Межпроцедурный анализ помеченных данных на базе инфраструктуры LLVM.

Кошелев В.К., Избышев А.О, Дудина И.А
ИСП РАН, Москва
{vedun, izbyshev, eupharina}@ispras.ru

Аннотация. В данной работе рассматривается задача анализа помеченных данных. Для её решения предлагается статический межпроцедурный котекстно-потоковый объектно-чувствительный алгоритм, производится оценка характеристик данного алгоритма, обсуждаются особенности реализации на базе компиляторной инфраструктуры LLVM и приводятся результаты практического тестирования.

Ключевые слова: статический анализ; межпроцедурный анализ; класс IFDS; анализ потоков данных; анализ помеченных данных

1. Введение

Разработка программного обеспечения (ПО) с повышенными требованиями к безопасности сопряжена с необходимостью дополнительного контроля качества. В частности, важной задачей является включение в процесс разработки ПО поиска уязвимостей, связанных с нарушением политик безопасности. Данный поиск может осуществляться с помощью статического анализа программного кода. Статический анализ производится без реального выполнения программы и затрагивает весь программный код, анализируя редко выполняемые фрагменты кода, тестирование которых затруднено сложностью воспроизведения.

Для поиска таких уязвимостей как, «утечка критических данных», «использование константных паролей» или же «отправка нешифрованных критических данных», необходимо использовать методы статического анализа, способные эффективно отслеживать потоки данных в программе. В данном случае задача отслеживания потоков данных может быть сведена к задаче анализа помеченных данных.

В данной работе предлагается межпроцедурный контекстно-зависимый алгоритм анализа помеченных данных, позволяющий реализовать поиск данных уязвимостей. Алгоритм поиска реализован на базе компиляторной инфраструктуры LLVM. В качестве представления программ, над которым осуществляется анализ, используется LLVM-биткод. Использование LLVM-биткода позволяет унифицировать представление программ на различных

языках программирования и использовать обширный набор анализов, уже реализованных в инфраструктуре LLVM.

Данная работа разбита на восемь частей. Во второй части производится обзор существующих подходов к анализу помеченных данных. В третьей части рассматривается решение задач из класса IFDS (класс IFDS представляет собой класс межпроцедурных задач анализа потоков данных с конечным множеством фактов и дистрибутивными передаточными функциями). Четвёртая часть посвящена сведению задачи анализа помеченных данных к задаче из класса IFDS. В пятой части рассматривается проблема анализа псевдонимов в контексте анализа помеченных данных. В шестой части приводится рад оптимизаций, позволяющих существенно уменьшить потребление памяти алгоритма. В седьмой части обсуждаются результаты тестирования реализации алгоритма анализа. Восьмая часть представляет собой заключение данной работы.

2. Существующие подходы к анализу помеченных данных.

Для начала сформулируем задачу межпроцедурного анализа помеченных данных. Пусть в программе заданы два множества функций: истоки и стоки. Задачей анализа помеченных данных является поиск всех таких параметров вызовов функций из множества стоков, что они зависят по данным от какого-либо параметра вызова функции из множества истоков. При этом считается, что все переходы по ветвлениям всегда являются допустимыми. Результатом такого анализа помеченных данных является набор путей в программе, вдоль которых помеченные данные распространяются от функции из множества истоков до функции из множества стоков.

На листинге 1 приведён пример программы на языке C, содержащий утечку помеченных данных. Данная программа обрабатывает события, полученные в ходе вызова функции `get_event`. Функция `get_event` возвращает либо событие `RECEIVE`, `PROCESS` или `SEND`, либо 0, если необходимо завершить работу. Предположим, что было получено событие `RECEIVE`, тогда в строке 11 функция source поместит помеченные данные в переменную `c`, после чего управление передаётся на строку 9. Далее, при получении события `PROCESS`, указатель на помеченные данные будет скопирован из переменной c в переменную d (строка 12). Наконец, при получении события `SEND`, помеченные данные, доступные из переменной d, передадутся в функцию sink. Таким образом, путь, на котором произойдёт утечка, будет выглядеть следующим образом: 7 – 9 – 11 – 9 – 12 – 4 – 12 –9 – 13.

```
1 char get_event(void); void source(void **c); void
sink(void **c);
2
3 void copy(void **a, void *b) {
```

```
4       *a = b;
5  }
6  int main(void) {
7       char *c, *d, e;
8       do {
9             e = get_event();
10            switch (e){
11                  case RECEIVE: source(&c); break;
12                  case PROCESS: copy(&d, c); break;
13                  case SEND: sink(&d); break;
14            }
15      } while (e);
16      return 0;
17 }
```

Листинг 1. Пример утечки помеченных данных.

Разрабатываемый в ИСП РАН инструмент статического анализа Svace осуществляет анализ помеченных данных для поиска уязвимостей типа «использование непроверенных входных данных»[1]. В настоящий момент Svace использует инфраструктуру анализа общего назначения для поддержки широкого класса проверок. Данная инфраструктура отвечает за построение модели памяти, основанной на графе объектов, и множества допустимых значений для данных объектов. Анализ функций программы осуществляется в обратном топологическом порядке относительно графа вызовов. Данный подход позволяет одновременно использовать целый набор проверок, что положительно сказывается на суммарном времени анализа, однако в случае, если функции из множеств истоков и стоков находятся достаточно далеко друг от друга, точность инструмента Svace оказывается недостаточной.

Недостаточная точность связана с компромиссом между точностью и масштабируемостью, к которому вынуждены были прибегнуть разработчики Svace. Данный компромисс заключается в жёстком ограничении размера графа объектов: если в ходе анализа функции размер графа объектов превышает допустимый максимум, система автоматически удаляет одну из вершин из графа в соответствии с заранее заданным критерием. Так как максимально допустимое число вершин в графе, как правило, намного меньше количества объектов в программе, то для достаточно длинной цепочки зависимостей можно почти гарантированно утверждать, что один из объектов, участвующих в данной цепочке, будет удалён при анализе. При этом изменение стратегии удаления вершин не меняет суть дела, так как анализ идёт в обратном топологическом порядке и в момент удаления ещё неизвестно, какие объекты будут помечены, а какие нет.

В университете Райса был предложен подход[2] к анализу помеченных данных на основе адаптации алгоритма Sparse Conditional Constant

Propagation(SCCP)[3]. Данный подход предполагает использование в SCCP вместо решётки для констант решётки из двух элементов «помеченные данные» и «непомеченные данные». Для моделирования памяти предлагается использовать Array SSA Form, представляя всю память как один массив. Однако в оригинальной статье утверждается[4], что Array SSA From для SCCP работает лишь с константными индексами массива и поддержка символьных значений является предметом дальнейших исследований. С другой стороны, тезис об использовании Array SSA Form никак более не поясняется, таким образом невозможно причины выбора данного представления.

Команда разработчиков проекта Flowdroid из EC SPRIDE предложила свести задачу анализа помеченных данных к задаче из класса IFDS[5]. Их главной целью было создание средства анализа Java-приложений для платформы Android, позволяющего находить утечки пользовательских данных, поэтому точность анализа была основным приоритетом. Предложенный ими подход является потоково-,объектно- и контекстно-чувствительным, однако объём потребляемой памяти оставляет желать лучшего. В ходе тестирования средства анализа Flowdroid было установлено, что на достаточно больших реальных приложениях (около десяти тысяч рёбер в графе вызовов) анализ не успевает завершиться до того, как израсходует всю доступную память (При тестировании для анализа выделялось 24 гигабайта оперативной памяти). Более того, другие пользователи проекта FlowDroid также подтверждают нехватку памяти для завершения анализа [6]. Так как целью данной работы является, прежде всего, анализ достаточно больших приложений, то подход проекта FlowDroid должен быть доработан в сторону уменьшения потребления памяти.

3. Класс задач IFDS

Как уже отмечалось ранее, класс задач IFDS[7] представляет собой класс межпроцедурных задач анализа потоков данных с конечным множеством фактов и дистрибутивными передаточными функциями (В данной работе нас интересуют только задачи с операцией объединения в качестве meet-оператора). При этом дистрибутивность передаточных функций означает, что для любого факта d, подмножества фактов анализа потоков данных D' и передаточной функции $F_{<v,u>}$ верно, что $F_{<v,u>}(d \cup D') = F_{<v,u>}(d) \cup F_{<v,u>}(D')$. Из данного определения следует, что для пустого множества и любого факта d всегда верно, что $F_v(\emptyset) \subseteq F_v(d)$. Поэтому введём дополнительный факт λ и новую передаточную функцию F'_v, такую, что:

$$F'_{<v,u>}(d) = F_{<v,u>}(d) \setminus F_{<v,u>}(\emptyset);\ F'_{<v,u>}(\lambda) = F_{<v,u>}(\emptyset) \cup \lambda;\ F'_{<v,u>}(\emptyset) = \emptyset.$$

Далее будем считать, что над передаточными функциями уже было проделано данное преобразование и выделен так называемый «пустой» факт λ.

Введём понятие точного решения для задачи из класса IFDS. Для этого рассмотрим все корректные пути в программе (под корректностью понимается, что при очередном возврате из функции управление должно быть передано именно в точку возврата предыдущего вызова данной функции в данном пути). Будем говорить, что для вершины u межпроцедурного графа потока управления верен факт анализа потока данных d, если найдётся такой путь $P = (e, v_1, v_2, \dots v_m, u)$, где e – точка входа в программу, что $d \subseteq F_{<v_m,u>}(F_{<v_{m-1},v_m>}(\dots F_{<e,v_1>}(\lambda)))$, т.е. d принадлежит к композиции передаточных функций. Тогда под точным решением будем понимать множество пар $< v, D_v >$, где v – вершина из межпроцедурного графа потока управления, а D_v – множество всех верных для неё фактов анализа потоков данных.

Для нахождения точного решения[7] задачи из класса IFDS было предложено сведение задачи межпроцедурного анализа потока данных к задаче достижимости на графе. Для этого межпроцедурный граф потока управления G преобразуется в расширенный граф G'. Вершины расширенного графа представляют собой декартово произведение $V_{G'} = V_G \times D$, где V_G – множество вершин в исходном графе, а D – множество фактов анализа потоков данных. Наличие ребра между вершинами в графе G' определяется следующим образом: между вершинами $< v, d > u\ < u, d' >$ есть ребро тогда и только тогда, когда ребро $< v, u > \in G\ u\ d' \in F_{<v,u>}(d)$. Такое определение графа G' с учётом дистрибутивности передаточных функций гарантирует тот факт, что существование корректного пути из начальной вершины $< e, \lambda >$ в вершину $< v, d >$ равносильно тому, что факт d верен в вершине v. Стоит отметить, что контекстная чувствительность анализа достигается как раз за счёт требования существования корректного пути.

Для поиска точного решения был предложен алгоритм, основанный на динамическом программировании, позволяющий найти его за время $O(|D|^3 * |E|), где\ |D|$ – мощность множества фактов анализа потока данных, а $|E|$ – мощность множества ребёр в графе G. Оригинальный алгоритм требовал построения графа G' в явном виде, однако граф G' может содержать до $O(|D| * |E|)$ вершин и $O(|D|^2 * |E|)$ рёбер, что делает невозможным его построение для реальных программ при хоть сколько-нибудь большой мощности множества D. К счастью, граф G' зачастую оказывается

несвязанным, и из начальной вершины $< e, \lambda >$ доступна лишь незначительная его часть. Поэтому впоследствии были предложены улучшения оригинального алгоритма, не требующие более построения G' в явном виде. В данной работе используется вариация такого алгоритма: пожертвовав полной поддержкой рекурсивных функций, была достигнута возможность полного анализа пары «функции и её контекста».

```
1 Global FunctionValues = Map(),Result = Map()
2 Function DFSBFS(F, C)
3   V = F.getEntry(), Q = Result.getQueue(<F, C>)
4   while (!Q.isEmpty())
5     V, D = Q.pop()
6     if (V.isCall())
7       DestF = V.getDest()
8       DestD = caller2callee(V, D)
9       if (not FunctionValues.contains(<DestF, DestD>))
10         DFSBFS(DestF, DestD)
11      for D' in callee2caller(V, FunctionValues
[<DestF, DestD>])
12        Q.addIfNotVisited(<V.getReturnSide(), D'>)
13     for D' in call2return(V, D)
14        Q.addIfNotVisited(<V.getReturnSide(), D'>)
15    else if (V.isReturn()) FunctionValues[<F,
C>].insert(D)
16    else for V' in V.succs()
17      for D' in normal(V, V', D)
18.       Q.addIfNotVisited(<V', D'>)
```

Листинг 2. Псевдокод алгоритма решения задачи IFDS.

Рекурсивная функция DFSBFS (листинг 2) строит для каждой функции и контекста её вызова множество посещённых в ней вершин графа G'(глобальная переменная Result). Разберём подробнее алгоритм работы данной функции. В начале выполнения функции (строчка 3) происходит получение точки входа в функцию – вершины V – и получение очереди Q, поддерживающие две операции: добавить пару «вершина и факт анализа потока данных» в очередь, если они до этого ещё не были добавлены, и получить очередную пару из очереди. Далее, пока очередь не пуста, происходит обход в ширину вершин графа G'(4-18). Отдельно рассматривается случай, когда вершина графа G представляет собой вызов функции(6-14). Основная идея алгоритма заключается в замене повторного анализа функций с данным контекстом на использование уже ранее

посчитанного множества фактов анализа потоков данных. Поэтому, если вызываемая функция с полученным контекстом ещё не анализировалась, то необходимо вызвать её анализ при помощи рекурсивного вызова функции DFSBFS(9-10), после этого результат анализа будет гарантированно содержаться в переменной `FunctionValues`. Далее полученные значения преобразуются с помощью передаточной функции Callee2Caller, и новые вершины добавляются к обходу в ширину, если это необходимо (11-12). Несложно показать, что данный алгоритм, так же, как и исходный, будучи запущенным от точки входа программы и факта λ, будет иметь вычислительную сложность $O(|D|^3 * |E|)$, однако при этом будут проанализированы лишь вершины, доступные из начальной вершины.

Заметим, что в реальной реализации данного алгоритма для каждой посещённой вершины сохраняется её предок с целью обеспечить возможность восстановления пути после нахождения обнаружения утечки. Поэтому после окончания анализа данные о посещённых вершинах не могут быть освобождены и хранятся всё время работы алгоритма, составляя основную часть расходов оперативной памяти.

4. Сведение задачи анализа помеченных данных к задаче из класса IFDS

Данный раздел посвящён заданию множества фактов анализа потоков данных и передаточных функций, позволяющих решать задачу анализа помеченных данных. Так же, как и разработчики проекта Flowdroid, в данном алгоритме используется модель памяти, основанная на путях доступа (access path).

Путь доступа представляет собой имя переменной в программе, к которой последовательно применялись операции смешения на константу (в байтах) и разыменования. Соответственно, использование путей доступа в качестве фактов анализа потоков данных позволяет сделать анализ объектно-чувствительным. Например, если в программе имеется переменная int **p, то на её основе могут быть получены такие пути доступа, как *(p + 4) или же *(*(p + 8)). Формально путь доступа задаётся именем переменной N и последовательностью смещений и разыменований. Так как два подряд идущих смещения гарантированно можно объединить в одно, то будем считать, что для задания смещений и разыменований достаточно одной последовательности смещений, если считать, что между двумя смещениями обязательно происходит разыменование. Таким образом, путь доступа формализуется как имя переменной и последовательность чисел.

Для построения передаточных функций необходимо ввести операции над путями доступа. Всего в данной работе будут рассмотрены четыре операции по преобразованию путей доступа и один предикат, задающий частичный порядок.

Для пути доступа WholeAP путь SubAP является подпутём при выполнении следующих условий. Во-первых, пути доступа WholeAP и SubAP имеют одно и то же имя переменной. Во-вторых, длина последовательности смещений у SubAP меньше либо равна длине последовательности у WholeAP. В-третьих, все смещения, кроме последнего, у SubAP совпадают со смещениями у WholeAP, а последнее либо совпадает, либо равно 0. Например, для доступа *(*(p + 4)+2) + 0 путь p + 0 является подпутём доступа, а путь *(p + 0) + 0 – нет.

Операция Callee2Caller(CallerAP, CalleeAP) используется для объединения пути доступа CalleeAP с путём доступа CallerAP. Семантика данного объединения задаётся следующим выражением, где символ “;” отделяет имя переменной от последовательности смещений.

$$CallerAP = (A; a_0, \dots a_n);$$

$$CalleeAP = (B; b_0, \dots b_m);$$

$$Callee2Caller = (A; a_0, \dots, a_n + b_0, b_1, \dots, b_m)$$

Операция Caller2Callee(CallerAP, Fact, B) используется для передачи пути доступа из вызывающей функции в вызываемую функцию. Символ B означает имя аргумента для нового пути доступа в вызываемой функции. Предполагается, что путь CallerAP является подпутём для пути доступа Fact.

$$CallerAP = (A; a_0, \dots a_n);\ Fact = (A;\ a_0, \dots a_{n-1}, c_0, \dots c_m)$$

$$Caller2Callee = (B;\ c_0 - a_n, c_1 \dots c_m)$$

Операция ChangeSubPath(OldAP, NewAP, AP) используется для моделирования копирования из одного пути доступа в другой путь доступа. В данном случае OldAP является подпутём для AP.

$$OldAP = (A; a_0, \dots a_n);\ NewAP = (B; b_0, \dots b_m);$$

$$AP = (A; a_0, \dots a_{n-1}, c_0, \dots c_m)$$

$$ChangeSubPath = (B;\ b_0 \dots b_m - a_n + c_0, c_1 \dots c_m)$$

Операция Deref(AP) используется для разыменования пути доступа AP.

$$AP = (A; a_0, \dots a_n);$$

$$Deref = (A; a_0, \dots a_n, 0);$$

Для поддержки такого случая смещения переменной на не константное значение, введём специальное смещение «*» со следующей семантикой: «*» равно любому числу и в том числе «*», операции сложения и вычитания «*» с использованием в качестве операнда всегда дают результат «*». Множество значений смещений может быть дополнено значением «*» с сохранением семантики введённых операцией.

В LLVM-биткоде результат выполнения каждой инструкции, если он имеется, сохраняется в псевдорегистр, представленный в SSA-форме. Для введения передаточных функций рассмотрим аналоги наиболее важных инструкций из LLVM-биткода.

- *store(B, A)* – сохранить по адресу A значение B
- *val = load(A)* – загрузить значение по адресу A в псевдорегистр val
- *val = getelementptr(A, offset)* – прибавить к адресу A offset байт и поместить в псевдорегистр val (на самом деле в llvm инструкция getelementptr работает не байтовыми смещениями, а со смещениями, исчисляющихся в членах типа, на который указывает A, но для упрощения повествования, будем считать, что getlementptr оперирует непосредственно с байтовыми смещениями).
- *val = call(foo, [arg1, arg2, ...])* – вызов функции foo.
- *ret [A]* – возврат из функции, если необходимо возвращается значение A
- *val = phi(val1, label1, val2, label2, ...)* – Фи функция для значений val1, val2 и.т.д, соответствующих базовым блокам label1, label2 и т.д.
- *val = binop(A ,B)* – бинарная операция над значениями, результат сохраняется в псевдорегистр *val*.
- *val = unop(A)* – унарная операция, результат сохраняется в псевдорегистр *val*.
- *val = alloca(type)* – операция выделения sizeof(type) памяти на стеке, в псевдорегистр val помещается указатель на выделенную память.

Так как данный статический анализ игнорирует условия на переходах, заменим стандартные инструкции перехода на jmp и jmpcond.

- *jmp label* – безусловный переход на метку label.
- *jmpcond label* – переход на метку label либо же переход на следующую инструкцию.

Сопоставим каждому псевдорегистру и аргументу функции путь доступа следующим образом. Аргументу функции с именем name соответствует путь доступа $name + 0$, псевдорегистрам инструкций val = phi(…), val = call(…) и val = alloca(size) соответствует путь доступа val + 0. Псевдорегистру инструкции val = load(A) соответствует путь доступа *(AP(A)), где AP(A) – путь доступа, соответствующий значению A. Аналогично, для инструкции val = getelementptr(A, size), псевдорегистру val будет соответствовать путь доступа AP(A) + size.

На листинге 3 приведён пример биткода, соответствующего примеру программы, содержащей утечку данных (листинг 1). В комментариях после инструкций приведены пути доступа, соответствующие их псевдорегистрам.

```
1  declaration i8 get_event(void)
2
3  definition void copy(i8 **a, i8 *b) // a + 0 и b + 0
4    store b, a
5    ret
6
7  definition int main(void)
8   i8** c = alloca(i8*)// c + 0
9   i8** d = alloca(i8*)// d + 0
10  loop_start:
11    i8 e = get_event()// e + 0
12    jmpcond receive
13    jmpcond process
14    jmpcond send
15    jmp end
16    receive:
17      call source(c)
18      jmp end
19    process:
20      c_value = load(c) // *(c + 0)
21      call copy(d, c_value)
22      jmp end
23    send:
24      call sink(d)
25      jmp end
26    end:
27    jmpcond loop_start
28  ret 0
```

Листинг 3. Примера утечки помеченных данных .

Реализуем передаточные функции Normal, Caller2Callee, Callee2Callee и Call2Return для алгоритма DFSBFS, используя вышеописанные операции преобразования. Метод `getArg(i)` для функции возвращает путь доступа, соответствующий i-тому формальному параметру. Функция `ReportLeak(V, D)` сообщает о найденной утечке.

```
  normal(V, V', D)
1  Result = Set()
2  if (V is store(B, A))
3    if (not (deref(A) is subpath for D))
4      Result.insert(D)
5    if (B is subpath for D)
6      Result.insert(ChangeSubPath(B, deref(A), D))
7  else
8    if (V is val = phi(a1, … an))
9      for i in (1, n) if (ai is subpath for D)
10       Result.insert(ChangeSubPath(ai,val,D))
```

```
11    if (V is val = binop(A, B) and (D is subpath for A or B))
12      Result.insert(val)
13   if (V is val = unop(A) and (D is subpath for A))
14      Result.insert(val)
15   Result.insert(D)
16   return Result

  caller2callee(V, D)
1   Result = Set()
2   if V is val = call Func(a1, … , an)
3     for i in (1, n)
4       if (ai is subpath for D)
5         Result.insert(Caller2Callee(ai, D,
Func.getArg(i))
6   return Result

  callee2caller(V, D)
1   Result = Set()
2   if V is ret from val = call Func(a1, … , an)
3     for i in (1, n)
4       if (Func.getArg(i) is subpath for D)
5         Result.insert(Callee2Caller(ai, D)
6   return Result

  call2return(V, D)
1   Result = Set()
2   Flag = True
3   if V is call Func(a1, … , an)
4       for i in (1, n)
5         If (ai is subpath for D)
6           Flag = false
7     if (Flag) Result.insert(D)
8     if (Func == Source)
9       Result.insert(deref(a1))
10    if (Func == Sink and a1 is subpath for D)
11      ReportLeak(V, D)
12  return Result
```

Листинг 4. Реализация передаточных функций.

Имея передаточные функции, применим алгоритм поиска утечек помеченных данных к примеру из листинга 3. Анализ начинается со строки 8 и пустого факта λ, далее с ним посещаются все инструкции. При посещении инструкции вызова функции source (строка 17) порождается факт *(c + 0) и посещает все инструкции с 11 по 28, в том числе и вызов функции copy. При обработке

вызова функции copy с фактом *(c + 0), передаточная функция устанавливает, что *(c + 0) является подпутём для параметра c_value, которому соответствует путь доступа *(c + 0), и производит преобразование Caller2Callee, получая из факта *(c + 0) факт b + 0. После этого происходит анализ функции copy с контекстом b + 0. Он начинается с посещения фактом b + 0 инструкции копирования (строка 4), где в ходе вызова передаточной функции устанавливается, что записываемое значение является подпутём для факта b + 0, поэтому, с учётом разыменования, выполняется преобразование ChangeSubPath и порождается факт *(a + 0). После этого факт *(a + 0) достигает инструкции возврата ret и записывается в результат вызова функции copy с контекстом b + 0. После завершения анализа функции copy с контекстом b + 0, анализ функции main с контекстом λ продолжается с обратного преобразования результатов вызова. В результате преобразования Callee2Caller факт *(a + 0) функции copy преобразуется в факт *(d + 0) функции main. Далее данный факт также посещает все инструкции с 11 по 28 и инициирует анализ функции copy с контекстом *(a + 0). После достижения фактом *(d + 0) инструкции на строке 24 передаточная функция обнаружит, что данный вызов является вызовом функции из множества стоков, и путь доступа параметра вызова функции d + 0 является подпутём для факта *(d + 0), следовательно, найдена утечка помеченных данных.

5. Анализ псевдонимов

Ключевой задачей для анализа помеченных данных является анализ псевдонимов. В данной работе используется подход, схожий с подходом проекта Flowdroid. Данный подход решает задачу анализа псевдонимов как задачу из класса IFDS. Главное преимущество использующегося подхода заключается в поиске псевдонимов не для всех переменных, а лишь для переменных, содержащих помеченные данные.

Стоит заметить, что данный алгоритм ищет не псевдонимы в обычном смысле, а такие пути доступа (назовём их неактивными путями доступа), которые при выполнении определённой инструкции (назовём её инструкцией активации) будут содержать помеченные данные. Иными словами, они являются псевдонимом для какого-то другого пути доступа, при выполнении записи помеченных данных в который данные в них также окажутся помеченными.

Поиск данных путей доступа осуществляется лениво: при очередной операции записи в память или же возврата из функции, в ходе которых образовались новые помеченные данные, для каждого нового помеченного пути доступа происходит поиск связанных с ним неактивных путей доступа. Для поиска неактивных путей доступа используется комбинация прямой и обратной IFDS-задач. Оба анализа используют множество фактов, основанное на паре «путь доступа и инструкция активации». Для начального факта инструкция активации совпадает с инструкцией, породившей новый факт, а путь доступа с

новым помеченным путём доступа. Передаточные функции для обратного анализа задаются следующим образом. Функции caller2callee и callee2caller не порождают никаких фактов, т.к. на данный момент используется внутрипроцедурная версия обратного анализа. Функция call2return передаёт полученный факт через точку вызов, реализация функции normal приведена на листинге 5. Функция `InjectForward(V, D)` добавляет пару `<V, D>` в прямой анализ псевдонимов, в тот же контекст, из которого была вызвана функция normal.

```
    normal(V, V', D)
1   Result = Set()
2   if (V is store(B, A))
3     if (deref(A)is subpath for D)
4       injectForward(V, D)
5       Result.insert(ChangeSubPath(deref(A), B, D))
6     else
7       Result.insert(D)
8   else if (V is val = phi(a1, …, an))
9     if (val is subpath for D)
10     for i in (1, n)
11       Result.insert(ChangeSubPath(val, ai, D))
12     injectForward(V, D)
13   else
14     Result.insert(D)
15  else if (V is val = alloca(…) or val = call(…) or V
is entry)
16    injectForward(V, D)
17  else
18    Result.insert(D)
19  return Result
```

Листинг 5. Обратная передаточная функция.

Обратим внимание на то, что как только факты анализа достигают точки своего определения (строки 4, 12 и 16), они передаются обратно прямому анализу. В данном случае прямой анализ использует те же передаточные функции, что и обычный анализ помеченных данных, с той лишь разницей, что инструкции unop и binop более не порождают помеченные данные. Более того, если прямой анализ обнаруживает запись помеченных данных в новый путь доступа, он запускает обратный анализ, беря инструкцию активации из факта. Если же факт достигает своей инструкции активации, то он дальше не передаётся, а вместо этого возвращается в анализ помеченных данных уже активированным.

```
1  definition copy(i8 **a, i8 *b)
2  a.addr = alloca (i8***)
```



```
3  b.addr = alloca (i8**)
4  store a, a.addr
5  store b, b.addr
6  a_value = load a.addr
7  b_value = load b.addr
8  store b_value, a_value
9  ret
```

Листинг 6. Альтернативная реализация функции copy.

Для демонстрации работы анализа псевдонимов, рассмотрим альтернативную реализацию функции copy из листинга 3. Данная реализация (листинг 6) генерируется компиляторной инфраструктурой LLVM в случае, когда не используется оптимизация отображения переменных из памяти на регистры. Так же, как и в примере из листинга 3, будем считать, что функция copy анализируется с контекстом b + 0. Несложно заметить, что оригинальный алгоритм не способен найти передачу помеченных данных в путь доступа относительно аргумента a, так как она происходит неявно. Для записи фактов будем использовать следующую нотацию: «путь доступа» для активных фактов и «путь доступа, число» для неактивных фактов с инструкцией активации на строке «число».

Рассмотрим работу алгоритма с учётом анализа псевдонимов. В начале работы алгоритм проталкивает факт «b + 0» до инструкции на строке 5, где происходит его запись в путь доступа «*(b.addr + 0)» далее запускается анализ псевдонимов, который, ничего нового не обнаружив, завершается. Далее путь факт «*(b.addr + 0)» доходит до инструкции на строке 8, порождая новый факт «*(*(a.addr + 0) + 0)», для данного факта также запускается анализ псевдонимов с фактом «*(*(a.addr + 0) + 0), 8». Идя назад, данный факт встречает инструкцию на строке 4, что приводит к тому, что в прямой анализ псевдонимов добавляется факт «*(*(a.addr + 0) + 0), 8», а обратный продолжается с фактом «*(a + 0), 8». После достижения точки входа в функцию факт «*(a + 0), 8» также добавляется в прямой анализ. В свою очередь, прямой анализ проталкивает факт «*(a + 0), 8» до его инструкции активации на строке 8, активирует его и возвращает в анализ помеченных данных. После этого анализ помеченных данных посещает инструкцию на строке 9 с фактом «*(a + 0)» и добавляет его к результатам анализа функции.

К сожалению, если считать, что факты анализа потока данных, различающиеся только инструкциями активации, разные, то количество фактов анализа может увеличиться в $O(E)$ раз, что недопустимо с точки зрения масштабируемости. Для решения данной проблемы факты, различающиеся лишь инструкцией активации, считаются одинаковыми. Эта эвристика позволяет существенно уменьшить число фактов, получаемых при анализе псевдонимов, однако в случае, если один и тот же путь доступа помечается в нескольких различных инструкциях, то инструкция для

активации неактивных фактов будет выбрана произвольно. Из-за этого может возникнуть ситуация, когда неактивный факт попал в функцию из множества стоков, пройдя породившую его инструкцию, но не пройдя выбранную. На данный момент неясно, насколько такая ситуация распространена в реальных приложениях, однако без данной эвристики анализ приложений даже из ста тысяч инструкций нецелесообразен.

6. Оптимизация времени работы алгоритма

Работа анализа псевдонимов устроена таким образом, что большинство неактивных фактов при своей генерации сначала поднимаются обратным анализом до точки входа в функцию, после чего спускаются прямым анализом, посещая каждую инструкцию. Поэтому, если функция из нескольких тысяч инструкций анализируется с десятками различных контекстов и при этом генерируется несколько сотен неактивных фактов, то значительное количество памяти тратится на посещение инструкций, неспособных породить новых фактов. Для решения данной проблемы вместо передачи факта к смежным вершинам можно передавать его сразу в точку ближайшего использования относительно исходной вершины.

Если в инструкцию могут быть переданы только факты, имена переменных путей доступа которых используются в данной инструкции, то в рамках одного контекста общее число посещений данной инструкции ограничено произведением числа возможных путей доступа для данного имени на количество аргументов у инструкции. Так как число путей доступа может быть произвольно большим даже для ограниченной длины самого пути, то необходимо отдельно ограничить число поддерживаемых путей некоторой константой A. Осталось оценить число различных контекстов, в которых может быть проанализирована функция. Пусть количество аргументов у функции не превосходит K, тогда всего общее количество контекстов без учёта глобальных переменных будет равно $O(K * A)$. Так как всего в программе $O(E)$ инструкций, то суммарное количество посещённых вершин, и соответственно объём занятой памяти, будут равны $O(K * A^2 * E)$. При этом затраты времени окажутся равными $O(TF * K * A^2 * E)$, где $O(TF)$ – временная сложность вычисления передаточной функции.

Для того чтобы иметь возможность переходить на ближайшее использование, предлагается для каждого имени, использующегося в инструкции, предподсчитать ближайшие использования. Так как в каждой функции имеется $O(E_F)$ инструкций, а ближайшие использования могут быть найдены, например, поиском в ширину за время $O(E_F)$, то для всех инструкций они могут быть найдены за время $O(E_F^2)$. Т.е. суммарное время работы алгоритма для всех функций составит:

$$O(\sum_F E_F^2)$$

Соответственно, суммарная сложность анализа помеченных данных равна:

$$O\left(TF * K * E * A^2 + \sum_F E_F^2\right)$$

Однако данная оценка неверна для прямой фазы алгоритма анализа псевдонимов, т.к. инструкции, способные выступать в роли инструкций активации, должны быть посещены неактивными фактами. Данный факт приводит к тому, что инструкции store и call могут быть посещены $O(E_F * A)$ раз. Учитывая, что в реальных приложениях доля таких инструкций достигает 25%, суммарная оценка времени работы алгоритма составляет:

$$O\left(\sum_F TF * K * A^2 * E_F^2\right)$$

7. Результаты

Предложенный алгоритм поиска помеченных данных реализован в качестве прохода для компиляторной инфраструктуры LLVM. Кроме того, в LLVM были также реализованы вспомогательные проходы, а именно проход, отвечающий за построение межпроцедурного графа потока управления, проход, отвечающий за анализ виртуальных вызовов в C++ и проход, реализующий инфраструктуру для разработки дополнительных проверок.

Межпроцедурный граф потока управления отличается от внутреннего представления LLVM, а именно, у него выделены точки входа в функцию, точки возврата после вызова функции и построены все межпроцедурные рёбра. Для построения межпроцедурного графа потока управления был реализован анализирующий проход, преобразующий внутреннее представление LLVM в межпроцедурный граф потока управления. Для построения межпроцедурных рёбер для виртуальных вызов в C++ данный проход использует результаты анализатора виртуальных вызовов, разработанного в ИСП РАН.

Дополнительные проверки, основанные на данном анализе, предполагается реализовать с помощью модификации передаточных функций. Разработчику новой проверки предоставляется возможность проанализировать построенный межпроцедурный граф потока управления и пометить вершины, для которых должна быть запущена дополнительная реализация передаточных функций. Также он имеет возможность указать, должна ли эта реализация быть запущена до, после или же вместо оригинальной реализации. В качестве тестовой проверки была реализована проверка того, что константные значения не передаются в указанные функции.

Отчёты о найденных трассах могут быть либо показаны последовательным списком инструкций, либо же представлены в виде размеченного межпроцедурного графа потока управления, демонстрирующего последовательность распространения помеченных данных.

Прежде чем перейти к результатам тестирования, необходимо заметить, что алгоритм использует целый набор эмпирических констант, от значения которых напрямую зависят результаты анализа. В данном тестировании максимальная длина пути доступа была ограничена четырьмя, а максимальное количество путей доступа для одного имени – тридцатью.

Тестирование проводилось как на синтетических тестах, разработанных специально для тестирования данной реализации, так и на наборе реальных приложений, а именно stl vector, md5 и openvpn. Необходимо заметить, что дополнительную сложность при тестировании представляет задача сборки тестируемого проекта в LLVM-биткод, т.к. такая сборка почти никогда не поддерживается разработчиками.

Целью тестирования на stl vector была проверка способности анализа отслеживать помеченные данные, даже если они были добавлены в контейнер. Так как stl vector является шаблонным классом, то его исходный код автоматически оказывается доступным в виде LLVM-биткода при анализе. Тестирование показало, что реализованный алгоритм может успешно отслеживать добавление и извлечение помеченных данных из контейнера stl vector.

Целью тестирования на алгоритме шифрования md5 была проверка возможности отслеживать зависимость зашифрованной строки от пароля, который изначально подвергался шифрованию. При тестировании было установлено, что выходной буфер алгоритма зависит от изначальной строки.

Целью тестирования на приложении openvpn (свободное клиент-серверное приложения для установления vpn-соединений) было установление того, что данные, посылаемые в зашифрованный канал, зависят от данных, полученных из сети. Данная зависимость была успешно установлена.

Название	Количество LLVM-инструкций	Время работы	Потребление памяти	Количество шагов анализа помеченных данных	Количество шагов анализа псевдонимов
stl vector	порядка 100	0.03 сек.	Меньше 10МБ	7383	18181
md5	порядка 1000	0.1 сек	Меньше 10МБ	25540	32806
openvpn	порядка 100 000	58 сек	Около 2ГБ	13840217	26471821

Табл. 1. Результаты тестирования.

Тестирование проводилось на компьютере с процессором Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz и 32 гигабайтами оперативной памяти. В табл 1. содержатся численные результаты тестирования. Однако, как было замечено [8], в случае анализа помеченных данных, эффективность подходов и время работы может кардинально различаться в зависимости от классов приложений и использующихся стоков и истоков, поэтому данные, приведённые в табл. 1, носят относительный характер.

8. Заключение

В данной работе рассмотрена задача анализа помеченных данных и предложен алгоритм её эффективного решения. Данный алгоритм реализован на базе компиляторной инфраструктуры LLVM. Проведено тестирование данной реализации на наборе реальных программ. Тестирование показало, что предложенный алгоритм способен эффективно находить утечки данных в программах среднего размера (несколько сотен тысяч инструкций). Учитывая, что основной целью данной работы является достижение линейной зависимости числа шагов от количества инструкций в анализируемой программе на достаточно широком наборе приложений, необходимо создать широкую базу тестовых приложений с размеченными стоками и истоками. Данный набор приложений позволит уточнить эмпирические значения, использующиеся в алгоритме, и адаптировать их к анализу больших приложений (несколько миллионов инструкций).

Список литературы

[1] В.П. Иванников, А.А. Белеванцев, А.Е. Бородин, В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 25, 2013, с. 231-249. DOI: 10.15514/ISPRAS-2014-26(1)-7.

[2] Vivek Sarkar. Security Analysis of LLVM Bitcode Files for Mobile. High confidence software and system conference, 2014.

[3] Mark N. Wegman, F. Kenneth Zadeck. Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13, 2 (April 1991), pp.181-210.

[4] Vivek Sarkar, Kathleen Knobe. Enabling Sparse Constant Propagation of Array Elements via Array SSA Form, 5th International Symposium, SAS'98, 1998, pp 33-56.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps, PLDI'14. 2014

[6] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, Patrick McDaniel. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. http://arxiv.org/abs/1404.7431. 29 Apr 2014.

[7] Thomas Reps, Mooly Sagiv, Susan Horwitz. Interprocedural Dataflow Analysis via Graph Reachability. University of Copenhagen. 1994

[8] Yannis Smaragdakis, George Kastrinis, George Balatsouras. Introspective Analysis: Context-Sensitivity, Across the Board. , PLDI'14. 2014

Interprocedural taint analysis for LLVM-bitcode

V.K. Koshelev, A.O. Izbyshev, I.A. Dudina
ISP RAS, Moscow, Russia
{vedun, izbyshev, eupharina}@ispras.ru

Abstract. Today the development cycle of many application classes requires a security analysis stage. Taint analysis is widely used to check programs for different security vulnerabilities. This paper describes static interprocedural flow, context, and object-sensitive taint analysis approach for C/C++ applications. Our taint analysis algorithm is based on the Flowdroid project's approach, but in contrast to Flowdroid, which aims to analyze Java bytecode, our approach handles LLVM bitcode and pointer arithmetic. Primary drawback of the Flowdroid approach is a memory usage issue which arises during analysis of medium size applications (around 10 000 edges in the call graph). To achieve scalability of the approach, we suggest a set of heuristics which helps to significantly decrease memory usage of the algorithm. The testing of real-world applications shows that such heuristics make precise taint analysis suitable for the medium size programs. Using our approach, we implemented general taint analysis framework as an LLVM pass. Additional security checks (e.g. Use of Hard-coded Password, Information Exposure, etc.) can be implemented on top of this framework. We have also developed auxiliary passes which resolve targets of virtual calls and build interprocedural control flow graph according to the results.

Keywords: static analysis, taint analysis, IFDS, dataflow, alias analysis.

References

[1] V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignat'ev, D.M. Zhurihin, A.I. Avetisjan, M.I. Leonov. Staticheskij analizator Svace dlja poiska defektov v ishodnom kode program [Static analyzer Svace for finding of defects in program source code]. Trudy ISP RAN [Proceedings of the ISP RAS], tom 25, 2013, s. 231-249. DOI: 10.15514/ISPRAS-2014-26(1)-7. (in Russian)

[2] Vivek Sarkar. Security Analysis of LLVM Bitcode Files for Mobile. High confidence software and system conference, 2014.

[3] Mark N. Wegman, F. Kenneth Zadeck. Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13, 2 (April 1991), pp.181-210.

[4] Vivek Sarkar, Kathleen Knobe. Enabling Sparse Constant Propagation of Array Elements via Array SSA Form, 5th International Symposium, SAS'98, 1998, pp 33-56.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps, PLDI'14. 2014

[6] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, Patrick McDaniel. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. http://arxiv.org/abs/1404.7431. 29 Apr 2014.

[7] Thomas Reps, Mooly Sagiv, Susan Horwitz. Interprocedural Dataflow Analysis via Graph Reachability. University of Copenhagen. 1994
[8] Yannis Smaragdakis, George Kastrinis, George Balatsouras. Introspective Analysis: Context-Sensitivity, Across the Board. , PLDI'14. 2014