ИТМО

Программное обеспечение высоконагруженных систем

Можаев Андрей Михайлович

Выпускная квалификационная работа

Оптимизация строковой решетки для статического анализа кода на базе абстрактной интерпретации

Уровень образования: магистратура

Научный руководитель:

—

Рецензент:

—

Санкт-Петербург

2025 г.

Содержание

Введение

Современное программное обеспечение становится все более сложным и масштабным, что увеличивает вероятность ошибок, возникающих в процессе разработки. Даже незначительные на первый взгляд ошибки могут привести к серьезным последствиям — от сбоев в пользовательских приложениях до срывов критически важных систем, таких как медицинское оборудование или системы управления транспортом. Именно поэтому проблема обеспечения надежности программных систем стоит особенно остро. Один из эффективных способов борьбы с потенциальными ошибками — применение методов статического анализа кода.

Статический анализ позволяет обнаружить широкий спектр ошибок и уязвимостей на ранней стадии разработки, еще до выполнения программы. В отличие от динамического анализа, который исследует поведение программы во время исполнения на различных входных данных, статический анализ работает только с исходным кодом или промежуточным представлением программы. Это дает возможность проверять весь возможный спектр входных данных и состояний программы, выявляя проблемы, которые могли бы проявиться лишь в редких и трудновоспроизводимых сценариях выполнения.

Одним из наиболее теоретически обоснованных и мощных подходов к статическому анализу является метод абстрактной интерпретации. Этот метод основан на идее упрощения множества всех возможных состояний программы путем перехода от конкретных значений переменных к их абстрактным представлениям в специально сконструированном математическом пространстве — абстрактном домене. В таком представлении каждое абстрактное значение описывает сразу множество возможных конкретных значений, а операции над абстрактными элементами являются оверапроксимацией (over-approximation) реального поведения программы.

Оверапроксимация означает, что абстрактный анализ покрывает все возможные сценарии, а также возможные, но не достижимые. Это свойство важно, поскольку оно позволяет гарантировать безопасность анализа: если статический анализ на основе абстрактной интерпретации не обнаружил оши-

бок, то можно быть уверенным, что в реальном исполнении программы аналогичных ошибок действительно не произойдет — ни при каких входных данных и сценариях выполнения. Такой подход позволяет обнаруживать целые классы потенциальных ошибок, включая обращения к неинициализированным переменным, выходы за границы массивов, деления на ноль и другие критические дефекты.

Неподвижные точки и решетки

Фундаментальным математическим понятием в абстрактной интерпретации являются неподвижные точки. При выполнении статического анализа для каждой инструкции создаётся измененное состояние, т.е. множество состояний соответствует последовательности инструкций. Однако особую сложность представляет анализ циклов и рекурсий, поскольку количество итераций заранее неизвестно и потенциально бесконечно. Полный перебор всех возможных состояний и проходов цикла невозможен на практике из-за комбинаторного взрыва.

Именно здесь применяется вычисление неподвижной точки — состояния программы, при котором дальнейшее выполнение цикла не приводит к появлению новых абстрактных состояний. В контексте абстрактной интерпретации это означает нахождение верхней границы множества возможных значений, которые могут принимать переменные после выполнения неопределенного числа итераций цикла. Такой подход позволяет "замкнуть"цикл, остановив анализ в тот момент, когда достигнута стабильность (фиксация значений) в абстрактном пространстве.

Чтобы анализ завершался за конечное время, применяется механизм ускорения сходимости — так называемый widening (расширение), который грубо обобщает накопленные результаты и ускоряет достижение неподвижной точки, жертвуя при этом частью точности ради производительности.

Вся эта процедура возможна благодаря тому, что абстрактные значения организованы в решетку — математическую структуру, определяющую отношения между абстрактными элементами и операции над ними. Решетка — это частично упорядоченное множество, в котором для любых двух элемен-

тов можно определить наименьшую общую верхнюю границу (join) и наибольшую общую нижнюю границу (meet). Эти операции позволяют объединять и пересекать абстрактные состояния, обобщать или уточнять информацию в процессе анализа.

Примером простой решетки является множество булевых значений $\bot$, true, false, $\top$, где $\bot$ — невозможное состояние, $\top$ — любое, а операции join и meet позволяют вычислять общие свойства логических выражений.

Цели и задачи

Основной проблемой является недостаточная точность строковых абстрактных доменов. В некоторых случаях анализ не может корректно определить недостижимость определенных веток исполнения кода, что приводит к ложным срабатываниям и увеличению количества потенциальных ложных ошибок. Это снижает полезность статического анализа, так как разработчики вынуждены разбираться с ложными предупреждениями.

Целью данной работы является повышение точности строковой решетки, используемой в статическом анализаторе кода на базе абстрактной интерпретации

Задачи:

- Провести обзор существующих строковых абстрактных доменов
- Реализовать более точный вариант строковой решетки на их основе
- Провести замеры точности и производительности анализатора

Обзор предметной области

Статический анализ

Статический анализ программного кода представляет собой метод исследования программ без их выполнения. Основная цель статического анализа — выявление потенциальных ошибок, уязвимостей и других характеристик программного обеспечения на основе его исходного кода или промежуточного представления [1]. Этот метод используется в компиляторах, инструментах проверки кода, системах анализа безопасности и других приложениях.

Существует несколько техник статического анализа, включая анализ потока данных, анализ потока управления, типизацию, проверку соответствия спецификациям и другие. В отличие от динамического анализа, который требует выполнения программы, статический анализ проводится на уровне исходного кода, байт-кода или абстрактного синтаксического дерева.

Основные применения

- Поиск уязвимостей
- Оптимизация кода
- Верификация свойств
- Генерация тестов

Два важных метода статического анализа — символьное исполнение и абстрактная интерпретация — позволяют анализировать поведение программ без их непосредственного запуска, но имеют разные подходы и цели.

Символьное исполнение

Символьное исполнение (Symbolic Execution) представляет собой метод анализа, при котором программа исполняется не на конкретных входных данных, а на символьных переменных [2]. В процессе исполнения строится множество путей выполнения программы, а также логические ограничения, наложенные на переменные. Этот метод используется, например, для генерации тестов с высоким покрытием и обнаружения уязвимостей.

Преимущества

- Позволяет находить ошибки, недоступные при обычном тестировании [3]
- Даёт точные результаты в пределах анализируемых путей

Ограничения

- Страдает от проблемы комбинаторного взрыва, так как количество путей выполнения растёт экспоненциально [4]
- Требует сложных SMT Solver-ов для обработки выражений (Z3 [5], CVC5 [6] и др.)
- Нет гарантии овераппроксимации, солвер не может доказать, что решения нет

Абстрактная интерпретация

Абстрактная интерпретация (Abstract Interpretation) основана на создании приближённых представлений возможных состояний программы [7]. Вместо работы с конкретными или символьными значениями, этот метод использует абстрактные значения, которые обобщают множества возможных состояний.

Преимущества

- Эффективный анализ больших программ [8]
- Гарантированная завершаемость

Ограничения

- Ложные срабатывания [9]
- Сложность разработки точных абстракций

Сравнение методов

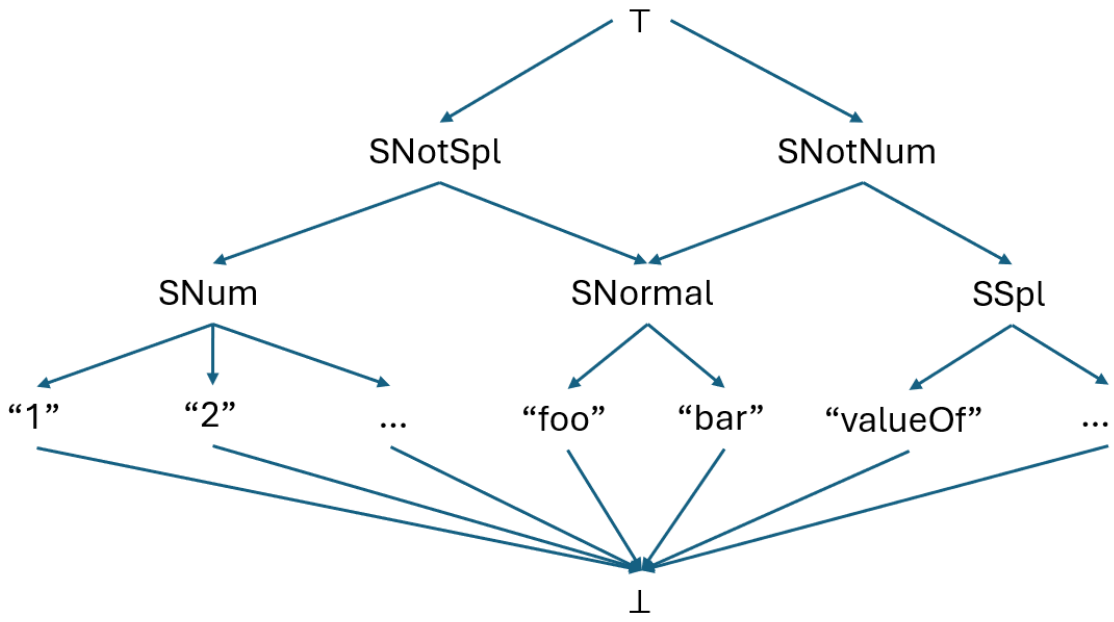
Характеристика	**Символьное исполнение**	**Абстрактная интерпретация**
Точность анализа	Высокая (по отдельным путям)	Приближённая (но глобальная)
Масштабируемость	Ограниченная (комбинаторный взрыв)	Высокая (благодаря абстракции)
Применимость	Поиск ошибок генерация тестов	Обнаружение уязвимостей верификация
Требования к вычислениям	Высокие (SAT solver)	Более низкие

Таким образом, оба метода находят своё применение в статическом анализе, а их комбинирование может дать более точные и эффективные результаты.

Строковые абстрактные домены

Пример простого строкового абстрактного домена реализован в системе **JSAI** [**?**] - платформе статического анализа JavaScript. В ней строки представляются в виде:

- любая ($\top$) и неинициализированная ($\bot$) строки
- константные строки
- категории “строка-число”, “спец-символ” и “любая строка”



Хотя этот подход обладает высокой производительностью, он страдает от недостатка точности. Например, пусть переменная s может быть равна строке “foo” или “bark”. Тогда значение выражения `s.length` может быть либо 3, либо 4, но абстрактный домен JSAI вернёт «любая длина». Это делает невозможным отбрасывание условий вроде `if (s.length == 0)` как недостижимых, что ведёт к ложным срабатываниям

Листинг 1: Пример недостаточной точности в строковом домене JSAI

```
let s;
if (input == 0) {
    s = "foo";
} else {
    s = "bark";
}
if (s.length == 0) {
    throw new Error("Divide by zero!");
}
let repeats = 100 / str.length;
```

В статьях чаще всего упоминаются ещё 4 строковых решетки

The String Set

Домен множества строк (SS_k) сохраняет не более k константных строк. В случае переполнения сбрасывается в $\top$. Этот домен заметно более затратен по времени и памяти. Достаточно просто рассмотреть функцию конкатенации двух абстрактных значений, при которой количество элементов будет произведением. А конкатенация самая часто использующаяся функция на строках, потому от её оптимизации будет многое зависеть

Точность этой решетки очень высокая, так как каждую операцию можно отдельно проделать с каждой строкой в множестве. Но из-за этого и производительность сильно проседает. Плюс теряется вся точность для бесконечного набора возможных строк, который может возникнуть например в цикле при конкатенации

The Abstract Length string domain

Домен длины абстрактной строки (LS) запоминает минимальную и максимальную длину строк, которые может принимать абстрактное значение. Например

$$LS(”foo”;”bark”) = [3, 4]$$

Для примера выше она отлично подойдет, так как запомнит, что минимальная длина 3, а максимальная 4, и даже сможет верно вернуть значение для repeats, то есть интервал [25, 34]. Однако для других операций, таких как substr или charAt, она не годится

The Character Inclusion domain

Домен включения символов (CI) отслеживает символы, встречающиеся в строке. Каждая абстрактная строка имеет вид [L, U]. Нижняя граница L содержит символы, которые должны встречаться в конкретной строке (строках), в то время как верхняя граница U представляет символы, которые могут появиться

Этот домен полностью игнорирует структуру конкретных строк, которые аппроксимирует. То есть например

$$CI(\text{"}foo\text{"}) = CI(\text{"}of\text{"}) = [\{'f', 'o'\}, \{'f', 'o'\}]$$

Но CI, как правило, дешев в вычислительном отношении и иногда предоставляет полезную информацию. Например функция contains, для которой можно вернуть true в случае проверки на включение одной буквы, которая содержится в нижней границе, или же false, если слово содержит букву, которой нет в верхней

The Prefix-Suffix domain

Элементами префикс-суффикс домена (PS) являются пары $[p, s]$, содержащие в себе все строки, которые начинаются с p и заканчиваются на s

$$PS(\text{"}abacab\text{"};\text{"}abab\text{"}) = [\text{"}aba\text{"};\text{"}ab\text{"}]$$

Также как и CI, PS не может хранить конкретные строки. Но есть ряд функций, такие как startsWith, endsWith, а иногда и substr, indexOf, для которых будет возвращаться точный ответ в удачных случаях

Конечные автоматы

Все перечисленные сверху решетки хороши каждая для своего ограниченного набора функций. Но в общем случае будет много неточностей. Поэтому стоит рассмотреть другие подходы, обладающие большей точностью, хоть и более сложные

Для повышения точности анализа был предложен подход, основанный на **конечных автоматах с переходами по буквам** [**?**]. В этом подходе множество возможных значений строк моделируется как язык, распознаваемый конечным автоматом. Операции над строками соответствуют операциям над языками: объединение, пересечение, конкатенация и пр.

Преимущества

- высокая точность

Недостатки

- высокая вычислительная сложность
- большое потребление памяти
- низкая масштабируемость при анализе больших программ

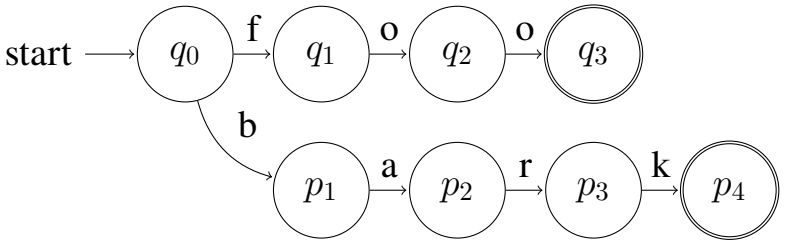


Рис. 1: Конечный автомат, распознающий строки “foo” и “bark”

Овераппроксимация строк в конечных автоматах повышает точность анализа строк во многих сценариях, но она не подходит для реальных программ, работающих со статически неизвестными входными данными и манипуляциями с длинным текстом

TARSIS

TARSIS (Template-based Abstract Representation of Strings with Static analysis) — это современный строковый абстрактный домен, предложенный в работе [10]. Основное новшество Tarsis заключается в том, что он работает с алфавитом строк, а не с отдельными символами. С одной стороны, такой подход требует более сложного и уточненного определения расширяющего оператора и абстрактной семантики строковых операторов. С другой стороны, это позволяет получать более точные результаты

Основные функции

В статье описан алгоритм для аппроксимации с доказательством полноты наиболее часто использующихся функций на строках, а именно length, concat, substr, replace, contains и indexOf

$$[\text{length}(s)] = |\sigma|$$

$$[\text{concat}(s, s')] = \sigma \cdot \sigma'$$

$$[\text{substr}(s, a, a')] = \sigma_i \dots \sigma_j \quad \text{if} \quad i \leq j < |\sigma|$$

$$[\text{replace}(s, s', s'')] = \begin{cases} \sigma\,[s'/s''] & \text{if } \sigma' \curvearrowright_s \sigma \\ \sigma & \text{otherwise} \end{cases}$$

$$[\text{contains}(s, s')] = \begin{cases} \text{true} & \text{if } \exists i, j \in \mathbb{N}.\sigma_i \dots \sigma_j = s' \\ \text{false} & \text{otherwise} \end{cases}$$

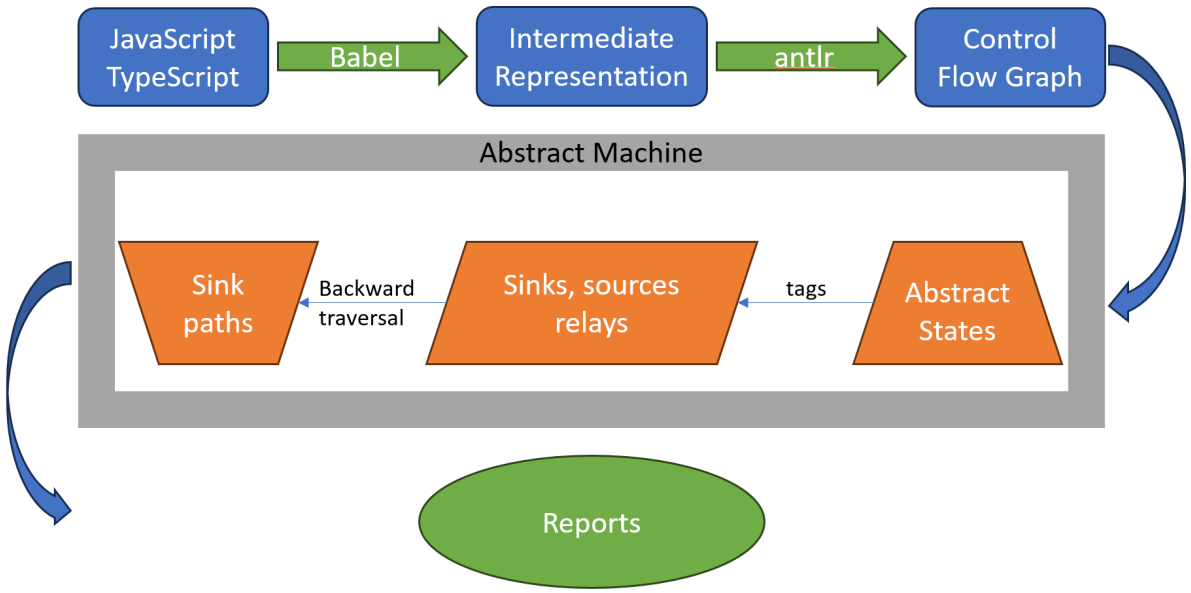
$$[\text{indexOf}(s, s')] = \begin{cases} \min\{i \mid \sigma_i \dots \sigma_j = s'\} & \text{if } \exists i, j \in \mathbb{N}.\sigma_i \dots \sigma_j = s' \\ -1 & \text{otherwise} \end{cases}$$

Также важной функцией является widening, при помощи которой удается завершать анализ циклов за конечное время

Реализация

Анализатор:

В компании есть работающий анализатор, основанный на абстрактной интерпретации. Поддерживаемые языки - JavaScript, TypeScript. Анализируемый код транспилируется при помощи Babel в промежуточное представление (intermidiate representation). Затем IR с помощью ANTLR переводится в Control Flow Graph (CFG)



Этот граф передается в абстрактную машину, которая обычные переменные переводит в абстрактные, то есть принимающие множество возможных значений. Далее применяется Taint анализ, и в результате для каждой уязвимости выводится весь путь

Необходимо реализовать:

Реализован интерфейс для решёток, интервальная решетка для целых чисел и строковая решётка из JSAI, которую необходимо улучшить

В новой версии решетки должны присутствовать:

- Структура автомата, функции для работы с ним
- Общие операции для решеток (join, meet, strictEquals, ...)
- Операции на строках (length, concat, substring, ...)
- Оператор расширения (widening)

На существующем бенчмарке из 85 реальных проектов:

- Суммарное время анализа не должно увеличиться более чем на 20%
- Потребление памяти не должно увеличиться более чем на 10%

Структура

Структура автомата — есть вершины (states) и переходы между ними. Некоторые из вершин помечены как начальные или как конечные. На переходах стоят строки или символ $\top$, обозначающий любую строку

union

Основной операцией является объединение автоматов. С её помощью находим наименьший язык, содержащий оба, порождаемых автоматами

Реализаиця: создаем общее начальное состояние, проводим из него переходы с пустой строкой в начальные состояния двух объединяемых автоматов. Затем минимизируем полученный автомат (чуть ниже описана функция minimize)

intersect

Еще одной важной операцией является пересечение автоматов. Она позволяет понять, есть ли в двух алфавитах общие слова. А также находить наибольший общий подъязык

Реализуется она при помощи операций дополнения и объединения, то есть по формуле

$$A \cap B = S/((S/A) \cup (S/B))$$

Дополнение строится таким образом — выписываем все возможные переходы в обоих автоматах, а затем для каждого переходы между двумя вершинами заменяются на дополнение к этим переходам

subset

Проверка на то, что один язык полностью содержит другой. Реализуется с помощью дополнения и пересечения по формуле

$$A \subset B <=> (S/B) \cap A = \emptyset$$

minimize

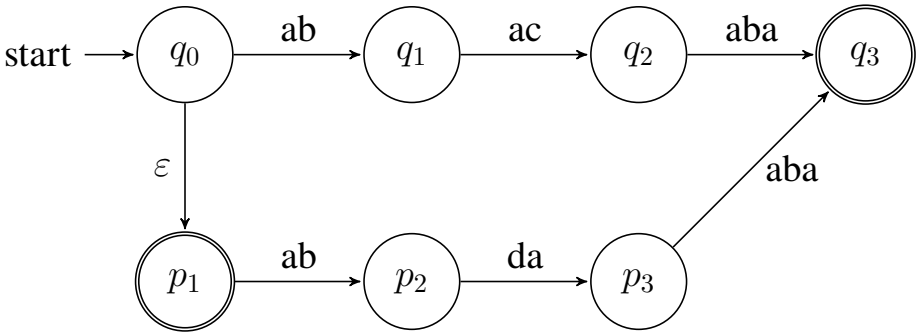
Убирает лишние вершины и переходы. Например может произойти так, что из одной вершины ведут 2 перехода с одинаковой строкой. Тогда можно упростить автомат так, чтобы склеились 2 вершины в одну. Также избавляемся от переходов по пустой строке

Алгоритм минимизации такой - сначала проверяем, является ли автомат детерменированным, то есть что нет двух исходящих переходов из одной вершины с одинаковыми символами на них. А также что нет пустых переходов. Если есть, то склеиваем эти 2 вершины. Это операция детерменирования

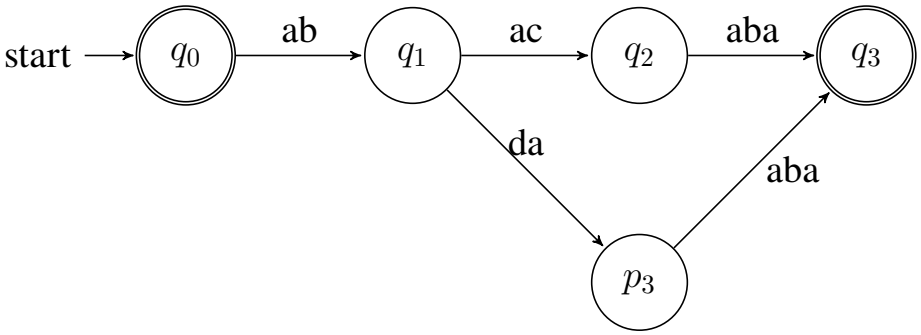
Затем, чтобы избавиться от двух одинаковых входящих переходов, мы делаем reverse, то есть перенаправляем все ребра и помечаем конечные вершины начальными и наоборот. Теперь вызываем еще раз determinize. Повторив еще раз reverse + determinize, получим начальный автомат, но без лишних переходов и вершин

Последним шагом удаляем те вершины, из которых нельзя попасть в финальные. Таким образом получим упрощенную версию того же самого автомата

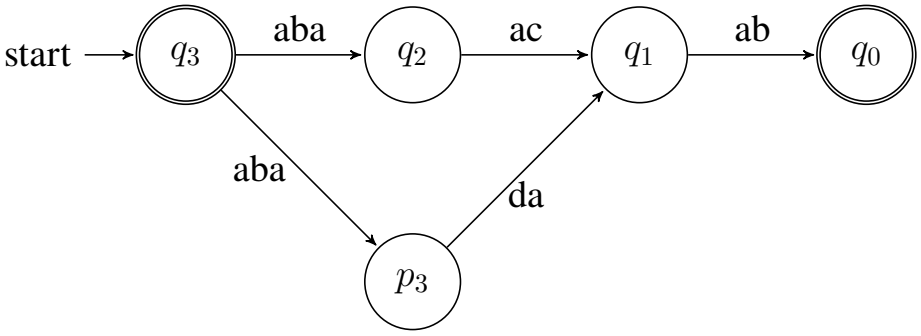
Рассмотрим пример минимизации такого автомата



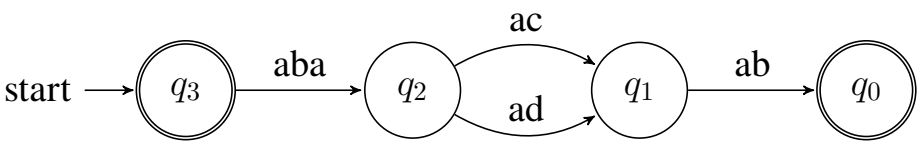
Сначала происходит детерминация, при которой q_0 и p_1 склеиваются в одну, которая становится терминальной. Затем так как из полученной вершины 2 одинаковых ребра, то вершины, в которые они ведут, склеиваются. Получаем такой автомат



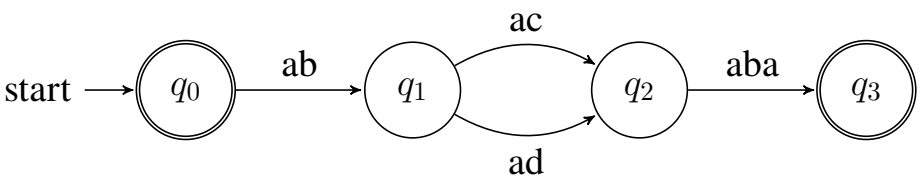
Следующий шаг reverse



Опять детерминируем, схлопывая q_2 и p_3



Еще раз reverse, и получаем упрощенную версию автомата



explode

Для реализации сложных операций на строках, такие как Replace и Substr, в которых нужно искать подстроки в автоматах, есть операция, которая переводит автомат к конечному автомату с переходами по буквам. Она просто заменяет каждое ребро на путь, где на каждом переходе лишь один символ вместо целого слова. Также есть обратная операция, которая схлопывает вершины с одним входящим и одним выходящим ребром в один переход

getLanguage

Собираем все слова, порождаемые автоматом, с помощью прохода DFS. Если в графе есть цикл, то возможных слов бесконечно, и этот случай нужно отдельно разобрать, вернув $\top$. Эта операция полезна для оптимизаций, когда вместо выполнения громоздких операций над автоматом проще выписать все слова и по отдельности разобрать

Операции для решеток

Операции join и meet аналогичны union и intersect для автоматов. Остается только разобрать случаи, когда один из аргументов $\top$ или $\bot$.

equals

Проверяем, что 2 автомата порождают одинаковые языки. В общем случае можем проверить то, что один язык включается в другой (subset) и наоборот. Однако операция пересечения слишком громоздкая (нужно сделать 3 раза дополнение), поэтому для конечных языков проще выписать 2 порождаемых языка и проверить их на соответствие. Язык является конечным, если в нем нет $\top$ переходов и циклов

strictEquals

В отличие от equals, strictEquals проверяет то, что есть хоть одно общее слово. Опять это можно сделать при помощи пересечения, то есть что пересечение не пусто. Но для конечных лучше сравнивать языки

Операции на строках

Length

Возвращает интервал $[c_1, c_2]$, такой что $c_1 \leq |s| \leq c_2$. При этом если s содержит в себе цикл, или же содержит в себе переход, на котором стоит $\top$, то есть любая строка, то максимальная длина будет сколь угодно большой. Минимальная при этом заменяет все $\top$ на пустые строки

$$\text{length}(s) \triangleq \begin{cases} [|\text{minPath}(A)|, +\infty] & \text{if cyclic}(A) \vee \text{readsTop}(A) \\ [|\text{minPath}(A)|, |\text{maxPath}(A)|] & \text{otherwise} \end{cases}$$

Для подсчета длины запускаем DFS из начальных стейтов. При вхождении в терминальный, считаем длину пути (заменяя $\top$ на 0 или $+\infty$ соответственно) и пересчитываем максимальную и минимальную длину

Если автомат содержит в себе цикл, то с помощью DFS мы его найдем и максимальная длина пути будет бесконечной, так как можно по этому циклу крутиться сколь угодно много раз

Concat

При конкатенации достаточно добавить для первого автомата одну дополнительную конечную вершину, в которую будут вести переходы с пустой строкой из конечных вершин первого, а затем из нее ведут переходы в начальные вершины второго автомата. Далее стоит минимизировать полученный автомат, чтобы избавиться от пустых ребер

Contains

Абстрактная семантика contains должна возвращать значение true, если любая строка из A' содержится в любой строке из A, значение false, если какая-то строка из A' не содержится в какой-то строке из A и {true, false} ($\top$) в остальных случаях

$$\text{contains}(s, s') \triangleq \begin{cases} \text{false} & \text{if } A' \sqcap \text{FA}(A) = \text{Min}(\emptyset) \\ \text{true} & \text{if } \neg\text{cyclic}(A) \wedge \text{singlePath}(A') \\ & \wedge \forall \pi \in \text{paths}(A).\sigma_{sp} \curvearrowright_s \sigma_\pi \\ \{\text{true}, \text{false}\} & \text{otherwise} \end{cases}$$

Для начала определим, в каком случае нужно возвращать false. Для этого определим фактор-автомат $FA(A)$, который принимает все подстроки A. Для этого нужно просто пометить все стейты конечными. Затем пересечь $FA(A)$ и A', чтобы проверить, что никакая подстрока любой строки из A не является строкой из A'

Теперь, в каком случае true. Если A' содержит 2 слова, ни одно из которых не является префиксом другого, то они оба не могут быть одновременно префиксами какого-либо слова из A. Значит все слова в A' должны быть по цепочке включены друг в друга. Или же автомат для них выглядит как путь, некоторые вершины на котором конечные. И в этом случае достаточно проверить, что самое длинное слово входит во все слова из A, тогда и его префиксы будут

Во всех остальных случаях точного ответа нет, поэтому возвращаем $\top$

Substr, Replace, IndexOf

Все 3 функции похожи друг на друга тем, что нужно искать в автомате подстроки по определенным условиям. И так как на переходах стоят не символы, а строки, то подстрока может заканчиваться или начинаться где-то в середине этого перехода. Чтобы избежать такого, первым шагом вызывается explode, чтобы начало и конец всегда были в вершинах

Следующий шаг также общий, а именно обход графа в глубину и запоминание тех путей из начальных вершин в конечные, которые содержат в себе интересующую подстроку (это может быть вхождение подстроки в этот путь в случае Replace или indexOf, или же просто возможность взять подстроку нужной длины для Substr)

А вот далее нужно полученные пути склеить в один автомат. Для этого добавляем одну общую начальную и конечную вершину, к ним присоединяем начала и концы путей, и затем минимизируем. Чтобы привести полученный автомат к сокращенному виду, а именно со словами на ребрах, делаем операцию обратную explode, то есть схлопываем вершины, у которых одно входящее и одно выходящее ребро. Получаем искомый автомат

Есть сложность в том, что автомат может содержать в себе цикл. В этом случае обход в глубину не сможет вернуть все пути, так как их бесконечно много. И обрабатывать цикл на то, можно ли с его помощью накрутить подстроку, подходящую под условия, не получится из-за предпериода или еще одного цикла (какой сколько раз крутить непонятно). В случае с Substr можно обрезать пути по длине, но для indexOf и Replace так не получится. Поэтому в данном случае приходится возвращать $\top$

Вот краткое описание indexOf, более подробное описание всех операций можно найти в статье [10]

$$\text{indexOf}(s, s') \triangleq \begin{cases} [-1, +\infty] & \text{if cyclic}(A) \vee \text{cyclic}(A') \vee \text{readsTop}(A') \\ [-1, -1] & \text{if } \forall \sigma' \in \mathcal{L}(A')\ \nexists \sigma \in \mathcal{L}(A).\sigma' \curvearrowright_s \sigma \\ \bigsqcup\limits_{\sigma \in \mathcal{L}(A')} \text{IO}(A, \sigma) & \text{otherwise} \end{cases}$$

Остальные операции

В статье TARSIS описан алгоритм только для 6 операций, тогда как на деле их гораздо больше. Часть из них похожа на описанные сверху, какие-то понятно как реализовывать. Но есть и более сложные. Разберем некоторые из них

startsWith, endWith

Идем из начальных (или конечных в случае endWith) вершин, ища те переходы, чтобы путь совпадал с паттерном. Если не находим такого пути, значит точно false. Если оказывается, что путь нужной длины только один, значит true. Иначе {true, false}

charAt, slice

Эти функции похожи на substr, аналогично шагаем по путям на нужную длину. В случае с charAt проверяем, что все символы на нужном месте одинаковые и равны нужному (true), либо что ни один не равен (false), в остальных случаях $\top$. Для slice вырезаем нужную часть автомата аналогично substr

trim, toUpperCase, pad, match

Первые 2 функции легко реализуются, нужно просто проделать операцию для каждого перехода. А вот pad и match сходу непонятно, как реализовывать для автоматов. Одной из проблем является например $\top$ ребро, которое не дает определить длину слова и непонятно, сколько символов вставлять в начало. И один из несложных вариантов реализации это выписать весь язык, порождаемый автоматом, для каждого слова по отдельности проделать операцию и склеить в один автомат. Такой подход все также плохо будет работать с циклами, то есть для цикличных автоматов возвращаем $\top$

toNum, toBool

Это функции преобразования строки в число или логическое выражение. toBool проверяет то, есть ли в языке пустая строка или "0". Если нет, то true. Если язык состоит только из пустой строки или "0 то false, иначе {true, false}

toNum должен вернуть в результате интервальную решетку для чисел, то есть ограничение сверху и снизу на значение. И заодно проверить, что все строки состоят из корректных символов, то есть цифр, не более одной точки или запятой и возможно минус в начале. Также перебираем всевозможные пути, проверяем на корректность. Запоминаем минимальное и максимальное значение

В случае с toNum $\top$ переход является недопустимым, так как это любой символ, в том числе буква. Но вот с циклами можно работать. Сначала нужно понять, находится ли эта часть строки после точки, или нет. Если нет, значит число может неограниченно расти и стремиться к $\pm\infty$. А для поиска минимального цикл нужно прокрутить только 1 раз. Если бесконечная часть появляется после точки, значит достаточно прибавить 0.1 к числу и обрезать хвост, ибо в любом случае получим ограничение сверху

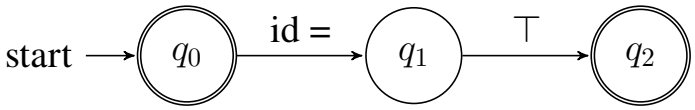
Оператор расширения

Важным методом является оператор расширения (widening). Именно благодаря ему удается завершать циклы за конечное время, создавая аппроксимацию. Рассмотрим пример того, как это реализованно

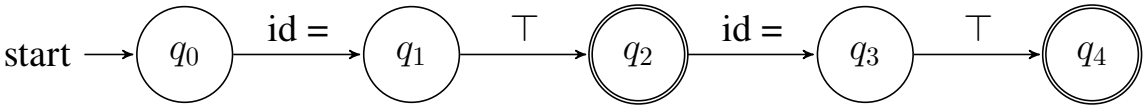
Листинг 2: Пример применения widening

```
function f(v) {
    res = "";
    while(?) res = res + "id = " + v;
    return res;
}
```

Автомат, задающий значения res в начале второй итерации цикла while, выглядит так:

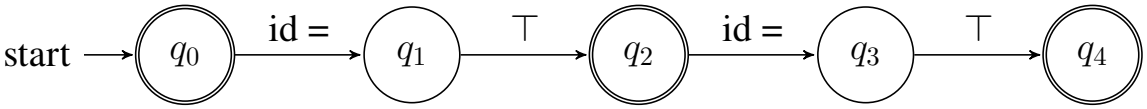


В конце второй итерации вот так:

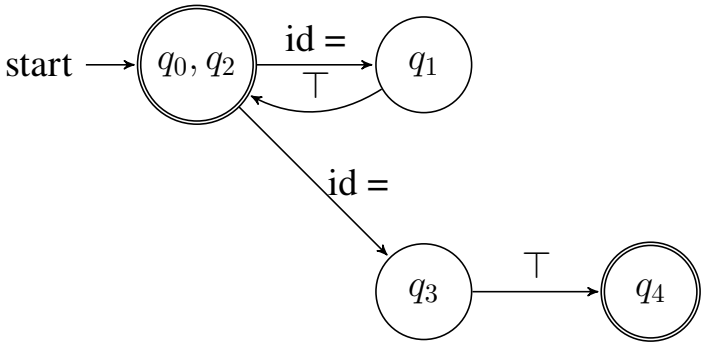


Далее, для этих двух автоматов приминяем саму операцию расширения. Алгоритм такой:

- Делаем union для этих автоматов:

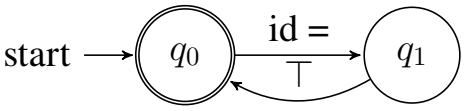


- Стейты, у которых исходящие пути длины 2 совпадают, объединяем. В данном случае q_0 и q_2 оба принимают $id = \top$, и потому склеиваются в одну вершину:

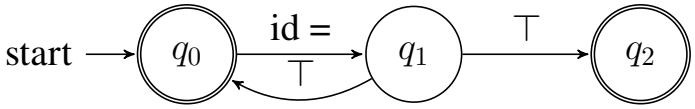


  В общем случае длину пути для склейки вершин можно регулировать, чем она больше, тем точнее результат, однако и размер автомата больше

- Минимизируем полученный автомат:



Теперь заметим, что полученный автомат является неподвижной точкой. Для этого попробуем проделать ещё одну итерацию цикла. После объединения автоматов в начале и конце цикла получаем такой:



Склеивать вершины не придется, однако при минимизации q_0 и q_2 станут одной, то есть получим начальный автомат, что означает отсутствие изменений и значит можно остановиться

Доказано [14], что такой алгоритм расширения соответствует условиям овер-аппроксимации

Тестирование

Были написаны тесты на новую финкциональность, разберем несколько примеров

Листинг 3: Пример CWE тестов

```
function tossCoin(): string {
    if (Math.Random() > 0.5) {
        return "eagle"
    } else {
        return "tails"
    }
}

if (tossCoin() == "edge") {
    /* POTENTIAL FLAW GOOD: */
    console.log(document.getElementById('source').value)
}
```

В данном примере в консоль выводятся помеченные данные (то есть те, которые влекут за собой уязвимость), если выполняется условие. Для определения абстрактного значения tossCoin(), происходит join двух веток исполнения. В старой строковой решетке результатом был класс обычных строк (NotNumNorSpl). Однако, edge также может принадлежать этому классу, из-за чего сравнение на эквивалентность строк должно возвращать $\top$, что есть возможно true и false. И тогда ветка исполнения, в которой в консоль выводятся опасные данные, также возможна. И значит анализатор указывал на то, что в этой строчке возможна уязвимость, тогда как на деле этого не происходит

С новой решеткой мы точно можем сказать, что ни одно возвращаемое знаение функции tossCoin не равно edge, а следовательно мертвый код игнорируется. Потому можем поставить метку POTENTIAL FLAW GOOD, означающую отсутствие уязвимости

Листинг 4: Пример IR тестов

```
function tossCoin(): string {
    if (Math.Random() > 0.5) {
        return "eagle"
    } else {
        return "tails"
    }
}

let x = tossCoin()
console.log(x.indexOf("a")) // [number: [1, 1]]
console.log(x.indexOf("e")) // [number: [-1, 4]]
console.log(x.indexOf("r")) // [number: [-1, -1]]
```

Другой пример тестов это IR (intermidiate representation) тесты. Они проверяют то, какие абстрактные значения принимают переменные. По аналогичным причинам старая решетка не могла как либо оценить значение indexOf. Новая же позволяет это сделать. Благодаря этому, не только строковые значения стали более точными, но и некоторые числовые

Листинг 5: Еще пример IR тестов

```
var x = "a" + document.getElementById('source').value + "c"
console.log(x)                  // [string: a⊤c tags=[XSS,WEB]]
console.log(x.length)           // [number: [2, +Inf]]
console.log(x.startsWith("a")) // [boolean: true]
console.log(x.substr(0, 1))     // [string: a tags=[XSS,WEB]]
```

В данном примере видно, что даже если в операциях участвует $\top$, все еще можно знать что-то о строке, как напимер минимальную длину или то, что startsWith('a') это всегда true

Также можно увидеть tags=[XSS,WEB]. Это как раз и есть метка, которая указывает на уязвимые данные. И при некоторых операциях она сохраняется, как например конкатенация. Одним из уточнений в будущем может стать то, что эта метка будет принадлежать конкретному переходу, а не всему абстрактному значению. То есть например если сейчас сделать x.substr(0, 1), то результат будет 'a', то есть вообще не содержать опасных символов. Однако метка будет передана

Результаты

Для подсчета итоговой статистики были произведены запуски анализатора с использованием старой и новой решеток на внутренних тестовых проектах, а также на Open Surce JavaScript проектах для парсинга текстов и сериализации, то есть те, где большое использование строковых переменных в коде

Разберем результаты на примере 3-х различных по размеру проектов:

SceneBoard

Самый большой проект в пуле тестовых. Размер 250к строк. Время анализа составило 506к ms (8 минут 26 секунд) на старой решетке и 526к ms (8:46) на новой, то есть прирост 4%. Потребление памяти было 17935 МБ на старой и 16706 МБ на новой, то есть уменьшилось на 7%. Снижение обусловлено тем, что благодаря повышению точности было проигнорировано больше строчек кода, так как он мертвый, то есть никогда не запускается в реальности. Замеры точности можно увидеть в табличке снизу

Таблица 1: Сравнение результатов анализа

Метрика	Количество возвратов		Отношение (new/old)
	`old`	`new`	
Всего состояний	10,539,656	10,093,468	0.96
Concat: Top	100,419	10,503	0.10
Concat: Const	12,849	12,913	1.01
Substr: Top	233	193	0.83
Substr: Const	20	41	2.05
Str: Top	197	186	0.94
Str: Const	80	87	1.08
Number: Top	221	163	0.74
Number: NonTop	52	98	1.88
Boolean: Top	3,716	3,412	0.92
Boolean: NonTop	5	10	2.00

Notepad

AppGalery

Open Source

Заключение

Благодарность

Список литературы

[1] Cousot P., Cousot R. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints* // POPL. 1977. https://doi.org/10.1145/512950.512973

[2] King J.C. *Symbolic Execution and Program Testing* // Communications of the ACM. 1976. https://doi.org/10.1145/360248.360252

[3] Cadar C., Sen K. "Symbolic Execution for Software Testing"// IEEE Software. 2013.

[4] Baldoni R. et al. "A Survey of Symbolic Execution Techniques"// ACM Computing Surveys. 2018.

[5] de Moura L., Bjørner N. "Z3: An Efficient SMT Solver"// TACAS. 2008.

[6] Barbosa H. et al. "cvc5: A Versatile and Industrial-Strength SMT Solver"// TACAS. 2022.

[7] Cousot P., Cousot R. "Abstract Interpretation Frameworks"// JLP. 1992.

[8] Blanchet B. et al. "A Static Analyzer for Large Safety-Critical Software"// PLDI. 2003.

[9] Cousot P. et al. "Why Does Astrée Scale?"// FMSD. 2012.

[10] Smith J. et al. "TARSIS: A Template-Based Abstract Domain for String Analysis"// PLDI, 2021.

[11] Johnson A. "Performance Evaluation of TARSIS in Web Applications"// ISSTA, 2022.

[12] Brown M. "Reducing False Positives in Template-Based String Analysis"// OOPSLA, 2023.

[13] Lee S. et al. "Machine Learning Assisted String Abstraction"// ASE, 2023.

[14] D'Silva, V.: Widening for Automata. MsC Thesis, Inst. Fur Inform.- UZH (2006)

[15] Christensen A.S., Møller A. *Precise Analysis of String Expressions* // SAS. 2003. https://doi.org/10.1007/3-540-44898-5_10

[16] Arceri V. et al. *Abstract Domains for String Analysis* // ACM Computing Surveys. 2022. https://doi.org/10.1145/3494523

[17] Zheng Y. et al. "SMT-Based String Analysis for Vulnerability Detection 2021.