Сравнительный анализ двух подходов к статическому анализу помеченных данных

[1] *М.В. Беляев <mbelyaev@ispras.ru>*
[1] *Н.В. Шимчик <shimnik@ispras.ru>*
[1] *В.Н. Игнатьев <valery.ignatyev@ispras.ru>*
[1,2] *А.А. Белеванцев <abel@ispras.ru>*
[1] *Институт системного программирования РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1*

Аннотация. В настоящее время одним из наиболее эффективных средств поиска проблем безопасности ПО является анализ помеченных данных. Он может быть реализован на основе статического анализа и успешно обнаруживать ошибки, приводящие к уязвимостям, таким как внедрение кода или утечка непубличных данных. Возможны несколько существенно различных подходов к реализации алгоритма распространения пометок по внутреннему представлению программы: на основе анализа потоков данных или на базе символьного исполнения. В данной работе описаны особенности реализации обоих подходов в рамках существующей инфраструктуры статического анализатора для поиска ошибок в программах на C#, а также проведено сравнение этих подходов в различных аспектах: область применения, качество результатов, производительность и требовательность к ресурсам. Поскольку оба подхода используют единую инфраструктуру доступа к информации о программе и реализованы одной командой разработчиков, результаты сравнения близки к объективным и могут быть использованы при выборе оптимального варианта в контексте поставленной задачи.

Ключевые слова: taint analysis; static analysis; IFDS; symbolic execution

DOI: 10.15514/ISPRAS-2017-29(3)-7

Для цитирования: Беляев М.В., Шимчик Н.В., Игнатьев В.Н., Белеванцев А.А. Сравнительный анализ двух подходов к статическому анализу помеченных данных. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 99-116. DOI: 10.15514/ISPRAS-2017-29(3)-7

1. Введение

В настоящее время статический анализ все чаще используется для поиска серьёзных уязвимостей в ПО, таких как внедрение вредоносного кода или утечка непубличных данных. Помимо очевидного преимущества раннего обнаружения проблемы, статический анализ указывает точное место возникновения и проявления дефекта, а также предоставляет последовательность диагностических точек, содержащих путь распространения проблемы. С помощью тестирования сложно проверить все нетривиальные сценарии использования ПО, где часто содержатся уязвимости. Детекторы на основе статического анализа покрывают практически все возможные пути выполнения программы, что является существенным преимуществом, особенно в случае проблем безопасности.

Одним из наиболее эффективных методов статического анализа программ с целью поиска проблем безопасности является анализ помеченных данных. Полученные в результате работы анализатора предупреждения могут быть также использованы автоматизированными средствами проверки на основе динамического анализа или для генерации входных данных, вызывающих уязвимость.

В основе анализа помеченных данных лежит идея продвижения пометок по всем возможным путям распространения «интересных» данных в программе от некоторого истока (например, функции чтения пользовательских данных из формы) до стока (например, функции исполнения SQL запроса), тем самым вычисляя, как злоумышленник может внедрить свой код или какие секретные данные могут оказаться доступны (в примере – выполнить внедрение SQL кода). Анализ помеченных данных может быть реализован на основе как статического, так и динамического анализа. Помимо перечисленных ранее преимуществ поиска ошибок в статике, возможны ситуации, когда утечка пользовательских данных специально реализована автором программы и маскируется, если распознает запуск внутри динамического анализатора. Данная проблема особенно актуальна для широко распространённых в настоящее время мобильных приложений и должна быть обнаружена до начала распространения вредоносной программы.

В работе рассматриваются два подхода к статическому анализу помеченных данных. Первый подход основан на решении IFDS-задачи и называется алгоритмом табуляции [1]. Данный метод решает задачу анализа потоков данных, сводя её к задаче достижимости на расширенном межпроцедурном графе потока управления и имеет сложность $O(ED^3)$, а для локально разделимых задач — за $O(ED)$, где E — количество рёбер (нерасширенного) межпроцедурного ГПУ, а D — количество фактов анализа данных. Данный подход реализован в широко известном инструменте FlowDroid [4,5].

Второй подход основан на символьном выполнении и производит однократный обход графа развёртки функции, симулируя эффект выполнения каждой встреченной инструкции в графе. Обход функций выполняется в

топологическом порядке по графу вызовов, чтобы анализ всех вызываемых функций был завершён к началу рассмотрения вызывающей.

При решении задачи анализа потоков данных, помимо выбора схемы распространения пометок, существенное влияние оказывают методы моделирования окружения выполнения программы (моделирование библиотечных функций), построение модели графа вызовов программы для симуляции активности пользователя, анализ виртуальных вызовов, анализ исходных текстов на языках описания интерфейсов и другой семантики программы (например, XAML, для сбора доступных пользователю интерфейсов). Оба рассмотренных в данной работе подхода используют единую инфраструктуру анализа, в том числе представления окружения, поэтому в рамках данной статьи перечисленные проблемы не рассматриваются.

Работа состоит из следующих частей. Во второй части рассмотрена общая инфраструктура анализатора SharpChecker, необходимая для реализации обоих методов. В частях 3 и 4 описаны модель работы и особенности реализации анализа помеченных данных с помощью IFDS и символьного выполнения соответственно. В части 5 собраны результаты работы обоих алгоритмов как на специальном проекте WebGoat.NET, моделирующем распространённые уязвимости, так и на наборе проектов с открытым кодом на языке C#. Шестая часть содержит обобщение полученных результатов и рассматривает основные направления дальнейших исследований для их улучшения.

2. Инфраструктура анализатора SharpChecker

Сравнение алгоритмов анализа помеченных данных будет проводиться на основе SharpChecker [2,3] — статического анализатора для поиска ошибок в программах на языке C#. Он способен находить более 100 различных типов ошибок, производя поиск на основе синтаксического анализа (сигнатурный), анализа потоков данных и чувствительного к контексту вызова и путям выполнения символьного выполнения. Общая схема работы анализатора представлена на рисунке 1.

На первых этапах SharpChecker строит статический граф вызовов анализируемой программы, который в дальнейшем уточняется за счёт анализа виртуальных методов, вызовов через интерфейсы и делегаты. Подсистема анализа неявных вызовов предоставляет возможность рассматривать все методы, возможные для вызова в данной инструкции, позволяя детекторам или другим подсистемам анализа выбирать способ использования данной информации. Например, детектор разыменования null последовательно применяет все резюме кандидатов, а другие детекторы могут эвристически выбрать только один, основываясь, например, на ресурсоёмкости требуемых операций. Поскольку большинство атак производятся с помощью пользовательских интерфейсов, которые, в свою очередь, на C# чаще всего

реализованы на основе интерфейсов, качество алгоритмов девиртуализации оказывает очень существенное влияние на результаты анализа помеченных данных.

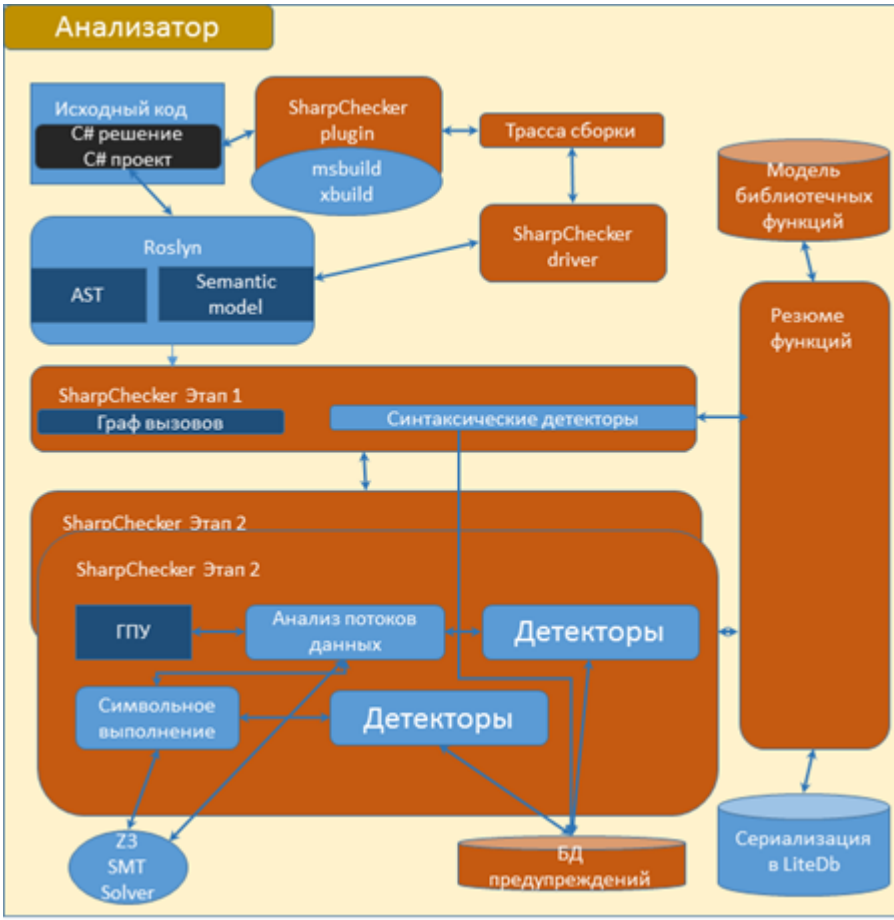

Рис. 1. Схема работы анализатора SharpChecker
Fig. 1. Schematic diagram of the SharpChecker analyzer

Символьное выполнение основано на обходе графа вызовов в порядке от вызываемых функций к вызывающим. Это позволяет свести межпроцедурный анализ к внутрипроцедурному за счёт построения резюме каждого метода. Резюме содержит всю необходимую информацию об эффектах вызова данной функции с учётом контекста вызова. Это достигается благодаря

использованию предусловий, вычисляемых с помощью SMT решателя в случае необходимости.

Во время внутрипроцедурного анализа для каждого метода строится граф потока управления, вершинами которого являются базовые блоки - последовательности подряд идущих инструкций. Инструкциями являются вершины абстрактного синтаксического дерева, семантически соответствующие отдельным операторам. Рёбра в графе потока управления являются ориентированными и показывают, в какие базовые блоки может быть совершён переход после завершения выполнения данного. Особыми (пустыми) базовыми блоками являются точка входа в метод и точка выхода из метода, в которых сходятся рёбра от всех базовых блоков, которые могут завершить выполнение тела метода (например, содержащие инструкцию return или возможное исключение). Следует отметить, что граф потока управления может содержать циклы, а значит, в нём может быть бесконечное число различных путей от точки входа в метод до точки выхода. Чтобы ограничить количество рассматриваемых путей, вместо графа потока управления строится его ациклическая развёртка (граф развёртки), в которой переходы по обратным рёбрам заменяются на переходы в отдельные компоненты, копирующие часть графа потока управления. Поскольку развёртка графа строится для фиксированной глубины и в ней нет обратных рёбер, то максимальная длина и суммарное количество различных путей от точки входа до точки выхода ограничены.

Символьное выполнение предполагает обход графа развёртки, при котором считается, что формальные параметры метода, а также все поля данного класса и статические поля других классов имеют произвольные значения. По этой причине в точке входа в метод формальные параметры и начальное состояние памяти параметризуются набором символьных переменных, тип которых совпадает с типом исходных полей и переменных, а конкретное значение никак не задаётся. Инструкциям в программе задаётся новая семантика, которая позволяет работать не с конкретными значениями, а с символьными выражениями: формулами над символьными переменными и константами. Так, если выполняется операция сложения двух целочисленных символьных переменных V1 и V2, то результатом символьного выполнения будет считаться формула V1 + V2. Аналогичные действия выполняются для унарных операторов и условного оператора. Значения, которые не являются константными и не могут быть выражены через существующие символьные переменные, моделируются с помощью введения новых символьных переменных. Вызовы методов моделируются с использованием информации, сохранённой в резюме вызываемого метода. Для сокращения количества интерпретируемых путей также применяется техника объединения состояний. В точке слияния путей выполнения ГПУ создаётся новый контекст анализа, строящийся на основе слияния (по заданным для каждого типа правилам) соответствующих пар фактов, пришедших с входящих рёбер.

Для моделирования окружения в анализаторе реализовано несколько подсистем. Первая основана на хранении множества «интересных» свойств библиотечных функций, например, что в результате её работы создаётся объект, требующий освобождения, или что первый параметр не может быть равен null. Для анализа помеченных данных данная подсистема может быть расширена с целью поддержки библиотечных передаточных функций, а также множеств истоков и стоков.

Другой метод моделирования окружения позволяет задавать поведение библиотечных функций на языке C#. Это особенно актуально для реализации коллекций, имеющих внутреннее состояние, которое необходимо учитывать для качественного анализа. Этот механизм также был доработан для поддержки продвижения пометок через вызовы библиотечных функций.

Таким образом, существующая инфраструктура анализатора SharpChecker позволяет единообразно решать проблему моделирования окружения, построения внутреннего представления и сбора данных, необходимых для реализации обоих подходов анализа помеченных данных.

3. Алгоритм анализа помеченных данных на основе IFDS

В основе алгоритма лежит распространение информации о помеченных данных по базовым блокам и рёбрам межпроцедурного графа потока управления. При этом пометки могут продвигаться от источников к стокам и наоборот. Будем говорить, что *прямой* анализ начинается от источников, *обратный* — от стоков. Правила распространения помеченных данных задаются *передаточными функциями*. Анализ для каждого источника (стока – в обратном) проводится независимо.

3.1 Задача анализа помеченных данных как IFDS-задача

Введём необходимые определения.

Путь доступа — это список, состоящий из начального объекта и полей. Начальным объектом пути доступа может являться литерал, локальная переменная, параметр, возвращаемое значение метода, ссылка `this`, статическое поле класса, выражение (например, вызов метода или оператора) или временный начальный объект, используемый в реализации передаточных функций и не представляющий никакого реального объекта, содержащегося в программе. Полями могут являться нестатические поля и общий элемент массива. Примеры путей доступа: `x`, `x.y.z`, `x.[...].w`, `this.f`, `a.f()`, `"password"`, `return`, `(a + b).p`. *Длиной* непустого пути доступа называется количество полей в нём + 1. Пути доступа хранятся в виде префиксного дерева, корнем которого является специальный пустой путь доступа `{root}`, имеющий длину 0.

Факт анализа данных — это информация о том, что путь доступа помечен. Факт содержит помеченный путь доступа и предшествующий факт, используемый для восстановления пути распространения помеченных данных.

Точка анализа данных — это информация о том, что в данной точке программы путь доступа, содержащийся в факте, помечен. Точка содержит факт и базовый блок, в котором этот факт рассматривается.

Алгоритм анализа работает с точками анализа данных, а передаточные функции — непосредственно с фактами анализа данных и путями доступа. Различные дефекты безопасности имеют различное соотношение количества источников и стоков, поэтому для возможности реализации детекторов дефектов был реализован как прямой (от источника к стоку), так и обратный (от стока к источнику) анализ.

Каждый факт распространяется независимо от других. Это позволяет создавать резюме, содержащие результаты анализа метода, и повторно использовать их, если входная точка ещё раз встретится в программе. Резюме для входной точки содержит выходные точки, полученные в результате анализа, включая пути распространения данных. Использование резюме значительно ускоряет анализ, так как это избавляет от необходимости повторно рассматривать не только метод, для которого построено резюме, но и все вызываемые им (в т. ч. транзитивно) методы.

Основная функция (`Solve`, листинг 1) осуществляет обработку большинства инструкций базового блока и вызывается для каждой начальной точки анализа (точки источника для прямого анализа и точки стока для обратного), а также для точек, входящих в вызываемый метод. Эта функция управляет очередью обработки точек, а также применением резюме. Если для входной точки существует резюме, применяются точки из резюме и результат возвращается. Применение точек заключается в замене предшествующего факта у факта входной точки резюме на факт входной точки функции `Solve`, чтобы восстановить путь распространения данных. Если же резюме не существует, то входная точка добавляется в очередь, и запускается основной цикл обработки. В этом цикле из очереди последовательно извлекаются точки, для них вызывается функция `SolveOnePoint`, выполняющая распространение помеченных данных, и результат помещается в очередь. Цикл выполняется, пока в очереди есть точки. В результат записываются точки, достигшие конца метода.

```
1: function Solve(point) → {Point}
2:   if ∃ a summary for point
3:     return Apply(summary points for point)
4:   worklist ← {point}, outPoints ← ∅
5:   while worklist ≠ ∅
6:     pt ← worklist.Get()
```

```
7:   if pt ∈ visited
8:     continue
9:   visited ← visited ∪ {pt}
10:  worklist.PutAll(SolveOnePoint(pt))
11:  if BasicBlock(pt) is exit block
12:     outPoints ← outPoints ∪ {pt}
13:  store outPoints as a summary for point
14:  return outPoints
```

Листинг 1. Псевдокод функции Solve для прямого анализа
Listing 1. Pseudo-code of the Solve function for direct analysis

Функция `SolveOnePoint` распространяет помеченные данные, используя передаточные функции `NormalTF`, `CallTF`, `RetTF`, `Call2RetTF`. В строках 10–12 оригинальный алгоритм расширяется для поддержки несбалансированных возвратов, что позволяет проводить анализ, начиная из источников, а не из метода `main`, которого может не быть, и просматривать в ходе анализа только те методы, в которые попадают помеченные данные. Возврат из метода называется сбалансированным, если метод был достигнут во время анализа по цепочке вызовов. Для сбалансированного вызова известно, в какую точку программы произойдёт возврат. Когда анализ возвращается в метод, с которого он начался, возврат из этого метода является несбалансированным, поэтому требуется перебирать все точки вызова этого метода, чтобы продолжить анализ. Примеры сбалансированных и несбалансированных возвратов приведены на рисунке 2.

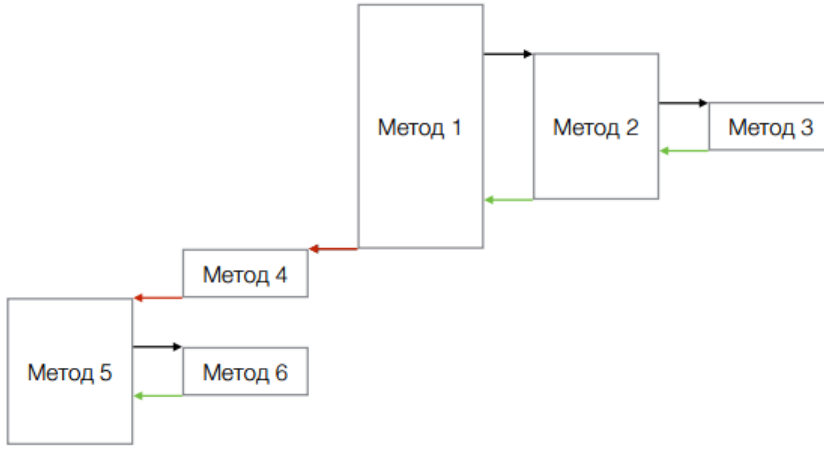

Рис. 2. Сбалансированные и несбалансированные возвраты
Fig. 2. Balanced and unbalanced returns

Здесь Метод 1 — метод, с которого начинается анализ, чёрные стрелки обозначают вызовы, зелёные — сбалансированные возвраты, красные — несбалансированные возвраты.

```
1: function SolveOnePoint(point) → {Point}
2:   points ← NormalTF(point)
3:   if last statement of BasicBlock(point) is call
4:     points_old ← points, points ← ∅
5:     foreach p in points_old
6:       points_call ← CallTF(p, call)
7:       points_callee ← ⋃_{p'∈points_call} Solve(p',call)
8:       points_ret ← ⋃_{p''∈points_callee} ReTF(p'',call)
9:       points ← points ∪ points_ret ∪
                          Call2RetTF(p, call)
10:  if BasicBlock(point) is exit block ∧ point
wasn't created by processing a call
11:    foreach call site call of current function
12:      points ← points ∪ ⋃_{p'∈RetTF(point,call)} Solve(p'))
13:  return points
```

Листинг 2 Псевдокод функции SolveOnePoint для прямого анализа
Listing 1. Pseudo-code of the SolveOnePoint function for direct analysis

В строках 7–9 применяются резюме для внутренних точек, строки 2–3 предотвращают постоянное пересоздание резюме для одних и тех же точек в строке 16.

Поскольку при обратном анализе частым случаем является передача «управления» в середину функции при возврате из вызова, то для частей таких методов также строится резюме. Оно может быть повторно использовано в дальнейшем для ускорения анализа. Это и является наиболее существенным отличием реализации метода `Solve` обратного анализа.

Практические реализации функций Solve, SolveOnePoint и передаточных функций содержат дополнения для вычисления не только результирующих точек, но и диагностических точек — точек анализа данных, в которых детектированы дефекты.

3.2 Передаточные функции для анализа помеченных данных

Анализ использует четыре вида передаточных функций — внутрипроцедурная передаточная функция, передаточная функция вызова, передаточная функция возврата и передаточная функция от вызова до возврата. Каждая из них принимает на вход одну точку анализа данных и необходимую дополнительную информацию и возвращает множество точек анализа данных, а внутри себя работает с фактами, а не точками.

Для определения передаточных функций введём следующие обозначения:

T — множество помеченных путей доступа, x, y, z — некоторые пути доступа, $\diamondsuit$ — унарный оператор, $\circ$ — бинарный оператор, $x.[f]$ — произвольный путь доступа, полученный из пути доступа x добавлением списка полей $[f] = .f_1.f_2 \ldots f_p$, в т. ч. пустого, $"t_0{a_1}...{a_n}t_n" — выражение интерполяции строк в языке C#, где t_i — фрагменты текста ($0 \le i \le n$), а a_i — аргументы ($1 \le i \le n$).

Передаточные функции очевидно различаются для прямого и обратного анализа, однако для понимания достаточно рассмотреть только один случай прямого анализа.

Обычная передаточная функция `NormalTF`:

Функция осуществляет внутрипроцедурное распространение помеченных данных внутри базового блока, последовательно обрабатывая его инструкции в прямом порядке. Выходные точки строятся по множеству T, используя ГПУ для определения следующих базовых блоков.

$$\texttt{var x = y} : T \to T \cup \{x.[f] : y.[f] \in T\};$$
$$\texttt{x = y} : T \to T \cup \{x.[f] : y.[f] \in T\} \setminus \{x.[f] : y.[f] \notin T\};$$
$$\diamondsuit\texttt{x} : T \to T \cup \{(\diamondsuit x).[f] : x.[f] \in T\};$$
$$\texttt{x} \circ \texttt{y} : T \to T \cup \{(x \circ y).[f] : x.[f] \in T \vee y.[f] \in T\};$$
$$\texttt{x} \circ\texttt{= y} : T \to T \cup \{x.[f] : y.[f] \in T\};$$
$$\texttt{\$"t0\{a1\}...\{an\}tn"} : T \to T \cup \{(\$"t0\{a1\}...\{an\}tn").[f] : \exists i;\ 1 \le i \le n \wedge arg_i.[f] \in T\};$$
$$\texttt{return x} : T \to T \cup \{return.[f] : x.[f] \in T\}.$$

Передаточная функция вызова `CallTF`:

Функция осуществляет распространение помеченных данных из текущего метода в вызываемый метод, сопоставляя фактические и формальные параметры, включая неявный параметр $this$, а также оставляя без изменений статические поля. Всё остальное вызываемому методу недоступно.

Объявление метода: `ReturnType f(param1, param2, ..., paramN)`.

Вызов метода: `a.f(arg1, arg2, ..., argN)`.

В результирующих точках указывается базовый блок $Entry$ вызываемого метода.

$$T \to \{param1.[f] : arg1.[f] \in T\} \cup \{this.[f] : a.[f] \in T\} \cup$$
$$\cup \{x.[f] : IsStatic(x) \wedge x.[f] \in T\}.$$

Передаточная функция возврата `RetTF`:

Функция распространяет помеченные данные из текущего метода в вызвавший его метод, сопоставляя формальные и фактические параметры вызова, включая неявный параметр $this$, а также оставляя без изменений статические поля. Всё остальное вызывающему методу недоступно. Заметим,

что пути доступа длины 1, соответствующие формальным параметрам, не распространяются, поскольку их изменения в языке C# локальны для текущего метода, если не указано ключевое слово **`ref`** или **`out`**. Возврат происходит в базовый блок, следующий за базовым блоком вызова.

$$T \to \{argI.x.[f]:paramI.x.[f] \in T\} \cup$$
$$\cup \{argI:IsOutParameter(paramI) \wedge paramI \in T\} \cup$$
$$\cup \{a.x.[f]:this.x.[f] \in T\} \cup \{x.[f]:IsStatic(x) \wedge x.[f] \in T\} \cup$$
$$\cup \{a.f(arg1, arg2, \ldots, argN).[f]:return.[f] \in T\}.$$

Передаточная функция от вызова до возврата `Call2RetTF`:

Эта передаточная функция внутрипроцедурно распространяет пометки из базового блока вызова в блок возврата для тех путей доступа, которые недоступны вызываемому методу. Такими путями доступа являются нестатические поля, переменные и параметры, которые не были переданы в вызываемую функцию как аргументы. В результирующих точках указывается базовый блок возврата, как и в `RetTF`.

$$T \to \{x.[f] : \neg IsStatic(x) \wedge a \neq x \wedge \ !\exists\, I : argI = x \wedge x.[f] \in T\} \cup$$
$$\cup \{x : \neg IsStatic(x) \wedge (a = x \vee \exists I : argI = x) \wedge x \in T\}.$$

3.3 Результаты тестирования

Тестирование проблем безопасности в активно используемом ПО затруднено тем, что как правило большинство реальных уязвимостей уже исправлено, поэтому количество сообщений анализатора очень мало. В связи с этим, сравнение подходов удобно проводить на специальном проекте для обучения информационной безопасности WebGoat.NET, который намеренно содержит ошибки. Однако приводятся также результаты запуска на других открытых проектах, содержащих искомые ошибки.

Испытания проводились на двух одинаковых компьютерах с процессором Intel Core i7-6700, 32 ГБ оперативной памяти, ОС Windows 10 x64. Проведённые испытания показали следующие результаты:

Табл. 1. Результаты работы метода анализа IFDS
Table. 1. Results of the work of the IFDS analysis method

Тип дефекта	Проект	SNAP	WebGoat		OmniDB + Spartacus		netMQ	ShareX	shadowsoc	OpenRA	banshee	cassandra	Всего
LDAP INJECTION	TP	8											8
	FP												
SQL INJECTION	TP		15	(29)	35	(630)							50
	Специально				2	(36)							2

	FP												
COMMAND INJECTION	**TP**												
	FP				1								1
CONSTANT CREDENTIALS	**TP**				75		1	1					77
	Тесты	1							6		1	10	18
	Инициализация				4								4
	Пустой пароль							3	1	12			16
	FP				1								1
INFORMATION_ EXPOSURE	**TP**												
	Tests												
	FP						64						
RESOURCE INJECTION	**TP**		1										1
	FP				2								2

Результаты работы анализатора были проанализированы вручную с целью оценки соотношения истинных срабатываний. Было установлено, что часть ошибок обнаружено в тестах и при инициализации переменных, а также некоторые предупреждения оказались ложными, поскольку указали на предусмотренные сценарии использования программы.

4. Анализ помеченных данных на основе символьного выполнения

В отличие от подхода IFDS, в данном методе помечаются символьные значения. Множества истоков, стоков и передаточных функций могут состоять из:

- вызовов методов, с отмеченными входными и/или выходными параметрами;
- обращений к полям класса на запись или чтение;
- формальных параметров метода.

Базовый алгоритм можно описать следующим образом. Пусть состояние программы в ходе символьного выполнения задаётся текущей инструкцией I, предикатом текущего пути P, множеством символьных значений V и отображением множества переменных на множество символьных выражений.

Дополним это состояние множеством помеченных значений $T \in V$.

В ходе символьного выполнения для каждого состояния выполняются следующие проверки:

- если I входит в множество истоков, то символьные значения, соответствующие её выходным параметрам, добавляются в множество T (то есть становятся помеченными);
- если I входит в множество передаточных методов и хотя бы один её входной параметр является помеченным, то все выходные параметры

также становятся помеченными;

- если I входит в множество стоков и хотя бы один входной параметр является помеченным, то алгоритм обнаружил дефект в программе.

Если используется символьное выполнение с объединением состояний, то при объединении множество помеченных символьных значений получается объединением двух исходных множеств: $T = T_1 \cup T_2$.

Приведённый алгоритм может констатировать факт достижения помеченными данными стока, однако для его практического использования, например, проверки истинности выданного предупреждения, необходимо построение множества диагностических точек, демонстрирующих пользователю анализатора путь в программе от истока до стока. Для этого каждому элементу из множества T требуется сопоставить структуру данных TInfo, сохраняющую и накапливающую информацию об истоке помеченности данного значения и пройденных передаточных методах. Путь распространения пометки будем называть трассой. Следует отметить, что эта структура может хранить больше одной трассы ввиду наличия передаточных методов с несколькими входными параметрами (например, операция соединения строк сохраняет помеченность обоих своих аргументов), а также в случае объединения символьных состояний (когда по разным путям одна переменная была помечена из разных источников)

Кроме того, рассмотренный алгоритм является внутрипроцедурным. Чтобы была возможность применять его для обнаружения межпроцедурных путей, можно использовать резюме методов - структуру данных, сохраняющую информацию о свойствах метода, выявленных в результате внутрипроцедурного анализа. Следует отметить, что в этом случае становится важным порядок анализа методов: он должен проводиться в обратном топологическом порядке (начиная с листовых вершин) по графу вызовов, чтобы вызывающие методы имели информацию о свойствах вызываемых методов.

В символьное состояние программы следует добавить множество PT потенциально помеченных значений, полностью аналогичное ранее введённому множеству T помеченных значений, с тем исключением, что:

- изначально в множество PT добавляются символьные значения, соответствующие формальным параметрам исследуемого метода, а в соответствующем TInfo сохраняется информация о данной точке входа в метод;
- инструкции из множества истоков не добавляют элементы в множество PT;
- при достижении потенциально помеченным значением инструкции из множества стоков в резюме метода добавляется информация о том, что метод становится межпроцедурным стоком с указанием точки входа в метод и участком трассы;

- аналогичным образом при выходе из тела метода в резюме добавляется информация о том, что метод становится межпроцедурным истоком (для значений из множества T) и/или передаточным методом (для значений из множества PT), с указанием выходных символьных значений и участком трассы.

Межпроцедурные истоки, стоки и передаточные методы обрабатываются таким же образом, как и обычные, за тем исключением, что связанный с ними участок трассы прибавляется к внутрипроцедурной трассе, формируя межпроцедурную трассу распространения помеченности данных.

Для поиска различных типов ошибок необходимо использование различных категорий пометок. В связи с этим некоторые из них приходится анализировать раздельно, тем самым повторно выполняя анализ одних и тех же путей передачи помеченных данных (так как множества передаточных методов, как правило, не отличаются). Одним из способов решения данной проблемы является введение понятия тега. Тег - это наследуемый от истока идентификатор, который приписывается точке входа в структуре TInfo. Теги прозрачны для передаточных методов и проверяются по достижении стока: если теги помеченных данных и стока не совпадают, то никакие действия не выполняются.

Для борьбы с ложными срабатываниями, возникающими из-за наличия в программе недостижимого кода, в структуре данных, содержащий информацию о помеченности, предусмотрено хранение предикатов пути и используются следующие правила объединения состояний A с предикатом пути P_A и B с предикатом P_B:

- если значение переменной v является помеченным в состоянии A, но не помечено в состоянии B значение переменной v будет считаться помеченным с предикатом P_A;
- если значение переменной v является помеченным и в состоянии A и в состоянии B, при этом у них общий исток значение переменной v считается помеченным с предикатом $(P_A \wedge P_B)$;
- если значение переменной v является помеченным и в состоянии A и в состоянии B, но у них различные истоки в TInfo сохраняются обе трассы, каждая со своим предикатом.

При достижении стока помеченным значением с предикатом P_t в состоянии с предикатом пути P строится формула, соответствующая предикату $P_t \wedge P$, которая передаётся SMT-решателю. Если решатель приходит к выводу, что формула не является выполнимой, сообщение о дефекте не выдаётся.

5. Результаты тестирования

Тестирование обоих методов проводилось на одинаковых машинах, использовался один и тот же набор проектов с открытым исходным кодом. У рассматриваемых методов анализа помеченных данных есть только один

общий детектор – поиск внедрения SQL кода, и оба детектора выдают одинаковое множество предупреждений на всём наборе тестов. Таким образом, вне зависимости от способа реализации, возможно достигнуть достаточной полноты анализа.

Тестирование производительности рассмотренных методов показывает следующие результаты:

Табл. 2. Сравнение производительности методов анализа
Table. 2. Comparison of the performance of analysis methods

	WebGoat.NET	OmniDB	CodeContracts
IFDS	10 сек.	117 сек.	377 сек.
Символьное выполнение	7 сек.	182 сек.	573 сек.
Символьное выполнение без построения резюме передаточных функций	7 сек.	129 сек.	512 сек.

Данные результаты соответствуют запуску анализатора с только одним включённым детектором SQL_INJECTION. Результирующий набор выданных предупреждений одинаков для всех запусков. Таким образом, можно отметить, что указанные методы имеют сравнимую производительность, однако основанный на IFDS, при условии работы только одного детектора, оказывается быстрее. В случае символьного исполнения необходимо вычислять существенно большее количество информации, необходимое для работы остальных детекторов, которое невозможно отключить. Кроме того, этот метод вычисляет информацию о возможных распространениях меток для всех функций, даже если такая информация не потребуется при дальнейшем анализе. Тестирование показало, что для указанных проектов эта информация несущественна, и отключение её вычисления существенно ускоряет работу метода, основанного на символьном выполнении, что показано в третьей строке.

Основные отличия в производительности подходов возникают при масштабировании. Тестирование показало, что существенное увеличение количества истоков оказывает неравномерное влияние. Это происходит при добавлении новых детекторов. Так, например, если часто используемую функцию ToString() объявить истоком, то время работы на OmniDB у IFDS подхода составит 278 секунд вместо 117 (замедление в 2.4 раза), а у второго - 223 сек. вместо 182 (замедление всего в 1.22 раза). Таким образом, при разработке новых детекторов необходимо принимать во внимание количество источников помеченных данных.

Заключение

В работе рассмотрены два подхода к реализации анализа помеченных данных с целью поиска ошибок в программах на языке C#: на основе IFDS и символьного выполнения. Оба метода были реализованы в рамках единой

инфраструктуры статического анализатора SharpChecker и протестированы на одинаковом наборе проектов с использованием одинакового аппаратного обеспечения. Исследования результатов и производительности показали, что возможно достижение сравнимой полноты анализа вне зависимости от использованного подхода. Метод IFDS имеет значимые преимущества в производительности при условии небольшого количества истоков, однако плохо масштабируется при их увеличении, что допустимо для реализации большинства детекторов.

Практика разработки детекторов для ошибок безопасности на основе анализа помеченных данных также показывает важность полноты множеств истоков, стоков и передаточных функций, а также точности моделирования окружения (библиотечных функций). Кроме того, наличие двух методов позволяет существенно повысить качество и производительность детекторов за счёт сравнения результатов их работы.

Дальнейшие исследования будут посвящены исследованию методов моделирования окружения, в том числе построения модели работы программы, которое заключается в дополнении графа вызовов рёбрами, позволяющими симулировать поведение пользователя. Для подхода IFDS актуальной является задача анализа псевдонимов, которая может повысить полноту результатов.

Список литературы

[1] Reps T., Horwitz S., Sagiv M. Precise Interprocedural Dataflow Analysis via Graph Reachability. Proceedings of the 22Nd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL ’95), San Francisco, California, USA, ACM, 1995, pp. 49–61

[2] Кошелев В.К., Игнатьев В.Н., Борзилов А.И. Инфраструктура статического анализа программ на языке C#. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 21–40. DOI: 10.15514/ISPRAS-2016-28(1)-2

[3] Кошелев В.К. Дудина И.А. Игнатьев В.Н. Борзилов А.И. Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 59–86. DOI: 10.15514/ISPRAS-2015-27(5)-5

[4] FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycleaware Taint Analysis for Android Apps. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14), Edinburgh, United Kingdom, ACM, 2014, pp. 259–269

[5] Christian Fritz et al. Highly Precise Taint Analysis for Android Applications. Tech. rep. EC SPRIDE, May 2013, 14 p. http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf, дата обращения 20.06.2017

Comparative analysis of two approaches to the static taint analysis

[1]*M.V. Belyaev <mbelyaev@ispras.ru>*
[1]*N.V. Shimchik <shimnik@ispras.ru>*
[1]*V.N.Ignatyev <valery.ignatyev@ispras.ru>*
[1,2] *A.A.Belevantsev <abel@ispras.ru>*
[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. Currently, one of the most efficient ways to find software security problems is taint analysis. It can be based on static analysis and successfully detect errors that lead to vulnerabilities, such as code injection or leaks of private information. Several different ways exist for the implementation of the algorithm for the taint data propagation through the program intermediate representation: based on the dataflow analysis (IFDS) or symbolic execution. In this paper, we describe how to implement both approaches within the existing static analyzer infrastructure to find errors in C# programs, and compare these approaches in different aspects: the scope of application, practical completeness, results quality, performance and scalability. Since both approaches use a common infrastructure for accessing information about the program and are implemented by a single development team, the results of the comparison are significant and can be used to select the best option in the context of the task. Our experiments show that it's possible to achieve the same completeness regardless of chosen approach. IFDS-based implementation has higher performance comparing with symbolic execution for detectors with small amount of taint data sources. In the case of multiple detectors and a large amount of sources the scalability of IFDS approach is worse than the scalability of symbolic execution.

Keywords: taint analysis; static analysis; IFDS; symbolic execution

DOI: 10.15514/ISPRAS-2017-29(4)-7

For citation: Belyaev M.V., Shimchik N.V., Ignatyev V.N., Belevantsev A.A. Comparative analysis of two approaches to the static taint analysis. *Trudy ISP RAN/Proc. ISP* RAS, vol. 29, issue 3, 2017, pp. 99-116 (in Russian). 10.15514/ISPRAS-2017-29(3)-7

References

[1]. Reps T., Horwitz S., Sagiv M. Precise Interprocedural Dataflow Analysis via Graph Reachability. Proceedings of the 22Nd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL '95), San Francisco, California, USA, ACM, 1995, pp. 49–61

[2]. Koshelev V.K., Ignatyev V.N., Borzilov A.I. C# static analysis framework. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 1, 2016, pp. 21-40 (in Russian). DOI:10.15514/ISPRAS-2016-28(1)-2

[3]. V. Koshelev, I. Dudina, V. Ignatyev, A. Borzilov. Path-sensitive bug detection analysis of C# program illustrated by null pointer dereference. Trudy ISP RAN / Proc ISP RAS, vol. 27, issue 5, 2015, pp.59–86 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-5

[4]. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycleaware Taint Analysis for Android Apps. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14), Edinburgh, United Kingdom, ACM, 2014, pp. 259–269

[5]. Christian Fritz et al. Highly Precise Taint Analysis for Android Applications. Tech. rep. EC SPRIDE, May 2013, 14 p. http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf, accessed 20.06.2017