Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт
(национальный исследовательский университет)»
Физтех-школа Радиотехники и Компьютерных Технологий
Кафедра инфокоммуникационных систем и сетей

**Направление подготовки / специальность:** 03.03.01 Прикладные математика и физика
**Направленность (профиль) подготовки:** Радиотехника и компьютерные технологии

# РЕАЛИЗАЦИЯ ЭФФЕКТИВНОГО АЛГОРИТМА АНАЛИЗА БЕЗОПАСНОСТИ МОДЕЛИ «БРАТЬ-ДАВАТЬ»

(бакалаврская работа)

**Студент:**
Лифантьев Данил Александрович

_____
*(подпись студента)*

**Научный руководитель:**
Промыслов Виталий Георгиевич,
канд. физ.-мат. наук

_____
*(подпись научного руководителя)*

**Консультант** *(при наличии)***:**
_____
_____
*(подпись консультанта)*

Москва 2023

## Аннотация

Дипломная работа посвящена практическим аспектам применения формальных (математических) моделей информационной безопасности. Основное внимание уделено реализации эффективного алгоритма проверки безопасности в дискреционной модели безопасности типа take-grant («брать-давать»). Для нее сделан теоретический обзор основных методов, входящих в классическую и расширенную модель take-grant, и предложена программная реализация алгоритма одного из основных методов проверки безопасности $can\_share$ на языке Python с линейной вычислительной сложностью на основе алгоритма Снайдера (L. Snyder). В отличии от исходного алгоритма, разработанная программная реализация использует современные методы распараллеливания вычислений, что позволяет повысить скорость вычислений на современных процессорных архитектурах. Проанализирована эффективность полученного решения и произведено его тестирование. Результаты работы могут быть использованы для улучшения систем, связанных с применением модели take-grant в информационной безопасности.

## Содержание

# Введение

В настоящее время компьютерные системы и сети определяют надежность большинства систем реального мира. Массовое внедрение автоматизированных процессов обработки и хранения информации решает многие задачи, но также делает эти процессы более уязвимыми, что приводит к появлению новой проблемы – информационной безопасности. Один из способов решения этой проблемы – создание политики безопасности, которая включает в себя нормы и правила для обработки информации, обеспечивающие защиту от различных угроз и являющиеся необходимым условием безопасности системы [2]. Таким образом, использование политики безопасности – это эффективный способ защиты информации от угроз и повышения уровня безопасности системы.

Политика безопасности часто выражена в неформальном (текстовом) виде, например, в виде наборов организационных мер, правил, целей безопасности, принятой для организации. Однако известно, что если для системы необходим высокий уровень информационной безопасности, то в основе политики безопасности должна лежать некая формальная, математическая модель [3, требование ADV_SPM].

В настоящее время разработано большое количество математических моделей безопасности и реализаций алгоритмов для их проверки, но одной из первых была модель дискреционного доступа ХРУ [4], которая реализована в большинстве компьютерных систем. Её достоинствами являются относительно простая реализация механизмов защиты информации в информационных системах. Управление доступом в дискреционном подходе основывается на двух свойствах: идентификации всех субъектов и объектов и определении прав доступа субъектов к объектам на основании внешних правил [5]. Субъект доступа – активная

сущность, которая может изменять состояние системы через порождение процессов над объектами, включая порождение новых объектов и инициализацию порождения новых субъектов. Объект доступа – пассивная сущность, процессы над которой могут в определенных случаях быть источником порождения новых субъектов [1]. При использовании дискреционной политики безопасности возникает необходимость определения правил распространения прав доступа и анализа их влияния на безопасность системы. Поэтому важно тщательно анализировать реализацию политик безопасности для обеспечения эффективной защиты информации и повышения уровня безопасности системы.

В случае использования дискреционной политики безопасности возникает необходимость определения правил распространения прав доступа и анализа их влияния на безопасность системы.

Недостатком классической модели ХРУ является то, что задача проверки безопасности для системы в общем виде алгоритмически неразрешима. Поэтому одним из развитий классической дискреционной модели безопасности является модель take-grant («брать-давать»), которая позволяет провести анализ безопасности системы за линейное время от размера системы [6].

Компьютерная система в модели take-grant представлена в виде ориентированного графа, вершины которого являются субъектами и объектами моделируемой системы и дуги, которые представляют собой набор прав доступа. Анализ графа позволяет определить, возможно ли конкретному субъекту получить доступ к тому или иному объекту за определенное количество шагов. Имеется в виду, что он выявляет возможность модификации исходного графа так, чтобы между двумя вершинами появилась дуга, которой не было в исходном графе.

Прикладным результатом подобного анализа может являться: перечень возможных успешных атак, не учтенных при разработке системы; соотношение реализуемых мер безопасности и уровня защищенности системы; перечень наиболее критичных уязвимостей; наименьшее множество мер, реализация которых сделает систему защищенной от атак. Анализ графа также используется при расследовании компьютерных инцидентов, для поиска рисков и корреляций с предупреждениями систем обнаружения атак.

Модель take-grant с момента своего появления и до сегодняшнего времени используется для анализа безопасности как телекоммуникационных, офисных, так и промышленные систем [7][8]. В работе [7] представлена формальная модель безопасности дистрибутивов Linux, в которой структура анализа информационных потоков, описана как расширенная модель take-grant. В статье [8] предлагается подход к визуализации систем контроля доступа на основе треугольных матриц. Подход используется для визуализации модели безопасности управления доступом, основанной на методах моделей ролевого разграничения доступа и take-grant. В работе утверждается, что по сравнению с обычными матрицами доступа разреженность треугольных матриц меньше, и подход способен визуализировать вложенность на уровне прав.

В общей практике известна классическая и расширенная модель take-grant. Расширенная модель является продолжением классической с поддержкой дополнительных прав доступа между объектами системы для анализа информационных потоков. В работе будет произведен обзор обеих моделей.

В качестве основной цели решалась задача разработки эффективной программной реализации предиката $can\_share$ для проверки возможности получения прав доступа в модели take-grant. Предикат $can\_share$ является одним из основных предикатов анализа безопасности, который

используется как в классической, так и в расширенной модели. Дополнительная задача ставилась – получить простое, легко интегрируемое в более сложные средства анализа безопасности решение для программной реализации, которое в дальнейшем можно будет совершенствовать, дополняя новыми предикатами анализа безопасности модели take-grant, построенными на основе имеющихся наработок, и применять для реальных информационных систем.

# Основная часть

## 1 Теоретические сведения

### 1.1 Классическая модель take-grant

Модель take-grant — это формальная модель, используемая в области компьютерной безопасности, для анализа систем дискреционного разграничения доступа; подтверждает либо опровергает степени защищенности данной автоматизированной системы, которая должна удовлетворять регламентированным требованиям [9]. Данная модель распространения прав доступа использует графы для описания отношений доступа между объектами и субъектами политики безопасности.

Основными элементами классической модели take-grant являются [5]:

- $O$ – множество объектов;
- $S \subseteq O$ – множество субъектов;
- $R = \{r_1, r_2, \dots, r_m\} \cup \{t, g\}$ – множество видов прав доступа, где $t$ (take) – право брать права доступа; $g$ (grant) – право давать права доступа;
- $G = (S, O, E)$ – конечный помеченный ориентированный ациклический граф, представляющий текущие доступы в системе. Элементы множеств $S$ и $O$ являются вершинами графа, которые на диаграммах обозначаются:
  - $\otimes$ – объекты (элементы множества $O \setminus S$),
  - $\bullet$ – субъекты (элементы множества $S$).
- Элементы множества $E = |S| \times |O| \times |R|$ являются ребрами графа. Каждое ребро помечено непустым подмножеством множества видов прав доступа $R$. Основная цель классической модели take-grant – определение и обоснование алгоритмически проверяемых условий проверки возможности утечки права доступа по исходному графу

доступов, соответствующего некоторому состоянию системы. Порядок перехода системы модели take-grant из состояния в состояние определяется правилами преобразования графа доступов, которые в классической модели носят название де-юре правил. Преобразование графа $G$ в граф $G'$ в результате выполнения правила $\mathrm{op}$ обозначается следующим образом: начальный граф доступа $G$, задаваемый формальной моделью безопасности, может быть трансформирован последовательным применением правил в новый граф $G'$, трансформация обозначается как $G \vdash_{op} G'$.

В классической модели take-grant рассматриваются четыре де-юре правила преобразования графа, выполнение каждого из которых может быть инициировано только субъектом, являющимся активной компонентой системы [10]:

- **Определение.** $take$ – право брать права доступа;

  Пусть $x, y$ и $z$ — три различные вершины графа доступов $G$, и пусть $x$ — субъект. Пусть существует ребро из $x$ в $y$ с правом доступа $\gamma$, таким что $t \in \gamma$; ребро из $y$ к $z$ с правом $\beta$, таким что $\alpha \subseteq \beta$. Тогда правило $take$ определяет новый граф доступов $G'$, добавляя ребро от $x$ до $z$, с правом $\alpha$. Графическое представление на Рис. 1.

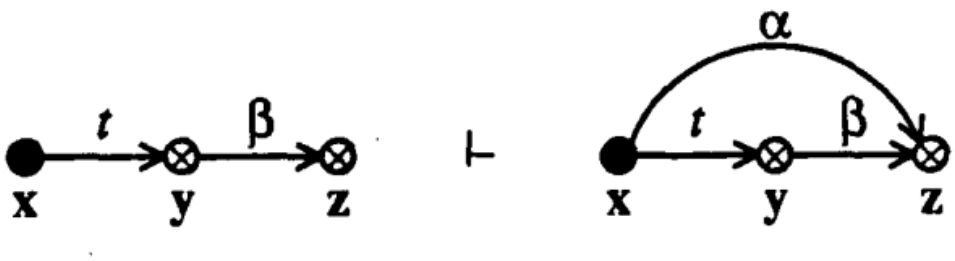



Рис. 1: Применение правила $take$

- **Определение.** $grant$ – право давать права доступа;

  Пусть $x$, $y$ и $z$ — три различные вершины графа доступов $G$, и пусть $x$ — субъект. Пусть существует ребро из $x$ в $y$ с правом доступа $\gamma$, таким что $g \in \gamma$; ребро из $x$ к $z$ с правом $\beta$, таким что $\alpha \subseteq \beta$. Тогда правило $grant$ определяет новый граф доступов $G'$, добавляя ребро от $y$ до $z$, с правом $\alpha$. Графическое представление на Рис. 2.

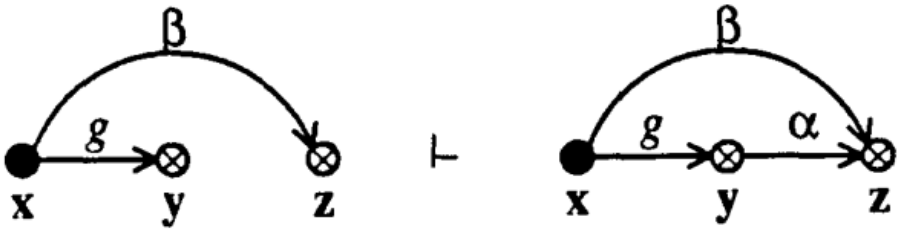



Рис. 2: Применение правила $grant$

- **Определение.** $create$ – право создавать новый объект или субъект; при этом субъект-создатель может взять на созданный субъект любые права доступа (по умолчанию предполагается, что при выполнении правила $create$ создается объект);

  Пусть $x$ — любой субъект в графе защиты $G$, $\alpha$ — произвольное право доступа из $R$. $create$ определяет новый граф доступов $G'$, добавляя к графу новую вершину $y$ и ребро от $x$ до $y$ с правом $\alpha$. Графическое представление на Рис. 3.

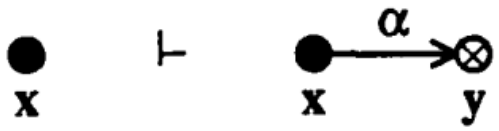



Рис. 3: Применение правила $create$

- **Определение.** $remove$ – право удалять права доступа.

  Пусть $x$ и $y$ — любые различные вершины в графе доступов $G$, такие

что $x$ является субъектом. Пусть существует ребро из $x$ в $y$, с множеством прав $\beta$, и $\alpha$ — произвольное право доступа из $R$. Тогда $remove$ определяет новый граф $G'$, удаляя право $\alpha$ из множества $\beta$. Если в результате $\beta$ станет пустым, само ребро будет удалено. Графическое представление на Рис. 4.

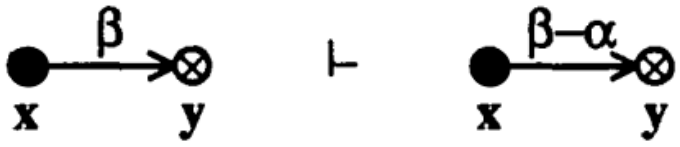



Рис. 4: Применение правила $remove$

Начальное и конечное состояния графов после действия упомянутых правил также удобно представить в виде таблицы:

Табл. 1: Де-юре правила классической модели take-grant

| Правила | Исходное состояние $G = (S, O, E)$ | Результирующее состояние $G' = (S', O', E')$ |
|---|---|---|
| $grant(\alpha, x, y, z)$ | $x \in S,$<br>$(x, y, \{g\}) \subset E,$<br>$(x, z, \beta) \subset E,$<br>$y \neq z, \alpha \subseteq \beta$ | $S' = S$<br>$O' = O$<br>$E' = E \cup \{(y, z, \alpha)\}$ |
| $take(\alpha, x, y, z)$ | $x \in S,$<br>$(x, y, \{t\}) \subset E,$<br>$(y, z, \beta) \subset E,$<br>$x \neq z, \alpha \subseteq \beta$ | $S' = S$<br>$O' = O$<br>$E' = E \cup \{(x, z, \alpha)\}$ |
| $remove(\alpha, x, y)$ | $x \in S,$<br>$(x, y, \beta) \subset E,$<br>$\alpha \subseteq \beta$ | $S' = S$<br>$O' = O$<br>$E' = E \setminus \{(x, y, \alpha)\}$ |
| $create(\beta, x, y)$ | $x \in S,$<br>$y \notin O,$<br>$\beta \neq \emptyset$ | $O' = O \cup \{y\},$<br>$S' = S \cup \{y\}$ (если $y$ – субъект),<br>$E' = E \cup \{(x, y, \beta)\}$ |

## 1.2 Расширенная модель take-grant

Кроме классической модели также существует расширенная модель take-grant, которая строится на основе первой. В ней в большей степени рассматриваются не абстрактные отношения связанные с передачей прав, как в классической модели, а более реалистичные отношения доступа типа чтения и записи данных $r$ $(read)$ и $w$ $(write)$, наличие которых у субъектов системы является причиной возникновения информационных потоков. Отличие от изначальной модели в том, что $R = \{r_1, r_2, \dots, r_m\} \cup \{t, g, r, w\}$ – множество прав доступа расширено; $G = (S, O, E \cup F)$ – конечный помеченный ориентированный ациклический граф, представляющий текущие доступы в системе, где множества $S$, $O$ соответствуют вершинам графа. $E = |S| \times |O| \times |R|$ представляют реальные ребра графа, помеченные непустыми подмножествами из множества прав доступа. Ребра $F = |S| \times |O| \times \{r, w\}$ соответствуют мнимым информационным потокам (пунктирные линии на диаграммах). Порядок перехода графа из одного состояния в другое определяется де-юре и де-факто правилами преобразования графа доступов и информационных потоков. Де-факто правила применяются к «реальным» или «мнимым» ребрам (элементам множества $E$ и $F$), помеченным $r$ или $w$. Результатом применения де-факто правил является добавление новых «мнимых» ребер в множество $F$. Де-факто правила и результаты их применений отражены в таблице [5]:

Табл. 2: Де-факто правила расширенной модели take-grant

| Правила | Исходное состояние $G = (S, O, E \cup F)$ | Результирующее состояние $G' = (S, O, E \cup F')$ |
|---|---|---|
| Первое правило | $x \in S, y \in O,$ $(x, y, r) \in E \cup F,$ | $F' = F \cup \{(y, x, w), (x, y, r)\}$ |

| | | |
|---|---|---|
| Второе правило | $x \in S, y \in O,$ $(x, y, w) \in E \cup F,$ | $F' = F \cup \{(y, x, r), (x, y, w)\}$ |
| $pass(x, y, z)$ | $x \in S; y, z \in O; z \neq y$ $\{(x, y, w), (x, z, r)\} \subset E \cup F$ | $F' = F \cup \{(y, z, r), (z, y, w)\}$ |
| $post(x, y, z)$ | $x, y \in S; z \in O; x \neq y$ $\{(x, z, r), (y, z, w)\} \subset E \cup F$ | $F' = F \cup \{(x, y, r), (y, x, w)\}$ |
| $find(x, y, z)$ | $x, y \in S; z \in O; x \neq z$ $\{(x, y, w), (y, z, w)\} \subset E \cup F$ | $F' = F \cup \{(x, z, w), (z, x, r)\}$ |
| $spy(x, y, z)$ | $x, y \in S; x \neq z$ $\{(x, y, r), (y, z, r)\} \subset E \cup F$ | $F' = F \cup \{(x, z, r), (z, x, w)\}$ |

### 1.3 Постановка проблемы

Теперь, когда есть понимание правил в модели, можно изучить практический пример. Рассмотрим граф на Рис. 5 и ответим на вопрос: есть ли у субъекта $y$ право доступа $read$ к субъекту $z$ ?

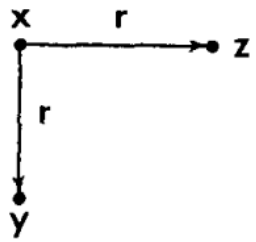



Рис. 5: Начальный граф доступов

Очевидный ответ – нет, так как нет ребра с правом чтения из $y$ до $z$ . Но на самом деле мы спрашиваем: существует ли последовательность правил, которая преобразует систему к графу с ребром чтения от $y$ до $z$ ? Отойдем от примера и поставим более общий вопрос, может ли субъект $p$ применять право $\alpha$ к объекту $q$ , или иначе существует ли последовательность правил, примененных к изначальному графу такая, что приводит к графу с присутствующей дугой с правом доступа $\alpha$ от $p$ к $q$ . Возвращаясь к изначальному вопросу, рассмотрим последовательность правил, изображенных на Рис. 6.

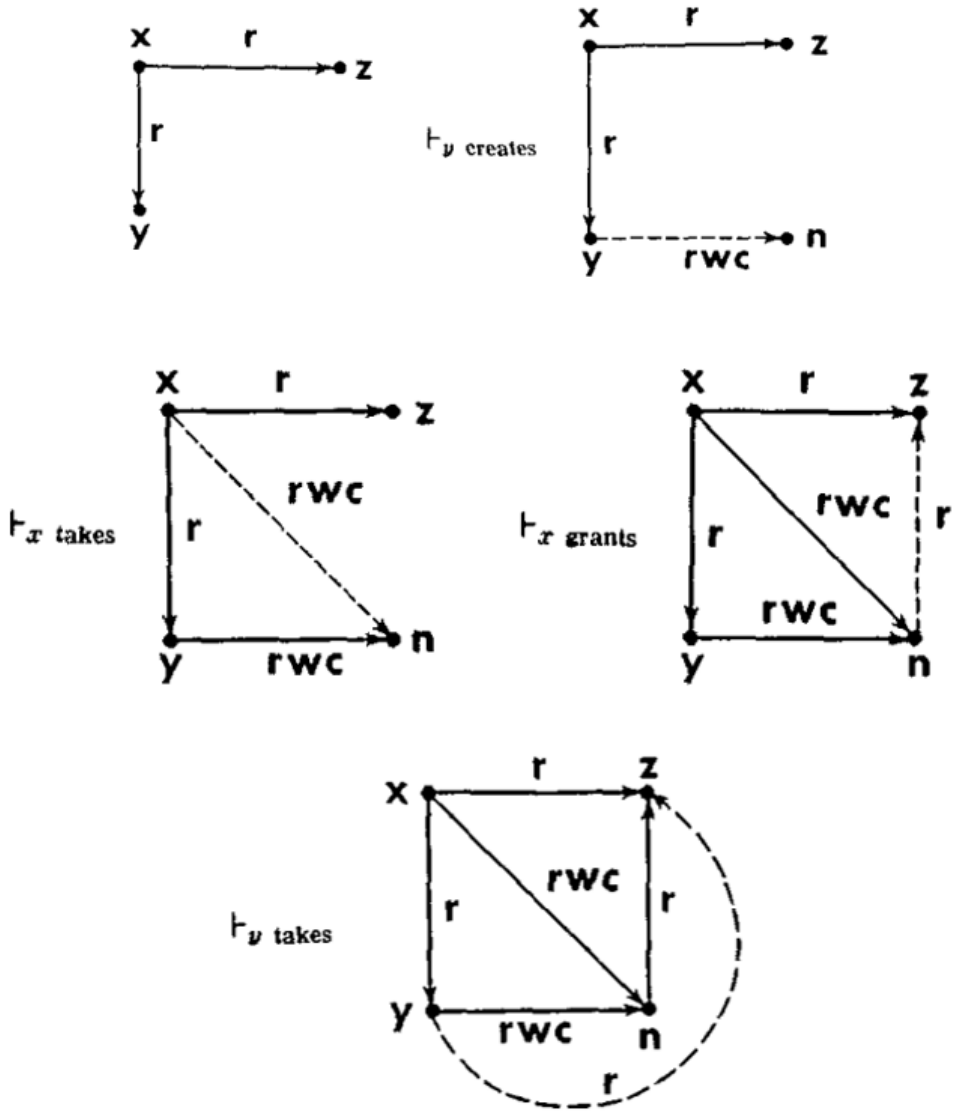



Рис. 6: Последовательность правил,
которая приводит к появлению ребра $(y, z, r)$ [9]

Тогда видим, что у субъекта $y$ право доступа $read$ к субъекту $z$ на самом деле существует. Этот пример демонстрирует тип проблем, с которыми работает модель take-grant:

- Какая информация объекта может быть доступна другим?
- К какой информации объектов другой субъект может получить доступ?

Так же известно, что в графовой модели с произвольным набором правил (по существу подобных $take, grant$ и т. д.) невозможно ответить на общим образом поставленный вопрос: может ли субъект $p$ применять право $\alpha$ к объекту $q$ [9]? Поэтому рассматривать имеет смысл лишь системы с фиксированным множеством правил, определенных в классической и расширенной моделях take-grant [9].

## 1.4 Анализ безопасности

Анализ безопасности модели проводится с целью ответить на вопросы, обозначенные в предыдущей главе. Алгоритм, решающий конкретную проблему безопасности, принято обозначать определенным предикатом. Основным является предикат $can\_share$, который формально определяет понятие передачи полномочий в классической модели take-grant и вводится следующим образом [10]:

- **Определение.** Предикат $can\_share(\alpha, x, y, G_0)$ истинен для права $\alpha$ и двух объектов $x$ и $y$ тогда, и только тогда, когда существуют графы доступов $G_1, \dots, G_n$ такие, что $G_0 \vdash^* G_n$ (символ $*$ означает многократное, в том числе нулевое, повторение) используя только де-юре правила, и в $G_n$ есть ребро от $x$ до $y$ с правом $\alpha$.

Анализ истинности предиката $can\_share$ непосредственно по определению является в общем случае алгоритмически неразрешимой задачей, так как

требует проверки всех траекторий функционирования системы [5]. По этой причине для проверки истинности предиката $can\_share$ вводятся необходимые и достаточные условия, при которых проверка возможна.

Введем понятие $tg$-связных вершин:

- **Определение.** $tg$-путь – это непустая последовательность $v_0, \dots, v_k$ различных вершин графа доступов таких, что для всех $i$: $0 \leq i < k$, $v_i$ соединяется с $v_{i+1}$ ребром с правом $t$ или $g$.
- **Определение.** Вершины $tg$-связные, если между ними существует $tg$-путь.

Приведем способ проверки предиката $can\_share$ для графов, все вершины которых являются субъектами, предложенный в работе [9]:

- Пусть $G = (S, S, \text{E})$ – граф доступов, содержащий только вершины субъекты, $\text{x}, \text{y} \in S, \text{x} \neq \text{y}$. Тогда предикат $can\_share(\alpha, \text{x}, \text{y}, G)$ истинен тогда и только тогда, когда выполняются условия 1 и 2.
  - Условие 1. Существует субъект $s \in S$, такой что $(s, \text{y}, \alpha) \in G$.
  - Условие 2. Субъекты x и $y$ tg-связны в графе $G$.

Важно отметить, что в данном случае существует алгоритм проверки предиката с линейной от размера графа доступов сложностью по времени [9].

Вспомогательные понятия:

- **Определение.** Остров — это максимальный $tg$-связный субъектный подграф.
- **Определение.** Начальным пролетом моста в графе доступов называется $tg$-путь, началом которого является вершина субъект, концом — объект, проходящий через вершины объекты, словарная запись которого имеет вид $\vec{t}^* \vec{g}$.

- **Определение.** Конечным пролетом моста в графе доступов G называется $tg$-путь, началом которого является вершина субъект, концом — объект, проходящий через вершины объекты, словарная запись которого имеет вид $\vec{t}^*$.
- **Определение.** Мостом в графе доступов называется $tg$-путь, концами которого являются вершины субъекты, проходящий через вершины объекты, словарная запись которого имеет следующий вид $\vec{t}^*, \overleftarrow{t}^*, \vec{t}^* \vec{g}\ \overleftarrow{t}^*, \vec{t}^* \overleftarrow{g}\ \overleftarrow{t}^*$.

Способ проверки предиката $can\_share$ для произвольных графов задается следующим образом [10]:

- Пусть $G = (S, O, E)$ – произвольный граф доступов, $x, y \in O,\ x \neq y$. Предикат $can\_share(\alpha, \text{x}, \text{y}, G)$ истинен тогда и только тогда, когда или $(x, y, \alpha) \subset E$, или выполняются условия 1–4.
  - Условие 1. Существует вершина $s \in G$: есть ребро от $s$ до $y$ с правом $\alpha$;
  - Условие 2. Существует субъект $x'$: $x = x'$ или $x'$ соединен начальным пролетом моста с $x$;
  - Условие 3. Существует субъект $s'$: $s = s'$ или $s'$ соединен конечным пролетом моста с $s$;
  - Условие 4. Существуют острова $I_1, \ldots, I_n$: $x'$ принадлежит $I_1$, $s'$ принадлежит $I_n$, и существует мост от $I_j$ до $I_{j+1}$ $(1 \leq j < n)$.

Наглядно можно видеть структуру графа доступов, при которой выполняется предикат $can\_share$ на Рис. 7:

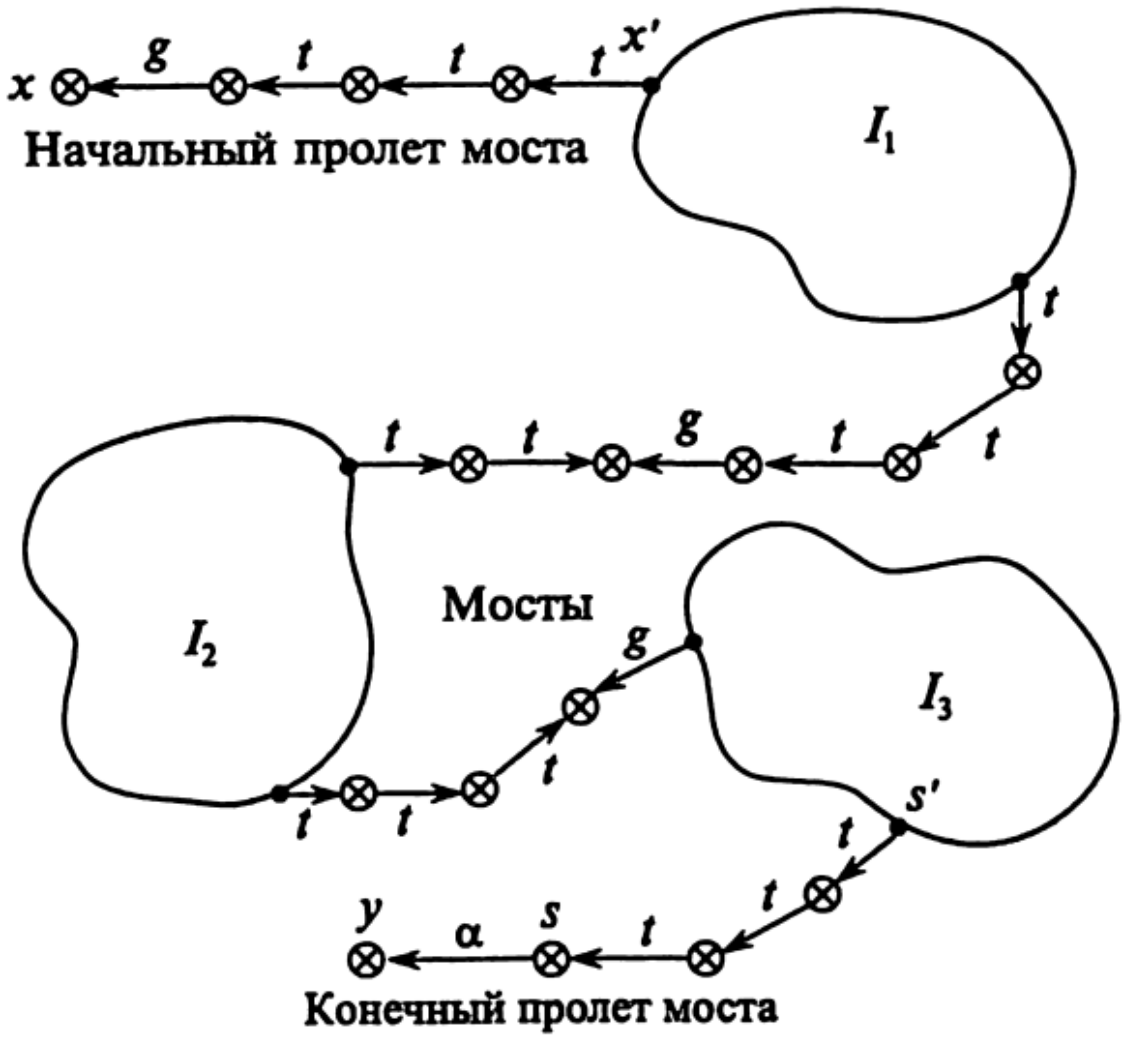



Рис. 7: Модель графа доступа с истинным предикатом $can\_share$ [5]

Если же право доступа может быть передано без участия какой-либо вершины, которая изначально имеет это право, то тогда говорят, что происходит похищение прав доступа. Обозначается эта операция предикатом $can\_steal(\alpha, x, y, G_0)$, который истинен тогда и только тогда, когда не существует ребра с правом $\alpha$ от $x$ до $y$ в графе $G_0$; существуют графы доступов $G_1, \ldots, G_n$, такие что $G_0 \vdash^* G_n$, используя только де-юре правила; в графе $G_n$ есть ребро от $x$ до $y$ с правом $\alpha$; и если есть ребро, с правом $\alpha$ от $s$ до $q$ в $G_0$, тогда не существует такого, что $s$ $grants$ α (для $q$) для любого $z \in G_j$, $(1 \leq j < n)$.

Из формулировки критерия истинности предиката $can\_steal$ можно заметить, что он истинен, только если истинен предикат $can\_share$ [10], плюс дополнительные условия. То же самое верно и для анализа информационных потоков, которые существуют в рамках расширенной модели take-grant — проверка истинности всех предикатов, включает в себя как часть проверку на истинность предиката $can\_share$ в том или ином виде [5]. Поэтому в практической части будут рассмотрены возможности эффективной программной реализации именно предиката $can\_share$, его анализ и тестирование.

# 2 Практическая часть

## 2.1 Выбор средств реализации

Для написания программного кода практической реализации алгоритма проверки предиката $can\_share$ был выбран язык Python. Выбор данного языка программирования был осуществлен исходя из нескольких причин. Во-первых, он является языком программирования высокого уровня, который обладает простым и понятным синтаксисом, что позволяет сосредоточиться именно на алгоритмической составляющей решаемой задачи и ускорить процесс написания кода. Во-вторых, Python предоставляет легкий для понимания и использования интерфейс для работы с параллельным и многопоточным программированием, а для реализации эффективного алгоритма эти техники по возможности должны быть использованы. В-третьих, этот язык имеет огромную экосистему библиотек, которые позволяют быстро реализовывать различные задачи, и так как в данном случае необходимо работать с графами, будет большим плюсом наличие уже готовых средств для эффективного хранения структуры данных графа и методов доступа к этим данным.

В качестве основной библиотеки для работы с графами был выбран пакет NetworkX (распространяется под 3-clause BSD license, которая является открытой и свободной для некоммерческого использования) предназначенный для создания, манипуляции и изучения сетевых структур. Основные возможности библиотеки [11]:

- Существуют классы для работы с простыми, ориентированными и взвешенными графами;
- В качестве узлов графа может использоваться практически любой объект, в том числе текст и изображение;

- В библиотеке реализовано большое количество алгоритмов работы с графами, таких как поиск циклов, проверка связности вершин, нахождение кратчайшего пути и других;
- Библиотека также позволяет удобно получать различные характеристики сетевых структур, например, степени вершин, длины путей, размеры графов;
- Имеются широкие возможности для визуализации графов.

Для текущей задачи наиболее значимыми являются возможности:

- Использовать ориентированные мультиграфы, которыми представлены графы доступов в модели take-grant [5];
- Наличие встроенной визуализации графов, так как это позволяет наглядно отслеживать структуру при написании тестов для алгоритма и контроля результатов его выполнения;
- Реализованные алгоритмы эффективного доступа к данным, получения свойств графа, так как эти операции очень часто используются при проверке предиката $can\_share$. А наличие большого числа других алгоритмов, например, поиска и т. п., является большим плюсом в случае оптимизации или дополнения текущего решения реализациями других предикатов.

Согласно заявлению разработчиков, библиотека NetworkX обладает высокой производительностью и может легко оперировать сетевыми структурами большого размера, достигающими уровня графа с 10 миллионами узлов и 100 миллионами дуг [11]. Такая производительность обусловлена тем, что графы в NetworkX базируются на низкоуровневой структуре данных языка Python, известной как "словарь-словарей" [11]. В этой структуре, каждый узел графа является ключом внешнего словаря, а его значениями являются внутренние словари, содержащие узлы, связанные с этим узлом и атрибуты для каждого ребра. Эта структура данных

обеспечивает быстрый доступ к соседним узлам и их атрибутам, что делает ее удобной для работы с большими графами и память расходуется эффективно [11].

## 2.2 Программная реализация алгоритма

Задача по написанию программного кода для решения задачи анализа безопасности модели take-grant реализована, как отдельный проект. Репозиторий с исходным кодом размещен на платформе GitHub (URL: https://github.com/lifantev/take-grant) под лицензией MIT (открытая и свободная для некоммерческого использования).

Как уже было описано в главе 1.4 Анализ безопасности, алгоритм проверки истинности предиката $can\_share$ был реализован через критерий сформулированный в работе [10].

Так как предикат $can\_share$ включает в себя совместное выполнение нескольких условий, основная функция программной реализации (в блоке исходного кода ИК. 1) состоит из последовательной проверки этих условий.

ИК. 1: Функция проверки истинности предиката $can\_share$

```
def can_share(graph: nx.MultiDiGraph, a: str, x: str, y: str) ->
bool | None:
    if x == y:
        return None
    # condition 0
    if x_y_a_edge_exist(graph, a, x, y):
        log.info('[can_share:condition #0] true')
        return True
    # condition 1
    s_ids = s_y_a_nodes(graph, a, y)
    log.info('[can_share:condition #1] s_ids=%s', s_ids)
```

```
    if not s_ids:
        return False
    # condition 2
    xi_ids = initially_spans(graph, x)
    log.info('[can_share:condition #2] xi_ids=%s', xi_ids)
    if not xi_ids:
        return False
    # condition 3
    si_ids = terminally_spans(graph, s_ids)
    log.info('[can_share:condition #3] si_ids=%s', si_ids)
    if not si_ids:
        return False

    # condition 4
    undirected_graph_view = graph.to_undirected(as_view=True)
    args = ((graph, undirected_graph_view, xi_si)
            for xi_si in product(xi_ids, si_ids))
    with mp.Pool() as pool:
        for res in pool.imap_unordered(island_bridge_paths_exist,
args):
            if res:
                return True
    return False
```

В условии 0 происходит проверка существует ли в графе доступов $(x, y, a) \subset E$. Алгоритм проверки следующий (блок ИК. 2): получаются все ребра от вершины $x$ к вершине $y$; осуществляется проход по этим ребрам пока не найдется любое с правом доступа $a$. Если такое ребро найдено, то условие истинно, иначе ложно. Так же нужно заметить, что истинность этого условия сразу гарантирует истинность предиката $can\_share$.

ИК. 2: Функция проверки условия 0

```
def x_y_a_edge_exist(graph: nx.MultiDiGraph, a: str, x: str, y: str)
-> bool:
    x_y_edge = graph.get_edge_data(x, y)
```

```
    if x_y_edge is not None and any(edge_data[EDGE_TYPE] == a for
edge_data in x_y_edge.values()):
        return True
    return False
```

В условии 1 происходит поиск вершин $s$, для которых есть ребро от $s$ до $y$ с правом $a$. Алгоритм поиска следующий (блок ИК. 3): получаются все ребра входящие в вершину $y$; осуществляется проход по этим ребрам; и исходящие вершины тех ребер из полученных, которые помечены правом доступа $a$, добавляются в список подходящих под условие вершин $s$. Если ни одна такая вершина $s$ не находится, предикат $can\_share$ сразу становится ложным.

ИК. 3: Функция поиска вершин $s$ в условии 1

```
def s_y_a_nodes(graph: nx.MultiDiGraph, a: str, y: str) -> set[str]:
    s_ids = set()
    for src, _, d in graph.in_edges(nbunch=y, data=True):
        if d[EDGE_TYPE] == a:
            s_ids.add(src)
    return s_ids
```

В условии 2 происходит поиск субъектов $x'$: $x = x'$ или $x'$ соединен начальным пролетом моста с $x$. Алгоритм поиска следующий (блок ИК. 4): если вершина $x$ является субъектом, то она добавляется в список подходящих субъектов $x'$; для того чтобы найти субъекты соединенные с $x$ начальным пролетом моста, сначала для вершины $x$ получаются все входящие в неё ребра и проходом по ним формируется список соседей $x$, ребро от которых помечено правом $grant$; дальше от этих соседей, используя модифицированный под поиск вершин, связанных пролетами мостов, алгоритм обхода графа в глубину (блок ИК. 5), идет поиск субъектов $x'$, которые соединены с ними путём со словарной записью $\vec{t}^*$.

Таким образом будет получен желаемый список вершин $x'$. Если он оказывается пустым, предикат $can\_share$ сразу становится ложным.

ИК. 4: Функция поиска вершин $x'$ в условии 2

```
def initially_spans(graph: nx.MultiDiGraph, x: str) -> set[str]:
    x_ids = set()
    # add x == x'
    x_node = graph.nodes.get(x)
    if x_node is not None and x_node[NODE_TYPE] == SUBJECT:
        x_ids.add(x)

    # if x' initially spans to x
    # get nodes that grant to x
    g_to_x = {src for src, _, d in graph.in_edges(
        nbunch=x, data=True) if d[EDGE_TYPE] == GRANT}

    # t->* paths from subjects to grant nodes
    visited = set()
    to_visit = [(v, x) for v in g_to_x]
    dfs_for_spans(graph, x_ids, visited, to_visit)
    return x_ids
```

ИК. 5: Модифицированный под поиск вершин, связанных пролетами мостов, алгоритм обхода графа в глубину

```
def dfs_for_spans(
        graph: nx.MultiDiGraph, ids: set[str],
        visited: set[str], to_visit: list[tuple[str, str]]):
    while to_visit:
        v, parent = to_visit.pop()
        if v not in visited:
            visited.add(v)
            if graph.nodes[v][NODE_TYPE] == SUBJECT:
                ids.add(v)
            for src, _, d in graph.in_edges(nbunch=v, data=True):
                if src != parent and d[EDGE_TYPE] == TAKE:
                    to_visit.append((src, v))
```

В условии 3 происходит поиск субъектов $s'$: $s = s'$ или $s'$ соединен конечным пролетом моста с $s$. Алгоритм поиска следующий (блок ИК. 6): осуществляется проход по найденным вершинам $s$ из условия 1; если вершина $s$ является субъектом, то она добавляется в список подходящих субъектов $s'$; дальше, используя уже упомянутый модифицированный под поиск вершин, связанных пролетами мостов, алгоритм обхода графа в глубину (блок ИК. 5), идет поиск субъектов $s'$, которые соединены с $s$ путём со словарной записью $\vec{t}^*$. Таким образом будет получен желаемый список вершин $s'$. Если он оказывается пустым, предикат $can\_share$ сразу становится ложным.

ИК. 6: Функция поиска вершин $s'$ в условии 3

```
def terminally_spans(graph: nx.MultiDiGraph, s_ids: set[str]) ->
set[str]:
    si_ids = set()
    # add s == s', get nodes that take to s
    to_visit = []
    for s in s_ids:
        s_node = graph.nodes.get(s)
        if s_node is not None and s_node[NODE_TYPE] == SUBJECT:
            si_ids.add(s)
        for src, _, d in graph.in_edges(nbunch=s, data=True):
            if d[EDGE_TYPE] == TAKE:
                to_visit.append((src, s))

    # t->* paths from subjects to s nodes
    visited = set()
    dfs_for_spans(graph, si_ids, visited, to_visit)
    return si_ids
```

В условии 4 происходит проверка, что между найденными вершинами $x'$ и $s'$ существует путь, состоящий из мостов и островов. Алгоритм проверки следующий: для каждой пары вершин $\{x', s'\}$ параллельно, используя

Python библиотеку multiprocessing, запускается функция по проверке наличия удовлетворяющего условию пути; эта функция представляет из себя модифицированный обход графа в глубину для поиска путей между вершинами, реализованный по описанию из работы [12] и дополненный ранним отсечением путей, не удовлетворяющих условию 4. Когда хотя бы для одной пары $\{x', s'\}$ находится верный путь, все параллельные процессы для других пар вершин завершаются, и результат условия 4 считается истинным, а вместе с ним истинным становится сам предикат $can\_share$. Если же ни для одной из пар вершин $\{x', s'\}$, удовлетворяющий условию 4, путь не находится, то предикат $can\_share$ считается ложным.

## 2.3 Тестирование

Для тестирования итоговой программной реализации использовался следующий подход:

- Для каждого из условий участвующих в проверке предиката $can\_share$ был написан собственный тест, подтверждающий корректность работы алгоритма по обработке этого условия;
- Каждый тест в свою очередь проверяется на нескольких, специально подготовленных для тестов, графах доступов модели take-grant с различными параметрами на вход функции по проверке конкретного условия.
- Тестовые графы и входные параметры для функций проверки условий подготавливались с целью проверить соответствующие алгоритмы как на правильность общего хода выполнения, так и на корректную работу в граничных случаях. Например, для проверки условия 3 использовался граф на Рис. 8, с целью убедиться, что конечным пролетам мостов в алгоритме действительно соответствуют пути

лишь со словарной записью $\vec{t}^{*}$, и что никакие другие вершины кроме субъектов не добавляются в список искомых субъектов $s'$.

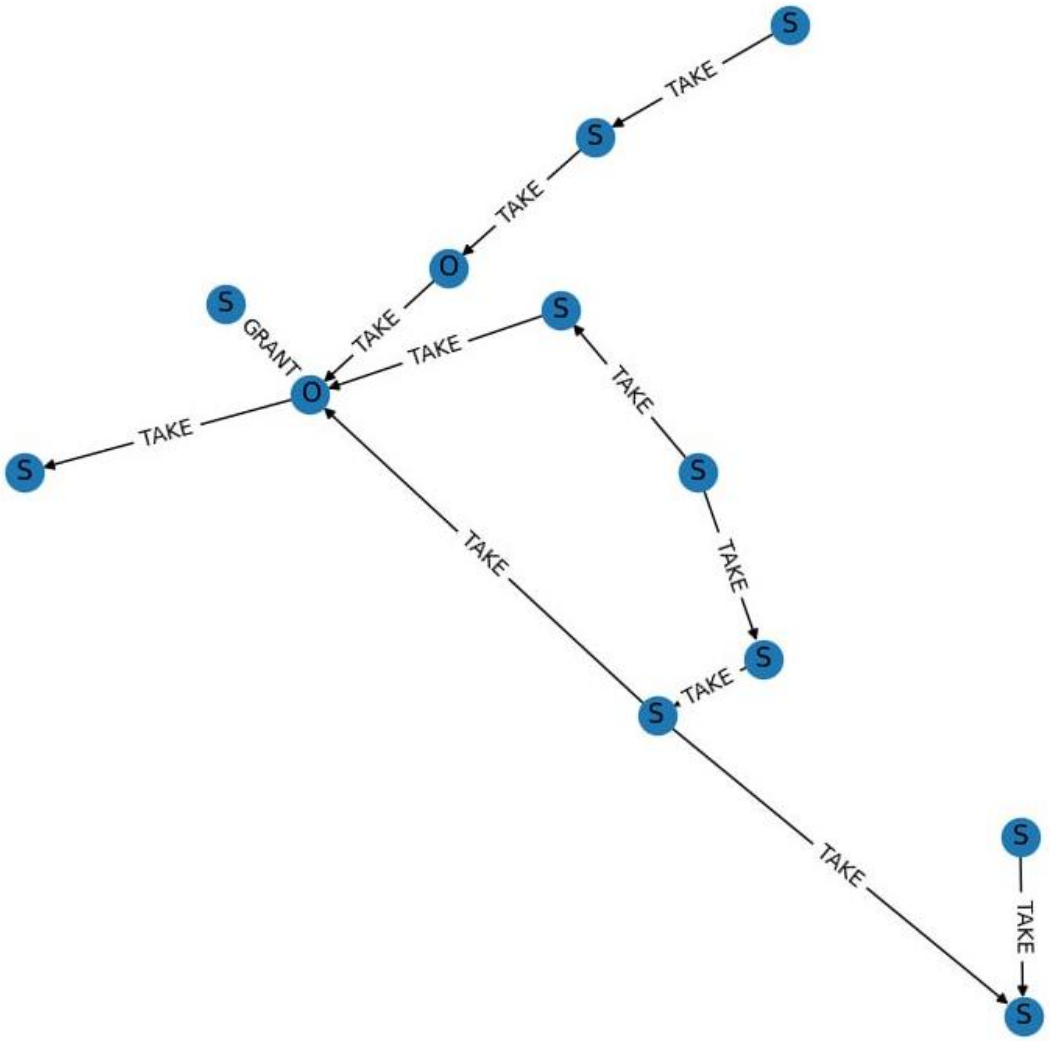



Рис. 8: Тестовый граф доступов для проверки условия 3 (визуализирован библиотекой NetworkX)

Общее количество написанных тест-кейсов составляет немногим больше 40. Используя Python утилиту coverage, предназначенную для измерения покрытия кода во время выполнения, можно видеть (Рис. 9), что удалось добиться покрытия тестами исходного программного кода, участвующего в проверке условий предиката $can\_share$, на 89%.

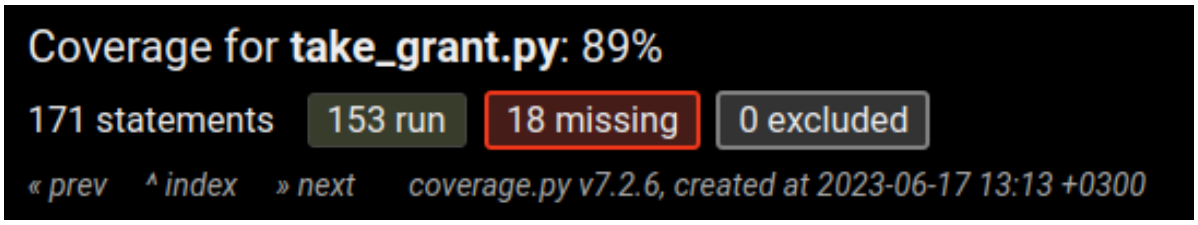
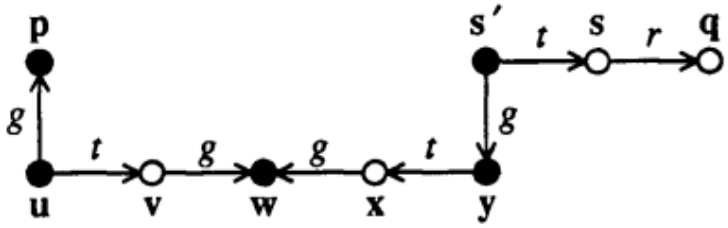



Рис. 9: Тестовое покрытие программного кода, участвующего в проверке условий предиката $can\_share$

Так же корректность работы алгоритма проверена на стандартных графах доступов, часто встречающихся в статьях по модели take-grant, для которых известен результат выполнения предиката. Получили следующее:

- Пример 1, из статьи [10]. Изображение графа доступов на Рис. 10.




| | |
|---|---|
| islands: | $I_1$={p,u},$I_2$={w},$I_3$={y,s′} |
| bridges: | u,v,w and w,x,y |
| initial span: | p with associated word: v |
| terminal span: | s′,s with associated word: $\vec{t}$ |

Рис. 10: Граф доступов $G$ для примера 1

Для такого графа предикат $can\_share(r, p, q, G)$ является истинным [10]. Результат работы алгоритма для примера 1 с этим согласуется.

- Пример 2, из статьи [6]. Изображение графа доступов на Рис. 11.

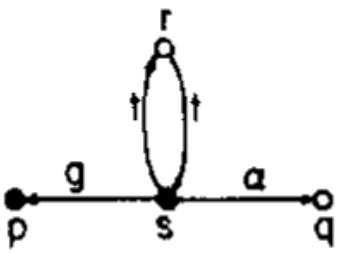



Рис. 11: Граф доступов $G$ для примера 2

Для такого графа предикат $can\_share(a, p, q, G)$ является истинным [6]. Результат работы алгоритма для примера 2 с этим согласуется.

- Пример 3, из статьи [6]. Изображение графа доступов на Рис. 12.

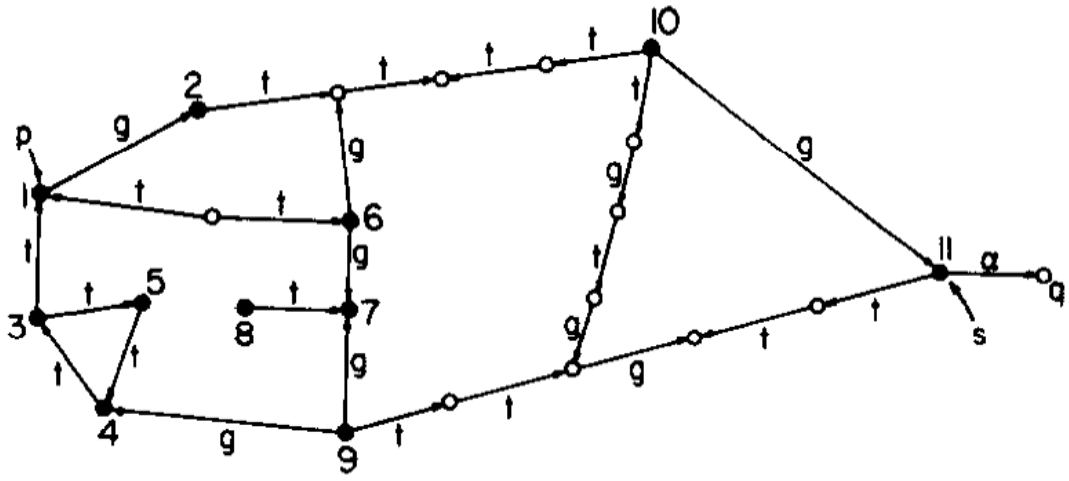



Рис. 12: Граф доступов $G$ для примера 3

Для такого графа предикат $can\_share(a, p, q, G)$ является истинным [6]. Результат работы алгоритма для примера 3 с этим согласуется.

## 2.4 Анализ эффективности алгоритма

Проведем оценку вычислительной сложности программной реализации предиката $can\_share$ в модели take-grant по времени. Для этого последовательно дадим оценку сложности всех алгоритмов, используемых в полученном решении, описанном в главе 2.2 Программная реализация алгоритма.

В условии 0 получение ребер между вершинами выполняется за $O(1)$, как операция доступа к элементу словаря. Затем начинается проход по полученным ребрам, который в худшем случае будет иметь сложность $O(N_1)$, где $N_1$ – число ребер между вершинами $x$ и $y$. Итоговая сложность проверки условия 0 лучшем случае будет $O(1)$, в худшем $O(N_1)$.

В условии 1 получаются все ребра, входящие в вершину $y$. Это действие так же выполняется за $O(1)$, как операция доступа к элементу словаря. Но так как в любом случае происходит проход по всем полученным ребрам, итоговая сложность будет $O(N_2)$, где $N_2$ – число ребер, входящих в вершину $y$.

В условии 2 получение вершин, от которых есть ребро с правом $grant$ в сторону $x$, аналогично условию 1 имеет сложность $O(N_3)$, где $N_3$ – число ребер, входящих в вершину $x$. А модифицированный под поиск вершин, связанных пролетами мостов, алгоритм обхода графа в глубину, который выполняется следом имеет сложность $O(V + E)$, где $V$ – число вершин графа, $E$ – число ребер [13]. Поэтому итоговая сложность проверки условия 2 будет $O(V + E)$.

В условии 3 вначале происходит проход по всем найденным вершинам $s$, и для каждой из них вложенный проход по входящим в неё ребрам. Сложность этой операции будет равна $O(N_4 * N_5)$, где $N_4$ – число вершин $s$, $N_5$ – число ребер, входящих в вершину $s$. Далее в этом условии так же выполняется модифицированный под поиск вершин, связанных пролетами мостов, алгоритм обхода графа в глубину, со сложность $O(V + E)$. Очевидно, что почти для любого графа верно соотношение $V + E \gg N_4 * N_5$ , поэтому можно считать, что итоговая временная сложность условия 3 будет $O(V + E)$.

Ещё раз заметим, что при проверке условия 4, каждая пара вершин $\{x', s'\}$ обрабатывается параллельно. Это позволяет заметно снизить время на анализ этого условия. Без использования параллелизма, очевидно, что итоговая сложность проверки условия в худшем случае (когда нужно проверить все пары, прежде чем удовлетворяющий условию путь будет найден или не найден вовсе) умножалась бы на $N_6$ – число найденных пар $\{x', s'\}$. Сам алгоритм представляет из себя модифицированный обход графа

в глубину для поиска путей между вершинами, в котором один путь может быть найден за время $O(V + E)$ [12]. Дополнение же его ранним отсечением путей, не удовлетворяющих связности остров-мост, определяет намного более быстрое выполнение, чем поиск вообще всех путей между парой вершин без ограничений и последующий его анализ (сложность в таком случае была бы $O(n!)$, где $n$ – порядок графа). В данном варианте при попытке входа алгоритма на новую глубину графа, чтобы понять удовлетворяет ли путь условию 4, необходимо анализировать лишь несколько подряд идущих вершин, так как предыдущая часть пути была уже проанализирована на меньшей глубине обхода графа. Сложность подобных операции константная, поэтому итоговая сложность анализа условия 4 остается равна сложности модифицированного обхода графа в глубину $O(V + E)$.

Тогда после анализа всех условий, необходимых для истинности предиката $can\_share$, можно сделать вывод, что временная вычислительная сложность всего алгоритма будет равна $O(V + E)$, где $V$ – число вершин графа, $E$ – число ребер.

В работе [6] утверждается, что существуют алгоритмы, которые осуществляют проверку предиката $can\_share$ за линейное от $V + E$ время, как для субъектных графов доступа, так и для произвольных, поэтому приведенная реализация для общего случая может считаться оптимальной.

## 2.5 Проверка эффективности

Проверим реальную зависимость скорости работы алгоритма от размера графа, с целью подтвердить линейность временной вычислительной сложности, определенную из теоретического анализа алгоритма выше. Для проверки передачи полномочий используем графы доступов разных

размеров, построенных на основе графа на Рис. 12; на каждом графе многократно выполняем алгоритм проверки предиката $can\_share$ и подсчитываем суммарное время выполнения; используем его для вычисления среднего времени работы алгоритма и выявления зависимости от размера графа. Полученные результаты приведем в виде таблицы:

Табл. 3: Результаты тестирования алгоритма на эффективность

| Размер графа, $V + E$ | Число запусков | Суммарное время, с / Среднее время, мс |
|---|---|---|
| 152 | 1000 | 19.169 |
| 305 | 1000 | 22.102 |
| 611 | 1000 | 26.468 |
| 1223 | 1000 | 35.694 |
| 2447 | 1000 | 55.573 |
| 4385 | 1000 | 95.531 |

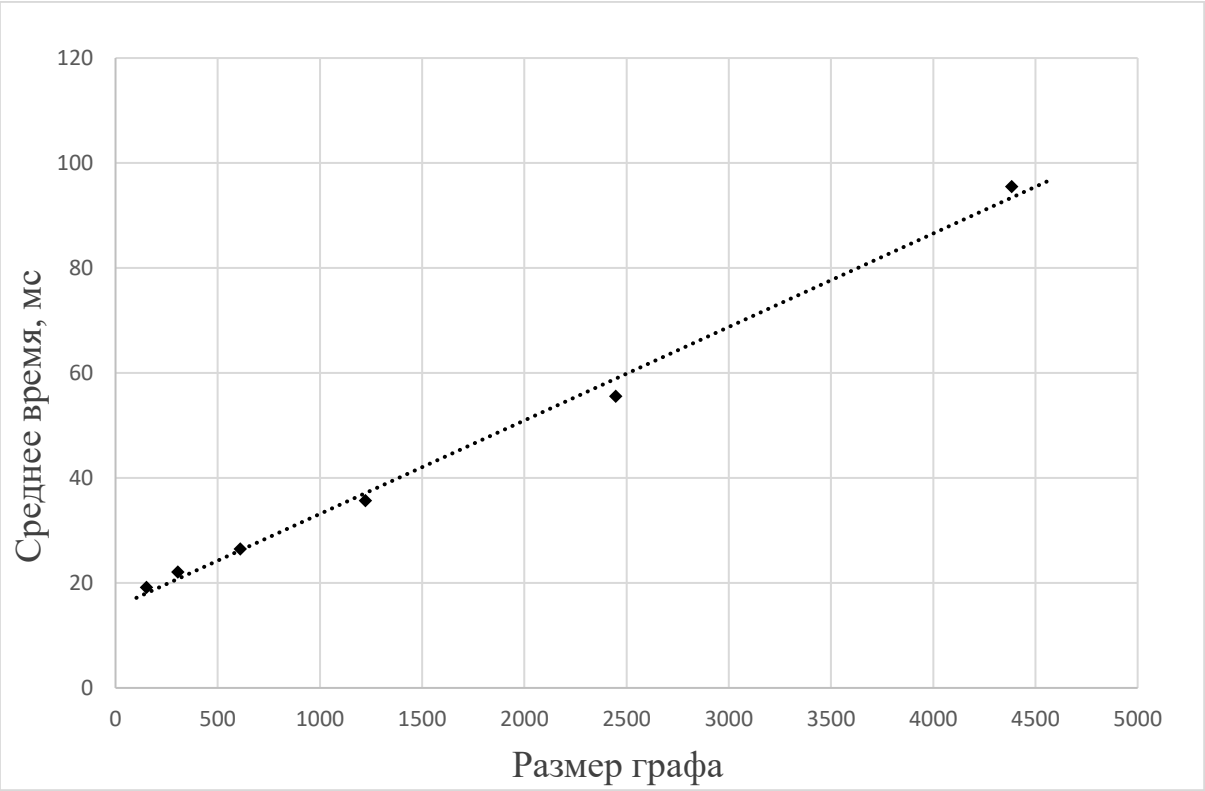



График 1: Зависимость времени работы алгоритма от размера графа

Из репрезентации результатов на Графике 1 видно, что действительно временная вычислительная сложность реализованного алгоритма линейно зависит от размера графа доступов.

Влияние распараллеливания условия 4 можно заметить уже на небольших графах. Например, для графа с $V + E$ равным $105$ и числом уникальных пар $\{x', s'\}$ равным $4$ без использования параллелизма среднее время выполнения составило $\sim 85$ мс, а при его наличии $\sim 23$ мс. Видно, что результат согласуется с утверждением “без использования параллелизма … итоговая сложность проверки условия в худшем … случае умножалась бы на $N_6$ – число найденных пар $\{x', s'\}$” из главы 2.4 Анализ эффективности алгоритма.

## Заключение

Цели и задачи, поставленные для работы выполнены полностью. В качестве практического результата получили алгоритм проверки истинности предиката $can\_share$ и реализовали его в программном коде на языке Python с использованием библиотеки NetworkX.

Произведено тестирование итогового решения, проанализирована и подтверждена его теоретическая эффективность в терминах временной вычислительной сложности. Удалось реализовать алгоритм с линейной сложностью от суммарного количества вершин и ребер в графе, что считается оптимальным решением данной задачи [6]. Также заметно уменьшили время работы программной реализации благодаря применению параллельных вычислений.

В ходе работы было подробно изучено большое количество статей по классической и расширенной модели безопасности take-grant; проанализированы критерии для анализа безопасности в этих моделях.

Полученная реализация позволяет в дальнейшем использовать исходный программный код в качестве базы для создания полноценного моделирующего комплекса.

## Список литературы


1. Вострецова, Е. В. Основы информационной безопасности: учебное пособие для студентов вузов. Екатеринбург: Изд-во Урал. ун-та, 2019. 204 с.
2. Зегжда, Д.П., Ивашко, А. М. Основы безопасности информационных систем. М.: Горячая линия – Телеком, 2000. 452 с.
3. ISO/IEC 15408-3:2013 Information technology – Security techniques – Evaluation criteria for IT security – Part 3: Security assurance requirements.
4. Harrison M., B., Ruzzo W., Ullman J. ESIGN: Protection in operating systems— 1976. — Август (т. 19, № 8). — С. 461–471. — ISSN 0001-0782.
5. Девянин П.Н. Модели безопасности компьютерных систем. Управление доступом и информационными потоками. Учебное пособие для вузов. 2-е изд., испр. и доп. М.: Горячая линия – Телеком, 2017. 338 с.
6. Snyder, L. "Theft and Conspiracy in the Take-Grant Protection Model". Journal of Computer and System Sciences, vol. 23, 1981, pp. 333-347.
7. Kuliamin, V., Khoroshilov, A., Medveded, D. "Formal Modeling of Multi-Level Security and Integrity Control Implemented with SELinux". 2019 Actual Problems of Systems and Software Engineering (APSSE), Moscow, Russia, 2019, pp. 131-136, doi: 10.1109/APSSE47353.2019.00024.
8. Kolomeets, M., Chechulin, A., Kotenko, I., Saenko, I. "Access Control Visualization Using Triangular Matrices". 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Pavia, Italy, 2019, pp. 348-355, doi: 10.1109/EMPDP.2019.8671578.
9. Lipton R.J., Snayder L. "A linear time algorithm for deciding subject security". Journal of ACM (Addison-Wesley), vol. 24, no. 3, 1977, pp. 455-464.
10. Bishop, M. "Applying the Take-Grant Protection Model". Computer Science Technical Report PCS-TR90-151, 1990.
11. Hagberg, A. A., Schult, D. A., Swart, P. J. "Exploring network structure, dynamics, and function using NetworkX". Proceedings of the 7th Python in

Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008.

12. Sedgewick, R. "Algorithms in C, Part 5: Graph Algorithms". Addison Wesley Professional, 3rd ed., 2001.
13. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. "Introduction to Algorithms", Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 22.3: Depth-first search, pp. 540–549.