DOI: 10.15514/ISPRAS–2020–32(6)–7

Внутрипроцедурный анализ для поиска ошибок на основе символьного выполнения

[1] *А.Е. Бородин, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1,2] *И.А. Дудина., ORCID: 0000-0002-5359-184X <eupharina@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова 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 работе описывается внутрипроцедурный анализ отдельных функций, использующийся в инструменте статического поиска ошибок Svace. Отличительные особенности анализа: анализ по графу потока управления, символьное выполнение с объединением состояний анализа в точках слияния путей, анализ только части путей в функциях с циклами, одновременный запуск всех анализаторов, моделирование достижимых ячеек памяти, нумерация значений переменных.

Ключевые слова: статический анализ; символьное выполнение; svace; поиск ошибок

Для цитирования: Бородин А.Е., Дудина И.А. Внутрипроцедурный анализ для поиска ошибок на основе символьного выполнения. Труды ИСП РАН, том 32, вып. 6, 2020 г., стр. 87-100. DOI: 10.15514/ISPRAS–2020–32(6)–7

Symbolic Execution Based Intra-Procedural Analysis for Search for Defects

[1] *A.E. Borodin, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1,2] *Dudina I.A., ORCID: 0000-0002-5359-184X <eupharina@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the RAS, 25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russian Federation*

Abstract. Svace is a static analysis tool for bug detection in C/C++/Java source code. To analyze a program, Svace performs an intra-procedure analysis of individual functions, starting from the leaves of a call-graph and moving towards the roots, and uses summaries of previously analyzed procedures at call-cites. In this paper, we overview the approaches and techniques employed by Svace for the intra-procedural analysis. This phase is performed by an analyzer engine and an extensible set of detectors. The core engine employs a symbolic execution approach with state merging. It uses value numbering to reduce the set of symbolic expressions, maintains points-to relationship graph for memory modeling, and performs strong and weak updates of program values. Detectors are responsible for discovering and reporting bugs. They calculate different properties of program values using a variety of abstract domains. All detectors work simultaneously orchestrated by the engine. Svace analysis is unsound and employs a variety of heuristics to speed-up. We designed Svace to analyze big projects (several MLOCs) in just a few hours and report as many warnings as possible, while keeping a good quality of reports ≥ 65 of true positives). For example, Tizen 5.5 (20MLOC) analysis takes 8.6 hours and produces 18,920 warnings, more than 70% of which are true-positive.

Keywords: static analysis; symbolic execution; svace; search for defects

For citation: Borodin A.E., Dudina I.A. Symbolic Execution Based Intra-Procedural Analysis for Search for Defects. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 6, 2020, pp. 87-100 (in Russian). DOI: 10.15514/ISPRAS-2020-32(6)-7

1. Введение

Статический анализатор Svace осуществляет поиск ошибок в программах, написанных на языках C/C++/Java [1, 2]. Анализатор за время, сравнимое с временем компиляции, осуществляет поиск ошибок с небольшим уровнем ложных срабатываний. Для анализа не требуется специальная подготовка программ, и не накладывается никаких ограничений на используемые конструкции языка.

Для поиска ошибок используются разные подходы: как анализ на основе абстрактного синтаксического дерева, так и межпроцедурный анализ с моделированием значений переменных и ячеек памяти.

На вход анализатору Svace подаётся исходный код вместе с командой сборки. Svace перехватывает команды запуска компилятора и компоновки. Затем запускается модифицированный компилятор (Clang для C/C++ [3] и OpenJDK javac для Java [4]). Компилятор строит абстрактное синтаксическое дерево (АСД) и запускает детекторы для поиска ошибок на АСД, а также генерирует промежуточное представление программы (LLVM биткод [5] для C/C++ и байткод для Java). Промежуточное представление подаётся на вход основному анализатору SvEng[1]. Основной анализатор строит граф вызовов и запускает поочерёдный анализ каждой функции начиная с листьев графа. В данной работе описывается используемый при этом внутрипроцедурный анализ отдельных функций, который является базой для межпроцедурного анализа.

В разд. 2 описывается обобщённый анализ на основе символьного выполнения [6], который может использоваться различными анализаторами для поиска ошибок в исходном коде программ. Приводится общая схема анализа, даётся описание основных используемых абстракций: идентификатор значения, ссылка, граф указателей, атрибут, абстрактное состояние. В подразделе 2.7 дано описание расширения анализа для поддержки чувствительности к путям. В разд. 3 описывается реализация анализа, используемая в инструменте Svace для внутрипроцедурного анализа функций.

2. Обобщённый анализатор

Разработанный анализ предназначен в первую очередь для выполнения неконсервативного анализа. Неконсервативным будем называть анализ, который для выполнения каких-либо требований (скорость анализа, потребление памяти, простота реализации и др.) в некоторых случаях может выдавать некорректные результаты.

2.1 Символьное выполнение с объединением состояний анализа

Опишем анализ для поиска ошибок на основе символьного выполнения с объединением состояний анализа в точках слияния путей.

Рассмотрим граф потока управления для некоторой функции. С рёбрами графа потока управления будем ассоциировать абстрактные состояния, описывающие интересующие нас свойства функции. Символьное выполнение осуществляется в топологическом порядке. Шаг анализа для каждой вершины графа из абстрактного состояния на её входном ребре формирует абстрактное состояние для выходного ребра. Анализ точек слияния путей для ациклических подграфов проводится после того, как получены состояния на входных рёбрах.

[1] Сокращение от Svace Engine – движок анализатора Svace.

Анализ формирует абстрактное состояние на выходном ребре функции, которое будет описывать свойства программы для всех рассматриваемых путей.

Для анализа сильно связанных компонент (ССК) графа потока управления производится несколько итераций цикла с сохранением всех абстрактных состояний на выходных рёбрах из ССК. После анализа ССК, анализ выполняет объединение состояний на выходных рёбрах. При этом используются эвристики для получения состояния, которое будет описывать все возможные пути выполнения ССК. В этом случае анализ может некорректно описывать свойства для части путей, если эвристики сработали неверно.

Если внутри ССК можно выделить внутреннюю ССК, то на каждом шаге анализа внешней ССК производится несколько итераций анализа внутренней ССК. Такая схема анализа циклов имеет экспоненциальную сложность от количества вложенных циклов. Чтобы уменьшить сложность для внутренних ССК уменьшается количество обходов. При достижении некоторого порога вложенности анализатор перестаёт выделять внутренние ССК.

Будем использовать следующие обозначения:

- S – множество символов языка,
- I – множество инструкций,
- P – множество вершин графа потока управления,
- E – рёбра графа потока управления,
- Γ – абстрактные состояния.

Анализ параметризуется следующими компонентами:

- дополнительные анализы A_i, на основе которых выдаются предупреждения;
- детекторы C_i для поиска ошибок;
- передаточные функции для каждой инструкции $T[I]$;
- количество обходов ССК N;
- функции для создания состояния для точек слияния путей $U[P]: \Gamma \times \Gamma \mapsto \Gamma$;
- функции для создания состояния из нескольких состояний для выходных рёбер ССК $U'[P] : \Gamma \times \Gamma \mapsto \Gamma$.

Для каждой инструкции выполнение всех дополнительных анализов и детекторов производится одновременно.

Детектор запускается для каждой посещённой вершины и на основе абстрактного состояния на входном ребре выдаёт предупреждение об ошибке. Дополнительные анализы формируют абстрактные состояния на выходном ребре.

Любому реализованному анализу и детектору при проходе через некоторую вершину доступны результаты всех других анализов на входных рёбрах этой вершины, но недоступны результаты на выходных рёбрах. Важно, что последовательность применения анализов не должна влиять на результат.

Подобная реализация имеет следующие преимущества.

- Высокая скорость работы. Общие действия выполняются каким-либо одним анализатором.
- Возможность быстрого написания довольно сложных детекторов за счёт того, что каждому детектору доступны все проанализированные свойства.

2.2 Идентификаторы значений

Идентификатор значения[2] является абстракцией для обозначения разбиения значений переменных на классы эквивалентности для шага символьного выполнения, и подчиняется следующим правилам.

I. Если двум переменным сопоставлен один и тот же идентификатор значения, то и при выполнении эти переменные будут иметь одинаковые значения (задача нумерации значений[3]).

II. Для инструкции идентификаторы значения на входном и выходном ребрах равны для всех переменных и ячеек памяти, кроме тех, значения которых меняются в данной инструкции. Данное требование к свойствам идентификаторов значений позволяет значительно упростить написание передаточных функций: все созданные свойства не теряют корректность по мере выполнения анализа.

Идентификаторы значений играют центральную роль в описываемом анализе. Помимо задачи нумерации значений, идентификаторы значений используются для описания большинства анализируемых свойств значений переменных. При этом свойства ассоциируются с идентификаторами значений. Сами свойства в свою очередь могут описываться с помощью идентификаторов значений.

Для описания свойств используются *атрибуты*. Атрибуты могут ассоциироваться с идентификаторами значений и с рёбрами графа потока управления для некоторого абстрактного состояния. Атрибут обозначает некоторое анализируемое свойство (интервал значений переменной, необходимое условие достижимости точки, список идентификаторов значений заблокированых мьютексов и др). Атрибуты должны иметь функцию объединения двух атрибутов ⊔. Эта функция используется при объединении состояний в точках слияния путей. Во многих случаях удобно, чтобы атрибуты были решётками, или полурешётками, а функция объединения – наименьшим верхним элементом. Но это не является требованием для атрибута.

Атрибуты позволяют разделять абстрактное состояние между дополнительными анализами – каждый такой анализ работает со своим множеством атрибутов, поэтому не возникает конфликтов, когда несколько анализов по-разному формируют выходное состояние.

Абстрактное состояние содержит информацию о значении всех атрибутов для идентфикаторов значений. Для наглядности рассмотрим атрибуты: интервал значений **VI**, который описывает интервал возможных значений переменной, и атрибут **Null**, обозначающий что переменная имеет нулевое значение. Тип атрибута будем выделять жирным шрифтом, а значение интервала курсивным. VI – тип атрибута, а **VI** – множество значений.

Обозначим V – множество идентификаторов значений. Функция $val: S \hookrightarrow V$ для каждого символа возвращает ассоциированный идентификатор значения. Функция val является частью абстрактного состояния. Также в абстрактном состоянии для каждого типа атрибута и идентификатора значений содержится информация о значении атрибута:

- $\Gamma[\mathbf{VI}]: V \mapsto VI$,
- $\Gamma[\mathbf{Null}]: V \mapsto Null$.

Вместо записи $\Gamma(\mathbf{Null}, V)$ будем использовать сокращения $\Gamma[\mathbf{Null}](V)$ или $\mathbf{Null}(V)$.

[2] Идентификатор значения является символьной переменной. В данной статье будем использовать этот термин для обозначения символьных переменных вместе со способом их использования в анализаторе.

[3] Задача нумерации значений заключается в определении множества переменных, значения которых равны в данной точке для всех путей выполнения. Для случая символьного выполнения все пути выполнения можно заменить на все рассматриваемые пути выполнения.

Рассмотрим фрагмент кода на языке C, иллюстрирующий преимущества ассоциирования атрибутов не с переменными программы, а с идентификаторами значений (Листинг 1).

```
1:     void func(int*p) {
2:           int*q = p;
3:           if(!q) {
4:                 *p = 7;
5:           }
6:     }
```

Листинг 1. Разыменование нулевого указателя
Listing 1. Null pointer dereference

После анализа инструкции на строке 2 в абстрактном состоянии выполняется $val(q) = val(p)$. Пусть $val(q) = v_1$. При анализе инструкции на строке 3 идентификатору значения переменной q будет сопоставлено значение атрибута **Null**, обозначающее, что указатель имеет только нулевое значение $null$. При анализе инструкции разыменования на строке 4 абстрактное состояние будет иметь следующие свойства:

$$val(p) = v_1, val(q) = v_1,\ \Gamma[\mathbf{Null}](v_1) = null\,.$$

Фактически, анализу известно, что разыменовывается указатель, значение которого может быть только нулём. Этого достаточно, чтобы выдать предупреждение о разыменовании нулевого указателя. Если строка 4 достижима, то произойдёт ошибка.

Поскольку идентификаторы значений обозначают значения, которые не меняются в разных точках программы, многие свойства удобно выражать, используя идентификаторы значений. Иначе говоря, значения атрибутов ссылаются на идентификаторы значений. Идентификаторы значения при этом используются как символы некоторого алфавита.

Рассмотрим атрибут pt (см. подразд. 2.4), значениями которого являются множество идентификаторов значений для адресов указываемых ячеек.

```
1:     void func(int f) {
2:           int a, b, c;
3:           int*p = f>0? &a : f<0? &b : &c;
4:           int*q = p;
5:           p = 0;
6:     }
```

Листинг 2. Пример с присваиванием
Listing 2. Example with assignment

Состояние в точке 3 на Листинге 2 будет имеет следующие свойства:

$$val(\&a) = v_a, val(\&b) = v_b, val(\&c) = v_c,$$
$$val(p) = v_{pp}, pt(v_{pp}) = \{v_a, v_b, v_c\}.$$

Т.е. значение переменной p указывает на адреса переменных a, b и c.

Эти же свойства будут выполняться и для строки 6, несмотря на то, что значение переменной p поменялось:

$$val(p) = v_0, val(q) = v_{pp}, pt(v_{pp}) = \{v_a, v_b, v_c\}, \Gamma[\mathbf{Null}](v_0) = null.$$

Таким образом, анализу достаточно поменять информацию о значении переменной p, но не требуются менять свойства значений этой переменной, что позволяет оптимизировать время работы анализа.

2.3 Ссылки

Ссылками будем называть подмножество идентификаторов значений, для которых выполняется два условия:

I. значение является указателем;
II. анализ выполняет моделирование памяти для этого указателя, т.е. отслеживаются значения, находящиеся в указываемой ячейке памяти.

Расширим функцию val для моделирования памяти: $val: S \cup R \hookrightarrow V$, где R – множество ссылок. Запись $val(s) \mapsto v_1$, $val(v_1) \mapsto v_2$ означает, что переменная s имеет значение, описываемое идентификатором v_1, а значение в ячейке, на которую указывает s описывается идентификатором v_2.

Ссылки используются для моделирования указателей и всегда обозначают указатели. Моделируются только те указатели, для которых анализ может отследить, что они не являются алиасами. В наиболее простой реализации ссылки обозначают адреса локальных переменных, про которые точно известно, что они не равны друг другу.

2.4 Граф указателей

Анализ указателей позволяет ответить на вопрос, на какие моделируемые ячейки памяти указывают переменные. Результат этого анализа удобно представлять в виде графа указателей.

Графом указателей назовём направленный граф $P = < V, pt >$, вершинами которого являются идентификаторы значений, а рёбра идут от идентификаторов значений к ссылкам. Рёбра задаются функцией $pt: V \hookrightarrow 2^R$, которая для идентификатора значений возвращает множество указываемых ссылок. Если граф указателей имеет ребро $< v_1, r_2 >$, то это означает, что значение, обозначаемое идентификатором v_1, указывает на ячейку памяти, моделируемую ссылкой r_2. Либо, что тоже самое, значение v_1 может иметь алиас r_2.

Функция pt является ещё одной частью абстрактного состояния. Анализ указателей необходим для моделирования непрямых обращений к памяти. К анализу указателей не предъявляется каких-либо особых требований и его можно рассматривать как ещё одну параметризацию анализа.

2.5 Сильные и слабые обновления

Если анализ инструкции присваивания значения переменной или ячейке памяти может уничтожить эффект от присваивания всех предыдущих значений, то такое поведение называют сильным обновлением. Если же эффекты предыдущих присваиваний всё ещё учитываются анализом, то происходит слабое обновление.

В описываемом анализе сильные обновления можно производить для всех переменных и ячеек, для которых в графе указателей есть только одно исходящее ребро.

Рассмотрим инструкции языка C, выполняющие непрямые обращения к памяти: $r = *p$, $*p = a$. Опишем передаточные функции для этих инструкций для произвольного анализа с сохранением результатов в атрибут $\mathbf{A}_i$.

Инструкция чтения значения из памяти: $m_n = *p$. Пусть $pt(val(p)) = Pt$. Возможны 3 случая: множество Pt имеет один элемент, множество имеет более одного элемента и множество пусто. В первом случае пусть $Pt = \{m\}$, можно сделать сильное обновление, в результате в выходном состоянии $val(r) = val(m)$.

Для случая множества элементов $Pt = \{m_1, m_2, \ldots, m_n\}$ создадим новый идентификатор значения v_r, который сопоставим r в выходном состоянии $val(r) = val(v_r)$. Для всех атрибутов будем использовать функцию объединения свойств:

$$\Gamma[\mathbf{A}_i](v_r) = \sqcup(\Gamma[\mathbf{A}_i]val(m_1), \ldots, \Gamma[\mathbf{A}_i]val(m_n)).$$

Если множество пусто, то с переменной r будет ассоциирован новый идентификатор значения.

Инструкция записи в память: $*p = a$. Аналогично рассмотрим 3 случая в зависимости от множества Pt. Если $Pt = \{m\}$, то в выходном состоянии $val(m) = val(a)$. Для случая множества элементов $Pt = \{m_1, m_2, \ldots, m_n\}$ выполним слабое обновление, для этого для каждой ячейки памяти создадим новый идентификатор значения v_i, который сопоставим

ячейкам в выходном состоянии $val(m_i) = v_i$. Для всех атрибутов будем использовать функцию объединения свойств:

$$\Gamma[\mathbf{A}_i](v_r) = \Gamma[\mathbf{A}_i](val(a) \sqcup \Gamma[\mathbf{A}_i](prev(m_i))).$$

Здесь функция $prev$ вернёт значение $val(m_i)$, если оно определено, либо создаст новый идентификатор значения. Т.е. для каждой из возможных ссылок учитывается, что её значение могло измениться. Если множество пусто, то с переменной $val(p)$ будет ассоциирован новый идентификатор значения.

Для создания абстрактного состояния для точки слияния путей используется похожее правило, как для инструкции чтения значения из памяти. Правило для значений моделируемых ячеек памяти объединяет свойства из входных состояний.

$$\Gamma[\mathbf{A}_i](joinval(m)) = \Gamma_1[\mathbf{A}_i](val(m) \sqcup \Gamma_2[\mathbf{A}_i](prev(m_i))).$$

Функция $joinval$ возвращает идентификатор значения ссылки m, если в обоих состояниях для входных рёбер ссылке m присвоен один и тот же идентификатор значения; и создаёт новый идентификатор значения в остальных случаях.

2.6 Ядро анализа и дополнительные анализы

Все дополнительные анализы реализуются в виде обработчиков инструкций, оперирующих не переменными, а соответствующими идентификаторами значений. Ядро анализа отслеживает граф указателей, выполняет сильные, либо слабые обновления, и вызывает обработчики для соответствующих ситуаций. Фактически ядро анализа занимается следующими компонентами абстрактных состояний: val, pt, т.е. отслеживает значения переменных и осущесьвляет анализ указателей.

Рассмотрим анализ на небольшом примере (Листинг 3), где в качестве дополнительных анализов будем использовать анализ интервалов.

```
1:     int f(int a, int*p) {
2:           int x = 1;
3:           *p = x;
4:           if(a)
5:                 *p = 2;
6:           return *p;}
```

Листинг 3. Небольшой пример
Listing 3. Small Example

Абстрактные состояния анализа после соответвующих строк:

2: $val(x) = v_x, \mathbf{VI}(v_x) = [1; 1]$.

3: $\Gamma_2, val(p) = v_p, val(v_p) = v_x$.

5: $\Gamma_4, val(v_p) = v_2, \mathbf{VI}(v_2) = [1; 2]$.

6: $\Gamma_5, val(v_p) = v_{p2}, val(ret) = v_{p2}, \mathbf{VI}(v_{p2}) = [1; 2]$.

Значение атрибута **VI** в последнем абстрактном состоянии было получено применением функции объединения атрибутов для значений по правилу, описанному в подразд. 2.5 для точек слияния:

$$\Gamma[\mathbf{VI}](joinval(v_p)) = \Gamma_1[\mathbf{VI}](val(v_p) \sqcup \Gamma_2[\mathbf{VI}](prev(v_p))),$$
$$\Gamma[\mathbf{A}_i](joinval(v_{p2})) = \Gamma_1[\mathbf{A}_i](val(v_x) \sqcup \Gamma_2[\mathbf{A}_i](prev(v_2))) = [1; 1] \sqcup [2; 2] = [1; 2].$$

2.7 Чувствительность к путям

Анализ будем называть чувствительным к путям, если он способен различать пути выполнения программы в процессе анализа. Описываемая реализация чувствительности к путям основана на расширении использования идентификаторов значений, которые

используются как кирпичики в условных выражениях, и в определениях идентификаторов значений. Задача чувствительности к путям – отсеять пути, которые не выполнимы из-за несовместных условий.

Для конкретного анализатора добавление чувствительности к путям позволяет не только уменьшить количество ложных предупреждений, связанный с несовместными условиями, но также увеличить количество выдаваемых истинных предупреждений. Последнее достигается за счёт выдачи предупреждений в сложных случаях, для которых анализатор без чувствительности к путям не может обеспечить приемлемое качество анализа.

Например, в программе из Листинга 4 путь, проходящий через два разыменования p, не является выполнимым.

```
1:     void f(int a, int**p) {
2:           int x = a+2;
4:           if(a>10) {
5:                 *p = 0;
6:           }
7:           if(x<12) {
8:                 **p = 0;
9:           }
10:    }
```

Листинг 4. Функция с невыполнимым путём
Listing 4. Function with infeasible path

Если бы путь являлся выполнимым, то на строке 8 произошло бы разыменования нулевого указателя $* p$.

Условным будем называть атрибут, который характеризует условия наступления некоторых событий. Значением атрибута является формула – выражение логики высказываний, где могут использоваться константы языка программирования и идентификаторы значений для обозначения свойств значений переменных. Пример условия: $(v_x > v_y) \wedge (v_x \neq 0) \vee (v_y < 10)$.

Имеется предопределённый условный атрибут **Ness** – необходимые условия достижимости ребра графа потока управления.

Остальные атрибуты используются отдельными детекторами по следующей схеме. В точке, где происходит интересующее событие, значение атрибута C_i устанавливается в $true$ и ассоциируется с некоторым идентификатором значения. В точках слияния путей 1 и 2 значение атрибута вычисляется по следующей формуле:

$$\Gamma_{res}[C_i](v) = (\Gamma_1[\mathbf{Ness}](v) \wedge \Gamma_1[C_i](v)) \vee (\Gamma_2[\mathbf{Ness}](v) \wedge \Gamma_2[C_i](v)).$$

Для отсеивания несуществующих путей – перед выдачей предупреждения об ошибке – запускается SMT-решатель, которому на вход подаётся конъюнкция необходимого условия с условием отслеживаемого атрибута. Если SMT-решатель возвращает unsat, то ошибка на таких путях не существует.

Для примера из Листинга 4 значение атрибута **Ness** будет $v_x = v_a + 2 \wedge v_x < 12$, значение атрибута, отслеживающего присваивание нулевого указателя, – $v_a > 10$. Поскольку формула $v_x = v_a + 2 \wedge v_x < 12 \wedge v_a > 10$ не имеет модели, благодаря SMT-решателю, не будет выдано ложное срабатывание о разыменовании нуля.

Заметим, что передаточные функции для анализируемых свойств и необходимых условий, а также сам вид условных атрибутов, зависят от конкретной реализации. Более простая реализация может не добавлять условие $v_x = v_a + 2 \wedge v_x < 12$ в необходимые условия. В этом случае путь не будет отсеян. Фактически, это будет чувствительный к путям анализ, без чувствительности по данным.

SMT-решатель вызывается только тогда, когда имеется подозрение на ошибку. Если формула не разрешима, то предупреждение не будет выдано, во всех остальных случаях предупреждение будет выдано. SMT-решатель не вызывается, чтобы подсчитать какие-либо промежуточные данные.

Описанная реализации, чувствительного к путям анализа, имеет следующие преимущества:

- простота расширения для анализа на основе идентификаторов значений;
- высокая скорость, поскольку SMT-решатель вызывается только в том случае, когда есть подозрение на ошибку;
- отсутствие ограничений на конкретный вид анализируемых формул.

3. Реализация анализа в Svace

3.1 Используемые эвристики

Анализ не является консервативным и может использовать эвристики как для повышения точности анализа (иногда в ущерб корректности), так и для уменьшения времени анализа. Основные используемые эвристики:

- входные параметры процедуры, а также их смещения и разыменования, не являются алиасами;
- выбранное множество путей анализа описывает все существенные пути анализа (путь считаем существенным, если он может повлиять на результат анализа).

Анализ покрывает все возможные пути в функциях без циклов, и только часть путей в функциях с циклами.

Дополнительно используется ряд ограничений на различные параметры:

- тело цикла обходится только 2 раза; большее количество обходов не используется, чтобы не замедлять анализ;
- максимальное моделируемое количество ссылок для одного идентификатора значений 150;
- максимальное количество моделируемых неконстантных смещений для одного указателя 10;
- максимальная длина цепочки моделируемых разыменований и смещений для переменных равна 6;
- максимальное время анализа одной функции 5 минут.

Ограничение на время анализа одной функции является защитой от нестандартного кода. На всех известных нам проектах анализ всех функций укладывается в 5 минут.

3.2 Анализируемый язык svace0

Анализ выполняется для внутреннего представления на языке svace0. Язык svace0 представляет собой упрощённую версию языка LLVM с дополнительными инструкциями, позволяющими получить больше информации о программе.

Для анализа программ, написанных на языках C/C++, производится трансляция в промежуточное представление LLVM, которое затем преобразуется в язык svace0.

Для анализа программ, написанных на языке Java, производится трансляция в байткод, который затем преобразуется в язык svace0. Реализация большей засти детекторов и анализов не отличается для C/C++ и Java.

Язык является довольно низкоуровневым, что с одной стороны позволяет точно моделировать семантику программ, а с другой стороны затрудняет анализ некоторых высокоуровневых конструкций.

3.3 Используемые атрибуты

Одновременный запуск всех анализов и возможность использования результатов других анализов позволяют получать выигрыш как по времени выполнения анализа, так и по используемой памяти. Высокая скорость анализа обеспечивается за счёт того, что нет необходимости выполнять похожие вычисления, можно сразу использовать результаты других анализов. Минимизация использования памяти достигается за счёт того, что анализу не требуется хранить большинство абстрактных состояний: как правило, после анализа инструкции состояние на входном ребре больше не потребуется и может быть удалено.

Результаты анализов сохраняются в виде атрибутов. Добавление нового атрибута не требует значительных затрат вычислительных ресурсов. В настоящий момент в Svace используется более 350 атрибутов. Приведём некоторые виды атрибутов в качестве примера:

- возможный интервал значений целочисленный переменных;
- интервал размера массива и интервал смещения указателя на массив;
- мьютекс был заблокирован;
- переменная получена из непроверенного источника;
- интервал длины строки;
- указатель на динамическую память сравнили с константой;
- условия, при которых была выделена динамическая память;
- условия, при которых переменная не была инициализирована;
- условия того, что указателю присвоено нулевое значение (требуется для поиска разыменования нулевых указателей);
- условия того, что переменная может иметь нулевое значение (требуется для поиска ошибок деления на ноль);
- необходимые условия достижения точки в программе.

3.4 Другие анализы

В анализаторе svace описанный анализ является только небольшой частью, которая требуется для внутрипроцедурного анализа. Внутрипроцедурный анализ является базой для межпроцедурного анализа на основе резюме. На основе межпроцедурного анализа выполнен анализ конструкторов, деструкторов и операторов присваивания классов C++ для обнаружения неконсистентности их реализации [7].

Кроме этого перед запуском внутрипроцедурного анализа для каждой функции выполняется анализ потока данных, консервативно вычисляющий важные свойства функции (недостижимый код, функции, завершающие программу, живые переменные) [8].

Дополнительно используются анализы на основе абстрактного синтаксического дерева для реализации части детекторов. Эти анализы запускаются в модифицированных компиляторах (clang, javac).

3.5 Результаты

В табл. 1 приведено время анализа для проектов с открытым исходным кодом. Измерялось только время анализатора SvEng. Во время анализа были включены все реализованные детекторы.

Анализ производился на двух серверах, имеющих следующие характеристики:

- Сервер 1: Intel Xeon CPU E5-2650 2.00GHz, 32 ядра, 256 Гб ОП;
- Сервер 2: Intel Core CPU i7-6700 3.40GHz, 8 ядер, 32 Гб ОП.

Операционные системы tizen и android имеют существенный размер исходного кода, их анализ желательно проводить на сервере, имеющем хотя бы 32 Гб оперативной памяти.

Текущая скорость анализа позволяет проанализировать эти проекты во время ночной сборки. Для сравнения в таблице приведено время сборки этих проектов на сервере 1.

Табл. 1. Analysis time for big projects
Table 1. Время анализа больших проектов

Проект	**Размер кода, тыс. строк**	**Время анализа, мин.**		**Время сборки, мин.**
		сервер 1	**сервер 2**	
tizen 5.5	19 988	272	516	250
android 5	8 561	236	421	31
android 9 java	12 122	27	32	77

Для значительной части небольших проектов для анализа требуется не больше 2Гб оперативной памяти. В табл. 2 приведены данные анализа проектов busybox, cairo, xorg-server и nss, имеющих размер от 139 до 393 тысяч строк кода, на сервере 1 с ограничением используемой памяти.

Табл. 2. Analysis time for small projects
Table 2. Время анализа небольших проектов, сек.

Память, Гб	**Проект**			
	busybox-1.18.5	**cairo-1.12.14**	**xorg-server-1.12.3**	**nss-3.17.4**
	Размер, тыс. строк кода			
	139	270	393	355
16	310	176	524	336
8	314	160	521	336
4	319	156	520	335
2	338	158	594	378
1	419	231	1430	418

Размер исходного кода в строках позволяет примерно оценить сложность проекта. Скорость анализа зависит от сложности кода и используемых конструкций. Как видно из результатов скорость анализа варьируется от 448 до 1534 строк в секунду при использовании 16Гб оперативной памяти. Снижение доступной памяти до 2Гб практически не влияет на время анализа. Худший результат был 274 строк в секунду для проекта xorg-server при ограничении доступной памяти в 1 Гб, при этом увеличение доступной памяти до 2 Гб ускорило анализ этого проекта в 2,4 раза.

Табл. 3. Качество выдаваемых предупреждений
Table 3. Analysis quality

Проект	Всего	Размечено	Истинных
cairo-1.12.14	321	10.9%	94.2%
xorg-server-1.12.3	791	10.1%	78.7%
nss-3.17.4	826	22%	85.1%
busybox-1.18.5	1561	10.1%	80.5%
android 9 java	7327	10.68%	77.7%
android 5.02	10 414	10.7%	76.4%
tizen 5.5	18 920	22.1%	70.8%

Важной характеристикой анализатора является процент истинных предупреждений. Фактически постановка задачи – находить настолько много предупреждений, насколько можно при обеспечении приемлемого уровня истинных срабатываний. В табл. 3 представлены данные по качеству предупреждений среди 190 стабильных детекторов. В таблицу входят данные только для анализатора SvEng. В колонке «Всего» приводится общее

количество предупреждений, выданных для оцениваемых детекторов, колонка «Размечено» содержит процент размеченных вручную предупреждений среди выданных, и колонка «Истинных» показывает, сколько процентов предупреждений среди всех размеченных было классифицировано как истинные.

4. Похожие работы

Инструменты, основанные на символьном выполнении без объединения состояний в точках слияния путей: PREfix [9], Archer [10] и, по всей видимости, Prevent [11]. Наиболее ранней работой является описание ин-струмента PREfix, в котором по умолчанию анализируется 50 путей. Количество выдава-емых предупреждений почти перестаёт изменяться после рассмотрения больше 100 путей внутри функции.

Использование символьного выполнения без объединения путей имеет как свои достоинства, так и недостатки. Очевидным недостатком является проблема экспоненциального роста путей внутри функции. Из-за чего инструмент не может проанализировать значительную часть путей в функциях с большим количеством путей. Кроме этого общее время анализа выше по сравнению со схемой с объединением путей. Другой проблемой является создание резюме для анализируемой функции. Либо придётся писать дополнительный анализ для создания резюме, либо ограничиться резюме, описывающими неполное поведение функции (так как не все пути проанализированы). К преимуществам можно отнести то, что каждый путь моделируется более точно, нет необходимости в эвристической функции объединения абстрактных состояний, реализация многих детекторов упрощается. Помимо этого, легче сообщить об ошибке пользователю, поскольку достаточно показать рассматриваемый путь.

Инструмент Saturn [12] выполняет объединение состояний в точках слияния путей. Используется SAT-решатель. Детекторы запускаются по очереди. Способ анализа циклов зависит в том числе и от детектора. Абстрактные состояния между детекторами не разделяются. Время анализа существенно зависит от включённых детекторов. Похожую архитектуру имеет инструмент Calysto [13], основанный на SMT-решателе, имеющий лучшую скорость и точность.

SharpChecker [14, 15] – разрабатываемый в ИСП РАН инструмент для анализа программ на языке C\#. Инструмент имеет множество общих черт с описанной схемой (объединение состояний анализа в точках слияния путей, одновременный запуск всех анализаторов, моделирование достижимых ячеек памяти), при этом он изначально проектировался из расчёта использования вместе с SMT-решателем. Текущая схема анализа циклов, используемая в Svace, взята из этой работы. В более ранних версиях Svace объединение состояний производилось на обратных рёбрах.

5. Заключение

Описан внутрипроцедурный анализ, позволяющий относительно быстро и качественно анализировать большие объёмы исходного кода, что подтверждается реализацией в инструменте Svace.

Описана общая схема анализа, которая может быть улучшена и дополнена множеством способов:

- добавление девиртуализации для построения более точного графа вызовов;
- улучшение анализа указателей, в том числе использование анализа указателей с чувствительностью к путям;
- более точное моделирование циклов;
- уменьшение количества моделируемых ячеек памяти и значений переменных для ускорения анализа.

Список литературы / References

[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 26, вып. 1, 2014 г., стр. 231-250 / V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatyev, D.M. Zhurikhin, A.I. Avetisyan, M.I. Leonov. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 231-250 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7.

[2] A. Borodin, A. Belevantsev, D. Zhurikhin, and A. Izbyshev. Deterministic static analysis. In Proc. of the 2018 Ivannikov Memorial Workshop, 2018, pp. 10-14. DOI: 10.1109/IVMEM.2018.00009.

[3] Clang. URL: https://clang.llvm.org, accessed September 10, 2020.

[4] The javac compiler. URL: https: //docs.oracle.com/en/java/javase/11/tools/javac.html, accessed September 10, 2020.

[5] LLVM bitcode. URL: https: //releases.llvm.org/8.0.1/docs/BitCodeFormat.html, accessed September 10, 2020.

[6] J. C. King. Symbolic execution and program testing. Communications of the ACM, vol. 19, no. 7, 1976, pp. 385-394.

[7] А.Е. Бородин и А.А. Белеванцев. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 111-134 / A.E. Borodin, A.A. Belevancev. A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp.111-134 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-8.

[8] Р.Р. Мулюков, А.Е. Бородин. Использование анализа недостижимого кода в статическом анализаторе для поиска ошибок в исходном коде программ. Труды ИСП РАН, том 28, вып. 5, 2016 г., стр. 145-158 / R.R. Mulyukov, A.E. Borodin. Using unreachable code analysis in static analysis tool for finding defects in source code. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 5, 2016, pp. 145-158 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-9.

[9] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. Software-Practice and Experience, vol. 30, no. 7, 2000, pp. 775-802.

[10] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. ACM SIGSOFT Software Engineering Notes, vol. 28, no. 5, 2003, pp. 327–336.

[11] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. Communications of the ACM, vol. 53, no. 2, 2010, pp. 66–75.

[12] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the saturn project. In Proc of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2007, pp. 43–48. ACM.

[13] D. Babic and A.J. Hu. Calysto: scalable and precise extended static checking. In Proc. of the 30th International Conference on Software Engineering, 2008, pp. 211–220.

[14] В.К. Кошелев, И.А. Дудина, В.Н. Игнатьев, А.И. Борзилов. 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 / V.K. Koshelev, I.A. Dudina, V.N. Ignatyev, A.I. 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.

[15] V. Koshelev, V. Ignatiev, A. Borzilov, and A. Belevantsev. Sharpchecker: static analysis tool for C# programs. Programming and Computer Software, vol. 43, no. 4, 2017, pp. 268–276. DOI: 10.1134/S0361768817040041.

Информация об авторах / Information about authors

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, научный сотрудник. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Alexey Evgenevich BORODIN – PhD, researcher. Research interests: static analysis for finding errors in source code.

Ирина Александровна ДУДИНА – кандидат физико-математических наук, сотрудник ИСП РАН и ассистент кафедры системного программирования факультета ВМК МГУ. Сфера научных интересов: статический анализ, символьное исполнение, SMT-решатели.

Irina Aleksandrovna DUDINA – PhD, employee of ISP RAS and assistant of the Department of System Programming of the Faculty of CMC, Moscow State University. Research interests: static analysis, symbolic execution, SMT solvers.