Московский Государственный Университет имени М.В.Ломоносова
Факультет Вычислительной Математики и Кибернетики
Кафедра Системного Программирования

Курсовая работа

Управляемое построение предиката пути при символьной интерпретации трасс бинарного кода

Автор:
группа 327
Вишняков Алексей Вадимович

Научный руководитель:
к.ф.-м.н. Падарян Вартан Андроникович

Москва, 2017

Содержание

Введение

Обеспечение безопасности программного обеспечения является на сегодняшний день одной из важных задач. Программные продукты применяются в повседневно окружающих нас вещах: настольных компьютерах, смартфонах, автомобилях и технологиях «интернета вещей». Кроме того программное обеспечение используется на объектах критической инфраструктуры, сбои в работе которых могут привести к серьезным последствиям, а использование ошибок в злонамеренных целях может причинить колоссальный ущерб. В безопасности программных продуктов заинтересованы не только конечные пользователи, но и разработчики. Различные научные институты и крупные корпорации уделяют особое внимание разработке методов и технических средств для поиска ошибок в программах.

В настоящее время для поиска ошибок в программах активно используется техника символьного выполнения. Символьное выполнение представляет собой процесс интерпретации программы, где конкретным значениям переменных сопоставляются символьные переменные, которые могут принимать произвольные значения. Анализируемый путь выполнения программы может быть описан системой уравнений от символьных переменных, решением которой являются входные данные, обеспечивающее прохождение потока управления по этому пути. Такая система уравнений называется предикатом пути.

В работе [1] был предложен метод построения предиката пути при символьной интерпретации трасс машинных инструкций. Однако предложенному методу свойственны проблемы недостаточной помеченности и избыточной помеченности. Недостаточная помеченность возникает, когда некоторые зависимости по управлению не попадают в предикат пути. Это приводит к тому, что входные данные, полученные в результате решения предиката пути, не проведут программу по тому же самому пути выполнения, для которого строился предикат пути. В свою очередь, избыточная помеченность приводит к неминуемому усложнению уравнений в предикате пути за счет анализа инструкций, не оказывающих существенного влияния на ход выполнения программы.

В данной работе предлагается решение описанных выше проблем. Работа организована следующим образом. В разделе 1 описываются используемые техники динамического анализа бинарного кода, метод построения предиката пути (разд. 1.4) и подробное описание свойственных ему проблем (разд. 1.5). В разделе 3 приводится обзор суще-

ствующих моделей символьной памяти. В разделе 4 представлен метод управляемого построения предиката пути при символьной интерпретации трасс бинарного кода, позволяющий преодолевать озвученные проблемы. Результаты практического применения приводятся в разделе 5.

1 Обзор предметной области

1.1 Символьное выполнение

Символьное выполнение было изначально предложено для тестирования программного обеспечения [2]. При использовании модульного тестирования программист может убедиться в корректности выполнения программы на некотором наборе тестов, однако корректность программы вне тестового набора остается под сомнением. Вместо исследования поведения программы на фиксированных входных данных, символьное выполнение позволяет анализировать программу на целых классах входных данных.

Символьное выполнение представляет собой процесс интерпретации программы, где конкретным значениям переменных сопоставляются символьные переменные, которые могут принимать произвольные значения. Преобразования данных в программе описываются в виде формул над символьными переменными и константами. Каждое условное ветвление, условие выполнения которого зависит от символьных переменных, порождает уравнение, описывающее прохождение потока управления по определенной ветке.

Обычно свободные символьные переменные ставятся в соответствие входным данным программы и исследуется конкретный путь выполнения в программе. Система уравнений, построенная в результате символьной интерпретации этого пути, называется предикатом пути. Предикат пути подается на вход SMT-решателю, где символьные переменные выступают в качестве неизвестных. Решением системы уравнений является конкретный набор входных данных, обеспечивающий прохождение потока управления по исследуемому пути.

1.2 Динамический анализ помеченных данных

Динамический анализ помеченных данных — метод, предложенный в работе [3] для автоматического обнаружения программных дефектов. Для оценки безопасности программы исследуется, как данные из недоверенных источников могут изменять данные и путь выполнения программы. Как правило, входные данные программы являются таким недоверенным источником. Обработка таких данных может привести к проявлению ошибок в программах. Метод анализа помеченных данных основывается на динамическом анализе бинарного кода программы и позволяет выявить зависимости по данным

и по управлению от входных данных программы.

Суть метода заключается в добавлении пометок на байты буфера с входными данными и дальнейшем отслеживании выполнения программы для распространения пометок. Если результат выполнения инструкции зависит от помеченных операндов, операнд назначения также помечается. Ситуация, когда операнд потенциально опасной операции (обращения к памяти или передачи управления) оказывается помеченным, сигнализирует о наличии возможной ошибки в программе.

1.3 Промежуточное представление машинных инструкций

В настоящее время существует множество процессорных архитектур с различными инструкциями. Для того, чтобы абстрагироваться от специфики конкретной архитектуры для написания универсальных алгоритмов, традиционно используется промежуточное представление машинных инструкций. В этом случае алгоритмы анализа бинарного кода работают с более простым промежуточным представлением, а не с архитектурой целевого процессора.

В работе [4] предлагается промежуточное представление инструкций, удовлетворяющее SSA-форме и имеющее трехадресный код, что упрощает разработку алгоритмов анализа. Основные операторы модельной архитектуры приводятся ниже:

- **NOP.** Не имеет никакого эффекта.
- **INIT.** Инициализирует локальную переменную константным значением.
- **APPLY.** Применяет одну из модельных операций. В качестве параметров и результата используются локальные переменные.
- **BRANCH.** Осуществляет передачу управления (условную или безусловную).
- **LOAD.** Производит загрузку на локальную переменную значения из одного из адресных пространств.
- **STORE.** Записывает значение локальной переменной в одно из адресных пространств.

Адресные пространства памяти и регистров представляются в виде двух байтовых массивов. Адресное пространство регистров состоит из всех регистров машины с учетом

наложений и пересечений. Для учета побочных эффектов используется слово состояния, аналогичное регистру флагов x86. На листинге 1 приводится промежуточное представление инструкции `ADD EAX, EDX` архитектуры Intel 64.

```
 1. I o.0:i16  = 0000h                ; адрес регистра EAX в пространстве
                                      ; регистров
 2. I o.1:i16  = 0010h                ; адрес регистра EDX
 3. L t.0:i32  = r[o.0]               ; загрузка значения EAX
 4. L t.1:i32  = r[o.1]               ; загрузка значения EDX
 5. A t.2      = add.i32(t.0, t.1)    ; сложение
 6. I t.3:i16  = 0088h                ; адрес регистра EFLAGS
 7. L t.4:i16  = r[t.3]               ; загрузка значения EFLAGS
 8. I t.5:i16  = 08D5h                ; подготовка маски для EFLAGS
 9. A t.6      = x86.uf(t.4, t.5)     ; обновление EFLAGS с учетом маски
10. S r[t.3]   = t.6                  ; сохранение EFLAGS
11. A t.7      = zx.i32.i64(t.2)      ; беззнаковое расширение EAX до RAX
12. S r[o.0]   = t.7                  ; сохранение RAX
```

Листинг 1: Промежуточное представление инструкции `ADD EAX, EDX`

1.4 Построение предиката пути для трасс бинарного кода

В рамках данной работы используется метод построения предиката пути для трасс машинных инструкций, предложенный в статьях [1, 5]. Трасса машинных инструкций получается в результате работы полносистемного эмулятора [6], внутри которого запускается исследуемая программа, на вход которой подается буфер с данными фиксированного размера. Трасса содержит выполненные инструкции, значения регистров, информацию о прерываниях и взаимодействии с периферийными устройствами. Шаг трассы соответствует одной выполненной инструкции процессора. Предикат пути состоит из набора символьных уравнений и неравенств, описывающий преобразования входных данных с момента их получения и до завершения программы.

Перед построением предиката пути производится предварительное повышение уровня представления трассы, описанное в работе [7]. Среди прочего определяются нити, процессы и зоны, в рамках которых выполняются инструкции трассы. Инструкции с

общим виртуальным адресным пространством принадлежат одной зоне. Понятие зоны отличается от понятия процесса тем, что во время выполнения процесса может происходить смена контекста, приводящая к переключению виртуального адресного пространства. В трассе выявляются точки входа и выхода из прерываний. Выделяются вызовы и возвраты из функций с последующим построением графа вызовов функций в трассе.

Буфер с входными данными и начальный шаг (шаг трассы, на котором он целиком инициализирован) может задаваться явно аналитиком или определяться автоматизированно. В последнем случае аналитик может подготовить описание функции, которая считывает входные данные, и ее параметров. Шаг трассы принимается равным шагу после возврата из функции, а буфер определяется из параметров функции.

Глобальным символьным контекстом будем называть отображение из элементов адресного пространства (разд. 1.3) в формулы над символьными переменными. Под элементом адресного пространства понимается объединение последовательных байт с присвоенным уникальным идентификатором. Для каждого байта буфера с входными данными создается символьная переменная. В глобальный контекст добавляется отображение входного буфера в конкатенацию его символьных переменных.

Байты входного буфера помечаются на начальном шаге. Используя отслеживание помеченных данных, производится распространение пометок по трассе в пределах одной зоны, что гарантирует проведение анализа в едином виртуальном адресном пространстве. Также отбираются инструкции, участвующие в обработке помеченных данных. Эти инструкции транслируются в промежуточное представление (разд. 1.3). Промежуточное представление интерпретируется (загруженные значения берутся из трассы) для выявления выполненных условных переходов в промежуточном представлении инструкции.

Построение предиката пути производится в результате символьной интерпретации промежуточного представления каждой инструкции, отобранной в результате анализа помеченных данных. Для символьной интерпретации промежуточного представления в локальным символьном контексте хранится отображение из локальных переменных промежуточного представления в формулы над символьными переменными и константами. Локальный контекст существует во время символьной интерпретации промежуточного представления одной инструкции. Также для процесса интерпретации каждой инструкции трассы вводится локальное слово состояния, являющееся отображением

из бита слова состояния промежуточного представления в формулу над символьными переменными и константами. В процессе символьной интерпретации операторов промежуточного представления выполняются следующие действия:

- **NOP.** Ничего не делается.
- **INIT.** В локальный контекст добавляется отображение локальной переменной в константу.
- **APPLY.** Формула результирующей локальной переменной добавляется в локальный контекст в соответствии с операцией. Формулы параметров берутся из локального контекста. Побочные эффекты сохраняются в локальном слове состояния.
- **BRANCH.** Для условного ветвления в предикат пути добавляется уравнение или неравенство, описывающее условие перехода потока управления по выполненной ветке.
- **LOAD.** Формула для локальной переменной загружается из глобального символьного контекста. Значения, не являющиеся символьными, берутся из трассы. В локальный контекст добавляется формула для локальной переменной.
- **STORE.** Формула локальной переменной из локального контекста добавляется в глобальный контекст.

В результате символьной интерпретации всех отобранных инструкций будет построен предикат пути. Решением предиката пути будет являться набор входных данных, обеспечивающий выполнение программы по трассе, для которой он был построен.

1.5 Проблемы при построении предиката пути

Описанному выше методу построения предиката пути (разд. 1.4) свойственны проблемы схожие с проблемами, встречающимися при динамическом анализе помеченных данных [8, 9].

1.5.1 Недостаточная помеченность

Недостаточная помеченность — это ситуация, когда значение не помечено, несмотря на то что оно зависит от входных данных. Это происходит из-за того, что во время

символьного выполнения программы символьные значения могут оказаться в адресном выражении, тем самым определяя значение адреса памяти. Для рассмотрения такого адреса как символьного необходимо предполагать обращение к любой ячейке адресуемой памяти, что приводит к генерации сложных для решения уравнений [10]. Конкретизация значений символьных адресов значениями из трассы позволяет значительно упростить уравнения, но приводит к недостаточной помеченности, что, в свою очередь, может привести к тому, что набор входных данных, полученный в результате решения предиката пути, не проведет программу по тому же самому пути выполнения, для которого строился предикат пути.

В некоторых случаях последствий недостаточной помеченности можно избежать, добавляя дополнительные ограничения на символьные переменные. Обладая некоторыми сведениями об устройстве программы, можно добавить соответствующие ограничения на входные данные, параметры функций, а также на произвольные ячейки памяти и регистры.

1.5.2 Избыточная помеченность

В процессе построения предиката пути возникает неминуемый рост количества помеченных инструкций, не все из которых оказывают существенное влияние на ход анализа. Это явление называется избыточной помеченностью. Избыточное количество помеченных инструкций приводит к генерации более сложных для решения систем уравнений.

В основном, такие инструкции содержатся в коде библиотечных функций. Возможным решением будет указание списка неинтересных для анализа функций, в рамках которых не будет происходить распространение пометок.

2 Постановка задачи

Необходимо разработать и реализовать метод управляемого построения предиката пути для борьбы с проблемами недостаточной помеченности и избыточной помеченности, описанными в разделе 1.5.

Метод должен позволять:

- Добавлять ограничения на предикат пути для борьбы с недостаточной помеченностью.
- Пропускать неинтересные для анализа функции для борьбы с избыточной помеченностью.

Ограничения и список неинтересных для анализа функций должны задаваться на основе априорных знаний о программе.

3 Обзор известных решений

3.1 Моделирование символьных адресов

Проблема символьных адресов [8, 10] появляется, когда значение адресного выражения зависит от входных данных, что часто встречается в бинарном коде. В частности, символьные адреса возникают в табличных преобразованиях входных данных. Например, выражение языка Си `switch(c)` компилируется в таблицу переходов, где входной символ `c` выступает в качестве индекса. Стандартные функции преобразования строк (конвертация между ASCII и Unicode, `tolower`, `toupper` и т.д.) также используют таблицы.

Обработка символьных адресов является сложной задачей, так как символьный адрес (в худшем случае) может указывать на произвольную ячейку адресуемой памяти. Существуют два основных подхода к обработке символьных адресов:

- Конкретизация адреса.
- Работа с адресом как с полностью символьным.

Первый подход предполагает конкретизацию адреса некоторым фиксированным значением, что позволяет упростить получаемые формулы и уменьшить время решения предиката пути. Однако фиксирование значения символьного адреса может привести к недостаточной помеченности. Конкретизация символьных адресов — распространенный выбор при динамическом символьном выполнении, например, в SAGE [11].

Второй подход предполагает рассмотрение адреса как полностью символьного. Тем самым разрешение символьных адресов ложится на SMT-решатель. Основной недостаток по сравнению с первым подходом — производительность. Формулы, использующие символьную память, более дорогие для вычислений, соответственно, время решения предиката пути обычно больше.

Символьный интерпретатор в инструменте Mayhem [10] использует оба подхода. Символьные адреса, по которым производится чтение, рассматриваются как символьные. Символьный адрес записи при этом всегда конкретизируется.

3.1.1 Индексная модель памяти (Mayhem)

Индексная модель памяти — модель, предложенная в Mayhem [10], для обработки операций чтений символьных адресов.

Глобальная память представляется как отображение $\mu : I \mapsto E$ из индексов i (адресов памяти) в выражения e (формулы над символьными переменными и константами). Таким образом, $\mu[i]$ представляет содержимое i-ой ячейки памяти.

Для моделирования чтения по символьному адресу используются объекты памяти, представляющие собой аналогично с глобальной памятью отображение $M : I \mapsto E$ из индексов в выражения. Каждый раз, когда во время символьного выполнения происходит чтение по символьному адресу, создается новый объект M, который содержит все значения, к которым можно обратиться по индексу. M, по сути, — частичный снимок глобальной памяти.

Для создания объекта памяти требуется определить все возможные значения, которые может принимать индекс. Mayhem пытается определить границы $[L, U]$, в которых лежат значения индекса. Чтобы вычислить эти границы, с использованием SMT-решателя производится бинарный поиск по значению индекса в контексте текущего предиката пути. Например, изначально нижняя граница 32-битного индекса i: $L \in [0, 2^{32} - 1]$. Если $i < \frac{2^{32}-1}{2}$ выполняется, тогда $L \in [0, \frac{2^{32}-1}{2} - 1]$, в то время как невыполнимость означает, что $L \in [\frac{2^{32}-1}{2}, 2^{32} - 1]$. Процесс повторяется, пока границы не будут вычислены. Используя вычисленные границы, можно инициализировать объект памяти следующим образом: $\forall i \in [L, U] : M[i] = \mu[i]$.

Любое значение, загружаемое из объекта памяти M, может быть символьным. Чтобы разрешить уравнения, содержащие загружаемое значение ($M[i]$), SMT-решатель должен найти выражение в M, удовлетворяющее уравнениям, и убедиться, что индекс этого выражения выполним для данной системы уравнений. Чтобы облегчить работу SMT-решателя, Mayhem использует бинарные деревья поиска, которые представляются в виде формулы загрузки выражения $IST(E)$. Так для (отсортированного по адресу) списка выражений E объекта памяти M и индекса i: $IST(E) = ite(i < addr(E_{right}), IST(E_{left}), IST(E_{right}))$, где ite — if-then-else выражение, E_{left} (E_{right}) — левая (правая) часть выражений из E, а $addr(\cdot)$ возвращает наименьший индекс из данного списка выражений. Для списка из одного выражения IST возвращает это выражение без создания ite выражений.

Чтобы уменьшить количество листьев в IST, Mayhem объединяет несколько выражений в один контейнер, который представляет собой выражение, параметризованное индексом и возвращающее значения объекта памяти по индексам объединенных выражений. Выражения объединяются по принципу линейной зависимости значения от индекса. Два выражения с индексами i_i и значениями v_i добавляются в один контейнер, если $v_i = \alpha i_i + \beta$. Таким образом, объект памяти разбивается на контейнеры, где каждый контейнер представляет собой либо линейное выражение, либо изолированную точку. Например, IST для функции `tolower` выглядит следующим образом:

$$ite(n < 91, ite(n < 64, n, n + 32), n)$$

3.1.2 S^{2}E

Символьный интерпретатор инструмента S^{2}E [12] использует свою модель символьной памяти. Во время разыменования символьного указателя S^{2}E определяет страницы памяти, к которым можно обращаться по этому указателю, и передает их содержимое SMT-решателю. Для облегчения работы SMT-решателя S^{2}E разбивает страницы на страницы меньшего размера (например, 128 байт).

Также поддерживается конкретизация символьных адресов. S^{2}E использует бинарный поиск для определения интервала, которому могут принадлежать значения символьного указателя, и создает n состояний, каждое из которых имеет одно фиксированное значение указателя, удовлетворяющее текущему предикату пути.

3.1.3 Используемая модель памяти

В данной работе используется конкретизация символьных адресов. При обращении к памяти по символьному адресу значение адреса берется из трассы и уже по конкретному адресу производится чтение или запись. Однако в рамках описанной модели символьной памяти предлагается альтернативное решение проблемы символьных адресов путем добавления ограничений на символьные переменные.

4 Метод управляемого построения предиката пути и детали реализации

В данной работе предлагается метод управляемого построения предиката пути для трасс машинных инструкций. Метод позволяет преодолевать проблемы, связанные с недостаточной и избыточной помеченностями, возникающие при построении предиката пути. Это достигается путем добавления ограничений на символьные переменные и пропуска функций соответственно. Список ограничений и функций задается на основе априорных знаний о программе, полученных аналитиком, например, в результате анализа трассы программы.

4.1 Формальное описание функций и их параметров

Способ формального описания функций и их параметров, предложенный в статье [13], позволяет по заданной модели функции обнаруживать в трассе конкретные экземпляры вызовов функций и восстанавливать значения их параметров. В модели функции задаются следующие атрибуты:

- Имя функции.
- Имя модуля, в котором располагается функция.
- Список описаний входных и выходных параметров функции.

Описание параметра содержит его имя и элемент адресного пространства (регистр, стековый параметр, буфер памяти), содержащий его значение, или адрес нуль-терминированной строки. В качестве адреса и размера буфера памяти могут выступать арифметические выражения над регистрами, константами и ранее описанными параметрами.

Значения входных параметров некоторого вызова функции вычисляются из значения регистра или буфера памяти на шаге с первой инструкцией функции, а значения выходных параметров — на шаге возврата из функции.

4.2 Преодоление недостаточной помеченности

Недостаточная помеченность (разд. 1.5.1) приводит к тому, что некоторые ограничения на символьные переменные отсутствуют в предикате пути, а входные данные,

полученные в результате решения предиката пути, не проводят программу по тому же пути выполнения. В некоторых случаях предикат пути можно дополнить недостающими ограничениями на символьные переменные.

Распространена ситуация, когда входные данные программы не должны содержать некоторых символов. В этом случае можно добавить ограничения на символы входного буфера. Например, когда получение входных данных программы происходит с использованием функции `scanf`, можно добавить ограничения, чтобы входные данные не содержали терминальных символов и пробелов.

Предикат пути строится по трассе программы, получающей на вход буфер фиксированного размера. Если в программе используется значение длины входного буфера, которое оказалось символьным, можно добавить ограничение, чтобы символьная длина буфера равнялась фактической его длине.

В некоторых случаях необходимо соблюсти формат данных входного буфера. Например, программе, осуществляющей сетевое взаимодействие, может потребоваться выполнять GET или POST запросы. Тогда отправляемый буфер должен начинаться с префикса «GET» или «POST» соответственно. Для этого нужно добавить ограничения на входной буфер функции `send`, чтобы он начинался с соответствующего префикса.

В данной работе ограничения описываются на языке выражений, поддерживающем:

- Арифметические операции. `x + y, x - y, x * y, x / y, x % y, +x, -x`
- Битовые операции. `x | y, x ^ y, x & y, x << y, x >> y, ~x`
- Логические операции. `x || y, x && y, !x`
- Операции сравнения. `x == y, x != y, x < y, x > y, x <= y, x >= y`
- Описание элементов адресного пространства. Регистры описываются их именем (например, `eax`), а буферы памяти — двумя выражениями для адреса и размера. Например, элемент адресного пространства `v(esp + 8, 4)` имеет размер 4 байта и располагается по адресу `esp + 8`.
- Операции взятия адреса и размера элемента адресного пространства. `&v(esp + 8, 4) = esp + 8, #v(esp + 8, 4) = 4`

Выражения транслируются в символьные уравнения и добавляются в предикат пути перед символьной интерпретацией инструкции (разд. 1.4) на указанном шаге трассы. Поддерживаются ограничения следующих видов:

- **Ограничения на шаге трассы.** Добавляются на указанном шаге.
- **Ограничения на инструкции по определенному адресу.** Добавляются на всех шагах трассы, содержащих инструкцию по заданному адресу.
- **Ограничения на параметр модели функции.** Параметр модели (разд. 4.1) выступает в качестве переменной в выражении. Ограничения на входные параметры накладываются на шаге с первой инструкцией функции, а на выходные — на шаге возврата из функции.

Также поддерживается итерационная переменная, которая может быть использована в выражении: ограничения добавляются для всего диапазона ее значений. Например, если 100-байтовый буфер, располагающийся по адресу `0xdeadbeaf`, должен содержать только заглавные буквы английского алфавита, необходимо добавить ограничение:
$\forall i \in [0, 99]$: `v(0xdeadbeaf + i, 1) >= 0x41 && v(0xdeadbeaf + i, 1) <= 0x5a`.

Предложенный подход может также использоваться в следующем случае. Разыменование указателя, значение которого зависит от входных данных, может привести к аварийному завершению программы. Этого можно избежать, приравняв значение указателя конкретному значению или добавив ограничения, чтобы значения указателя были из диапазонов памяти, доступных для чтения или записи. Тогда программа успешно продолжит выполнение.

4.3 Преодоление избыточной помеченности

Избыточная помеченность (разд. 1.5.2) приводит к росту количества отобранных инструкций, не все из которых оказывают существенное влияние на построение предиката пути, а его решение усложняется.

Функции работы с кучей часто используются в программах, но их работа не оказывает влияния на построение предиката пути. Однако распространена ситуация, когда параметры этих функций оказываются помеченными, что приводит к избыточности

предиката пути. Чтобы этого избежать, можно снять пометки с параметров соответствующих функций на время анализа тела функции.

При анализе трасс выполнения программы на буфере фиксированного размера часто не требуется учитывать символьную длину буфера. Однако современные реализации функций вычисления длины строки, таких как `strlen`, устроены таким образом, что их обработка приводит к тому, что вычисляемая длина исходного буфера зависит от входных данных и становится символьной. В этом случае целесообразно пропустить обработку этих функций.

Для решения проблемы избыточной помеченности можно указать модель функции и параметры, с которых необходимо снять пометки в процессе символьной интерпретации. Символьные пометки снимаются с входных параметров перед вызовом соответствующей функции и восстанавливаются после возврата из нее. С выходных параметров пометки снимаются после возврата из функции. Таким образом, в предикат пути не попадут те инструкции внутри функции, на результат выполнения которых могли повлиять выбранные параметры. Использование данного подхода позволяет значительно снизить количество отобранных инструкций.

5 Результаты практического применения

5.1 Преодоление недостаточной помеченности

Механизм добавления дополнительных ограничений на символьные переменные применялся для преодоления возникающей недостаточной помеченности при построении предиката пути для программы faad [5]. В качестве гостевой системы для получения трассы выполнения использовался 32-х битный дистрибутив Debian 8.3.0. Недостаточная помеченность возникала при копировании функцией `vsscanf` буфера с входными данными на стек. Во время копирования функция `vsscanf` проверяет наличие в буфере-источнике терминальных символов, при обнаружении которых копирование заканчивается. При построении предиката пути такие проверки не попадают в предикат. Для того чтобы компенсировать отсутствие таких проверок, были добавлены необходимые ограничения (отсутствие пробелов) на параметр буфера-источника функции `vsscanf`. В результате был получен предикат пути, решение которого, поданное на вход программе, провело программу по пути выполнения, для которого снималась трасса.

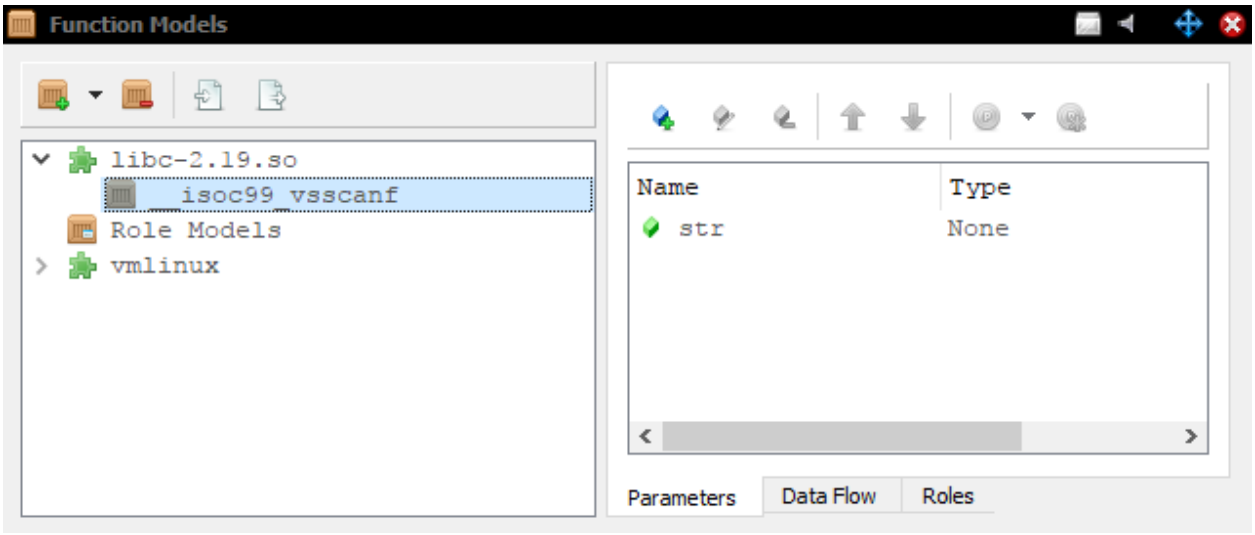

Рис. 1: Модель функции `vsscanf` из библиотеки libc

На рисунке 1 приводится модель функции `vsscanf`, а на рисунке 2 описание параметра буфера-источника `str` этой функции, который является нуль-терминированной строкой и располагается по адресу `rdi`. Для буфера-источника добавляется ограничение (рис. 3), чтобы он не содержал пробелов (символ `0x20`). Выражение для ограничения

выглядит следующим образом: $\forall i \in [0, \texttt{\#str}]$: `v(&str + i, 1) != 0x20`, где `#str` — длина строки, а `&str` — ее адрес, который равен `rdi`.

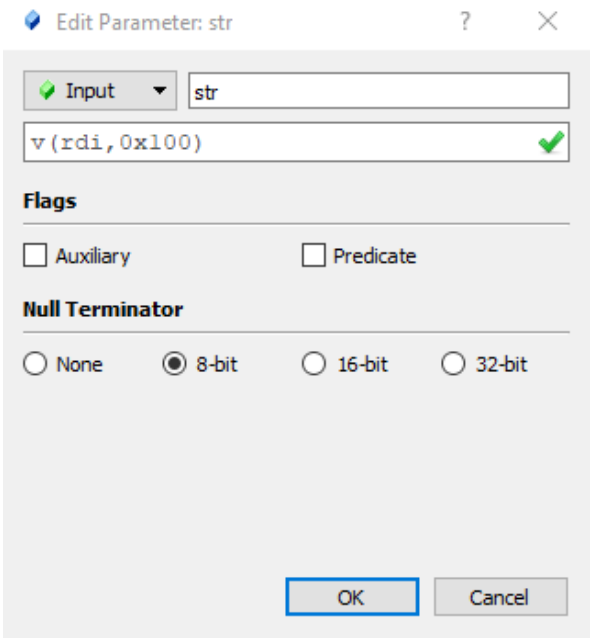

Рис. 2: Описание входного параметра (нуль-терминированной строки) `str` модели функции `vsscanf`

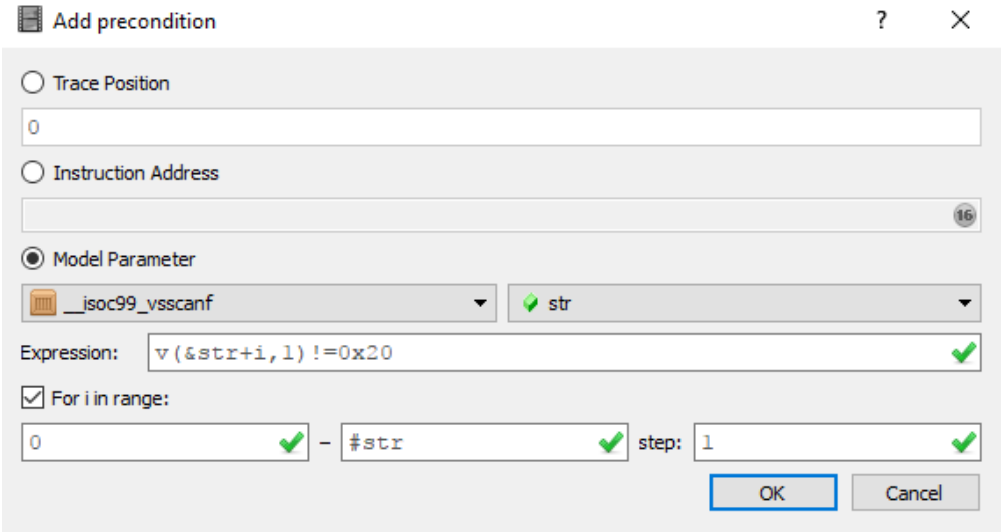

Рис. 3: Описание ограничения на параметр `str` модели функции `vsscanf`

5.2 Преодоление избыточной помеченности

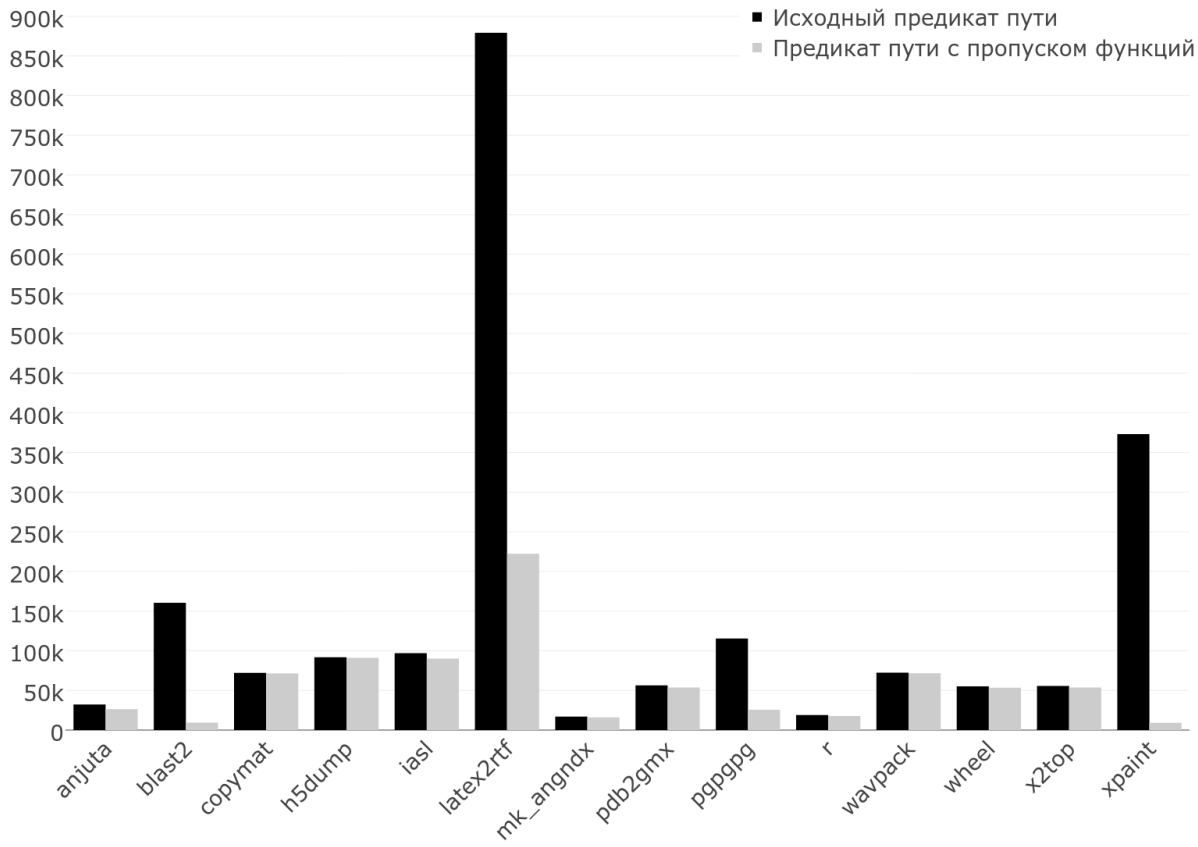

Рис. 4: Количество отобранных инструкций при построении предиката пути

Измерения производились на 32 программах из 32-х битного дистрибутива Debian версии 6.0.10. Во время построения предиката пути пометки снимались с параметров функций работы с кучей `free`, `malloc` и `calloc`, что помогло уменьшить количество отобранных инструкций для 14 программ (рис. 4). В остальных 18 программах функции работы с кучей не были отобраны во время построения предиката пути.

Программа	Число инструкций без пропуска функций	Число инструкций с пропуском функций	Число пропущенных инструкций	Доля пропущенных инструкций
anjuta	32349	26392	5957	0.18
blast2	160382	9368	151014	0.94
copymat	72124	71494	630	0.01
h5dump	91806	91049	757	0.01
iasl	97037	89973	7064	0.07
latex2rtf	879249	222309	656940	0.75
mk_angndx	17092	15877	1215	0.07
pdb2gmx	56286	53753	2533	0.05
pgpgpg	115539	25622	89917	0.78
r	18870	17766	1104	0.06
wavpack	72306	71672	634	0.01
wheel	55032	53500	1532	0.03
x2top	55716	53674	2042	0.04
xpaint	373192	9219	363973	0.98
Среднее				**0.28**

Таблица 1: Результаты работы инструмента

Для каждой программы, для которой удалось уменьшить число отобранных инструкций, в таблице 1 приводятся число отобранных инструкций при построении предиката пути с пропуском функций и без, число пропущенных инструкций и отношение числа пропущенных инструкций к числу инструкций без пропуска функций. Средняя доля пропущенных инструкций для 14 примеров составила 0.28.

6 Заключение

В данной работе был предложен и реализован метод управляемого построения предиката пути при символьной интерпретации трасс бинарного кода, который позволил преодолеть проблемы недостаточной и избыточной помеченностей. Применение предложенного метода позволит получать более точные предикаты пути и уменьшить число отобранных инструкций во время их построения.

Перспективным направлением дальнейших исследований является проведение анализа помеченных данных, начиная с файлов, что позволит добавлять ограничения на содержимое файлов. Еще одно перспективное направление исследований — автоматизированное вычисление ограничений на входные данные по трассе, что позволит облегчить работу аналитика.

Список литературы

[1] *Падарян В. А., Каушан В. В., Федотов А. Н.* Автоматизированный метод построения эксплойтов для уязвимости переполнения буфера на стеке // *Труды Института системного программирования РАН.* — 2014. — Т. 26, № 3. — С. 127–144.

[2] *King J. C.* Symbolic execution and program testing // *Communications of the ACM.* — 1976. — Vol. 19, no. 7. — Pp. 385–394.

[3] *Newsome J., Song D.* Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software // Proceedings of the Network and Distributed System Security Symposium. — NDSS '05. — 2005.

[4] *Падарян В. А., Соловьев М. А., Кононов А. И.* Моделирование операционной семантики машинных инструкций // *Труды Института системного программирования РАН.* — 2010. — Т. 19. — С. 165–186.

[5] *Федотов А. Н., Падарян В. А., Каушан В. В., Курмангалеев Ш. Ф., Вишняков А. В., Нурмухаметов А. Р.* Оценка критичности программных дефектов в условиях работы современных защитных механизмов // *Труды института системного программирования РАН.* — 2016. — Т. 28, № 5. — С. 73–92.

[6] *Довгалюк П. М., Макаров В. А., Падарян В. А., Романеев М. С., Фурсова Н. И.* Применение программных эмуляторов в задачах анализа бинарного кода // *Труды Института системного программирования РАН.* — 2014. — Т. 26, № 1. — С. 277–296.

[7] *Падарян В. А., Гетьман А. И., Соловьев М. А., Бакулин М. Г., Борзилов А. И., Каушан В. В., Ледовских И. Н., Маркин Ю. В., Панасенко С. С.* Методы и программные средства, поддерживающие комбинированный анализ бинарного кода // *Труды Института системного программирования РАН.* — 2014. — Т. 26, № 1. — С. 251–276.

[8] *Schwartz E. J., Avgerinos T., Brumley D.* All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask) // Proceedings of the IEEE Symposium on Security and Privacy. — SP '10. — 2010. — Pp. 317–331.

[9] *Kang M. G., McCamant S., Poosankam P., Song D.* DTA++: Dynamic taint analysis with targeted control-flow propagation // Proceedings of the Network and Distributed System Security Symposium. — NDSS '11. — 2011. — Pp. 269–282.

[10] *Cha S. K., Avgerinos T., Rebert A., Brumley D.* Unleashing Mayhem on binary code // Proceedings of the IEEE Symposium on Security and Privacy. — SP '12. — 2012. — Pp. 380–394.

[11] *Godefroid P., Levin M. Y., Molnar D.* Automated whitebox fuzz testing // Proceedings of the Network and Distributed System Security Symposium. — NDSS '08. — 2008. — Pp. 151–166.

[12] *Chipounov V., Kuznetsov V., Candea G.* The S2E platform: design, implementation, and applications // *ACM Transactions on Computer Systems.* — 2012. — Vol. 30, no. 1. — Pp. 2:1–2:49.

[13] *Аветисян А. И., Гетьман А. И.* Восстановление структуры бинарных данных по трассам программ // *Труды Института системного программирования РАН.* — 2012. — Т. 22. — С. 95–118.