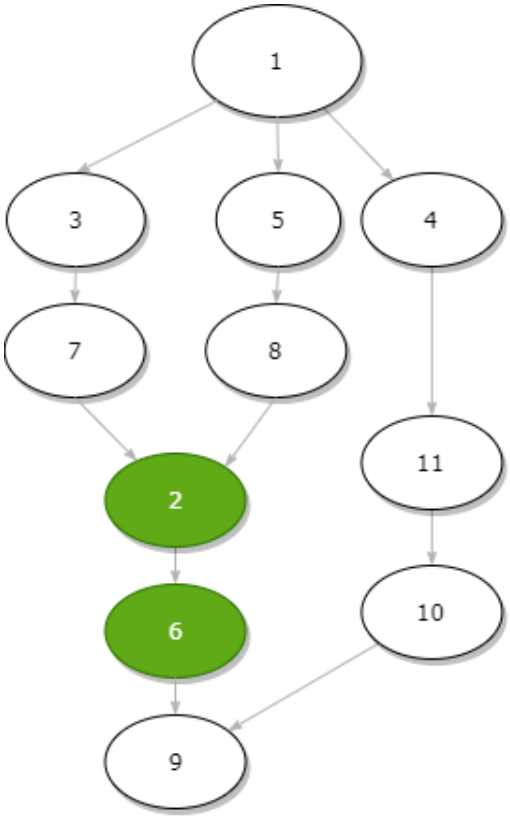

Рис. 2: Пример графа потока управления. Необходимо добавить механизм синхронизации между участками 3-7 и 5-8

5 Реализация прототипа

Представленный в обзоре механизм синхронизации не в полной мере отвечает нашим требованиям, а именно – он не пригоден для применения в произвольном месте программы, поэтому в начале модифицируем его для дальнейшего использования в решении поставленной задачи.

5.1 Модификация механизма синхронизации

Указанный в работе [10] механизм синхронизации нельзя добавить в случае, когда в защищаемый участок входит более одной дуги графа потока управления (рис. 2)

Для обработки такой ситуации модифицируем исходный механизм: будем создавать несколько линейных участков, сохраняющих состояние регистров процессора, а перед этим сохранять на стек адреса линейных участков, куда

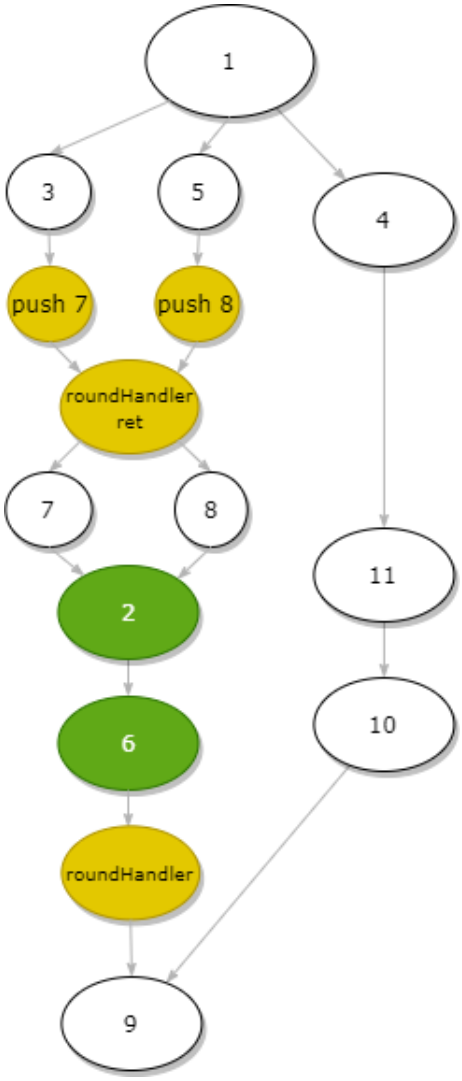

Рис. 3: Пример графа потока управления после модификации

ведут ребра графа потока управления до встраивания обработчика раунда. В этом случае на выходе из обработчика раунда после восстановления значений регистров нам достаточно будет выполнить команду *ret* и в зависимости от того, какой адрес был сохранен в стеке, мы вернемся в соответствующее места графа потока управления. В приведенном примере это будут номера 7 и 8 соответственно. Итоговый граф потока управления представлен на рис. 3

5.2 Реализация прототипа

Прототип модуля синхронизации должен решать задачу добавления одного или нескольких раундов кодирования. В начале рассмотрим решение этих задач в общем виде.

5.2.1 Добавление одного раунда

Формализуем условие задачи: у нас есть граф потока управления G_1 и граф с обратными ребрами G_2, оба связные, а также набор защищаемых линейных участков. Не умаляя общности, можно считать, что защищаемые линейные участки образуют одну компоненту связности, в противном случае можно рассмотреть каждую компоненту связности по отдельности либо расширить набор вершинами до образования одной общей компоненты связности. Областью, для которой решается задача, будем называть набор вершин, соответствующих защищаемым линейным участкам, и набор дуг, начальные и конечные вершины которых – защищаемые линейные участки.

Для добавления одного раунда нам необходимо определить на графе потока управления набор ребер, входящих в область, для которой решается задача, и набор ребер, исходящих из этой области. Если на всех входящих ребрах разместить раскодирующие блоки, а на исходящих – закодирующие, то можно уверенно утверждать, что в момент исполнения любого участка из области он будет раскодирован, а в то время, когда управление находится вне данной области – он будет закодирован.

5.2.2 Добавление нескольких раундов

Задача добавления нескольких раундов кодирования может быть сведена к задаче добавления одного раунда следующим образом: пронумеруем раунды кодирования “от центра”: нулевым будет раунд, на выходе которого получается готовый к исполнению линейный участок, последним (n-ым) – раунд, с которого начинается работа по подготовке линейного участка.

Тогда задача добавления нулевого раунда аналогична задаче с одним раундом. Переопределим входные данные следующим образом: включим в область, для которой рассчитывается встраивание раундов, предыдущие раунды (с меньшими номерами) и линейные участки, которые должны отделять очередной раунд от предыдущего. И для этой задачи решим задачу добавления одного раунда.

5.2.3 Общие принципы работы прототипа

Важной деталью описанного выше подхода к решению задачи синхронизации исполнения защищаемых линейных участков является то, что раунды имеют небольшие, но важные отличия в зависимости от номера: например, раундам с номерами больше 0 может потребоваться возвращение к более раннему (с меньшим номером) раунду с закодированием, если по каким-то причинам оно не было выполнено ранее, но у 0 раунда такой возврат, очевидно, невозможен. В то же время все раунды, кроме 0 и n-ого, будут иметь схожую функциональность. Кроме этого, секции ожидания во всех раундах одинаковы.

Кроме этого, в случае кодирования с одним раундом необходима комбинация функциональности первого, последнего и промежуточного раундов из описанной выше задачи.

Поэтому для решения задачи для произвольного числа раундов были созданы шаблоны для генерации раундов кодирования различных номеров, на основании которых, в зависимости от номера раунда, генерируется необходимый механизм синхронизации.

6 Тестирование

Для оценки влияния синхронизации на производительность была выбрана задача перемножения матриц размером 128*128 элементов, как наиболее "классическая"задача, которая при этом хорошо изучена, имеет точную оценку сложности $O(n^3)$, не зависящую от входных данных. Также эта задача может решаться в условиях идеального параллелизма – каждый элемент итоговой матрицы может быть вычислен независимо от других, а значит можно исследовать до 16384 потоков. Добавим в нее дополнительное условие: в итоговую матрицу будем добавлять только положительные элементы. Это необходимо для того, чтобы была возможность проверить сразу два случая: когда защищаемый участок исполняется всеми потоками на каждой итерации (в данном случае – перемножение элементов исходных данных) и когда защищаемый участок может быть выполнен либо не выполнен в зависимости от какого-то условия (в данном случае – запись в итоговую матрицу в зависимости от зна-

ка вычисленного результата). Для получения более достоверных данных была выполнена серия запусков с одинаковыми параметрами и рассматривалось среднее значение и отклонение. В качестве меры времени был взят условный квант времени – время выполнения одного линейного участка. Такой выбор был сделан по следующим соображениям: с одной стороны, на сегодня существует огромное количество различных вычислительных систем, и измерение в единицах времени (секундах, минутах, часах) будет справедливо для конкретной модели или даже конкретного экземпляра в конкретный момент времени. С другой стороны, линейный участок является базовым элементом программы, сгенерированной компилятором МАК, этот элемент достаточно хорошо изучен. Также характеристики этого элемента не меняются с момента начала его исследования в независимости от архитектур вычислительных систем. В начале рассмотрим граф потока управления данной программы:

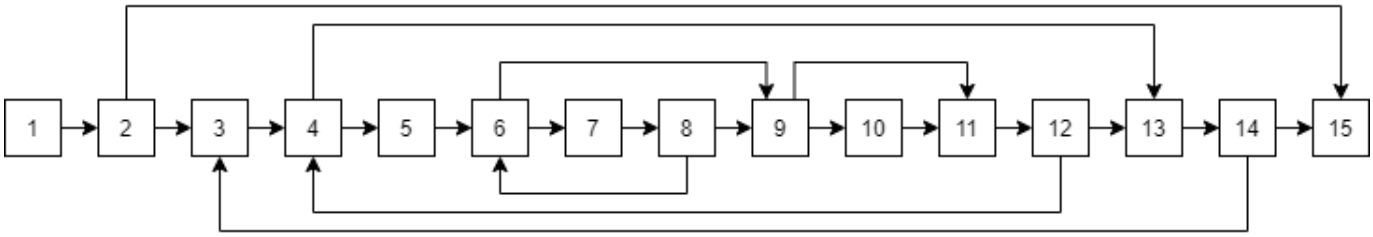

Рис. 4: Граф потока управления функции перемножения матрицы тестовой программы

Здесь видны все три вложенных цикла, сложение произведений элементов матрицы (участок 7), проверка знака полученного элемента матрицы (участок 9). В начале рассмотрим случай кодирования участка 7, он выполняется всегда, всеми потоками. Расположение раундов кодирования будет следующим:

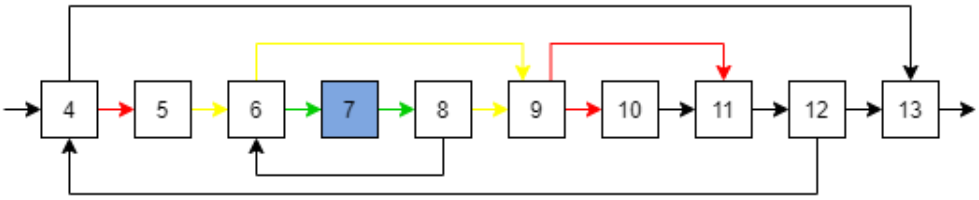

Рис. 5: Схема модификации графа потока управления. Цветные ребра – ребра, в которые встраивались механизмы синхронизации

Здесь цветами показаны ребра, в которые будут встроены механизмы синхронизации и кодирования. Результаты испытаний показаны ниже:

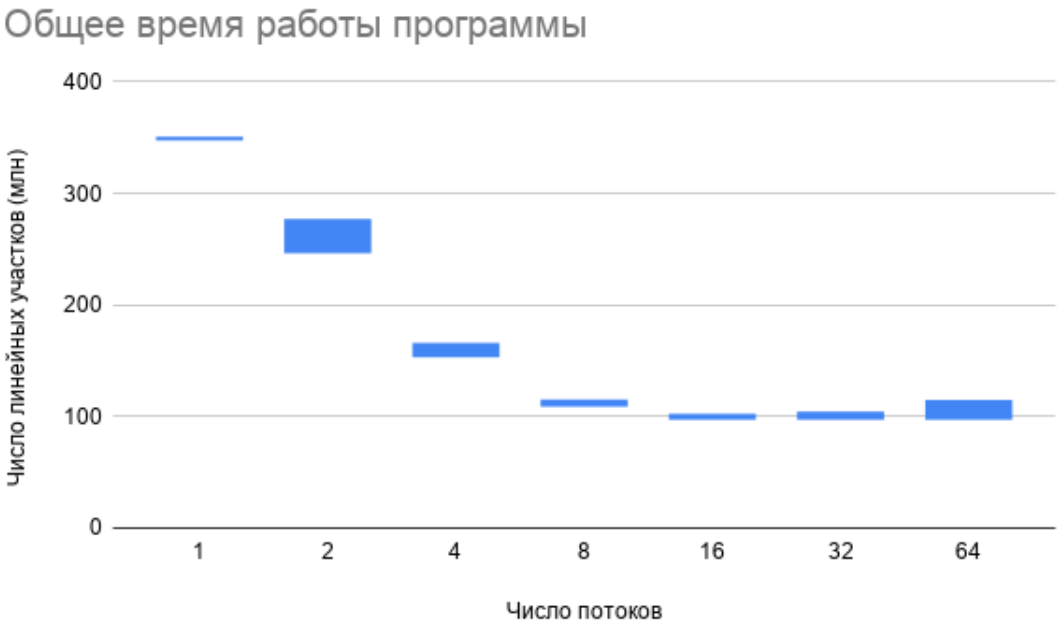

Рис. 6: Зависимость времени работы приложения от числа потоков. Верхние и нижние границы соответствуют максимальным и минимальным значениям в серии экспериментов

На данном графике видно, что разброс значений близок к нулю при 1 потоке, как и следовало ожидать, так как в этом случае один запуск от другого может отличаться только числом добавлений элементов в итоговую матрицу. Затем наблюдается увеличение разброса, с последующим снижением. Это показывает, что, начиная с двух потоков, появляется вторая причина различий числа выполненных линейных участков: согласованность работы различных потоков по кодированию защищаемой области. Это говорит о том, что потоки до определенного количества работают достаточно синхронно. Этим же объясняется снижение общего числа выполненных линейных участков на Рис. 7, части потоков не требуется раскодирование линейного участка перед его выполнением. Также важным параметром является время нахождения защищаемого участка в раскодированном состоянии, поскольку одной из основных целей защиты является сокрытие защищаемых линейных участков, и чем меньше времени они будут находиться в раскодированном состоянии, тем более надежно они

будут защищены.

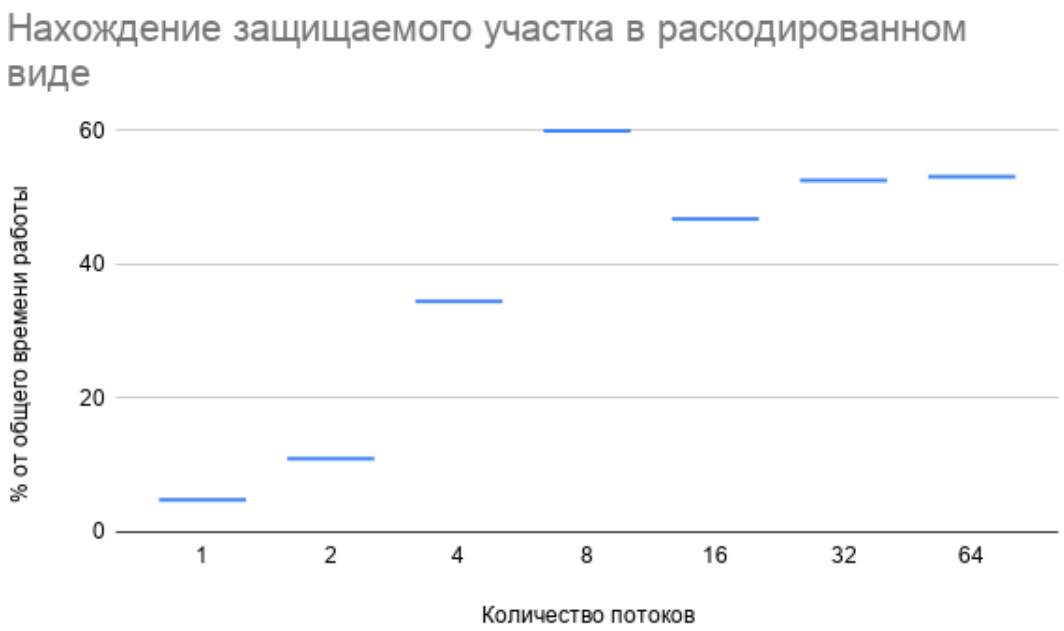

Рис. 7: Время нахождения защищаемого участка в раскодированном состоянии

При сравнении этого графика с рис.7 видно, что значительную часть времени защищаемый участок закодирован, а из роста разброса этой величины и общего поведения обоих графиков можно сделать вывод, что во там, где на графике отклонений происходит снижение величины, здесь виден рост. Учитывая его, а также то, что на первом графике видно снижение общего числа выполненных линейных участков, можно сделать вывод, что снижение выбросов связано с тем, что кодируемый участок значительное время находится в раскодированном состоянии и потоки редко его раскодируют, а следовательно, уменьшается различие в числе выполненных линейных участков. Также в рамках исследования следовало бы изучить количество выполнений защищаемого линейного участка, однако во всех экспериментах он составлял 2097152 раза что является 128^3, как и должно быть из оценки сложности наивного перемножения матриц.

Рассмотрим ситуацию, когда линейный участок может быть выполнен или не выполнен в зависимости от неких вычислений. В нашем случае в зависимости от знака результата может быть выполнен либо не выполнен линейный участок

номер 10.

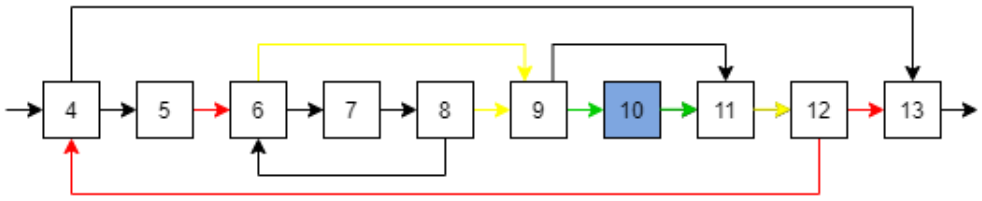

Рис. 8: Схема модификации графа потока управления

В этом случае были получены следующие результаты (Рис. 9).

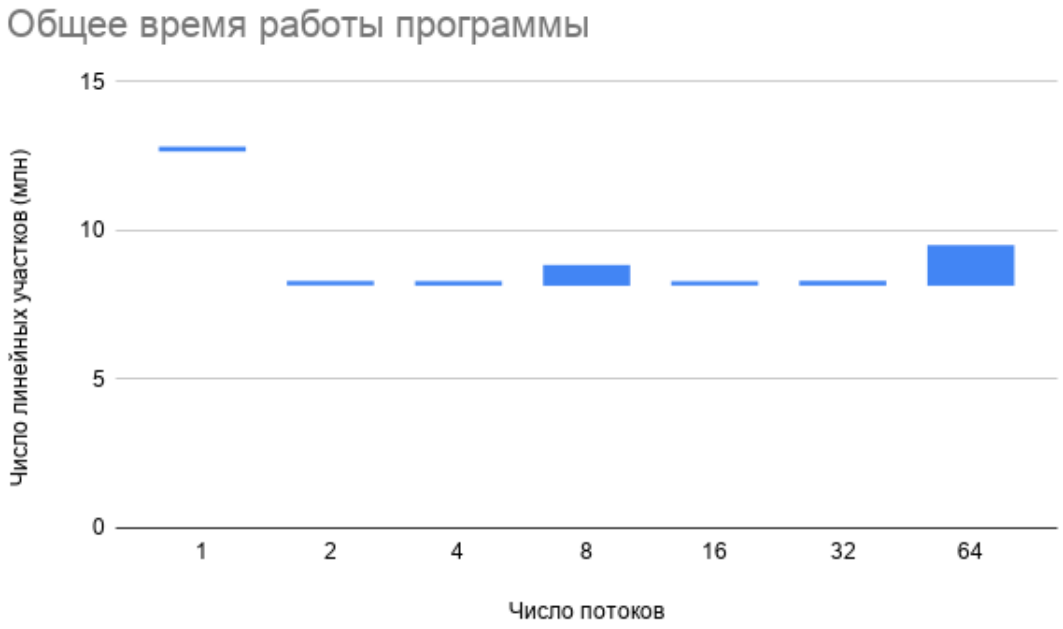

Рис. 9:

Здесь видно аналогичное явление: с ростом числа потоков, число выполненных линейных участков уменьшается, однако характер несколько другой: сильно снизившись после увеличения числа потоков до 2, оно остается почти постоянным. Для более подробного изучения этого явления рассмотрим время, в течении которого кодируемый участок был раскодирован.

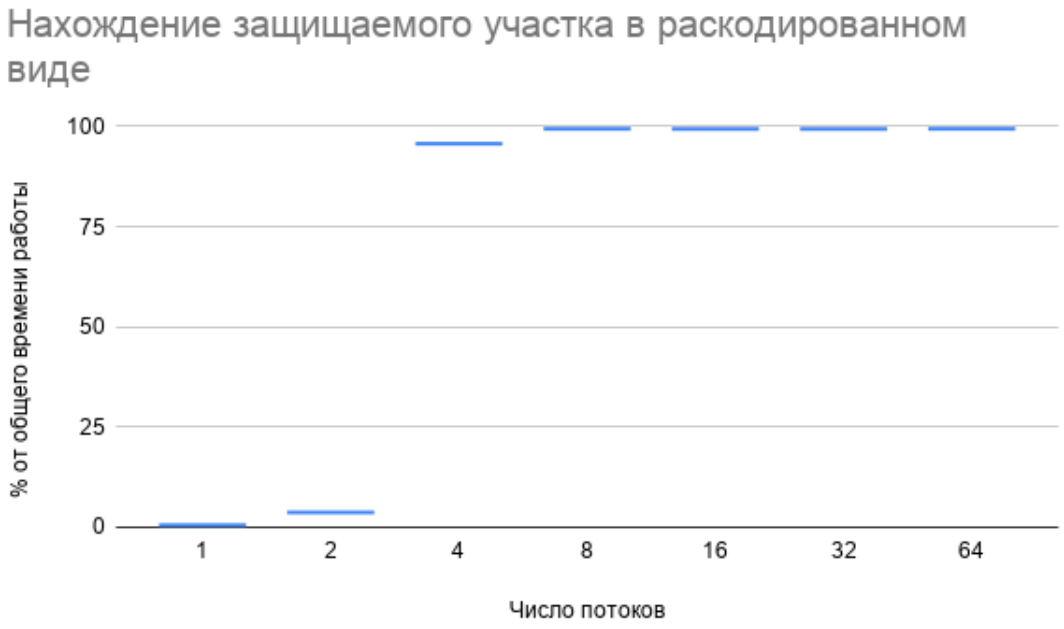

Рис. 10:

Из этой диаграммы видно что основное время участок находится в раскодированном состоянии. Это объясняет, почему число выполненных линейных участков снизившись в начале, далее остается практически постоянным: сокращению числа выполненных линейных участков больше негде происходить, участок и так остается раскодированным. В теории число потоков не должно влиять на количество выполнений кодируемого линейного участка, чтобы это проверить, рассмотрим число выполнений этого линейного участка в зависимости от числа потоков:

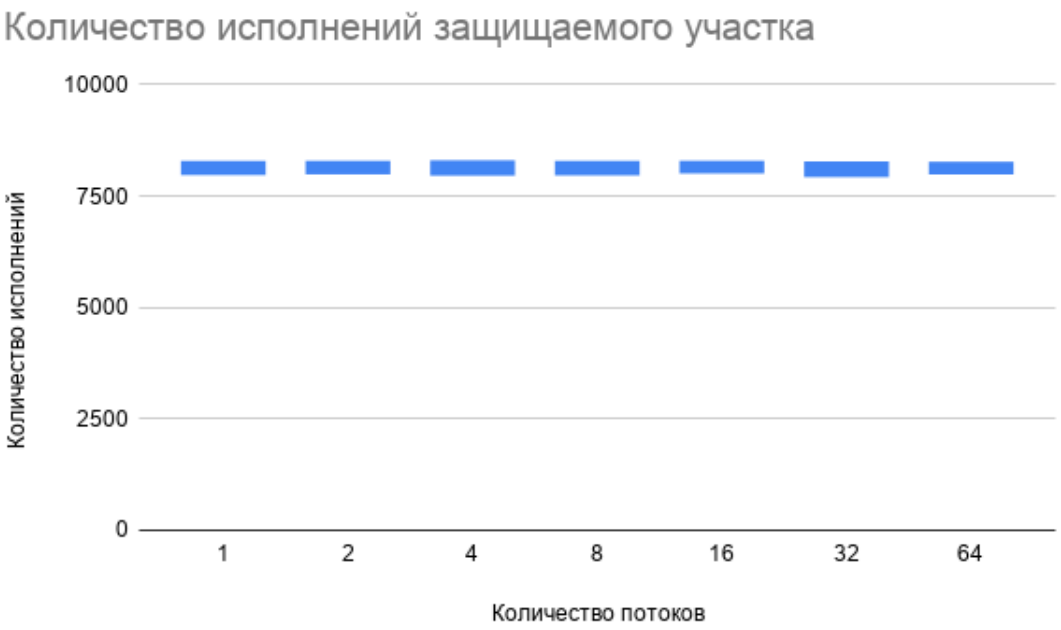

Рис. 11:

Отсюда видно, что и та, и другая величина являются практически константными, как это и предполагалось изначально. Таким образом была обнаружена уязвимость в таком методе синхронизации потоков: при отсутствии обработки условных операторов на этапе встраивания возможен сценарий, при котором защищаемый участок большую часть времени работы программы будет находиться в раскодированном состоянии. Из всего вышеизложенного можно сделать вывод, что наиболее оптимальным и с точки зрения производительности, и с точки зрения защиты является первый случай применения защиты.

7 Заключение

В рамках данной работы была достигнута поставленная цель.

- Был проведен обзор существующих решений задачи синхронизации работы потоков с разделяемыми ресурсами с применением методов неблокирующей синхронизации.
- Была проведена разработка архитектуры модуля автоматизированного встраивания механизмов синхронизации в произвольную MAK-компилируемую программу, в рамках которой были установлены требования к обфускаторам.
- Реалзиован прототип модуля компилятора, отвечающего за встраивание механизмов синхронизации.
- Проведено его тестирование и исследование влияния механизмов синхронизации на работу исходной программы, показавшие некоторые уязвимости такого решения.

Список литературы

[1] Дмитрий Намиот On lock-free programming patterns WSEAS TRANSACTIONS on COMPUTERS Volume 15 E-ISSN: 2224-2872, 2016 МГУ, Москва

[2] Ю. Лифшиц. Обфускация (запутывание) программ. Обзор. URL: https://logic.pdmi.ras.ru/~yura/of/survey1.pdf (дата обращения: 27.05.2020)

[3] Herlihy, M., Luchangco, V., & Moir, M. (2003, May). Obstruction-free synchronization: Double-ended queues as an example. In Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on (pp. 522-529). IEEE.

[4] John D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 214–222, August 1995

[5] Fraser, K. (2004). Practical lock-freedom (Doctoral dissertation, University of Cambridge)

[6] Rajwar, R., & Goodman, J. R. (2002). Transactional lock-free execution of lock-based programs. ACM SIGOPS Operating Systems Review, 36(5), 5-17.

[7] Kogan, A., & Petrank, E. (2012, February). A methodology for creating fast wait-free data structures. In ACM SIGPLAN Notices (Vol. 47, No. 8, pp. 141-150). ACM

[8] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. ACM Transactions on Programming Languages and Systems, 15(5):745 – 770, November 1993. (pp 17, 18)

[9] Maurice Herlihy. Wait-free synchronization. ACM Transactions on Programming Languages and Systems. 13(1):124 - 149 Jan. 1991

[10] Бабанов П. А. Многопоточное исполнение МАК обфусцированных программ. Выпускная квалификационная работа. Санкт-Петербургский Государственный Университет. 2018 год