УДК 004.052.42

DOI: 10.15827/0236-235X.121.152-157

Дата подачи статьи: 20.06.17

2018. Т. 31. № 1. С. 152–157

ПОСТРОЕНИЕ МОДЕЛИ ПРОГРАММНОГО КОДА ДЛЯ ОБНАРУЖЕНИЯ ПРОГРАММНЫХ ДЕФЕКТОВ ПРИ ПОМОЩИ СИСТЕМ ТИПОВ

А.В. Цветков¹, аспирант, laurentii.tsvetkov@corp.ifmo.ru**А.И. Спивак**¹, к.т.н., доцент, anton.spivak@gmail.com

¹ Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики (Университет ИТМО), Кронверкский просп., 49, г. Санкт-Петербург, 197101, Россия

Неотъемлемая часть процесса разработки ПО – анализ его качества. Дефекты программного кода являются компонентой, которую можно уменьшить при помощи статического анализа кода, одна из форм которого – статическая проверка типов. Многие языки программирования используют системы типов для снижения количества ошибок. Однако системы типов большинства языков не способны выразить базовые требования безопасности относительно исключительных ситуаций. Для обеспечения возможности расширения существующих систем типов императивных языков необходимо создать модель программного кода, учитывающую ряд факторов, таких как наличие изменяемого состояния, алиасинг указателей и др.

В статье дано описание модели программного кода, выражающей инварианты безопасности и допускающей эффективное использование расширенных систем типов для нахождения уязвимых участков кода. Предлагаемые решения проблем алиасинга и изменяемого состояния основаны на использовании формы статического одиночного присваивания путем применения правил перезаписи, сохраняющих функциональную эквивалентность программ, что позволяет использовать принципы проверки, применимые к функциональным языкам программирования.

Рассмотрены понятие линейного ресурса, связанные с ним проблемы и возможные решения в рамках предлагаемого подхода. На примере непрерывного массива данных показаны принципы индексирования зависимых типов.

Предлагаемая модель не исключает, а дополняет систему типов исследуемого языка программирования. Построение предлагаемой модели с расширенной системой типов не требует отдельных аннотаций исходного кода, что снижает необходимые трудозатраты и позволяет использовать ее для анализа существующей кодовой базы.

Ключевые слова: модель кода, системы типов, программные дефекты.

Статический анализ уязвимостей является формой статического анализа исходного кода, основная цель которого – нахождение программных дефектов, создающих потенциальную брешь в безопасности программы. Статический анализ предпочтителен для обеспечения безопасности, поскольку атаки зачастую нацелены на редко исполняемые ветви выполнения программы, которые не были покрыты средствами тестирования, а статический анализ нацелен на нахождение именно таких ветвей.

Многие языки программирования имеют механизм защиты от компиляции и исполнения некорректно сформированных в терминах языка программ – систему типов. Каждому терму языка назначается соответствующий тип, который служит маркером для значения, сохраненного в переменной или используемого как аргумент функции. Аннотация типов ограничивает операции, выполняемые над переменной, обеспечивая осмысленность операций относительно семантики языка. Поэтому статическая проверка типов может считаться формой статического анализа программ.

Возможность использования выразительных систем типов для устранения широкого спектра ошибок была давно отмечена и активно развивалась в функциональных языках программирования. Идеи использования систем типов развиваются в таких языках, как Agda [1] и Idris [2], позволяя типам зависеть от значений, то есть предоставлять зависимую типизацию. В результате типы могут

быть более точными и отражать те характеристики программы, которые необходимо обеспечить или проверить.

У систем типов императивных языков программирования нет подобных механизмов. Попытки создать императивные языки с мощными системами типов сталкиваются с необходимостью решения ряда вопросов, связанных с наличием изменяемого состояния. Одновременное обеспечение и высокоуровневых, и низкоуровневых свойств зачастую приводит к появлению большого количества аннотаций типов в коде, создает препятствия к его дальнейшему расширению и поддержке. В частности, язык программирования Rust [3] способен проверять корректность заимствования ссылок. В данном языке сделан акцент на явном выражении отношений владения, несмотря на то, что внутри выражений языка проверка выполняется неявным образом. Необходимость в предоставлении дополнительных аннотаций не позволяет использовать преимущества данного языка для анализа существующих программ, написанных на языках, традиционно более подверженных ошибкам работы с памятью – C/C++.

Дополнение существующих языков более выразительными системами типов способно дать новые сведения о структуре программ, вывести отношения между переменными и функциями, а также формами их взаимодействия. Такая информация может быть использована для обнаружения и потенциального эксплуатирования уязвимостей, при-

существующих в анализируемом ПО. Для поддержки таких систем типов необходимо создать модель программного кода, в которой будут выражаться зависимости между объектами системы, впоследствии анализируемые системой типов.

В данной работе представлена модель программного кода, в которой выражаются зависимости объектов, необходимые для обнаружения уязвимостей, связанных с работой с памятью в императивных языках программирования. Модель учитывает наличие изменяемого состояния и побочных эффектов, создает базис для создания системы типов, использующих механизм зависимых типов. В работе рассмотрены следующие вопросы.

- **Мутабельность.** Императивные языки предполагают использование изменяемых переменных. Если назначение типа переменной происходит в момент объявления, то последующее присваивание новых значений может приводить к изменению связанных доказательств, представляющих текущее понимание о безопасности программы. В предыдущих работах или подобное было полностью запрещено [4], или связанные типы отслеживались и модифицировались [5]. При отражении требований по мутабельности непосредственно в модели программного кода возможно использование существующих наработок для решения поставленных проблем.

- **Линейность.** Немаловажную роль в появлении различных уязвимостей играет неверное использование различных функций управления ресурсами. Модель программного кода должна выражать текущее состояние ресурса в линейной иерархии возможных состояний. Линейная иерархия выбрана как наиболее простая, но часто используемая форма управления ресурсом. Ресурсы, работа с которыми ведется в линейно упорядоченной манере, будем называть линейными. В дальнейшем модель линейного ресурса может быть расширена на модель владения и на понятие протокола взаимодействия в высокоуровневых языках.

- **Алиасинг.** Возможность доступа к переменным по множеству указателей (алиасинг) при наличии изменяемого состояния может сделать и самые продвинутые системы типов неспособными к выражению даже простых программ, поэтому необходимо отслеживать подобные переменные или ограничить возможность создания ссылок. Эту проблему предпочтительно решать на уровне построения модели программы, когда возможно внедрение дополнительных фрагментов кода, необходимых для анализируемости программы.

Существующие подходы

Существующие подходы к обнаружению ошибок, связанных с работой с памятью, можно разделить на две категории: статические и динамические.

CETS [6] применяет инструментирование кода во время компиляции к операциям над указателями для обнаружения висячих указателей. Указатель расширяется отдельно хранимой метаинформацией. Когда происходит обращение по указателю, проверяется блок метаинформации, чтобы удостовериться в его корректности. CETS способен обнаруживать висячие указатели как в куче, так и в стеке и даже в тех случаях, когда освобожденная память была вновь размещена в том же самом месте. Согласно авторам, замедление работы программы находится в диапазоне 48–116 %.

Address Sanitizer [7] использует инструментирование с моделью теневой памяти для обнаружения выхода за границы глобальных, локальных и размещенных в куче объектов. Теневая память отслеживает состояние адресного пространства программы относительно доступности для непосредственного чтения/записи. Также возможно обнаружение ошибок использования освобожденной памяти. Обнаружение ошибочного доступа стало возможным благодаря внедрению специальных зон между последовательными объектами. Используя сжатие состояния теневой памяти, авторам удалось добиться среднего замедления в 73 % при полном отсутствии ложных срабатываний.

И Address Sanitizer, и CETS эффективны для нахождения ошибок работы с памятью, но для обеспечения корректности необходимо иметь полное покрытие ветвей исполнения. Для крупных программ это означает неоправданно большой набор тестов. Замедление работы программы на настоящий момент не является значимым препятствием, хотя и представляет сложности для систем с высокими требованиями к скорости реакции.

Ранее было опубликовано несколько статей по статическому анализу с целью включить зависимые типы в императивные языки.

Noare Type Theory (НТТ) [8] предлагает подход, в котором логика Хоара встроена в систему типов. Императивные вычисления ведутся внутри индексированной монады Хоара. Зависимая типизация разрешена в тройках Хоара, так что, постусловия могут зависеть от возвращаемых значений. Проверка типов в НТТ происходит в два этапа для преодоления теоретической неразрешимости задачи останова. На первом этапе проводятся базовая проверка типов и генерация условий корректности. На втором этапе условия доказываются человеком или программой, что позволяет регулировать глубину верификации в соответствии с необходимостью.

Ynot [9] является дальнейшим развитием НТТ и ее реализацией в среде доказательств Coq. Ynot позволяет реализовывать императивные программы с побочными эффектами на чистых функциональных языках. Зависимые типы для НТТ предоставляются средой Coq. Используя данный подход, авторы реализовали несколько императивных алгоритмов и доказали их корректность.

Xanadu [4] – одна из первых попыток внедрить зависимые типы в императивные языки. Примечательной особенностью Xanadu является возможность замены типа во время вычисления.

Deputy [5] предоставляет гибкую систему типов для низкоуровневых императивных языков и поддерживает изменяемые переменные. Зависимые типы выводятся для локальных переменных.

Инвалидация выведенных типов происходит на основе аксиомы присваивания логики Хоара. Система отслеживает связанные типы для поддержания корректной типизации. Выражения внутри зависимых типов ограничены использованием только локальных переменных. В программу внедряются утверждения (assert), покрывающие случаи, не найденные системой типов.

HTT и Ynot служат более как модели вычисления императивных программ, а не модели самого кода. Данные модели не допускают автоматического преобразования в них существующих программ для проведения анализа.

Представленная авторами статьи система отличается от Deputy (и Xanadu) некоторыми важными аспектами. Во-первых, ни одна из этих систем не способна выражать доступ по отрицательным смещениям в массиве. Такая операция возможна при наличии в языке арифметики указателей. Во-вторых, в них не были рассмотрены проблемы управления памятью – Deputy предполагает корректное использование функций malloc/free. Более того, Deputy допускает только локальный вывод типов, а переменные, имеющие зависимый тип, не могут выступать в качестве возвращаемого значения функции. Необходимо отметить, что система предупреждения уязвимостей должна производить комплексный анализ использования и управления памятью. Это невозможно без поддержки межпроцедурного анализа со стороны используемой модели программного кода.

Модель программного кода

Основная задача данной статьи – описание созданной модели программного кода для высокоуровневых императивных языков, необходимой для поддержки возможности обнаружения дефектов и уязвимостей в программах путем выведения свойств и ограничений переменных и функций без поддержки со стороны пользователя, без дополнительных аннотаций и инструментирования. Создание такой модели невозможно без понимания требований к целевой системе типов и задач, которые она призвана выполнять.

Полиморфные типы предлагают целостный и всеобъемлющий подход к покрытию кода и определению ограничений. Зависимые типы расширяют возможности анализа, допуская объединение утверждений (постусловий и предусловий), сделанных в различных ветвях исполнения. Более

того, в низкоуровневом программировании допустимые над переменной действия определяются не только ее собственным значением, но и значениями других переменных в области видимости – от границ массивов, меток объединений и, конечно же, аргументов функций. Возможность создавать и распространять утверждения через вызовы функций является главным преимуществом зависимых типов, позволяющим проводить статический анализ. Подобный подход дает возможность скрывать реализацию функций от анализатора, допуская использование только внешнего по отношению к пользователю поведения. Это открывает возможность межпроцедурного анализа, необходимого для успешного обнаружения уязвимостей, так как многие трудноуловимые ошибки возникают из-за неправильного взаимодействия функций.

Мутабельность. Изменяемые переменные, использование которых предполагается в императивных языках, препятствуют созданию различных инструментов для языков: компиляторов и статических анализаторов. Для системы типов это означает, что либо типы не могут меняться после декларации переменной, либо все изменения переменных должны отслеживаться, а типы уточняться соответствующим образом. Ранее было предложено несколько подходов для решения этой проблемы: ограничение зависимостей только на константы и локальные переменные [5] и изменение типа во время исполнения [4].

Существует известный способ задания программы с изменяемыми переменными в виде программы с неизменяемыми переменными – SSA-представление (Static Single Assignment form [10]). В данном представлении каждая переменная присваивается лишь один раз, поэтому ее тип не может изменяться. Использование подобного представления открывает широкие возможности для разного рода оптимизаций и инструментов анализа. Внутри SSA переменные имеют различное именование в различных блоках графа управления, что позволяет различать наложенные на них ограничения.

Для объединения нескольких путей исполнения используются специальные Ф-функции. Значение Ф-функции зависит от ветви исполнения, из которой было получено управление. В высокоуровневых языках ветвление является иерархическим, поэтому объединяться могут только последние разделенные ветви. Благодаря свойству неизменяемости переменных в SSA можно использовать исходное выражение условного оператора для определения входящей ветви исполнения. Это хорошо согласуется с зависимыми типами: они изначально являются способом различать некоторые свойства в зависимости от определенного значения, в данном случае значения условия ветвления. Зависимые пары используются для задания подобных ситуаций, где первый элемент служит свидетельством, а второй доказательством выведенного свойства.

Линейность. Под линейностью будем понимать такую работу с некоторой переменной, граф состояния которой образует линейную иерархию. Линейные ресурсы и протоколы широко используются в программировании. Примерами таких протоколов служат управление памятью, ввод-вывод, инициализация объектов и многие другие команды, оперирующие состоянием. В общем случае жизненный цикл объекта будет следующим: объект не валиден $\rightarrow$ объект инициализирован $\rightarrow$ доступны операции над объектом $\rightarrow$ объект уничтожается $\rightarrow$ объект аннулирован.

В линейном протоколе существуют, по меньшей мере, два состояния, определяющие операции, доступные над объектом. Невалидный объект может быть только инициализирован, а для уничтожения объект должен быть ранее инициализирован. В данном случае операции являются линейно упорядоченными, поэтому такой протокол работы и следующие ему объекты называем линейными.

Неспособность следовать подобным протоколам приводит к множеству утечек памяти, падений и повреждений данных в существующих программах. И хотя проблема утечки памяти решается в современных языках путем введения сборщика мусора, по отношению к вводу-выводу по-прежнему используется ручное управление. Хорошее средство обеспечения корректности линейных протоколов – принцип RAII (Resource Acquisition Is Initialization), но он не поддерживается во многих затронутых проблемой языках, таких как Java и Python.

Если объект может находиться в большом количестве состояний, от которых зависит доступность методов, то обеспечение корректного использования объекта становится невозможным для используемых в императивном программировании парадигм. При помощи индексированных типов, то есть типов, имеющих дополнительные маркеры, возможно отслеживание подобных состояний до точки применения соответствующих функций.

Алиасинг. Алиасинг указателей препятствует назначению расширенных типов по двум причинам. Во-первых, если переменная использована как часть другого типа, такого как Φ , ее изменение может вызвать нарушение выведенных свойств. Для преодоления этой проблемы допускается использование в типах только локальных переменных. Форма SSA защищает локальные переменные от модификации до тех пор, пока не учитывается влияние указателей. Предлагается простой способ решения этой проблемы – разделение локальных переменных. Так как информация о взятии указателей доступна во время компиляции, возможно разделение локальных переменных на две составляющие: обеспечивающую работу с указателями и фиксирующую текущее значение переменной. Подобное разделение значительно упрощает правила вывода расширенных типов, поскольку операции доступа к памяти выходят за пределы ответственности

системы типов. Вся работа с памятью происходит прозрачно для системы.

Описание модели

В данном разделе дается описание модели и ключевых принципов, в соответствии с которыми она построена. Модель представляет собой промежуточное представление, в которое преобразуется исследуемая программа. В силу ограниченного набора команд возможно создание компактной системы типов, определяющей свойства программы относительно использования памяти. В рамках модели элементам языка назначаются типы согласно правилам вывода надстроенной системы типов. Если ошибок вывода не происходит, декларируется отсутствие ошибок доступа к памяти и множественных освобождений ресурса. Дополнительно контекст вывода может быть проверен на отсутствие неосвобожденных ресурсов, тем самым определяется наличие утечек памяти.

Язык. Для проведения анализа программа должна быть преобразована в *промежуточное представление* (ПП), использование которого делает систему анализа независимой от исходного языка, а также позволяет вводить собственные конструкции.

Грамматика ПП:

Виды	$k ::= * \mid * \rightarrow k \mid \tau \rightarrow k$
Типы	$\tau ::= B \mid L \mid \tau_1 \tau_2 \mid \tau e \mid \Phi(b, \tau_1, \tau_2) \mid \tau_1 \times \tau_2$
Встроенные типы	$B ::= \text{bool} \mid \text{int} \mid \text{uint} \mid \text{ref } L \tau i j$
Индексные выражения	$i, j ::= C \mid v \mid i \text{ op } j \mid \phi(b, v_1, v_2)$
Выражения	$e ::= j \mid l \mid \&l \mid e_1 (e_2) \mid \text{not } b$
L-значения	$l ::= v \mid * v \mid v_1 [v_2] \mid \langle v_1, v_2 \rangle$
Константы	$C ::= D \mid T \mid F \mid \text{Null} \mid 0 \mid 1 \mid \dots$
Инструкции	$s ::= \varepsilon \mid s_1; s_2 \mid l \leftarrow e \mid \text{if } b \text{ then } s_1 \text{ else } s_2$
Объявления	$D ::= M \mid \text{fun } (v_1, \dots, v_n) \{s; \text{return } v_r\}$
Источники указателей	$L ::= I \mid S \mid H \mid D$
Выдел. памяти	$M ::= \text{alloca } \tau j \mid \text{malloc } \tau j \mid \text{free } v$ $v \in$ Переменные $b \in$ Булевы переменные $\text{op} \in$ Бинарные операторы

В ПП никакая переменная не может быть назначена более одного раза, то есть оно находится в форме SSA. Это позволяет системе быть краткой, главным образом, потому, что назначение переменной значения и соответствующего типа происходит в момент ее объявления. Поэтому можно безопасно ссылаться на такую переменную, пока она находится в зоне видимости. Поскольку модификация переменных более недоступна, до преобразования в ПП требуется провести несколько синтаксических замен:

$v_1[v_2] \Rightarrow v' \leftarrow v_1 + v_2; *v'$
 $\&v_1[v_2] \Rightarrow v_1 + v_2$
 $\text{free } v \Rightarrow v ::= \text{free } v$
 $\&v \Rightarrow v' ::= v; \&v'$

Первые два правила перезаписывают операторы доступа к массиву в терминах арифметики указателей. Третье правило позволяет аннулировать указатель при освобождении через **free**. Функция **free** обычно определена в исходном языке как процедура с побочным эффектом. В данном случае добавляется возвращаемое значение в виде нулевого указателя для сброса типа переменной, так как последующие использования будут направлены на другую переменную после преобразования в ПП. Последнее преобразование разделяет обращаемые по указателям переменные, как было описано ранее. После этих преобразований невозможно нарушить выведенные свойства в виде типов путем изменения значений связанных переменных. Становится возможным упрощение правил вывода без потери корректности и общности.

Программа в ПП состоит из (глобальных) объявлений D , инструкций s и выражений e . Следуя традиции языка C, объявления функций считаются константами и не могут быть созданы во время работы программы. Представлены три вида инструкций: блочные, условные и присваивания. Они не имеют собственных типов, но объявляемые в них переменные пополняют контекст типизирования.

Существующие программы могут быть преобразованы в это ПП с помощью алгоритмов, реализованных во многих современных компиляторах [10]. Промежуточное представление не привязано к конкретному языку программирования, поэтому данный подход может быть применен к множеству императивных языков.

Типы

В системе представлено несколько видов типов. Типом операторов отношения является булев тип **bool**, типом целочисленных выражений – **int**. Тип **uint** ограничен только неотрицательными целыми. Также имеется тип пары, соответствующий определению структуры. Пары являются необходимым условием для успешной работы вывода, поскольку возврат экземпляра данного типа – единственный способ подтверждения выведенного свойства.

В качестве маркеров для типа **ref** используются конкретные незаселенные типы **I**, **S**, **H**, **D**, которые не имеют элементов.

Одним из основных типов в системе является **ref**. Это тип массивов, указателей, списков и т.п. Он индексирован двумя типами и двумя выражениями. Первый индекс, обозначаемый как L , отслеживает источник данного указателя: **I** для неверных указателей, **S** указателей на стековые переменные, **H** для указателей на кучу и **D** для указателей, которые были отсоединены от их оригинального источника. Этот индекс определяется функцией или оператором, использованным для получения данного указателя, и необходим для отслеживания и обеспечения корректного применения функций осво-

бождения памяти, тем самым предотвращая ее утечки и повреждения. Второй индекс, τ , идентифицирует тип элементов, на которые ссылается данный указатель. Наконец, индексные выражения i и j определяют количество элементов, доступных через данный указатель. На рисунке показан допустимый диапазон, соответствующий некоему указателю. Диапазон разделен на положительный и отрицательный для обеспечения возможности свободно перемещать указатель по массиву без потери информации. Изменение положения указателя является типичной практикой в низкоуровневых языках, таких как C, поэтому отслеживание доступных элементов по отрицательным индексам необходимо. Отметим также, что указатель может ссылаться на весь закрытый интервал $[-i, j]$, но только к элементам полуоткрытого интервала $[-i, j)$ можно иметь доступ, например, через оператор «*».

Второй тип, на который необходимо обратить внимание, – Φ . Это тип зависимой пары, в которой второй элемент зафиксирован в качестве оператора выбора между типами τ_1 и τ_2 . Свидетельством истинного типа выражения является значение первого элемента пары, имеющего тип **bool**. Данным типом наделяются ϕ -функции, поэтому источником такого типа могут стать только условные выражения. Как было отмечено выше, в силу неизменяемости переменных в промежуточном представлении и иерархической структуре условных выражений результирующий тип ϕ -функции может быть восстановлен при помощи переменной, на основе которой был совершен условный переход.

Также имеется функциональный тип, назначаемый во время компиляции определенным и предопределенным функциям. Использование функционального типа представляет интерес прежде всего благодаря возможности сокрытия информации. В связи с этим можно ставить вопрос о допустимости глубокого анализа внутреннего содержимого функции при каждом ее вызове, поскольку данный подход означает фактическое раскрытие ее внутренней структуры. Альтернативой глубокого анализа является следование ограничению на использование только наблюдаемого поведения функций.

Заключение

В данной статье представлена модель программного кода, в рамках которой возможно создание системы типов, обеспечивающей обнаружение

различного рода программных дефектов. Модель может представить сведения о возможных уязвимостях, связанных с работой с памятью, таких как переполнение буфера, утечка и повреждение памяти. Автоматическое преобразование программ в ПП с использованием данной модели может быть применено для обнаружения дефектов в редко используемых ветвях исполнения, не покрытых классическими средствами тестирования.

Планируется дальнейшее развитие модели путем включения в нее полноценных линейных типов. При наличии всех операторов линейной логики система типов будет обладать возможностью получения более полной информации о возможных будущих изменениях переменных. Это позволит безопасно использовать указатели и глобальные переменные в зависимых типах.

Языки с богатыми системами типов обычно следуют функциональной парадигме программирования и имеют множество преимуществ перед традиционными императивными языками. Теперь существующие решения могут получить преимущества от использования дополнительных схем типизации для обнаружения дефектов и уязвимостей.

Литература

1. Norell U. Dependently typed programming in Agda. Proc. 4th Intern. Workshop on Types in Language Design and Implementation. ACM, 2009, pp. 1–2.
2. Brady E. Idris, a general-purpose dependently typed programming language: Design and implementation. *Jour. of Functional Programming*. 2013, vol. 23, no. 05, pp. 552–593.
3. The Rust Programming Language. URL: <https://www.rust-lang.org/> (дата обращения: 13.06.17).
4. Xi H. Imperative programming with dependent types. *Logic in Computer Science*, 2000. Proc. 15th Annual IEEE Sympos. IEEE, 2000, pp. 375–387.
5. Condit J., Harren M., Anderson Z., Gay D., Necula G. Dependent types for low-level programming. *Europ. Sympos. on Programming*. Springer Berlin Heidelberg, 2007, pp. 520–535.
6. Nagarakatte S., Zhao J., Martin M., Zdancewic S. CETS: compiler enforced temporal safety for C. *ACM Sigplan Notices*, 2010, vol. 45, no. 8, pp. 31–40.
7. Serebryany K., Bruening D., Potapenko A., Vyukov D. AddressSanitizer: a fast address sanity checker. *Proc. USENIX Annual Technical Conf.*, 2012, pp. 309–318.
8. Nanevski A., Morrisett G., Birkedal L. Hoare type theory, polymorphism and separation. *Jour. of Functional Programming*, 2008, vol. 18, no. 5–6, pp. 865–911.
9. Nanevski A., Morrisett G., Shinnar A., Govereau P., Birkedal L. Ynot: dependent types for imperative programs. *ACM Sigplan Notices*, 2008, vol. 43, no. 9, pp. 229–240.
10. *Static Single Assignment Book*. 2015. URL: <http://ssabook.gforge.inria.fr/latest/book.pdf> (дата обращения: 13.06.17).

Software & Systems

DOI: 10.15827/0236-235X.121.152-157

Received 20.06.17

2018, vol. 31, no. 1, pp. 152–157

SOURCE CODE MODEL CONSTRUCTION FOR SOFTWARE DEFECT DETECTION WITH TYPE SYSTEMS

L.V. Tsvetkov¹, Postgraduate Student, lavrentii.tsvetkov@corp.ifmo.ru;

A.I. Spivak¹, Ph.D. (Engineering), Associate Professor, anton.spivak@gmail.com

¹ The National Research University of Information Technologies, Mechanics and Optics, Kronverksky Ave. 49, St. Petersburg, 197101, Russian Federation

Abstract. Quality assurance is an integral part of software development. Software defects are among the few components that can be mitigated using static code analysis. One of the forms of static analysis is static type checking. Most programming languages use type systems to reduce the number of software defects. Unfortunately, type systems of most programming languages are incapable of representing basic safety guaranties against exceptions. To allow extension of existing systems of imperative languages, it is necessary to create a source code model that considers a number of factors, such as an alterable state, pointer aliasing and others.

The paper describes a source code model that represents security invariants and allows effective use of extended type systems to locate vulnerable code locations. The proposed solutions of aliasing and alterable state problems are based on using a static single assignment form with applying several rerecording rules. This rules maintain functional equivalence of programs enabling use of principles that are already used for analyzing functional-style programs. The paper considers the notion of linear resource, related problems and possible solutions within the proposed approach. Principles of dependent types indexing are shown by example of a continuous data array. The proposed model extends type system of a given programming language. Construction of such model with extended type system does not require additional code annotations, minimizing labour costs needed to preform analysis on the existing code base.

Keywords: code model, type systems, software defects.

References

1. Norell U. Dependently typed programming in Agda. *Advanced Functional Programming*. Springer Publ., 2009, pp. 230–266.
2. Brady E. Idris, a general-purpose dependently typed programming language: Design and implementation. *Jour. of Functional Programming*. 2013, vol. 23, no. 5, pp. 552–593.
3. *The Rust Programming Language*. Available at: <https://www.rust-lang.org/> (accessed June 13, 2017).
4. Xi H. Imperative programming with dependent types. *Proc. 15th Annual IEEE Symp. on Logic in Computer Science*. 2000, pp. 375–387.
5. Condit J., Harren M., Anderson Z., Gay D., Necula G. Dependent types for low-level programming. *Programming Languages and Systems*. Springer Publ., 2007, pp. 520–535.
6. Nagarakatte S., Zhao J., Martin M., Zdancewic S. CETS: compiler enforced temporal safety for C. *ACM Sigplan Notices*. 2010, vol. 45, pp. 31–40.
7. Serebryany K., Bruening D., Potapenko A., Vyukov D. AddressSanitizer: A fast address sanity checker. *Proc. 2012 USENIX Annual Technical Conf. (USENIX ATC 12)*. 2012, pp. 309–318.
8. Nanevski A., Morrisett G., Birkedal L. Hoare type theory, polymorphism and separation. *Jour. of Functional Programming*. 2008, vol. 18, no. 5–6, pp. 865–911.
9. Nanevski A., Morrisett G., Shinnar A., Govereau P., Birkedal L. Ynot: dependent types for imperative programs. *ACM Sigplan Notices*. 2008, vol. 43, pp. 229–240.
10. *Static Single Assignment Book*. 2015. Available at: <http://ssabook.gforge.inria.fr/latest/book.pdf> (accessed June 13, 2017).