МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование)

09.04.03 Прикладная информатика
(код и наименование направления подготовки, специальность)

Управление корпоративными информационными процессами
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему «Верификация авторизационных политик для анализа надежности системы контроля доступа»

Обучающийся В.А. Ячевский
(И.О. Фамилия) (личная подпись)

Научный руководитель к.п.н., доцент, О.А. Крайнова
(ученая степень, ученое звание, И.О. Фамилия)

Тольятти 2023

Оглавление

Введение

Системы разграничения доступа обеспечивают контроль за тем, к каким ресурсам в системе имеют доступ те или иные пользователи. Такие системы являются одним из наиболее важных компонентов безопасности. Политики разграничения доступа, в свою очередь, применяются для того, чтобы упростить управление и обслуживание таких систем. Некорректные политики, неправильная конфигурация или недостатки в реализации программного обеспечения могут привести к серьезным уязвимостям, которые напрямую влияют на надежность СРД. Зачастую конфиденциальность и безопасность системы нарушаются именно из-за некорректной конфигурации политик разграничения доступа, а не из-за сбоя криптографических примитивов или протоколов. Эта проблема становится все более серьезной по мере усложнения программных систем, которые развертываются для управления большим количеством конфиденциальной информации и ресурсов, организованных в сложные структуры. Верификация, то есть выявление несоответствий между спецификациями политик и их предполагаемым назначением имеет решающее значение, поскольку правильная реализация и применение политик приложениями основывается на предпосылке, что спецификации политик верны. В особенности данная проблема является актуальной для систем разграничения доступа на основе атрибутов – одной из наиболее современных моделей, и в то же время во многих отношениях находящейся на исследовательском уровне вследствие своей новизны. Очевидным способом верификации является тестирование политик, основанных на данной модели. С его помощью возможно обнаружить противоречия в логике конструирования политик для дальнейшего устранения данных противоречий.

Объектом исследования магистерской диссертации является система разграничения доступа на основе атрибутов.

Предметом исследования является тестирование политик разграничения доступа на основе атрибутов.

Цель работы – обоснование подхода к верификации политик через тестирование путём определения метрики, позволяющей количественно оценить надежность тестового покрытия.

Для достижения поставленной цели необходимо решить следующие задачи:

– рассмотреть конструктивные особенности системы разграничения доступа на основе атрибутов и выявить её недостатки.

– рассмотреть существующие методы тестирования политик разграничения доступа на основе атрибутов.

– сформулировать модель отказов и условия обнаружения ошибок в политиках разграничения доступа, а также определить подходящую для цели исследования метрику.

– разработать алгоритм генерации тестов, соответствующих сформулированным условиям обнаружения ошибок.

– в ходе эксперимента провести количественную оценку эффективности существующих методов тестирования.

В качестве гипотезы исследования выступает предположение, что анализ надежности системы разграничения доступа может осуществляться на основе оценки эффективности выбранного метода верификации политик разграничения доступа.

Для решения поставленных в исследовании задач предполагается использовать методологический аппарат, включающий в себя такие методы исследования, как сравнительный анализ, моделирование и постановка эксперимента.

Научная новизна исследования заключается в разработке алгоритма генерации тестов, основывающегося на модели отказов и условиях обнаружения ошибок для систем разграничения доступа на основе атрибутов.

Теоретическая значимость исследования определяется тем, что сформулированные в нём положения формируют основу для практического

применения анализа надежности системы разграничения доступа на основе оценки тестового покрытия.

Структура работы представлена введением, тремя главами, заключением и списком использованных источников.

Первая глава посвящена рассмотрению теоретических аспектов работы с системой разграничения доступа на основе атрибутов, а также выявлению её недостатков с точки зрения надежности системы безопасности в целом. Кроме того, проводится анализ существующих методов тестирования политик разграничения доступа на основе изучения научной литературы по теме исследования.

Во второй главе формируются основные положения работы, в частности, понятия модели неисправностей и условий обнаружения ошибок. На их основе, на примере нескольких операторов мутации формируются ограничения, формально описанные в форме логических выражений, которым должен удовлетворять тест для обнаружения ошибки. С помощью сформулированных ограничений описывается процесс генерации тестов на основе сильного мутационного анализа.

В третьей главе приводится описание произведенного эксперимента и анализ его результатов.

В заключении приводится обобщение результатов проведенного исследования.

Диссертационное исследование изложено на 73 страницах, содержит 6 рисунков, 14 таблиц и 3 приложения.

Глава 1 Теоретические аспекты системы разграничения доступа на основе атрибутов

1.1 Конструктивные особенности атрибутной СРД

Система разграничения доступа – это фундаментальный механизм безопасности, который служит для ограничения доступа к системе с виртуальными или физическими ресурсами на основе субъекта, запрашивающего доступ. В таких моделях разграничения доступа, как IBAC (управление доступом на основе идентичности) или RBAC (ролевое управление доступом), решение основывается в первую очередь на идентификации субъекта, где доступ к объекту будет предоставляться локально идентифицированному субъекту или локально определенным ролям, которые назначены субъекту. Квалификаторы субъекта, такие как идентичность и роли, далеко не всегда достаточны для выражения потребностей бизнеса, поскольку такие требования, как правило, имеют дело с условиями окружения, соответствующими запросу. Однако традиционные модели, к числу которых относятся IBAC и RBAC, не способны эффективно учитывать факторы, относящиеся к условиям окружения [12]. Условия окружения, в свою очередь, включают оперативный или ситуационный контекст, которые являются определяемыми характеристиками окружения, в котором выполняется запрос на предоставление доступа. К таким характеристикам могут относиться текущее время, день недели, местоположение пользователя – то есть параметры, которые не зависят от субъекта или объекта [27].

ABAC (атрибутная система разграничения доступа) появилась как новое поколение систем разграничения доступа для решения вышеописанной проблемы. Преимущество неопределенности отношений между субъектом (пользователем) и объектом (запрашиваемым ресурсом), на котором основывается данная модель, позволяет обеспечить доступ к большому количеству комбинаций субъект-объект. Поскольку в ABAC имеется

возможность создания политик доступа на основе существующих атрибутов пользователей и объектов, он сводит к минимуму необходимость ручного вмешательства при настройке и развертывании системы разграничения доступа. Описываемый механизм позволяет ABAC модели быстро реагировать на изменения в системе, что позволяет считать её подход решением проблем, стоявших перед традиционными системами разграничения доступа [13].

В системе разграничения доступа на основе атрибутов, субъект выполняет запрос на предоставление доступа к ресурсу (объекту). Затем механизм ABAC оценивает политики, атрибуты субъекта и объекта, а также условия окружения, и на основе вышеперечисленного формирует решение о предоставлении или непредоставлении доступа для конкретного запроса. В случае, если запрос был авторизован системой, субъект получает доступ к ресурсу. В этом заключается основная концепция ABAC, которая проиллюстрирована на рисунке 1:

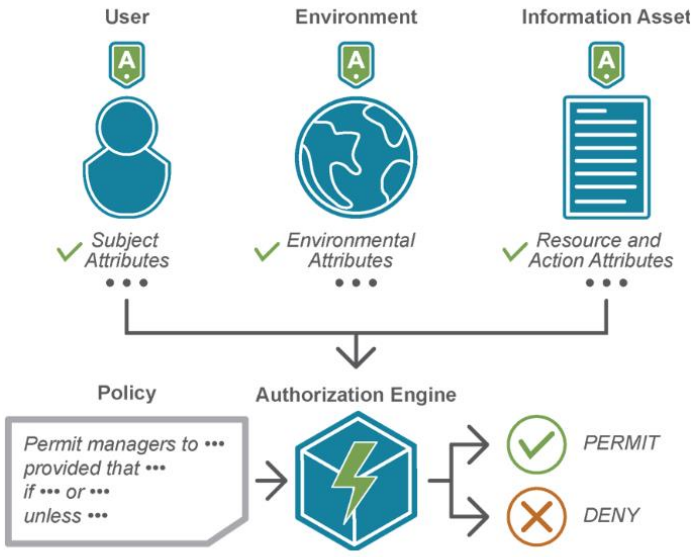

Рисунок 1 – Иллюстрация концепции системы разграничения доступа на основе атрибутов

Для описания политик разграничения доступа в модели ABAC наиболее распространенным на данный момент вариантом с открытым исходным кодом является XACML, стандарт, разработанный глобальным консорциумом OASIS, специфицирующий работу с ABAC-политиками. Он предоставляет и язык описания самих политик, и формализацию запросов и ответов при принятии решения на предоставление доступа. Язык политик используется для описания основных требований к разграничению доступа, и имеет точки расширения для введения новых функций, типов данных, комбинирования логики и так далее. В свою очередь, формализация запросов и ответов нужна для формулирования запроса на разрешение того или иного действия, а также интерпретации результата вычисления разрешения. Ответ состоит из решения о том, следует ли разрешать запрос или нет. К возможным значениям относятся следующие: «Permit» (разрешить), «Deny» (запретить), «Indeterminate» (невозможность принятия авторизационного решения) или «Not applicable» (система не может ответить на запрос).

Основным действием в ABAC системе является запрос пользователя (субъекта) на действие над каким-либо ресурсом (объектом). Субъект выполняет запрос в систему, который приходит в Policy Enforcement Point (PEP). Данный элемент системы предназначен для подготовки контекста операции по отношению к объекту путём сбора атрибутов, относящихся к пользователю. Это может быть и должность пользователя, и само действие, и условия среды, в которой выполняется запрос (адрес облачного окружения, на котором находится запрашиваемый объект). Все эти атрибуты собираются воедино и отправляются в Policy Decision Point (PDP), где определяется, какие из политик, содержащиеся в сервисе разграничения доступа применимы к текущему запросу. Здесь же происходит принятие решения о предоставлении доступа. В результате ответ от PDP возвращается в PEP, который и сообщает пользователю о принятом системой решении. На рисунке 2 представлена диаграмма потока данных в ABAC системе. Она, в том числе, содержит такой

элемент, как Policy Administration Point (PAP), который отвечает за управление и администрирование самих политик:

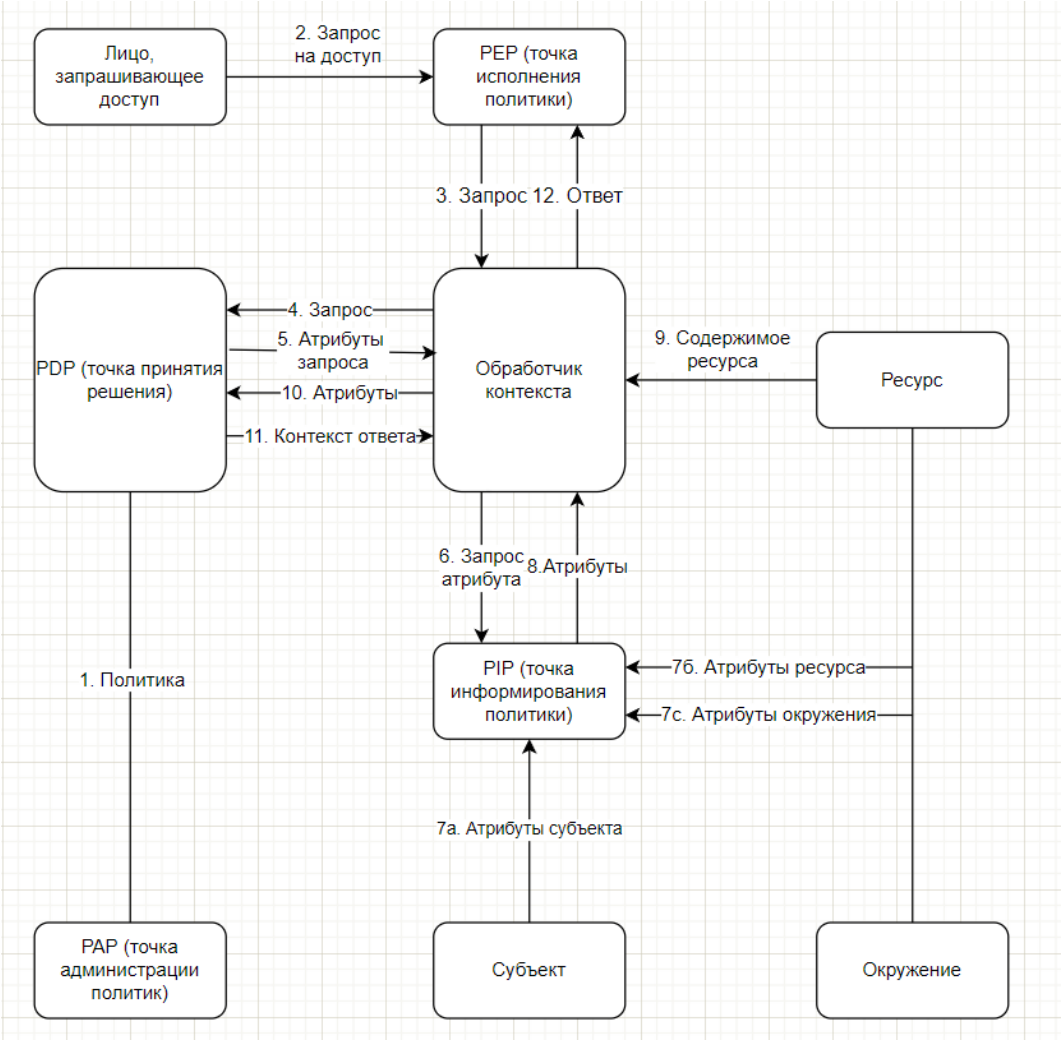

Рисунок 2 – Диаграмма потока данных в ABAC

Отдельно необходимо рассмотреть стандарт XACML 3.0, который определяет спецификацию политик разграничения доступа, поскольку на основе его аппарата предполагается дальнейшая работа над исследованием.

1.2. Рассмотрение стандарта XACML

XACML, как уже ранее было отмечено, является языком политик управления доступом общего назначения [25]. Корнем документа XACML являются такие элементы, как политика или набор политик. Элемент набора политик содержит другие дочерние элементы политики или элементы набора политик, каждый из которых оценивается PDP. Элемент политики, в свою очередь, содержит список элементов правил, которые также оцениваются PDP [18]. Кроме того, XACML описывает механизм для комбинации промежуточных вычислений, известный как комбинационный алгоритм. Комбинационный алгоритм, который согласовывает вычисления для правил внутри отдельно взятой политики, называется комбинационным алгоритмом правил. Аналогичным образом, комбинационный алгоритм, согласовывающий вычисления для политик или наборов политик, называется комбинационным алгоритмом политик.

Набор политик, политика и правило содержат элемент «target», или цель, определяющий набор запросов, к которым применяется тот или иной элемент. В дополнение, правило может содержать булевую функцию, известную как «condition», или условие, выполнение которой требуется для применения правила. На рисунке 3 представлена модель языка политик XACML:

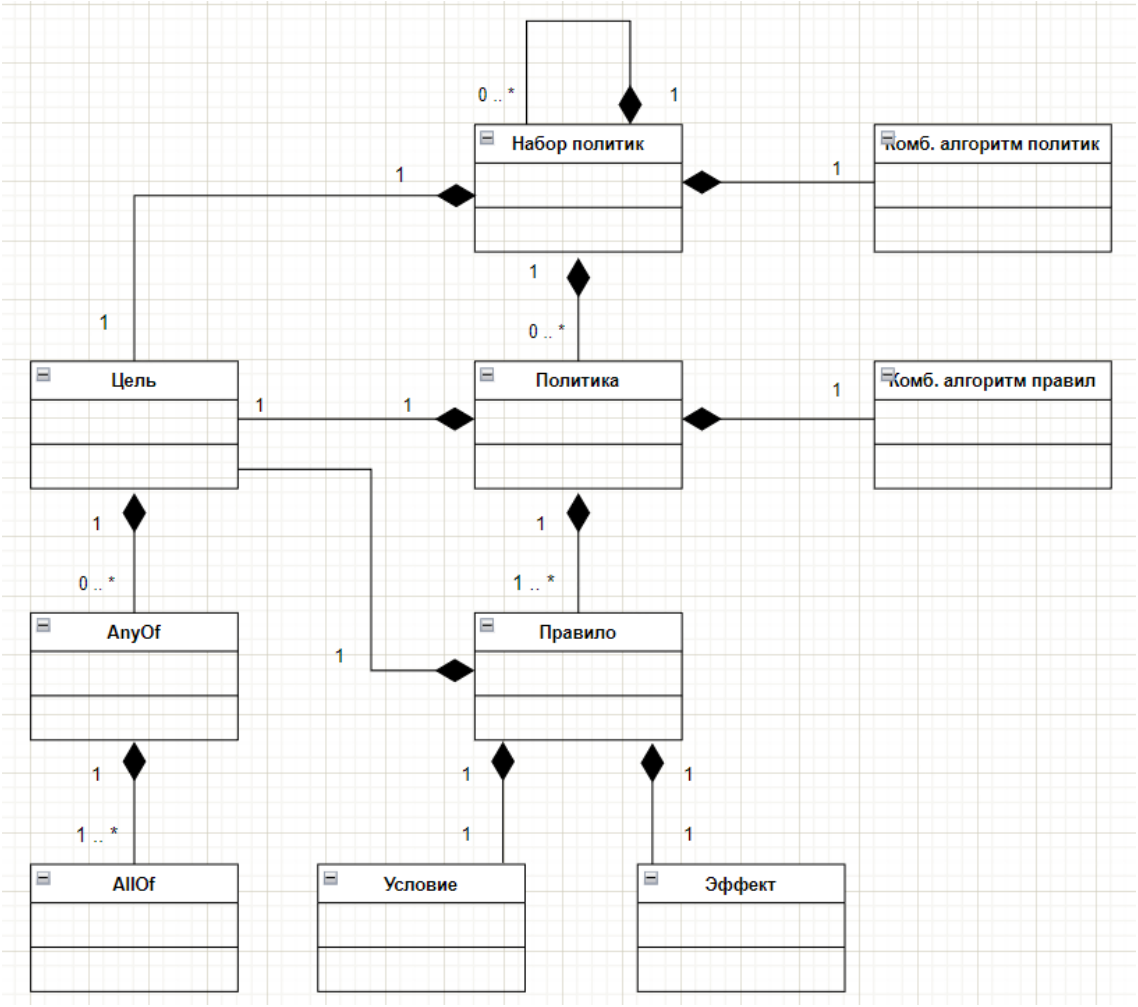

Рисунок 3 – Модель языка политик XACML

Исходя из представленной диаграммы, цели правил, политик и набора политик – это конъюнктивная последовательность элементов <AnyOf>. Каждый элемент <AnyOf>, в свою очередь, является дизъюнктивной последовательностью элементов <AllOf>, а каждый элемент <AllOf> является конъюнктивной последовательностью элементов <Match>. Элемент <Match> сопоставляет и сравнивает атрибуты в контексте запроса со встроенными значениями атрибутов. Логические выражения для условий правил определяются четырьмя категориями атрибутов – субъект, ресурс, действие и окружение. Они могут использовать большое разнообразие предопределенных функций и типов данных. Правила содержат обязательный

элемент эффекта, который определяет, какой результат PDP выдаст при успешном вычислении – «Permit» (разрешение) или «Deny» (запрет).

Формально, элемент набора политик PS является набором $< PST, PCA, [P_1, P_2, \dots, P_m], A, O >$, где PST – это цель набора политик, PCA – комбинационный алгоритм политик, и $[P_1, P_2, \dots, P_m]$ – список политик и/или вложенных наборов политик в рассматриваемом наборе политик. Каждое правило R_j представляет собой набор элементов $< rt_j, rc_j, re_j >$, где rt_j – это цель правила, rc_j – условие правила и $re_j \in \{Permit,\ Deny\}$ – эффект правила. Соответственно, правило называется разрешающим при $< rt_j, rc_j,\ Permit >$, в то время как $< rt_j, rc_j,\ Deny >$ называется запрещающим. В случае, если rt_j и rc_j в правиле опущены (по умолчанию являются true, будучи не заданными явно), то правило $< \_\_, \_\_, re_j >$ называется правилом по умолчанию. Более конкретно, $< \_\_, \_\_, Permit >$ является разрешающим правилом по умолчанию, $< \_\_, \_\_, Deny >$ - запрещающим правилом по умолчанию.

Для получения доступа к ресурсу, субъект отправляет системе запрос на предоставление доступа. Данный запрос содержит определенный набор атрибутов. На запрос по предоставлению доступа q, политика или набор политик отвечают решением о предоставлении доступа – положительным или отрицательным. Для данного запроса q, PS оценивается для получения ответа, что обозначается как $d(PS, q)$. Цель набора политик PST сначала оценивается в соответствии со значениями атрибутов в q. Если результатом оценки является false, тогда $d(PS, q) = N/A$, в противном случае выполняется оценка политик $P_1, P_2, \dots, P_m$ для случаев, когда цель набора политик оценивается как true или ошибка. Оценка набора политик $d(PS, q)$ зависит от комбинационного алгоритма политик PCA и результатов вычислений нижележащих политик с учётом q (обозначаемого как $d(Pi, q)$). Аналогично, для отдельно взятой политики $P_i = < PT_i,\ RCA_i, [r_1, r_2, \dots, r_n] >$, цель политики PT_i оценивается согласно значениям атрибутов в q. Если результат оценки равен false, тогда

$d(P_i, q) = N/A$, иначе выполняется оценка нижележащих правил $r_1, r_2, \dots, r_n$. Оценка набора правил $d(P_i, q)$ зависит от комбинационного алгоритма правил RCA_i и результатов вычислений отдельных правил. Решение правила $r_j = < rt_j, rc_j, re_j >$ с учётом q, обозначаемое как $d(r_j, q)$, определяется следующим образом:

– Permit: доступ разрешен, если $re_j = Permit$ и rtj с rtc равны true с учётом q;

– Deny: доступ запрещен, если rej = Deny и rtj с rtc равны true с учётом q;

– Not applicable, или N/A: запрос q не применим к рассматриваемому правилу, то есть при вычислении rtj и/или rcj равны false с учётом q;

– IndeterminateD, или I(D): возникновение ошибки при оценивании rtj или rcj, в то время как rej = Deny. Если бы ошибки не произошло, то доступ был бы запрещен. Синтаксически корректный запрос на доступ может привести к возникновению ошибки во время выполнения запроса по разным причинам, таким как отсутствие значения атрибута, несоответствие типа атрибута и др.;

– IndeterminateP, или I(P): возникновение ошибки при оценивании rtj или rcj, в то время как rej = Permit. Если бы ошибки не произошло, то доступ был бы разрешен.

Как уже ранее отмечалось, правило может иметь пустую цель, а также пустое условие – такие правила называются правилами по умолчанию. Для правила по умолчанию $r_j = <\_\_, \_\_, re_j>$, каждый запрос на предоставление доступа будет выглядеть как $d(r_j, q) = re_j$.

Корневым элементом документа с XACML политикой, P, является элемент политики или элемент набора политик. Если корневым элементом является набор политик, то комбинационные алгоритмы политик для корневого набора политик имеют вложенные комбинационные наборы правил

и/или комбинационные алгоритмы политик. Поскольку такая вложенность создаёт большое количество сочетаний комбинационных алгоритмов, для упрощения в данной работе будет рассматриваться документ политик XACML, P, в котором в качестве корневого элемента используется элемент политики. Это позволит избежать необходимости иметь дело с вложенными комбинационными алгоритмами правил и политик. В архитектуре ABAC также имеются такие элементы, как «Advice» (рекомендация) и «Obligation», которые в контексте данной работы не играют никакой роли, поэтому при представлении элемента политики они будут опущены. Таким образом, документ политик P будет представлен как P = <PT, RCA, [r1, r2, …, rn]>, где PT – это цель политики, RCA – комбинационный алгоритм правил и [r1, r2, …, rn] – список дочерних правил в политике P.

1.2.1 Оценка политик

Чтобы проиллюстрировать, каким образом работает схема авторизации в XACML, необходимо рассмотреть, как осуществляется оценка каждого элемента в данном стандарте. Когда запрос предоставляется системе разграничения доступа, она сначала должна определить, может ли доступный набор XACML политик применяться к полученному запросу или нет. Для этого используется элемент цели PT в корневом элементе политики P из XACML документа P. Если он является пустым, то данная политика применима к любому запросу q. В случае, когда PT не пустая и запрос соответствует ограничениям, указанным в цели политики (или при возникновении ошибки при оценке элемента), политика считается применимой к запросу q, иначе PDP выдает решение о неприменимости политики и её дальнейшее вычисление не производится. Дальнейшая оценка политики включает в себя оценку дочерних правил. После оценки дочерних элементов и получения авторизационного решения от каждого такого элемента, используется комбинационный алгоритм правил (КАПр), который согласует имеющийся набор авторизационных решений, и в результате

принимается окончательное решение об авторизации на основе результата вычисления комбинационного алгоритма правил.

1.2.2 Оценка правил

Оценка правила ri выполняется, если оценивание вышерасположенных в структуре политики правил не приводит к прекращению оценивания политики P, приводя к результату d(P, q).

Правило ri применимо к запросу q в случаях, когда:

– Цель правила rti и условие правила rci оцениваются как true,

– Цель правила rti оцениваются с ошибкой,

– Цель правила rti оценивается как true, в то время как условие правила оценивается с ошибкой.

Правило оценивается со значением «истина», если цель и условие правила оцениваются как «истина». Аналогично, если цель и условие оценены как «ложь», тогда правило оценивается со значением «ложь» или N/A. В случае ошибки при вычислении цели или условия, правило оценивается с ошибкой. В свою очередь, при вычислении правила со значением «истина», авторизационное решение на уровне правила d(P, ri) будет являться эффектом рассматриваемого правила, то есть если эффект указан как Permit, то и решение будет Permit. Для запроса q в рамках политики P может быть несколько применимых правил, каждое из которых имеет свой эффект. Задачей комбинационного алгоритма правил является согласование этих решений в одно для оценивания политики. Пусть rca(P, q) обозначает результат применения КАПр к правилам в политике P для запроса q. С условием, что список правил в P не является пустым, а согласно спецификации $rca(P, q) \in \{\text{Permit, Deny, NA, I(D), I(P)}\}$, тогда d(P, q) определяется следующим образом (1):

$$d(P,q) = \begin{cases} NA, & \neg PT \\ rca(P,q), & PT \\ rca(P,q), & Error(PT) \wedge RCA(P,q)\{NA, I(D), I(P)\} \\ I(P), & Error(PT) \wedge rca(P,q) = Permit \\ I(D), & Error(PT) \wedge rca(P,q) = Deny \end{cases} \quad (1)$$

1.2.3 Комбинационные алгоритмы

В XACML 3.0 существует одиннадцать комбинационных алгоритмов. Четыре из них – legacy ordered-deny-overrides, legacy ordered-permit-overrides, legacy deny-overrides и legacy permit-overrides – существуют в стандарте для поддержки обратной совместимости с предыдущими версиями XACML, в связи с чем они не рассматриваются в данной работе. Кроме того, алгоритмы first-applicable, ordered-deny-overrides и ordered-permit-overrides, хоть и актуальные для стандарта, на практике применимы редко – в связи с чем они тоже вынесены за рамки данного исследования. Таким образом, рассматриваются следующие четыре алгоритма:

– Deny-overrides: данный алгоритм предназначен для сценариев, где отказ в предоставлении доступа должен иметь преимущество над его получением. Если любое из правил в политике оценивается как Deny, то конечным результатом комбинации будет Deny. В случае отсутствия отрицательного решения и при наличии хотя бы одного I(D), то результатом является I(D) – для I(P) – аналогично. Если нет решений Deny, I(D) или I(P) и любое правило оценивается как Permit, то алгоритм вернет Permit. Если ни одно из представленных условий не сработало, запрос считается не применимым к данному набору правил.

– Permit-overrides: аналогично Deny-Overrides, но решение Permit имеет преимущество перед Deny. Порядок аналогичен представленному выше алгоритму с учетом инверсии авторизационного решения.

– Deny-unless-permit: предназначен для сценариев, когда разрешение преобладает над запретом, но при этом результат Indeterminate или N/A не

должен быть получен в процессе вычисления. Если любое из решений возвращает Permit, то результатом является Permit, иначе Deny.

– Permit-unless-deny: аналогично Deny-unless-permit с учётом инверсии авторизационного решения.

1.3. Анализ научной литературы по теме исследования

Одним из важных аспектов тестирования политик является формальная проверка таких свойств политик контроля доступа, как неполнота и несогласованность. Под несогласованностью понимается ситуация, когда две или более политики, заданные администраторами системы, приводят к различным результатам. К примеру, одна политика может позволять доступ пользователю к определенному ресурсу при определенных условиях, в то время как другая политика может запрещать доступ тому же пользователю при тех же условиях. В свою очередь, проблема неполноты возникает, когда при работе с системой с ограниченным доступом возникают ситуации, для которых администраторы системы не сформулировали правил. В работе Aqib M. были рассмотрены различные методы валидации политик контроля доступа с учётом данных свойств. К ним относятся:

– методы интеллектуального анализа данных [23, 24];
– методы проверки на модели [3, 9, 14, 20, 21, 28];
– формальные методы [8, 11, 22];
– подходы, основанные на матрицах [26];
– другие техники [15, 16, 17, 19].

Автор в своей работе подчеркивает, что сравнение описываемых методов показало, что большая их часть нацелена на обнаружение несогласованности в политиках контроля доступа. Только немногие из алгоритмов, рассмотренных автором, способны обнаруживать обе представленных проблемы одновременно [5].

Несмотря на то, что по данному вопросу существует большое количество статей, по-настоящему эффективными решениями являлись лишь те, что были предназначены для конкретных, специфических систем. В связи с этим, проблема тестирования с учётом данных свойств признаётся исследователями труднорешаемой либо невозможной для решения при работе с политиками, включающими в себя сложные ограничения.

Помимо вопроса проверки вышеупомянутых свойств, в литературе рассматривается также разработка различных инструментов, предназначенных для верификации свойств, задаваемых пользователями, в XACML политиках. Hughes et al. осуществляли перевод политик с языка XACML на язык Alloy и проверяли свойства, используя инструмент Alloy Analyzer. Fisler et al. разработал инструмент Margrave, использующий мультитерминальные диаграммы принятия решений для верификации заданных пользователями свойств и осуществления анализа влияния изменений [10]. Описанные подходы подразумевают, что политики, для которых эти методы используются, описаны с помощью упрощенного варианта языка XACML. В связи с этим, их использование для проверки политик контроля доступа, применяемых в реальных задачах, видится нецелесообразным.

В работе Bertolino et al. был предложен иной подход к тестированию политик – исследователями была предложена концепция покрытия политик, отсылающая к аналогичной методике тестирования программного обеспечения, а именно покрытия кода [6, 7]. Данную концепцию они описали с помощью следующих метрик – процента попадания по политике, процента попадания по правилу и процента попадания по условию. На основе этих метрик ими был разработан инструмент измерения охвата политик с учётом существующего набора XACML политик и набора запросов. Получаемые в процессе работы этого инструмента данные использовались затем для того, чтобы уменьшить количество тестов, генерируемых уже существующими методами генерации тестов.

Hu et al. развивают идею покрытия политик, представляя критерии покрытия политик тестами, а также методы для генерации тестов, удовлетворяющих данным критериям [15]. К критериям покрытия авторы отнесли:

– покрытие правила (RC),
– покрытие результатов решений (DC),
– модифицированный метод покрытия по веткам/условиям (MC/DC).

Данные критерии были определены для таких параметров в XACML3.0, как цель набора политик, цель политики, цель правила и условие правила. Они могут быть использованы для измерения адекватности тестов для разрабатываемой XACML политики и определения того, нужно ли создавать и проводить больше тестов для достижения ожидаемого уровня качества. Даже для политик, уже используемых в системах (т.е. тесты для нее представляют собой реальные запросы на предоставление доступа), критерии покрытия могут указывать на уровни уверенности в качестве политики. Кроме непосредственно критериев, авторами были презентованы методы, позволяющие генерировать тесты для политик с учётом этих критериев. В конечном итоге было выявлено, что тесты, генерируемые с учётом критерия покрытия измененного решения/условия, хоть и имеют высокие показатели эффективности обнаружения ошибок, но все же не способны обнаружить всех ошибок в отдельно взятых политиках [30].

Еще одним подходом, позаимствованным из теории тестирования ПО, является применение мутационного тестирования для проверки политик контроля доступа. В этой технике код существующей программы изменяется некоторым образом для получения кода, отличного от исходной программы. Измененные версии оригинальных программ называются мутантами и их вывод сравнивается с выводом изначальной программы. Если результаты выводов отличаются, то мутант считается убитым, и вывод исходной программы тестируется со следующим мутантом. Более высокий процент

уничтожения мутантов свидетельствует о высокой надежности исходной программы [9]. В случае проверки политик контроля доступа, некоторые исследователи использовали эту технику для целей тестирования. Xie T. et al. представил фреймворк для обнаружения ошибок в политиках, включающий в себя модель ошибок для проведения автоматического мутационного тестирования, а также операторы мутации для данной модели ошибок [29]. В данной работе были использованы ранее определенные критерии покрытия политик, о которых говорилось ранее. Пять элементов XACML политик были рассмотрены для генерации мутантов – это набор политик, политика, правило, цель и условие. Относительно невысокий показатель убитых мутантов в 50-60% от их общего количества обусловлен тем, что авторами был использован подход слабой мутации, при которой данный показатель эффективности не может достичь 100%. Это объясняется тем, что применение слабой мутации является классическим компромиссом между эффективностью и затрачиваемым временем, и в данном случае перевес идёт в сторону временных затрат.

В противовес слабой мутации существует понятие сильного мутационного тестирования – данный вид мутации характерен показателем мутации, близким к максимальному (вплоть до 100%), при этом время, затрачиваемое на генерацию тестов с его помощью, соответственно возрастает. Несмотря на то, что при тестировании программного обеспечения данный метод используется редко из-за его дороговизны, особенности языка XACML, такие как трёхуровневая структура политик, рассмотренная ранее, могут позволить применение такого способа в контексте политик [17].

Выводы и результаты по главе 1

– в ходе рассмотрения теоретических аспектов систем разграничения доступа на основе атрибутов, в частности, стандарта XACML 3.0, были выявлены особенности, позволяющие использование генерации тестов для политик контроля доступа, в частности, трёхуровневая структура политик, ограничивающая число возможных сценариев;

– анализ существующей литературы по вопросу верификации авторизационных политик продемонстрировал, что существующие методы имеют недостаточную эффективность для анализа надежности системы контроля доступа по разным причинам;

– в связи с вышеизложенным, определяется необходимость в проработке вопроса верификации политик контроля доступа с помощью мутационного анализа.

Глава 2 Разработка подхода к генерации тестов на основе сильного мутационного тестирования

2.1 Формулирование модели неисправностей

Как уже было сказано ранее, XACML политики могут содержать в себе различные ошибки ввиду недопонимания требований к системе контроля доступа, сложности языка описания политик, а также ошибок в их проектировании. В то время как ABAC является более выразительным в сравнении с традиционными методами контроля доступа, такими как ABAC, тем не менее, он является достаточно сложным, и ввиду его сложности увеличивается вероятность возникновения сбоев, ведущих к уязвимостям. Исследования показывают, что XACML политики подвержены различным ошибкам, таким как неправильные цели правил, неправильные условия правил, неправильные цели политик и наборов политик, а также неправильное использование правил или комбинационных алгоритмов политик [2, 3].

Для описания ошибок, подобным тем, что описаны выше, применяется модель неисправностей. В общем смысле, модель неисправностей представляет из себя инженерную модель того, что может пойти не так при создании или эксплуатации оборудования, структуры или программного обеспечения. В данном случае осуществляется моделирование того, что может пойти не так при построении политики контроля доступа. В таблице 1 представлена модель неисправностей с учётом стандарта XACML:

Таблица 1 – Модель неисправностей

Оператор мутации		**Вид ошибки**
Название оператора	**Значение**	
ИЭП	Изменение эффекта правила	Некорректный эффект правила
ЦПрИ	Цель правила со значением «истина»	Некорректная цель правила
ЦПрЛ	Цель правила со значением «ложь»	

Продолжение таблицы 1

УПИ	Условие правила со значением «истина»	Некорректное условие правила
УПЛ	Условие правила со значением «ложь»	
ДЛН	Добавить логическое НЕ в условие	
УЛН	Убрать логическое НЕ из условия	
УДП	Удалить правило	Отсутствующее правило
ППР	Первое правило разрешающее	Некорректный порядок правил
ППЗ	Первое правило запрещающее	
ЦПоИ	Цель политики со значением «истина»	Некорректная цель политики
ЦПоЛ	Цель политики со значением «ложь»	
ИКА	Изменение комбинационного алгоритма	Некорректный комбинационный алгоритм

Каждая из представленных неисправностей приводит к семантическим изменениям в политике, но в целом классификация ошибок на семантические и синтаксические будет осуществляться следующим образом: синтаксические ошибки являются результатом простых опечаток, в то время как семантические ошибки связаны с логическими конструкциями языка политик XACML [1].

Синтаксические ошибки допустить проще и, как уже было сказано, они состоят из обычных опечаток, приводящих к семантически некорректной политике. Синтаксические ошибки могут приводить и к синтаксически некорректным ошибкам, но предполагается, что для их выявления будут использоваться базовые средства статического анализа. К примеру, в случае XACML можно использовать XSD схемы, позволяющие выявлять очевидные изъяны в синтаксисе. Хотя и не все синтаксические ошибки нарушают XSD, такие как опечатка в имени атрибута, возможно также написать XSD схему, специфичную для той или иной информации, для проверки правильности значений атрибутов. В связи с этим, ошибки синтаксиса вынесены за пределы рассмотрения в данной работе.

2.2 Условия обнаружения ошибок в политиках контроля доступа

Основным средством поиска ошибок в XACML политиках является выполнение набора тестов над системой контроля доступа. Каждый тест, в свою очередь, состоит из входных данных (авторизационного запроса) и соответствующего тестового оракула (ожидаемого ответа). Тест завершается неудачно, если фактический ответ системы на запрос отличается от ожидаемого ответа. Такой отказ часто указывает на наличие неисправности, которая может привести к несанкционированному доступу или отказу в обслуживании. В свою очередь, сбоем в политике называется ошибка или недостаток, которая приводит к некорректному авторизационному решению. Различие в результате между ошибочной и корректной политикой можно использовать для того, чтобы выявлять ошибки. Идея заключается в том, чтобы записывать авторизационное решение для некоторых входных данных в том случае, когда достоверно известно, что политика корректна. Затем, когда необходимо определить, появились ли ошибки впоследствии, осуществляется подача вышеупомянутых входных данных на подозрительную политику, и если результат отличается от ранее зарегистрированного результата, можно заключить, что существует неисправность. Ошибочная политика, однако, не обязательно даёт результат, отличный от правильного, для всех возможных входных данных. Входные данные должны удовлетворять определенным ограничениям для того, чтобы получить результат, отличающийся от корректной политики, и тем самым выявить неисправность. Такие ограничения называются условиями обнаружения ошибок.

Условия обнаружения ошибок для отдельно взятой неисправности определяют ограничения (или условия), которым должен удовлетворять тест, чтобы выявить неисправность. Для выявления неисправности входные данные теста должны удовлетворять условиям достижимости, необходимости и распространения [2]. Это связано с тем, что для выявления ошибки в политике тест должен достичь неисправного элемента политики, что называется

условием достижимости. Когда он достигнут, тест должен оценить[1] неисправный элемент, чтобы выдать неправильный промежуточный результат, отличающийся от его правильного аналога, что называется условием необходимости. Если тест не удовлетворяет условию достижимости и/или условию необходимости, правильная и ошибочная политика будут вести себя одинаково, в таком случае, обнаружение ошибки будет невозможно. После получения неправильного промежуточного результата, он должен повлиять на получение некорректного конечного результата, что называется условием распространения. Следовательно, условие распространения также необходимо для обнаружения неисправности, поскольку условия достижимости и необходимости могут привести только к неправильному промежуточному результату, который может не сказаться на конечном решении о принятии авторизационного решения системой контроля доступа.

В случае, если условие распространения не выполняется, то неисправность обнаружить также не получится. В то же время проблема с ограничениями распространения в сфере программного обеспечения считается трудноразрешимой ввиду взрывного роста количества путей выполнения программы. Тем не менее, уникальные особенности языка описания политик, XACML 3.0, позволяют формально представить вышеописанные ограничения в трёхзначной логике, что существенно сократит объёмы вычислений для генерации тестов.

2.3 Применение условий обнаружения ошибок к операторам мутации

Рассмотрение модели ошибок и условий обнаружения ошибок позволяет перейти к разработке ограничений для описываемых моделью операторов мутации для того, чтобы убедиться в том, что тесты, генерируемые теми или

[1] Оценивание – промежуточное вычисление в процессе вычисления конечного авторизационного решения, осуществляется для элементов, нижележащих в трехуровневой иерархии XACML политик (правил, политик).

иными операторами, смогут находить ошибки в политиках. В качестве примера разработки таких ограничений предлагается рассмотреть оператор мутации ИЭП с типом ошибки «некорректный эффект правила».

Прежде чем перейти к применению условий обнаружения ошибок к операторам мутации, рассмотрим применяемую нотацию для описания формул. Нотации PT, $\neg PT$ и $Error(PT)$ используются для обозначения того, что цель политики вычисляется со значением true, N/A и Error соответственно. Правило будет вычисляться со значением true, если и цель, и условие правила вычисляются с тем же значением. Выражение $rb_i = (rt_i \wedge rc_i)$ обозначает, что как цель, так и условие правила i вычисляется со значением true. Аналогично, $Error(rb_i)$ обозначает, что результатом вычисления цели или условия правила r_i. В свою очередь, $\neg\, rb_i$ подразумевает, что либо rt_i, либо rc_i имеют значение false, в связи с чем правило вычисляется с конечным авторизационным решением N/A. Кроме того, если ни один из атрибутов в контексте запроса не совпадает с атрибутами в цели правила или условии правила, то правило не применимо к данному запросу. Нотация $I(P), I(D)$ и $I(DP)$ применяется для обозначения результатов комбинационного алгоритма правил – разрешить (Permit), запретить (Deny) и не определен (Indeterminate) соответственно, описывающих как промежуточный, так и конечный результат авторизационных вычислений. Для упрощения может использоваться нотация $r_i = < rb_i, re_i >$, когда возможно представить в такой форме правило i вида $< rt_i, rc_i, re_i >$.

Изменение эффекта правила (ИЭП) – это оператор мутации для типа ошибки «некорректный эффект правила», при котором возникает ошибка в эффекте элемента правила. Поскольку существует только два возможных эффекта правила – Permit или Deny – в случае, если эффект изменяется с Permit на Deny и наоборот, возникает ошибка в эффекте правила. Перевертывание эффекта правила является единственным оператором мутации для данной ошибки.

Рассмотрим XACML политику $P = < PT, RCA, RL >$, где PT – это цель политики, RCA – комбинационный алгоритм правил и $RL = [r_1, r_2, \dots, r_n]$ – список правил внутри политики. Если эффект правила r_i был изменен на Deny из-за какого-либо инцидента, то результирующей политикой будет P', как показано в таблице 2. Поскольку правило r_i должно иметь эффект Permit, но в политике P' эффект того же правила оказывается Deny, то P' является ошибочной политикой. В данном случае, P' является мутантом ИЭП по отношению к политике P.

Таблица 2 – Ошибочная политика с некорректным эффектом правила

	Корректная политика P		Ошибочная политика P'	
Цель политики	PT		PT	
Комбинационный алгоритм правила	Permit-Overrides		Permit-Overrides	
Правила R	r_1	$< rb_1, re_1 >$	r_1	$< rb_1, re_1 >$
	…	…	…	…
	r_i	$< rb_1, \boldsymbol{Permit} >$	r_i	$< rb_1, \boldsymbol{Deny} >$
	…	…	…	…
	r_n	$< rb_n, re_n >$	r_n	$< rb_n, re_n >$

Поскольку условия обнаружения ошибок опираются на комбинационный алгоритм правил (КАПр) политики, данные условия необходимо рассмотреть в рамках каждого существующего комбинационного алгоритма в XACML 3.0 [4]. Среди этих алгоритмов – Permit-Overrides, Deny-Overrides, Deny-Unless-Permit, Permit-Unless-Deny.

2.3.1 Условия обнаружения ошибок для оператора мутации ИЭП

Условие достижимости должно вызывать вычисление правила с ошибочным эффектом, то есть оно должно привести к оценке правила *i* в обеих политиках – P и P'. Правила в политике будут оцениваться только в том случае, если цель политики истинна или содержит ошибку. Кроме того, если алгоритм объединения правил установлен как Permit-Overrides, правило r_i не сработает, если перед правилом r_i есть разрешающее правило, эффект которого при вычислении возвращает Permit, поскольку ошибочное правило в

таком случае не будет затронуто. Это обусловлено тем, что алгоритм Permit-Overrides предназначен для случаев, когда промежуточное решение Permit должно иметь приоритет над промежуточным решением Deny. Следовательно, ограничение достижимости таково – цель политики должна оцениваться как true или Error и все предыдущие правила до текущего рассматриваемого правила должны оцениваться как N/A или Error. Формальная нотация выглядит следующим образом:

$$\{PT \vee Error(PT)\} \wedge \{\neg rb_j \vee Error(rb_j)\ \forall\ r_j =< rb_j, Permit > \text{при } j < i\} \quad (2)$$

Условие необходимости должно приводить правила r_i и r_i' в политике P и P' соответственно к различным вычислениям на уровне правил. Для этого необходимо, чтобы rb_i оценивалось либо со значением true, чтобы решение правила i было Permit для P и Deny для P соответственно, либо имело значение Error, чтобы решение правила было $I(P)$[2] в P и $I(D)$[3] в P'. Если же rb_i оценивается как N/A, решения на уровне правил будут N/A как в политике P, так и в ошибочной P', в связи с чем отличить ошибочную политику не будет представляться возможным. Таким образом, формально ограничение необходимости формулируется так: $rb_i \vee Error(rb_i)$.

Если неисправный элемент политики приводит к различным промежуточным результатам, то условие распространения должно заставлять P и P' вырабатывать различные решения на уровне политики. Другими словами, различный промежуточный результат от ограничения распространения должен способствовать выработке различных решений на уровне политики. Для этого любое разрешающее правило r_j (j > i) после r_i не должно оцениваться как разрешающее, иначе $d(P, q) = d(P', q) = Permit$. Тест

[2] I(P) – Indeterminate Permit – данная оценка обозначает, что запрос мог бы быть оценен со значением Permit, но при оценке цели или условия правила возникла ошибка.
[3] I(D) – Indeterminate Deny – данная оценка обозначает, что запрос мог бы быть оценен со значением Deny, но при оценке цели или условия правила возникла ошибка.

должен приводить к оцениванию rb_j как N/A или Error для каждого разрешающего правила r_j (j > i). Таким образом, ограничение распространения может быть формализовано в виде:

$$\neg\, rb_j \vee Error(rb_j)\ \forall\ r_j = < rb_j, Permit > (j > i) \qquad (3)$$

Допустим, Þ = "$\neg\, rb_j \vee Error(rb_j)\ \forall\ r_j = < rb_j, Permit > (j > i)$", то есть Þ – ограничение распространения. Ограничение Þ является достаточным для распространения ошибки, если rb_i оценивается со значением true. Однако, если rb_i оценивается со значением Error, то Þ не будет являться достаточным для распространения.

Для того, чтобы доказать представленное выше утверждение, необходимо рассмотреть таблицу 3, в которой можно увидеть, что ограничение Þ выполняется для всех возможных промежуточных вычислений других правил (всех правил в политике, кроме текущего рассматриваемого правила).

Таблица 3 – Возможные промежуточные вычисления остальных правил, когда rb_i оценивается как «истина» (true)

	Другие правила, кроме i-го правила	**i-е правило в политике *P***	**i-е правило в политике *P'***	**КАП в политике *P***	**КАП в политике *P'***
Производит эффект правила	Все другие правила оцениваются как N/A и не производят никакого эффекта	Permit	Deny	Permit	Deny

Продолжение таблицы 3

	Одно или более правил производит эффект Deny или I(D), остальные N/A	Permit	Deny	Permit	Deny
	Одно или более правил производит эффект I(P), остальные N/A	Permit	Deny	Permit	I(DP)
	Одно или более правил производит эффект I(P), одно или более правил производит эффект Deny или I(D), остальные N/A	Permit	Deny	Permit	I(DP)

Пусть i-е правило в корректной политике P имеет эффект Permit (разрешительный), в то время как тот же элемент в ошибочной политике P’ имеет эффект Deny (запретительный). Результат комбинационного алгоритма правил будет зависеть от результата оценки других правил, включая результат оценки i-го правила. Как показано в таблице 3, существует возможность для четырёх возможных оценок других правил.

Первая возможная оценка – все другие правила оцениваются как N/A и не производят никакого эффекта. В этом случае, эффект первого правила в P, а это Permit, будет решением уровня комбинационного алгоритма правил для P, а в P’ эффект будет Deny, так как его i-е правило имеет эффект Deny. Когда только одно правило оценивается как разрешающее, результатом вычисления по алгоритму Permit-Overrides будет Permit. Аналогично, когда только одно правило оценивается как запрещающее, результатом этого же алгоритма будет Deny. В результате, решение на уровне КАП – это Permit и Deny в политиках P и P’ соответственно, как показано в пятом и шестом столбцах первой строки

соответственно. Решение на уровне политики зависит от результата на уровне КАП и от того, как оценивается цель политики. Когда цель правила оценивается со значением true, результат P и P’ на уровне политики будет результатом соответствующих КАП, т.е. Permit в P и Deny в P’.

Вторая возможная оценка – это когда одно или несколько запрещающих правил производят эффект Deny или I(D), а остальные – N/A, В этом случае, поскольку КАП = Permit-Overrides, результатом комбинационного алгоритма в P будет Permit, поскольку i-е правило имеет эффект Permit, и если любое правило оценивается как Permit, результатом вычисления КАП будет Permit, что следует из названия алгоритма – разрешающий эффект (Permit) превалирует над запрещающим (Deny). В ошибочной политике P’ i-е правило оценивается как запрещающее, одно или несколько правил оценивается как запрещающее или I(D), а остальные правила оцениваются как N/A, то есть ни одно из правил не оценивается как разрешающее или I(P). Когда КАП производит вычисления с результатом Permit, в случае, если ни одно из правил не оценивается как разрешающее или I(P) и любое из правил оценивается как запрещающее, решение уровня КАП будет запрещающим для политики P’. Когда цель политики оценивается как true, результаты для политик P и P’ будут вычисляться как Permit и Deny соответственно. Аналогично, когда цель политики оценивается как Error, результаты для политик P и P’ оцениваются как I(P) и I(D).

Третья возможная оценка – когда одно или несколько разрешающих правил дают эффект I(P), а остальные – N/A. В этом случае результатом КАП в P будет Permit, поскольку i-е правило имеет разрешающий эффект. Для P’, каждое правило оценивается как запрещающее, и поскольку одно или несколько правил оцениваются как I(P), результатом КАП будет I(DP). В результате, когда цель политики оценивается как true, результаты для политик P и P’ оцениваются как Permit и I(DP). Аналогично, в случае если цель правила оценивается как Error, результаты для политик P и P’ оцениваются как I(P) и I(DP) соответственно.

Четвертая возможная оценка – когда одно или более правил производит эффект I(P), одно или более правил производит эффект Deny или I(D), а остальные – N/A. В данном случае результат КАП в P всё равно будет разрешающим, так как i-е правило имеет разрешающий эффект, а результатом P’ будет I(DP). В результате, когда цель политики оценивается как true, результаты для политик P и P’ оцениваются как I(P) и I(DP).

Следовательно, когда rb_i оценивается как true, ограничение Þ достаточно для соблюдения условия распространения, так как оно выполняется для всех возможных оценок других правил. В то же время, если rb_i оценивается со значением Error, данного ограничения недостаточно для соблюдения вышеописанного условия. В таблице 4 представлено обоснование данного утверждения:

Таблица 4 – Возможные промежуточные вычисления остальных правил, когда rb_i оценивается как «ошибка» (Error)

	Другие правила, кроме i-го правила	**i-е правило в политике *P***	**i-е правило в политике *P’***	**КАП в политике *P***	**КАП в политике *P’***
Производит эффект правила	Отсутствие эффекта от других правил, поскольку все они оцениваются N/A	I(P)	I(D)	I(P)	I(D)
	Одно или более правил производит эффект Deny или I(D), остальные N/A	I(P)	I(D)	Permit / I(P)	I(DP)
	Одно или более правил производит эффект I(P), остальные N/A	I(P)	I(D)	I(DP)	I(D)

Продолжение таблицы 4

	Одно или более правил производит эффект I(P), одно или более правил производит эффект Deny или I(D), остальные N/A	I(P)	I(D)	I(DP)	I(DP)

Результат комбинационного алгоритма правил, исходя из данных таблицы 4, для P и P’ в первых трёх случаях различается – это значит, что условие распространения для них соблюдается. Однако в четвертом случае, результат для P и P’ одинаков – I(DP). Поскольку результат при вычислении КАП не различается как для корректной, так и для ошибочной политик, ошибка в данном случае не будет обнаружена – это означает, что условие распространения не соблюдается. Поэтому ограничение Þ для четвертого случая недостаточно для условия распространения, когда rb_i оценивается как Error. В данном случае необходимо дополнительное ограничение в ограничении распространения. Требуемое дополнительное ограничение заключается в следующем – не должно существовать такой пары правил (исключая i-е правило), что одно из них имеет разрешающий эффект и оценивающееся как Error, а другое имеет запрещающий эффект и оценивающееся как true или Error. Если это ограничение будет выполняться, то четвертый случай из представленной таблицы не произойдёт, и ошибочное поведение будет возможно отличить от корректного. Формально данное ограничение к Þ описывается выражением (4):

$$\neg\,(\exists(p,d) : (i \neq \mathrm{p} \neq \mathrm{d}) \wedge \left(r_p = < rb_p, \mathrm{Permit} >\right)$$
$$\wedge\,(r_d = < rb_d, \mathrm{Deny} >) \wedge Error\left(rb_p\right) \wedge (Error(rb_d) \vee rb_d)) \qquad (4)$$

Таким образом, при объединении всех рассмотренных ограничений в одно, формулируется ограничение для обнаружения ошибок типа «изменение эффекта правила» в случае, когда комбинационный алгоритм правил является Permit-Overrides.

Для выполнения условия достижимости цель политики должна быть true или оцениваться как ошибка. Кроме того, в случае с КАП = Deny-Overrides, вычисление для правила r_i не сработает, если перед этим правилом вычислится правило с эффектом Deny, которое будет оценено с аналогичным промежуточным решением. Таким образом, ограничение достижимости заключается в том, что для любого запрещающего правила r_j (j $<$ i) перед правилом r_i, rb_j не должно иметь значение true (т.е. оно должно быть N/A или Error). Формально данное ограничение формулируется следующим образом: $\{PT \vee \text{Error(PT}\} \wedge \{\neg\, rb_j \vee \text{Error}(rb_j)\ \forall\ r_j =< rb_j, Deny > \text{при } j < i\}$.

Для выполнения условия необходимости rb_i должно быть либо true, чтобы решение для правила i было разрешающим и запрещающим для политик P и P’ соответственно, либо Error, чтобы решение по правилу было I(P) и I(D) в P и P’ соответственно. Если же rb_i будет оцениваться как N/A, то решения на уровне правил будут N/A как в P, так и в P’, и в этом случае выделить ошибочную политику не будет представляться возможным. Поэтому условие необходимости формулируется так: $rb_i \vee Error(rb_i)$.

Для соблюдения условия распространения каждое запрещающее правило r_j (j $>$ i) после r_i не должно приводить к решению Permit, в противном случае, $d(P,q) = d(P',q) = Deny$. Тест должен заставлять rb_j оцениваться как N/A или Error для каждого запрещающего правила r_j (j $>$ i). Таким образом, ограничение распространения может быть формализовано как $\neg\, rb_j \vee \text{Error}(rb_j)$ для каждого правила $r_j =< rb_j, Deny > (j < i)$.

Вышеупомянутое ограничение распространения является достаточным для соблюдения условия, если при ограничении необходимости rb_i оценивается как true, но не будет являться достаточным при оценивании rb_i

как Error. Поскольку аналогичная проблема возникала при рассмотрении ограничения распространения для комбинационного алгоритма Permit-Overrides, можно сделать вывод, что и в данном случае требуется дополнительное ограничение для соблюдения условия распространения.

Объединяя рассмотренные ограничения в одно, формулируется общее ограничение для обнаружения ошибок типа «изменение эффекта правила» в случае, когда комбинационный алгоритм правил является Deny-Overrides.

Поскольку условия обнаружения ошибок для алгоритма Permit-Overrides симметричны с теми же условиями для алгоритма Deny-Overrides, с той лишь разницей, что меняются местами лишь эффекты правил, для простоты при описании условий обнаружения ошибок для других операторов мутации можно брать в расчёт только алгоритм Permit-Overrides.

Для выполнения условия достижимости цель политики должна быть true или оцениваться как ошибка. В случае с КАП = Deny-Unless-Permit, правило r_i не вычислится, если до правила r_i будет существовать разрешающее правило, оценивающееся как истина. Поэтому, условие достижимости будет заключаться в том, что цель политики должна оцениваться как true или Error, а также все предыдущие правила с разрешающим эффектом до правила с внедренной мутацией должны оцениваться как N/A или Error. В формальной нотации, ограничение будет записано следующим образом:

$\{PT \vee \text{Error(PT)}\} \wedge \{\neg\, rb_j \vee \text{Error}(rb_j)\ \forall\, r_j =< rb_j, Permit > \text{при}\ j < i\}$.

Алгоритм Deny-Unless-Permit приводит к решению Permit только в случае, если хотя бы одно из правил с эффектом Permit оценивается как true, в противном случае, во всех остальных случаях результирующим эффектом будет Deny, то есть отказ в предоставлении доступа. В результате, нельзя рассматривать вариант с Error в правиле i, поскольку если результат Error будет получен в i-й цели правила и/или в условии, результатом для комбинационного алгоритма будет Deny вне зависимости от политик P и P’. В таком случае, и ошибочная, и корректная политика будут вести себя

одинаково при данном сценарии, поэтому ограничение необходимости является $rb_i = true$.

Для соблюдения условия распространения каждое разрешающее правило r_j (j > i) после r_i не должно приводить к решению Permit, в противном случае, $d(P, q) = d(P', q) = Permit$. Тест должен заставлять rb_j оцениваться как N/A или Error для каждого разрешающего правила r_j (j > i). Таким образом, ограничение распространения может быть формализовано как $\neg\, rb_j \vee \text{Error}(rb_j)$ для каждого правила $r_j =< rb_j, Permit > (j < i)$.

При объединении всех вышеописанных ограничений в одно, получается общее ограничение для обнаружения ошибок типа «изменение эффекта правила» в случае, когда комбинационный алгоритм правил является Deny-Unless-Permit. Для алгоритма Permit-Unless-Deny рассмотрение ограничений предлагается опустить, поскольку здесь существует аналогичная симметрия, как с алгоритмами Permit-Overrides/Deny-Overrides.

2.3.2 Условия обнаружения ошибок для оператора мутации ЦПрИ

Оператор мутации ЦПрИ («цель правила истина») изменяет цель правила таким образом, что данный элемент всегда оценивается как true. Одним из правил преобразования, которое позволит сделать оценивание цели правила как истина, является изменение его таким образом, чтобы содержимое данного элемента было пустым. Рассмотрим XACML политики $P =< PT, RCA, RL >$ и $P' =< PT, RCA, RL' >$, где $RL =< r_1, \dots, r_i, \dots, r_n >$, $RL' =< r_1, \dots, r_i', \dots, r_n >$, $r_i =< rt_i, rc_i, re_i >$ и $r_i' =< rt_i', rc_i, re_i >$. В данном случае политики P и P' аналогичны друг другу за тем исключением, что цель правила r_i', то есть rt_i', всегда оценивается как true. Следовательно, политика P' будет являться мутантом политики P, созданным оператором мутации ЦПрИ. Далее рассмотрим формулирование ограничений для данного оператора, основываясь на комбинационных алгоритмах правил, как было сделано ранее на примере оператора мутации ИЭП.

В случае комбинационного алгоритма Permit-overrides должны соблюдаться следующие условия:

- Ограничение достижимости: цель политики должна оцениваться как true или error, а все предыдущие правила с разрешительным эффектом до рассматриваемого правила должны оцениваться как N/A или Error (то есть не приводить к терминальному эффекту true). В формальной нотации логическое выражение, которое должно выполняться для соблюдения данного ограничения, может быть представлено следующим образом: $\{PT \vee \mathrm{Error(PT)}\} \wedge \{\neg\, \mathrm{rb_j} \vee \mathrm{Error(rb_j)}$ для каждого разрешающего правила $r_j = < rb_j, Permit >$ при $j < i$.
- Ограничение необходимости: цель правила должна оцениваться как N/A или Error, а условие правила должно оцениваться как истина. В формате логического выражения - $\neg\, rt_i \vee Error(rt_i) \wedge \mathrm{rc_i}$.

Отдельно необходимо рассмотреть условия, при которых выполняется ограничение распространения, поскольку оно является ключевым для выполнения сильного мутационного тестирования.

В случае, если цель политики оценивается как true, цель правила i оценивается как N/A и эффект правила i является запретительным, то все остальные правила должны оцениваться как N/A или Error. Кроме того, не должно существовать пары правил (исключая текущее рассматриваемое правило), таких, что одно из них – разрешающее правило, оценивающееся как Error, а другое – запретительное, оценивающееся как true или Error. Математически данное условие формулируется следующим образом:

$$\begin{aligned}\{(PT) \wedge \neg\, rt_i \wedge \mathrm{r_i} = < \mathrm{rb_i}, \mathrm{Deny} > \wedge \left\{\neg\, rb_j \vee Error\left(rb_j\right) \forall\, r_j \text{ при } j\ != i\right\} \wedge \\ \wedge \{\neg(\exists(p,d) \ni (i \neq p \neq d) \wedge (\mathrm{r_d} = < rb_d, Deny >) \wedge \\ \wedge \left(\mathrm{r_p} = < rb_p, Permit >\right) \wedge \mathrm{Error}\left(\mathrm{rb_p}\right) \wedge (\mathrm{Error(rb_d)} \vee \mathrm{rb_d}))\})\end{aligned} \quad (5)$$

Если же цель политики оценивается как Error, цель правила i оценивается как N/A, а эффект того же правила запретительный, тогда все остальные правила с запретительным эффектом должны оцениваться как N/A, а все прочие правила с разрешительным эффектом не должны оцениваться как true. При этом не должно существовать такой пары правил (не считая рассматриваемого), таких, что одно из них – разрешающее правило, оценивающееся как Error, а другое – запретительное, оценивающееся как true или Error, как и в предыдущем примере. Соблюдение перечисленных требований формально описывается так:

$$\begin{gathered}\{(PT) \wedge \neg\, rt_i \wedge \mathrm{r_i} = < \mathrm{rb_i}, \mathrm{Deny} > \wedge \\ \wedge \left\{\neg\, rb_j \vee Error\left(rb_j\right) \forall\, r_j = < rb_i, Permit > \text{при } j\ != i\right\} \wedge \\ \wedge \left\{\neg\, rb_j \vee Error\left(rb_j\right) \forall\, r_j = < rb_i, Deny > \text{при } j\ != i\right\} \wedge \\ \wedge \{\neg(\exists(p,d) \ni (i \neq p \neq d) \wedge (\mathrm{r_d} = < rb_d, Deny >) \wedge \\ \wedge \left(\mathrm{r_p} = < rb_p, Permit >\right) \wedge \mathrm{Error}\left(\mathrm{rb_p}\right) \wedge (\mathrm{Error}(\mathrm{rb_d}) \vee \mathrm{rb_d}))\}\end{gathered} \tag{6}$$

В случае, если цель политики оценивается как true, цель правила i оценивается как N/A и эффект правила i является разрешительным, то все остальные разрешительные правила должны оцениваться как N/A или Error. Для соблюдения данного условия необходимо выполнение следующего логического выражения:

$$\begin{gathered}\{(PT) \wedge \neg\, rt_i \wedge \mathrm{r_i} = < \mathrm{rb_i}, \mathrm{Permit} > \wedge \\ \wedge \left\{\neg\, rb_j \vee Error\left(rb_j\right) \forall\, r_j = < rb_i, Permit > \text{при } j\ != i\right\}\end{gathered} \tag{7}$$

В свою очередь, когда цель политики оценивается со значением Error, цель правила i оценивается как N/A и эффект того же правила является разрешительным, все остальные правила с разрешительным эффектом должны оцениваться как N/A, если все запретительные правила оцениваются как N/A.

В то же время если любое запретительное правило оценивается как true или Error, то другое разрешительное правило может оцениваться как N/A или Error. В представленной комбинации условий логическое выражение будет сформулировано в виде:

$$(\{Error(PT)\} \wedge \neg\, rt_i \wedge \mathrm{r_i} =< \mathrm{rb_i}, \mathrm{Permit} > \wedge$$
$$\wedge\, [\{\neg\, rb_j\ \forall\, r_j \text{ при } j\ !=i\} \vee \{\exists d(rb_d \vee \mathrm{Error(rb_d)}) \wedge (\mathrm{r_d} =< \mathrm{rb_d}, \mathrm{Deny} >) \wedge \quad (8)$$
$$\wedge\, \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j =< rb_i, Permit > \text{ при } j\ !=i\}])$$

Если цель политики оценивается как true, цель правила i оценивается как Error и эффект того же правила является запретительным, тогда все остальные правила с разрешительным эффектом должны оцениваться как N/A, в то время как правила с запретительным эффектом должны оцениваться как N/A или Error. Такое сочетание условий может быть описано следующим логическим выражением:

$$\{(PT) \wedge \mathrm{Error(rt_i)} \wedge \mathrm{r_i} =< rb_i, Deny > \wedge$$
$$\wedge\, \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j =< rb_i, Deny > \text{ при } j\ !=i\} \wedge \quad (9)$$
$$\wedge\, \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j =< rb_i, Deny > \text{ при } j\ !=i\}$$

Одним из краевых случаев является сценарий, при котором цель политики оценивается как Error, цель правила i оценивается как Error и эффект правила i является запретительным. При таком сочетании условий обнаружение ошибки типа «цель правила истина» не представляется возможным, а мутант, сгенерированный для подобной политики, будет работать аналогично исходной политике, то есть являться эквивалентным, для данного набора запросов.

Если цель политики оценивается как true, цель правила i оценивается как Error и эффект правила i является разрешительным, тогда все остальные

правила с разрешительным эффектом должны оцениваться как N/A. Соблюдение данного условия формулируется в виде выражения (10):

$$(\{(PT)\} \wedge \mathrm{Error}(\mathrm{rt_i}) \wedge \mathrm{r_i} = < rb_i, Permit > \wedge \\ \wedge \{\neg\, rb_j \; \forall\, r_j = < rb_i, Permit > \text{при } j\, !\, = i\}) \qquad (10)$$

Для случая, при котором цель политики оценивается как true, цель правила i оценивается как Error и эффект правила i является разрешительным, все остальные правила с разрешительным эффектом должны оцениваться как N/A, если все прочие запретительные правила оцениваются со значением Error. Тем не менее, если хотя бы одно запретительное правило оценивается как true или Error, разрешительное правило может оцениваться как error или N/A. Такой набор условий формализуется в виде логического выражения (11):

$$(\{Error(PT)\} \wedge \mathrm{Error}(\mathrm{rt_i}) \wedge \mathrm{r_i} = < \mathrm{rb_i}, \mathrm{Permit} > \wedge \\ \wedge \{\neg\, rb_j \;\; \forall\, r_j \text{ при } j\, !\, = i\} \vee \{\exists d\big(rb_d \vee \mathrm{Error}(\mathrm{rb_d})\big) \wedge (\mathrm{r_d} =< \mathrm{rb_d}, \mathrm{Deny} >) \wedge \\ \wedge \{\neg\, rb_j \vee Error\big(rb_j\big) \; \forall\, r_j = < rb_i, Permit > \text{при } j\, !\, = i\} \qquad (11)$$

При объединении всех ограничений (5-11) в одно, получается ограничение для условия обнаружения ошибок оператора ЦПрИ в случае, когда КАПр = Permit-Overrides. В полном виде оно представлено в Приложении А. Подтвердить достоверность полученного ограничения можно тем же путём, как это выполнялось для оператора ИЭП ранее. Пусть правило i имеет запретительный эффект. Результат вычисления комбинационного алгоритма правил будет зависеть от оценки других правил, а также от оценки самого правила i. В таблице 5 представлены возможные сценарии оценки правил и результат вычисления КАПр как для исходной политики P, так и для мутантной политики P’.

Таблица 5 – Возможные промежуточные вычисления остальных правил, когда rt_i оценивается как N/A и Ri является запретительным правилом в P

	Другие правила, кроме i-го правила	**i-е правило в политике *P***	**i-е правило в политике *P'***	**КАП в политике *P***	**КАП в политике *P'***
Производит эффект правила	Все другие правила оцениваются как N/A и не производят никакого эффекта	N/A	Deny	N/A	Deny
	Одно или более правил производит эффект Deny остальные N/A	N/A	Deny	Deny	Deny
	Одно или более правил производит эффект I(D), остальные N/A	N/A	Deny	I(D)	Deny
	Одно или более правил производит эффект I(P), остальные N/A	N/A	Deny	I(P)	I(DP)
	Одно или более правил производит I(P), одно или более правил производит Deny или I(D), остальные N/A	N/A	Deny	I(DP)	I(DP)

Рассуждая по аналогии с содержимым таблицы 3 можно рассмотреть записи в таблице 5. Последний из перечисленных случаев в данной таблице, когда одно или более правил (кроме правила i) производит эффект I(P), одно или более правил производит запретительный эффект и остальные правила вычисляются со значением N/A, результат комбинационного алгоритма аналогичен как в P', так и в P. Это означает, что при таких условиях ошибка не может быть обнаружена тестом. В дополнение, во втором сценарии,

перечисленном в таблице – когда одно или более правил оцениваются как запретительные, а остальные как N/A, результат комбинационного алгоритма в P и P’ также не отличается друг от друга. Таким образом можно заключить, что если эффект правила i является запретительным, то для обнаружения ошибки такого типа необходимо, чтобы все правила кроме правила i не оценивались со значением true, и не должна существовать такая пара правил (кроме правила i), в которой одно из правил имеет запретительный эффект, другое имеет разрешительный эффект – и при этом разрешительное правило оценивается как Error, а запретительное – как true или Error.

Далее, если цель политики оценивается как Error, тогда результат вычисления для исходной и мутантной политик P и P’ будет I(D) для третьего случая, когда одно или несколько других правил с запрещающим эффектом будут оценены как I(D). Следовательно, когда цель политики оценивается как Error, условие обнаружения ошибки таково, что другие правила с разрешающим эффектом могут оцениваться как Error, а правила с запрещающим эффектом должны оцениваться как N/A.

Пусть эффект правила i будет являться разрешающим, а не запретительным. Для такого правила возможные сценарии представлены в таблице 6:

Таблица 6 – Возможные промежуточные вычисления остальных правил, когда rt_i оценивается как N/A и Ri является разрешительным правилом в P

	Другие правила, кроме i-го правила	**i-е правило в политике *P***	**i-е правило в политике *P'***	**КАП в политике *P***	**КАП в политике *P'***
Производит эффект правила	Все другие правила оцениваются как N/A и не производят никакого эффекта	N/A	Permit	N/A	Permit
	Одно или более правил производит эффект Deny остальные N/A	N/A	Permit	Deny	Permit
	Одно или более правил производит эффект I(D), остальные N/A	N/A	Permit	I(D)	Permit
	Одно или более правил производит эффект I(P), остальные N/A	N/A	Permit	I(P)	Permit
	Одно или более правил производит I(P), одно или более правил производит Deny или I(D), остальные N/A	N/A	Permit	I(DP)	Permit

Исходя из сценариев, представленных в таблице 6, становится очевидно, что в случае, если эффект правила i является разрешающим, ошибка в политике может быть обнаружена, если ни одно из других разрешающих правил не оценивается со значением true.

Если же цель политики оценивается как Error, тогда результат вычисления КАПр для P и P' будет I(P) для четвертого случая в таблице 6, когда правило с разрешающим эффектом оценивается как Error. Поэтому,

когда цель политики оценивается как Error, условие обнаружения ошибки должно быть таково, чтобы правила с разрешающим эффектом оценивались только как N/A. При этом, в случае, если существует другое запрещающее правило, не оценивающееся как N/A, тогда может существовать и разрешительное правило, которое оценивается как Error – при таких условиях ошибка будет обнаружена.

Теперь необходимо рассмотреть ситуацию, в которой цель правила i оценивается как Error. Таблица 7 представляет описание возможных сценариев для оценивания других правил и результат КАПр для исходной и мутантной политик. Из содержания таблицы видно, что ошибка типа «ЦПрИ» может быть обнаружена только в случае, если прочие правила с разрешительным эффектом оцениваются как N/A и правила с запретительным эффектом не оцениваются со значением true.

Таблица 7 – Возможные промежуточные вычисления остальных правил, когда rt_i оценивается как Error и Ri является запретительным правилом в P

	Другие правила, кроме i-го правила	**i-е правило в политике *P***	**i-е правило в политике *P'***	**КАП в политике *P***	**КАП в политике *P'***
Производит эффект правила	Все другие правила оцениваются как N/A и не производят никакого эффекта	I(D)	Deny	I(D)	Deny
	Одно или более правил производит эффект Deny остальные N/A	I(D)	Deny	Deny	Deny
	Одно или более правил производит эффект I(D), остальные N/A	I(D)	Deny	I(D)	Deny

Продолжение таблицы 7

	Одно или более правил производит эффект I(P), остальные N/A	I(D)	Deny	I(DP)	I(DP)
	Одно или более правил производит I(P), одно или более правил производит Deny или I(P), остальные N/A	I(D)	Deny	I(DP)	I(DP)

В таблице также продемонстрировано, что в случае, если цель политики оценивается как Error, тогда результат вычисления исходной и мутантной политик одинаков во всех случаях и мутантная политика будет работать аналогично исходной.

Пусть эффект правила i является разрешительным, а не запретительным. Для данного случая в таблице 8 представлены возможные случаи промежуточных вычислений других правил и результатов КАПр для P и P'.

Таблица 8 – Возможные промежуточные вычисления остальных правил, когда rt_i оценивается как Error и Ri является разрешительным правилом в P

	Другие правила, кроме i-го правила	**i-е правило в политике *P***	**i-е правило в политике *P'***	**КАП в политике *P***	**КАП в политике *P'***
Производит эффект правила	Все другие правила оцениваются как N/A и не производят никакого эффекта	I(P)	Permit	I(P)	Permit

Продолжение таблицы 8

	Одно или более правил производит эффект Deny остальные N/A	I(P)	Permit	I(DP)	Permit
	Одно или более правил производит эффект I(D), остальные N/A	I(P)	Permit	I(DP)	Permit
	Одно или более правил производит эффект I(P), остальные N/A	I(P)	Permit	I(P)	Permit
	Одно или более правил производит I(P), одно или более правил производит Deny или I(D), остальные N/A	I(P)	Permit	I(DP)	Permit

Из сценариев, представленных выше, можно сделать вывод, что в случае, если эффект правила i является разрешающим, ошибка в политике может быть обнаружена, если ни одно из других разрешающих правил не оценивается со значением true. При этом, если хотя бы одно запрещающее правило оценивается как true или Error, то другое разрешающее правило может оцениваться как Error или N/A.

Для комбинационного алгоритма Deny-Unless-Permit ограничения достижимости, необходимости и распространения будут сформулированы в более компактном виде. Это обусловлено особенностями вычисления данного комбинационного алгоритма, а именно – отсутствует необходимость в проверке многочисленных условий для отдельно взятой политики, поскольку для достижения и вычисления исходного и мутантного правила достаточно, чтобы правила (кроме мутантного) не оценивались как true.

Для выполнения ограничения достижимости требуется, чтобы цель политики оценивалась как true или Error. Для каждого разрешающего правила до правила ri, rbj не должны оцениваться как true (то есть результатом оценивания должен быть N/A или Error). Формально такое ограничение будет описано в виде (10):

$$\{PT \vee \text{Error(PT)}\} \wedge$$
$$\wedge \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j = < rb_j, Permit > \text{при}\ j < i\} \qquad (10)$$

Чтобы соблюсти условие необходимости, требуется, чтобы цель рассматриваемого правила оценивалась как N/A или Error, а условие правила оценивалось как true. Данное ограничение формулируется в виде логического выражения (11):

$$\left(\neg\, rt_i \vee Error(rt_i)\right) \wedge \text{rc}_\text{i} \qquad (11)$$

В случае с ограничением распространения – если рассматриваемое правило является запрещающим, то мутант будет являться эквивалентным, поскольку промежуточное вычисление на уровне правил будет запретительным, а в исходной политике результатом будет N/A или Error. Тем не менее, как отмечалось ранее, для комбинационного алгоритма Deny-Unless-Permit все, что не приводит к разрешительному эффекту, приводит к запретительному. Таким образом, рассматриваемое правило должно быть разрешающим, при этом все последующие правила с разрешительным эффектом должны оцениваться как N/A или Error.

При объединении всех рассмотренных ограничений в одно становится возможным сформулировать следующее условие обнаружения ошибки «ЦПрИ» в случае, когда комбинационный алгоритм правил является Deny-Unless-Permit (12):

$$\{PT \vee \text{Error(PT)}\} \wedge$$
$$\wedge \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j = < rb_j, Permit > \text{при}\ j < i\} \wedge \qquad (12)$$
$$\wedge \{(\neg\, rt_i \vee Error(rt_i)) \wedge \text{rc}_\text{i}\}$$

2.4 Разработка сценариев для генерации тестов с учётом модели неисправностей

Генерация тестов на основе мутаций подразумевает генерацию таких тестовых наборов, которые используют условия обнаружения неисправностей с целью уничтожения всех неэквивалентных мутантов. Формализация полных условий обнаружения неисправностей делает возможным автоматическое создание тестов на основе сильного мутационного анализа. Если запрос на предоставление доступа удовлетворяет требованиям достижимости, необходимости и распространения для конкретной ошибки, то данный запрос даст разный результат в исходной и неисправной политике. Это различие в ответе, который формирует точка принятия решения в системе контроля доступа, может быть использовано для того, чтобы отличить ошибочную политику от правильной. Следовательно, задача состоит в том, чтобы сгенерировать входные данные, удовлетворяющие условиям обнаружения ошибок, для каждой возможной неисправной политики.

УОО могут включать несколько взаимоисключающих условий. Если одно из них выполняется, этого будет достаточно для обнаружения неисправности. В результате становится возможным использовать только одно из взаимоисключающих условий для каждой неисправности. Другими словами, устраняется необходимость в генерации нескольких запросов для каждого из взаимоисключающих условий, поскольку это приведет к появлению избыточных тестовых сценариев. В общих чертах алгоритм генерации тестов на основе сильного мутационного анализа (СМА) показан на рисунке 4:

Алгоритм 1 Генерация набора тестов на основе мутации

Импортируемые функции: kill(M,Q) возвращает список мутантов в M, которые были убиты тестовым набором Q; Z3request(FDC) возвращает решение для ограничения FDC

Входные данные: opsList - список операторов мутаций

Выходные данные: Q - набор запросов доступа

Переменные: M - набор мутантов, OPS - список операторов мутаций, FDC - условия обнаружения ошибок, q - тестовые входные данные

```
1:  procedure GENERATETESTS(opsList)
2:      Q ← e
3:      while opsList ≠ ∅ do
4:          OPS ← opsList   ▷ Выбрать один или более операторов мутации из opsList
5:          M ← OPS         ▷ Список мутантов, созданный мутационными операторами OPS
6:          M ← M − kill(M, Q)
7:          while M ≠ ∅ do
8:              FDC ▷ Составить УОО для одного или нескольких мутантов
9:              q ← Z3request(FDC)
10:             if q ≠ null then      ▷ Иначе мутант является эквивалентным
11:                 Q ← Q ∪ q
12:                 M ← M − kill(M, q)
13:             end if
14:         end while
15:     end while
16: end procedure
17: return Q
```

Рисунок 4 – Представление алгоритма CMA в формате псевдокода

На основе формализации условий обнаружения неисправностей было определено, что некоторые из операторов имеют общие УОО среди различных возможных взаимоисключающих УОО для каждого из них. В связи с этим исключается необходимость разбираться с каждым из операторов мутаций по отдельности. Другими словами, если мы сгенерируем запрос, который удовлетворяет общему условию обнаружению ошибок среди подобных операторов, он сможет убить мутантов для нескольких операторов. Рассмотрим представленный алгоритм подробнее. Сперва выбирается один (или несколько операторов, если они имеют общие УОО) за раз и выполняется генерация мутантов для них (строки 4-5), после чего мутанты запускаются против существующих тестов. Мутанты, которые уже убиты существующими тестами, удаляются из набора мутантов (строка 6). Затем выполняется составление ограничения для одного или нескольких совместимых мутантов (строка 8) и решаем полученное ограничение с помощью SAT-решателя, Z3

(строка 9) [10]. В случае, если ограничение решено, происходит преобразование решения в запрос доступа, который добавляется в набор тестов (строки 10-11), в противном случае мутант считается эквивалентным. Также в данном участке алгоритма осуществляется запуск нового теста против текущего набора мутантов. Мутанты, убитые новым тестом, удаляются из набора (строка 12). Когда все операторы мутации обработаны, алгоритм возвращает Q в качестве сгенерированного набора тестов (строка 17).

Представленный алгоритм является вычислительно дорогим для политики с большим количеством правил из-за применяемой в нём оптимизации (шаги 5, 6 и 12), в процессе которой происходит генерация мутантов и их тестирование. В частности, функция $kill(M, Q)$ имеет порядок $O(n^3)$, где n – количество правил в политике. Выполнение теста с каждым мутантом, в свою очередь, имеет сложность O(n). Количество мутантов имеет линейный порядок, как и размер набора тестов. Следовательно, сложность $kill(M, Q)$ составляет порядка $O(n^3)$, потому что в худшем случае необходимо выполнять операцию со сложностью O(n) для мутантов размера O(n) для каждого запроса в тестовом наборе размера O(n).

В случае, если n становится очень большим в крупных политиках, сложность шагов оптимизации растёт в кубическом порядке, что делает его нецелесообразным для использования в больших политиках. Поэтому в контексте данной работы будет рассматриваться также неоптимизированный алгоритм (нопт-CMA), в котором шаги для оптимизации были убраны. Однако отсутствие оптимизации приведет к тому, что тестовый набор, порожденный таким алгоритмом, будет иметь значительное количество избыточных тестовых сценариев. В этом заключается компромисс между временем генерации тестов и размером тестового набора.

Сложность шага 8 (для одной итерации), то есть генерации условий ограничения ошибок, составляет порядка O(n). Временная сложность шага 9 – это временная сложность SMT-решателя Z3, требуемая для решения

выражения, описывающего УОО. Пусть O(Z3) представляет собой временную сложность SMT-решателя на шаге 9. Функция $kill(M, \{q\})$ на шаге 12 будет иметь порядок $O(n^2)$, поскольку она аналогична $kill(M, Q)$, только выполняется для одного теста q. Поскольку размер M будет иметь нелинейный порядок, временная сложность с шага 7 по шаг 14 будет иметь порядок $O(n.(O(Z3) + n^2))$. Соответственно, временная сложность неоптимизированного алгоритма будет равна $O(n.(O(Z3) + n))$. Таким образом, становится очевидным, что алгоритм CMA не подходит для практического применения при генерации тестов для крупных политик контроля доступа. Однако в дальнейшем его можно использовать в качестве эталона для измерения относительной эффективности для прочих методов тестирования. В свою очередь, нопт-CMA будет сравниваться с иными методами в ходе эксперимента.

Далее будут рассмотрены алгоритмы, используемые для построения условий обнаружения ошибок для каждого оператора мутации в разработанной модели неисправностей. Первым из операторов мутации в модели неисправностей является «изменение эффекта правила», или ИЭП [1]. На рисунке 5 приведён псевдокод, описывающий алгоритм генерации тестов для данного оператора:

Алгоритм 2 Генерация тестов для оператора ИЭП

Входные данные: Политика pol = <ЦП, КАП, [r1, r2, ..., rn]>
Выходные данные: Q - набор запросов доступа

```
1:  procedure GENERATETESTFORCRE(pol)
2:      Q ← e
3:      constraint ← PT
4:      for each rule ∈ pol do
5:          ruleConstraint ← constraint
6:          ruleConstraint ← ruleConstraint ∧ ruleReachability(pol, ri)
7:          ruleConstraint ← ruleConstraint ∧ rti ∧ rci
8:          ruleConstraint ← ruleConstraint ∧ rulePropagation(pol, ri, CRE)
9:          Q ← Q ∪ Z3request(ruleConstraint)
10:     end for
11: end procedure
12: return Q
```

Рисунок 5 – Псевдокод для генерации тестов для оператора ИЭП

На вход алгоритму подаётся политика $pol =<$ ЦП, КАП, $[r_1, r_2, \dots, r_n] >$, в свою очередь, на выход он выдаёт набор сгенерированных запросов на доступ Q. Первоначально Q приравнивается к пустому значению (строка 1). Далее для каждого правила в политике, поданной на вход, выполняются следующие операции - для ограничительного выражения используется ограничение из цели политики, в случае же его отсутствия ограничение пустое (строка 5). Следующим шагом является итерирование по каждому правилу в политике (строка 4) с целью построения УОО для того, чтобы убить мутанта. Для каждого правила ограничение устанавливается таким же (строка 5), как ограничение, заданное в цели политики на второй строке алгоритма. Затем, для каждого правила в политике выполняется конкатенация ограничения правила с ограничением достижимости (строка 6). После выполнения ограничения достижимости, следующей задачей является объединение ограничения правила с ограничением необходимости (строка 7), а затем к полученному результату добавляется ограничение распространения (строка 8). Построенное ограничение правила является достаточным, чтобы убить мутанта типа ИЭП, поэтому полученное выражение передается в SAT-решатель для получения значений входных данных. Когда все правила

обработаны внутри цикла, функция возвращает набор входных данных Q. Дополнительные методы, использующиеся в данном алгоритме, представлены в приложениях Б и В соответственно. Коротко рассмотрим данные методы.

Для метода ruleReachability (приложение Б): в случае, если комбинационный алгоритм правил является Permit-Unless-Deny, то все предыдущие правила с запретительным эффектом становятся неприменимыми (строки 4-9). В случае с Deny-Unless-Permit выполняется аналогичная операция, только для разрешительного эффекта. При Deny-Overrides / Permit-Overrides выполняется отрицание только для запретительных / разрешительных правил с общими атрибутами в теле правила, а также отмечаются запретительные / разрешительные правила, не имеющие общих атрибутов, которые в дальнейшем используются в ограничении достижимости (строки 16-25 и строки 26-36).

В методе rulePropagation (приложение В) выполняются аналогичные операции с методом ruleReachability, но для использования запретительных / разрешительных правил в ограничении распространения. Если правило i имеет запретительный (или разрешающий) эффект, оно удовлетворяет ограничению распространения, и выполняется возвращение ограничения, в противном случае, если эффект разрешающий (или запретительный) и используемый оператор мутации не является CRE (ИЭП), выполняется проход циклом по набору правил, не имеющих общих атрибутов, с целью удостовериться в том, что отрицание рассматриваемого правила в итерации не приведет к срабатыванию другого правила. Если это всё же приводит к срабатыванию, то его эффект оценивается как неопределенный и для рассматриваемого правила устанавливается соответствующий флаг, dominantIndeterminateFlag, со значением true. В случае, если рассматриваемое правило оценивается с неопределенным эффектом, для обнаружения ошибки все разрешающие (запрещающие) правила должны оцениваться аналогичным образом.

Следующим оператором, для которого рассматривается алгоритм генерации тестов, является ЦПИ («цель правила истина»). Листинг алгоритма представлен на рисунке 6:

Алгоритм 5 Генерация тестов для оператора ЦПИ

Входные данные: Политика pol = <ЦП, КАП, [r1, r2, ..., rn]>
Выходные данные: Q - набор запросов доступа

```
1: procedure GENERATETESTFORRTT(pol)
2:     Q ← e
3:     dominantIndeterminateFlag ← false
4:     dominantRuleCollection ← null
5:     constraint ← PT
6:     for each rule ∈ pol do
7:         ruleConstraint ← constraint
8:         ruleConstraint ← ruleConstraint ∧ ruleReachability(pol, ri)
9:         ruleConstraint ← ruleConstraint ∧ ¬rti ∧ (rci ∨ Error(rci))
10:        ruleConstraint ← ruleConstraint ∧ rulePropagation(pol, ri, RTT)
11:        Q ← Q ∪ Z3request(ruleConstraint)
12:    end for
13: end procedure
14: return Q
```

Рисунок 6 – Псевдокод для генерации тестов для оператора ЦПИ.

Как и в случае с оператором ИЭП, в процессе выполнения алгоритма конструируются условия обнаружения ошибок для того, чтобы убить мутанта ЦПИ путём объединения ограничений достижимости, необходимости и распространения. Ограничение необходимости для ЦПИ заключается в том, что цель рассматриваемого правила должна вычисляться как N/A, а условие правила должно быть истинным.

Выводы и результаты по главе 2

- Для описания семантических ошибок, которые могут быть допущены при разработке политик контроля доступа, была сформулирована модель неисправностей;
- В ходе рассмотрения особенностей тестирования XACML политик были рассмотрены условия обнаружения ошибок, являющиеся одной из важных составляющих мутационного анализа;
- На основе вышеизложенных положений были описаны ограничения в виде логических выражений, которым должен удовлетворять тест для обнаружения той или иной неисправности, на примере нескольких операторов мутации, а также доказана корректность их формулировки;
- Дано описание алгоритма генерации тестов на основе сильного мутационного тестирования, а также рассмотрена его неоптимизированная версия с большей скоростью генерации сценариев.

Глава 3 Постановка эксперимента и анализ результатов

3.1 Пререквизиты для выполнения эксперимента

В данной главе описывается выполнение эксперимента, предназначенного для оценки эффективности существующих методов тестирования по сравнению с предложенным в работе подходом. Поскольку индикатор оценки мутации является общеупотребимым индикатором эффективности тестового метода, предлагается измерять эффективность путём соотнесения количества мутаций с размером тестового набора, то есть оценивать количество мутантов, убитых при данном размере тестового набора. Кроме того, размер тестового набора отражает среднее время выполнения теста. Поэтому отдельно будет рассмотрено и время генерации тестов, чтобы отразить эффективность методов тестирования в ходе выполнения.

Генерация тестов на основе мутаций, а также все методы генерации тестов на основе покрытия, рассмотренные в данной работе, были имплементированы с помощью инструмента с открытым исходным кодом – XPA (анализатор XACML политик). Данный инструмент, в свою очередь, основывается на Balana – имплементации XACML 3.0 с открытым исходным кодом. Эксперимент выполнялся на 64-битном ноутбуке с ОС Windows 11, с процессором AMD Ryzen 5 4600H (тактовая частота 3.0 ГГц) и 16 ГБ оперативной памяти DDR4. В ходе эксперимента использовались тестовые политики стандарта XACML 3.0 с различным уровнем сложности. Характеристика данных политик представлена в таблице 9:

Таблица 9 – Политики, используемые в эксперименте

Наименование политики	**КСК**	**Кол-во правил**	**Число эквивалентных мутантов**	**Число неэквивалентных мутантов**
Conference	228	15	1	91
Fedora-rule3	226	12	1	87
K-Market-blue	84	4	1	27
K-Market-silver	58	3	1	21
K-Market-gold	106	5	1	32
Sample	152	6	1	55
Sample-dup	80	4	1	33
Fedora-rule3-2	558	32	1	207
Fedora-rule3-3	2748	212	1	927
iTrust	1282	64	4	450
iTrust-5	6402	320	4	2242
iTrust-10	12806	640	4	4482
iTrust-20	25602	1280	4	8962

Политика K-Market является образцом политики из Balana. Остальные из представленных в таблице политик были взяты из литературных источников. В частности, iTrust-n и fedora-rule3-n были синтезированы из политик iTrust и fedora соответственно для изучения политик большого объёма. Понятие КСК в таблице 9 представляет собой количество строк кода в соответствующей политике. Отдельно необходимо отметить, что число эквивалентных и неэквивалентных мутантов для политик в таблице 9 указано при использовании метода нопт-CMA.

Эксперимент включает в себя генерацию мутантов для каждой политики, используя мутационные операторы из модели неисправностей. Каждый оператор мутации может сгенерировать несколько мутантов для заданной политики. Например, для политики с n правил, мутационный оператор «изменение эффекта правила» создает n мутантов, поскольку мутант создается путём изменения эффекта каждого правила в политике. Следующей задачей в ходе эксперимента являлось определение количества эквивалентных и неэквивалентных мутантов. Поскольку при генерации тестов использовалась в том числе сильная мутация (удовлетворяющая всем трём условиям обнаружения ошибок), мутанты, убитые таким набором, называются

неэквивалентными, в то время как выжившие являются эквивалентными. Следовательно, набор тестов, сгенерированный с помощью сильной мутации, запускался против созданных мутантов, чтобы определить среди них эквивалентные и неэквивалентные. Далее в процессе эксперимента было необходимо создать наборы тестов для каждого из методов генерации тестов для каждой из рассматриваемых политик, после чего запустить полученные наборы для каждой политики и сохранить фактический результат по каждому тесту. Каждый такой результат является правильным ответом политики для отдельно взятого тестового сценария. Следовательно, он может быть использован в качестве тестового оракула в дальнейшем, когда тесты будут выполняться для мутированных версий политик. Поскольку мутанты представляют собой ошибки, которые могут возникнуть в XACML политиках, индикатор оценки мутации может считаться показателем способности обнаружения неисправностей, который в свою очередь используется сообществом тестирования ПО.

3.2 Анализ результатов эксперимента

3.2.1 Размер тестового набора и время генерации тестов

В таблице 10 представлено количество тестов, сгенерированных для каждой политики используемыми методами тестирования. Как правило, набор тестов для нопт-CMA имеет большее количество тестов. Это обусловлено тем, что в нём не применяется оптимизация, предназначенная для сокращения количества избыточных тестов, благодаря чему время генерации тестов, которое будет рассмотрено позднее, является меньшим в сравнении с CMA. Размер тестового набора для MC/DC и CMA при этом примерно одинаковы. Размер каждого набора тестов для метода RC равен количеству правил в политике, и он всегда является наименьшим.

Таблица 10 – Количество тестов, сгенерированных различными методами

Политика	**RC**	**DC**	**MC/DC**	**CMA**	**нопт-CMA**
Conference	15	15	25	25	78
Fedora-rule3	12	19	30	23	62
kMarket-blue-policy	4	7	11	7	21
kMarket-gold-policy	3	6	9	5	16
kMarket-silver-policy	5	9	13	8	23
Sample	6	13	21	16	51
Sample-dup	4	7	12	8	23
Fedora-rule3-2	32	39	70	62	162
Fedora-rule3-3	212	219	250	242	702
iTrust3	64	65	197	196	387
iTrust3-5	320	321	983	982	1923
iTrust3-10	640	641	1965	1964	3843
iTrust3-20	1280	1281	3929	-	7683

В таблице 11, в свою очередь, рассматривается время генерации тестов для каждого из методов по каждой представленной политике. Соответственно, метод RC является наиболее быстрым из представленных, поскольку он генерирует меньшее число тестов. В целом, время генерации тестов пропорционально размеру тестового набора, за исключением CMA, поскольку в нём применяется вышеупомянутая оптимизация. Следовательно, все представленные методы за исключением CMA могут быть масштабированы. Поскольку время генерации тестов для политики iTrust3-10 составило около полутора дней, было решено не запускать CMA для политики iTrust3-20, поскольку приблизительное время генерации составило бы порядка десяти дней, и это доказывает отсутствие возможности масштабирования для CMA. При этом, для политик, количество правил в которых составляет менее ста, время генерации тестов у CMA сопоставимо с другими тестовыми методами.

Таблица 11 – Время генерации тестов (в миллисекундах)

Политика	**RC**	**DC**	**MC/DC**	**CMA**	**нопт-CMA**
Conference	488	408	716	2847	2019
Fedora-rule3	328	686	536	2222	1628
kMarket-blue-policy	96	252	307	467	519
kMarket-gold-policy	71	221	221	343	385
kMarket-silver-policy	130	328	335	704	626
Sample	174	506	557	1835	1309
Sample-dup	117	270	323	669	593
Fedora-rule3-2	847	1290	1935	14853	4534
Fedora-rule3-3	5368	8120	10304	930226	22625
iTrust3	1644	3511	8349	171716	10363
iTrust3-5	8711	24415	64749	15875846	48091
iTrust3-10	13890	84304	211230	129655675	176758
iTrust3-20	45908	562857	696428	-	709927

3.2.2 Оценка способности обнаружения неисправностей

В таблице 12 представлен индикатор оценки мутации для каждого тестового метода в отношении представленных политик. Поскольку оценка мутации для CMA и нопт-CMA составила 100% для всех политик, они не были включены в таблицу. При этом следует отметить, что для наибольшей по размеру политики, iTrust3-20, CMA не запускался. Однако для нопт-CMA известно, что для каждого мутанта существует как минимум один тест, который его убивает. Поэтому в данном случае оценка и составляет 100%.

Таблица 12 – Значения индикатора оценки мутации

Политика	**RC**	**DC**	**MC/DC**
Conference	75.82	78.02	100
Fedora-rule3	64.37	85.06	88.51
kMarket-blue-policy	81.48	96.3	100

Продолжение таблицы 12

kMarket-gold-policy	80.95	95.24	95.24
kMarket-silver-policy	81.25	100	100
Sample	69.09	90.91	96.36
Sample-dup	48.48	84.85	96.97
Fedora-rule3-2	36.71	55.07	56.52
Fedora-rule3-3	27.62	51.13	51.46
iTrust3	42.67	58.22	100
iTrust3-5	42.82	57.36	100
iTrust3-10	42.83	57.25	100
iTrust3-20	42.84	57.19	100

В таблице 13 продемонстрированы виды мутаций, которые не способен убить тот или иной тестовый метод. Для удобства, результаты для трёх политик kMarket-n были объединены в одну строку. Также политики iTrust3 и fedora-rule3-2 могут выступать как обобщающие для других политик из группы iTrust-n и fedora-rule-3-n, поэтому только они были включены в таблицу. Отсутствие оператора мутации в таблице означает, что все мутанты данного типа были убиты рассматриваемым методом, либо в случае, если оператор мутации не применим к политике. Оператор мутации, не отмеченный восклицательным знаком, подразумевает, что все мутанты данного типа выжили в результате тестов. Оператор мутации с одним восклицательным знаком означает, что было убито более 50% мутантов данного типа, в то время как с двумя восклицательными знаками – менее 50%.

Таблица 13 – Типы выживших мутантов после применения тестовых методов

Политика	**RC**	**DC**	**MC/DC**
Sample	ЦпрИ (!!) УпрИ ЦпоЛ ИКА (!!)	ЦпрИ (!)	ЦпрИ (!)

Продолжение таблицы 13

Sample-dup	ИЭП (!) ЦпрИ ЦпрЛ (!) УпрИ (!) УдПр (!) ИКА	ИЭП (!) ЦпрЛ (!) УдПр (!)	УпрИ
Conference3	ИКА (!!)		
Fedora-rule3-2	ЦпрИ (!!) ЦпрЛ (!!) УпрИ (!!) УдПр (!!) ИКА	ЦпрЛ (!!) УдПр (!!)	ЦпрЛ (!!) УдПр (!!)
kMarket	ЦпрИ (!!) УпрИ (!) ЦпоИ ИКА (!!)	ЦпрИ (!)	ЦпрИ (!)
iTrust	ЦпрИ ИКА		

Индикатор оценки мутации увеличивается для методов слева направо, что можно наблюдать в таблице 12, в то же время количество операторов, для которых не убиваются все мутанты, уменьшается по тому же шаблону. Для метода RC индикатор варьируется между 27% и 81% в зависимости от политик, и в таблице 13 заметно, что он не способен выявить значительную часть мутантов из модели неисправностей. Для методов DC и MC/DC индикатор ранжировался в пределах от 51% до 100%. Тем не менее, это не означает, что данные методы имеют схожую способность обнаружения ошибок. Далее, таблица 10 показывает, что методы MC/DC и CMA имеют сравнимый размер тестовых наборов, а в таблице 11 видно, что время генерации тестов для метода MC/DC намного меньше, чем у CMA. Из этого можно было бы сделать вывод, что метод MC/DC является более эффективным ввиду сравнительно высоких значений индикатора оценки мутации и меньшей по времени генерации тестов в сравнении с CMA. Однако метод MC/DC не удовлетворяет третьему условию обнаружения ошибок, а именно условию

распространения. В результате, при увеличении размера политики, как в fedora-rule3, значения индикатора снижаются с 88% до 51%.

Несмотря на то, что в большинстве политик метод MC/DC достигает высоких показателей индикатора оценки мутации, его применение не может гарантировать высокой надежности XACML политик, поскольку он удовлетворяет только ограничениям достижимости и необходимости. При этом, для большей части операторов в модели неисправностей он удовлетворяет двум из трёх условий обнаружения ошибок, что позволяет квалифицировать данный метод как близкий по эффективности к слабому мутационному тестированию. При этом методы RC и DC, как видно из таблицы 13, далеки от достижения условия необходимости для многих из ошибок, представленных в модели неисправностей.

3.2.3 Оценка эффективности рассматриваемых методов

В то время, как индикатор оценки мутации является хорошим показателем способности обнаружения ошибок у тестовых методов, он не учитывает эффективность сгенерированного тестового набора с точки зрения количества мутантов, убитых каждым тестом в данном тестовом наборе. В качестве индикатора эффективности рассматривается такой показатель, как количество мутантов, убитых одним тестом (сокращенно КМУТ). При этом он является усредненным показателем, то есть высчитывается не для каждого отдельно взятого теста, а путём деления количества мутантов на число тестов. В таблице 14 представлены значения индикатора КМУТ для каждой из рассматриваемых политик:

Таблица 14 – Значения индикатора «количество мутантов, убитых одним тестом»

Политика	**RC**	**DC**	**MC/DC**	**CMA**	**нопт-CMA**
Conference	4.6	4.44	3.64	3.64	1.17
Fedora-rule3	4.67	2.96	2.57	3.78	1.4

Продолжение таблицы 14

kMarket-blue-policy	5.5	2.6	2.45	3.86	1.29
kMarket-gold-policy	5.67	2.22	2.22	4.2	1.31
kMarket-silver-policy	5.2	2.46	2.46	4	1.39
Sample	6.33	2.63	2.52	3.44	1.08
Sample-dup	4	2.8	2.67	4.12	1.43
Fedora-rule3-2	2.38	2.53	1.67	3.34	1.28
Fedora-rule3-3	1.21	2.11	1.91	3.83	1.32
iTrust3	3	3.97	2.28	2.3	1.16
iTrust3-5	3	3.99	2.28	2.28	1.17
iTrust3-10	3	3.996	2.28	2.28	1.16
iTrust3-20	3	3.998	2.28	-	1.16

Исходя из данных в представленной таблице, метод RC достигает наибольших значений индикатора КМУТ в сравнении с остальными методами. Тем не менее, это не означает, что он наиболее эффективен, поскольку в политиках большего объема его эффективность резко снижается. Метод нопт-CMA, в свою очередь, хоть и способен достичь максимальных значений индикатора оценки мутации, но при этом имеет наименьший КМУТ для всех политик. Метод DC имеет большее значение КМУТ, чем у метода MC/DC. В то же время методы MC/DC и CMA имеют сходные показатели по данному индикатору. В некоторых политиках метод DC имеет ощутимо большие значения КМУТ в сравнении с методом CMA, однако опираться только на этот индикатор при оценке видится нецелесообразным, поскольку значение индикатора оценки мутации у DC ощутимо ниже – это, в свою очередь, не позволяет сделать вывод о том, что данный метод может обеспечить высокий уровень надежности XACML политик.

Выводы и результаты по главе 3

– в ходе эксперимента было проведено сравнение существующих методов верификации политик с методом, разработанным в данном исследовании;

– сравнение выполнялось с помощью нескольких метрик, а именно - индикатора оценки мутации и количества мутантов, убитых одним тестом;

– результаты эксперимента показали, что неоптимизированная версия алгоритма работает наравне со сравниваемыми методами, а для некоторых политик его эффективность оказалась выше;

– таким образом, подтверждается гипотеза исследования - оценка существующих методов тестирования продемонстрировала, что наиболее эффективно работает метод, основываемый на сильной мутации, однако при этом существует компромисс между временем генерации тестов и способностью тестов обнаруживать неисправности.

Заключение

В результате выполнения данной работы был представлен подход к сильному мутационному тестированию политик стандарта XACML 3.0. Кроме того, на основе модели неисправностей были сформулированы условия обнаружения ошибок. С одной стороны, данные условия позволили сгенерировать тестовые наборы для рассмотренных политик. С другой стороны, в ходе эксперимента с их помощью стало возможным оценить способность обнаружения ошибок у существующих тестовых методов. Кроме того, рассмотренные в данной работе условия ограничения ошибок не ограничиваются существующими операторами мутации. Потенциально они обеспечивают теоретическую основу для разработки новых методов и работы с неисправностями, создаваемыми новыми операторами мутации.

Далее был разработан подход для генерации набора тестов, с помощью которого были проведено сравнение эффективности методов тестирования XACML политик. Результаты эксперимента продемонстрировали, что метод MC/DC по эффективности, а также по индикатору оценки мутации, приближен к методу сильного мутационного анализа (CMA). CMA, хотя и имеет 100% значение оценки мутации, видится неприменимым на практике для больших политик вследствие значительного количества времени, требуемого на генерацию тестов. В то же время остальные рассмотренные в ходе эксперимента методы, будучи масштабируемыми, не могут во всех случаях обеспечить высоких показателей обнаружения ошибок. Следовательно, если необходимо максимально возможное обнаружение ошибок, и размер тестового набора не играет большого значения, возможно использовать набор тестов, генерируемый методом неоптимизированного CMA. В этом видится компромисс между способностью тестов обнаруживать неисправности, размером тестового набора и временем генерации тестов.

Список используемой литературы

1. Ячевский В.А. Построение модели отказов для применения мутационного анализа к политикам контроля доступа // Прикладная математика и информатика: современные исследования в области естественных и технических наук (Тольятти, 20-22 апреля 2022 года) – Тольятти : Изд-во ТГУ, 2022, стр. 55-58.

2. Ячевский В.А. Формализация условий обнаружения ошибок для XACML-политик // МОЛОДЕЖЬ. НАУКА. ОБЩЕСТВО – 2021 (Тольятти, 20-24 декабря 2021 года) – Тольятти : Изд-во ТГУ, 2021, стр. 277-279.

3. Котенко И. В., Левшун Д. С., Саенко И. Б. Верификация политик разграничения доступа на основе атрибутов в облачных инфраструктурах с помощью метода проверки на модели // Системы управления, связи и безопасности №4. 2019. URL: https://sccs.intelgr.com/archive/2019-04/17-Kotenko.pdf (дата обращения: 03.12.2022)

4. Саенко И.Б., Комашинский В.И., Левшун Д.С. Алгоритмы верификации политик разграничения доступа на основе модели ABAC // Перспективные направления развития отечественных информационных технологий, 2019. URL: https://www.elibrary.ru/item.asp?id=42944623 (дата обращения: 27.11.2022)

5. Aqib M., Shaikh R.A. Analysis and Comparison of Access Control Policies Validation Mechanisms // I. J. Computer Network and Information Security, 2015. URL: https://www.mecs-press.org/ijcnis/ijcnis-v7-n1/IJCNIS-V7-N1-8.pdf (дата обращения: 27.11.2022)

6. Bertolino A, Daoudagh S, Lonetti F, Marchetti E. Automatic XACML requests generation for policy testing // InSoftware Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference pp. 842-849.

7. Bertolino A, Le Traon Y, Lonetti F, Marchetti E, Mouelhi T. Coverage-based test cases selection for XACML policies // InSoftware Testing, Verification

and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference pp. 12-21.

8. Braghin C., Sharygina N., Barone-Adesi K. A model checking-based approach for security policy verification of mobile systems // Formal aspects of Computing, Springer Verlag. URL: https://hal.archives-ouvertes.fr/hal-00618198/document (дата обращения: 03.12.2022)

9. Cherneva G., Khalimov P. Mutation Testing of Access Control Policies // Advanced Information Systems. 2021. Vol. 5, No. 1. URL: http://ais.khpi.edu.ua/article/view/226846/226393 (дата обращения: 05.12.2022)

10. De Moura L., Bjorner N. Z3: An efficient SMT Solver [Электронный ресурс] // URL: https://researchgate.net/publication/225142568_Z3_an_efficient_SMT_solver

11. Fisler K., Krishnamurthi S., Meyerovich L. A. Verification and change-impact analysis of access-control policies // Proceedings of the 27th international conference on Software engineering, стр. 196-205.

12. Gouglidis A., Mavridis I., Hu V.C. Security policy verification for multi-domains in cloud systems [Электронный ресурс] // URL: https://infosec.uom.gr/wp-content/uploads/2013/07/IJIS-D-12-00249.pdf (дата обращения: 12.11.2022)

13. Hu V.C., Kuhn R., Ferraiolo D. F. Attribute-Based Access Control // IEEE Computer Society, February 2015. URL: https://www.profsandhu.com/cs5323_s18/Hu-2015.pdf (дата обращения: 12.11.2022)

14. Hu V.C., Kuhn R. Model Checking for Verification of Mandatory Access Control Models and Properties // International Journal of Software Engineering and Knowledge Engineering. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=903228 (дата обращения: 03.12.2022)

15. Hu V.C., Kuhn R. Property Verification for Generic Access Control Models // IEEE/IDIP International Conference on Embedded and Ubiquitous Computing. URL: https://ieeexplore.ieee.org/document/4755235 (дата обращения: 03.12.2022)

16. Hu V.C., Kuhn R., Yaga D. Verification and Test Methods for Access Control Policies/Model // NIST Special Publication 800-192. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-192.pdf (дата обращения: 05.12.2022)

17. Hu V.C., Kuhn R. Vulnerability Hierarchies in Access Control Configurations // 4th Symposium on Configuration Analytics and Automation. URL: https://csrc.nist.gov/CSRC/media/Publications/conference-paper/2011/12/27/vulnerability-hierarchies-in-access-control-configurations/documents/kuhn-safeconfig2011.pdf (дата обращения: 27.11.2022)

18. Hu V.C., Scarfone K. Guidelines for Access Control System Evaluation Metrics [Электронный ресурс] // URL: https://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7874.pdf (дата обращения: 11.12.2022)

19. Hu V.C., Scarfone K. Real-Time Access Control Rule Fault Detection Using a Simulated Logic Circuit [Электронный ресурс] // URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=915399 (дата обращения: 03.12.2022)

20. Hwang J.H., Xie T. ACPT: A Tool for Modeling and Verifying Access Control Policies [Электронный ресурс] // URL: http://taoxie.cs.illinois.edu/publications/policy10-demo.pdf (дата обращения: 03.12.2022)

21. Kikuchi S., Tsuchiya S. Policy Verification and Validation Framework Based on Model Checking Approach // Fourth International Conference on Autonomic Computing (ICAC). URL: https://ieeexplore.ieee.org/document/4273095 (дата обращения: 03.12.2022)

22. Kuhn R., Hu V.C., Ferraiolo D. Pseudo-exhaustive Testing of Attribute Based Access Control Rules [Электронный ресурс] // URL: https://csrc.nist.gov/CSRC/media/Presentations/Pseudo-exhaustive-Testing-of-Attribute-Based-Acces/images-media/abac-pseudo-ex-iwct.pdf

23. Li Y, Li Y, Wang L, Chen G. Automatic XACML requests generation for testing access control policies // SEKE, 2014. pp. 217-222.

24. Lin D., Rao P., Bertino E. EXAM – a Comprehensive Environment for the Analysis of Access Control Policies // CERIAS Tech Report 2008-13. URL: https://www.cerias.purdue.edu/assets/pdf/bibtex_archive/2008-13.pdf (дата обращения: 03.12.2022)

25. OASIS XACML 3.0 Specification [Электронный ресурс] // OASIS Standard. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html (дата обращения: 27.11.2022)

26. Saghafi S., Nelson T., Dougherty D.J. Geometric Logic for Policy Analysis // Department of Computer Science, Worcester Polytechnic Institute. URL: http://web.cs.wpi.edu/~tn/publications/snd-arsec13-geometric.pdf (дата обращения: 03.12.2022)

27. Servos D., Osborn S.L. Current Research and Open Problems in Attribute-Based Access Control [Электронный ресурс] // URL: https://cutt.ly/ER6ALcI (дата обращения: 12.11.2022)

28. Schaad A., Lotz V., Sohr K., “A model-checking approach to analysing organisational controls in a loan origination process”, in Proc ACM Symposium on Access Control Models and Technologies, pages 139–149, 2006.

29. Xie T., Martin E., Yu T. Defining and Measuring Policy Coverage in Testing Access Control policies [Электронный ресурс] // URL: https://link.springer.com/content/pdf/10.1007/11935308_11.pdf (дата обращения: 11.12.2022)

30. Xie T., Hu V.C. Paradigm in Verification of Access Control [Электронный ресурс] // URL: https://www.academia.edu/2693900/Paradigm_in_Verification_of_Access_Control (дата обращения: 11.12.2022)

Приложение А

Скомбинированное ограничение для обнаружения ошибки типа «ЦПрИ»

$$
\begin{aligned}
&\{(PT) \wedge \neg\, rt_i \wedge \mathrm{r_i} = <\mathrm{rb_i}, \mathrm{Deny}> \wedge \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j \text{ при } j\,!=i\} \wedge \\
&\wedge \{\neg(\exists(p,d) \ni (i \neq p \neq d) \wedge (\mathrm{r_d} = <rb_d, Deny>) \wedge \\
&\wedge (\mathrm{r_p} = <rb_p, Permit>) \wedge \mathrm{Error(rb_p)} \wedge (\mathrm{Error(rb_d)} \vee \mathrm{rb_d}))\}) \\
&\vee \\
&\vee \{(PT) \wedge \neg\, rt_i \wedge \mathrm{r_i} = <\mathrm{rb_i}, \mathrm{Deny}> \wedge \\
&\wedge \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j = <rb_i, Permit> \text{ при } j\,!=i\} \wedge \\
&\wedge \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j = <rb_i, Deny> \text{ при } j\,!=i\} \wedge \\
&\wedge \{\neg(\exists(p,d) \ni (i \neq p \neq d) \wedge (\mathrm{r_d} = <rb_d, Deny>) \wedge \\
&\wedge (\mathrm{r_p} = <rb_p, Permit>) \wedge \mathrm{Error(rb_p)} \wedge (\mathrm{Error(rb_d)} \vee \mathrm{rb_d}))\}) \\
&\vee \\
&\vee \{(PT) \wedge \neg\, rt_i \wedge \mathrm{r_i} = <\mathrm{rb_i}, \mathrm{Permit}> \wedge \\
&\wedge \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j = <rb_i, Permit> \text{ при } j\,!=i\} \\
&\vee \\
&\vee (\{Error(PT)\} \wedge \neg\, rt_i \wedge \mathrm{r_i} = <\mathrm{rb_i}, \mathrm{Permit}> \wedge \\
&\wedge [\{\neg\, rb_j\ \forall\, r_j \text{ при } j\,!=i\} \vee \{\exists d(rb_d \vee \mathrm{Error(rb_d)}) \wedge (\mathrm{r_d} =< \mathrm{rb_d}, \mathrm{Deny}>) \wedge \\
&\wedge \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j = <rb_i, Permit> \text{ при } j\,!=i\}]) \\
&\vee \\
&\vee \{(PT) \wedge \mathrm{Error(rt_i)} \wedge \mathrm{r_i} = <rb_i, Deny> \wedge \\
&\wedge \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j = <rb_i, Deny> \text{ при } j\,!=i\} \wedge \\
&\wedge \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j = <rb_i, Deny> \text{ при } j\,!=i\} \\
&\vee \\
&\vee (\{(PT)\} \wedge \mathrm{Error(rt_i)} \wedge \mathrm{r_i} = <rb_i, Permit> \wedge \\
&\wedge \{\neg\, rb_j\ \forall\, r_j = <rb_i, Permit> \text{ при } j\,!=i\}) \\
&\vee \\
&\vee (\{Error(PT)\} \wedge \mathrm{Error(rt_i)} \wedge \mathrm{r_i} = <\mathrm{rb_i}, \mathrm{Permit}> \wedge \\
&\wedge \{\neg\, rb_j\ \ \forall\, r_j \text{ при } j\,!=i\} \vee \{\exists d(rb_d \vee \mathrm{Error(rb_d)}) \wedge (\mathrm{r_d} =< \mathrm{rb_d}, \mathrm{Deny}>) \wedge \\
&\wedge \{\neg\, rb_j \vee Error(rb_j)\ \forall\, r_j = <rb_i, Permit> \text{ при } j\,!=i\}
\end{aligned}
$$

Рисунок А.1 – Логическое выражение, описывающее скомбинированное ограничение обнаружения ошибок

Приложение Б

Описание метода ruleReachability для алгоритма генерации тестов

Алгоритм 3 Метод ruleReachability, используемый в алгоритме генерации тестов для мутанта ИЭП

Импортируемые функции: hasCommonAttribute(rbi, rbj) - возвращает true, если правила i и j имеют общий атрибут

Входные данные: Политика pol = <ЦП, КАП, [r1, r2, ..., rn]>, rule = <rti, rci, rei>

Выходные данные: constr - ограничение

```
1:  procedure RULEREACHABILITY(pol, rule)
2:      constr ← e
3:      constraint ← PT
4:      if RCA = PermitUnlessDeny then
5:          for each rule ∈ pol do
6:              if rek = Deny then
7:                  constraint ← constraint ∧ ¬(rtk ∧ rck)
8:              end if
9:          end for
10:     else if RCA = DenyUnlessPermit then
11:         for each rule ∈ pol do
12:             if rek = Permit then
13:                 constraint ← constraint ∧ ¬(rtk ∧ rck)
14:             end if
15:         end for
16:     else if RCA = DenyOverrides then
17:         for each rule ∈ pol do
18:             if rek = Deny then
19:                 if hasCommonAttributes(rbi, rbk) then
20:                     constraint ← constraint ∧ ¬(rtk ∧ rck)
21:                 else
22:                     dominantRuleCollection.add(rk)
23:                 end if
24:             end if
25:         end for
26:     else if RCA = PermitOverrides then
27:         for each rule ∈ pol do
28:             if rek = Permit then
29:                 if hasCommonAttributes(rbi, rbk) then
30:                     constraint ← constraint ∧ ¬(rtk ∧ rck)
31:                 else
32:                     dominantRuleCollection.add(rk)
33:                 end if
34:             end if
35:         end for
36:     end if
37: end procedure
38: return Q
```

Рисунок Б.1 – Псевдокод, описывающий метод ruleReachability

Приложение В

Описание метода rulePropagation для алгоритма генерации тестов

Алгоритм 4 Метод rulePropagation, используемый в алгоритме генерации тестов для мутанта ИЭП

Входные данные: Политика pol = <ЦП, КАП, [r1, r2, ..., rn]>, правило rule = <rti, rci, rei>

Выходные данные: constr - ограничение

```
1:  procedure RULEPROPAGATION(pol, rule, mutationMethod)
2:      constr ← e
3:      constraint ← PT
4:      if RCA = PermitUnlessDeny then
5:          for each rule ∈ pol do
6:              if rek = Deny then
7:                  constr ← constr ∧ ¬(rtk ∧ rck)
8:              end if
9:          end for
10:     else if RCA = DenyUnlessPermit then
11:         for each rule ∈ pol do
12:             if rek = Permit then
13:                 constr ← constr ∧ ¬(rtk ∧ rck)
14:             end if
15:         end for
16:     else if RCA = DenyOverrides then
17:         for each rule ∈ pol do
18:             if rek = Deny then
19:                 if hasCommonAttributes(rbi, rbk) then
20:                     constr ← constr ∧ ¬(rtk ∧ rck)
21:                 else
22:                     dominantRuleCollection.add(rk)
23:                 end if
24:             end if
25:         end for
26:         if rei = Permit && mutationMethod ≠ CRE then
27:             for each rule ∈ dominantRuleCollection do
28:                 if ¬(rtl ∧ rcl) → (rtp ∧ rcp)for rp =< rbp, Permit > then
29:                     dominantIndeterminateFlag ← true
30:                 else
31:                     constr ← constr ∧ ¬(rtl ∧ rcl)
32:                 end if
33:             end for
34:             if dominantIndeterminateFlag = true then
35:                 permitRules ← get permit rules of P
36:                 for each rp ∈ permitRules do
37:                     constr ← constr ∧ ¬(rtp ∧ rcp)
38:                 end for
39:             end if
40:         end if
41:     else if RCA = PermitOverrides then
42:         for each rule ∈ pol do
43:             if rek = Permit then
44:                 if hasCommonAttributes(rbi, rbk) then
45:                     constr ← constr ∧ ¬(rtk ∧ rck)
46:                 else
47:                     dominantRuleCollection.add(rk)
48:                 end if
49:             end if
50:         end for
```

Рисунок В.1 – Псевдокод, описывающий метод rulePropagation