DOI: 10.15514/ISPRAS-2019-31(6)-5

Исследование и разработка межпроцедурных алгоритмов поиска дефектов в исполняемом коде программ

[1,2] Г.С. Иванов, ORCID: 0000-0002-0544-6816 <gregory@ispras.ru>
[3] П.М. Пальчиков, ORCID: 0000-0003-0449-2447 <pavel98@ispras.ru>
[3] А.Ю. Тарасов, ORCID: 0000-0002-6212-8144 <tematarasov48rus@ispras.ru>
[3] Г.С. Акимов, ORCID: 0000-0002-6601-4312 <akimg@ispras.ru>
[4] А.К. Асланян, ORCID: 0000-0002-7320-4835 <hayk@ispras.ru>
[4] В.Г. Варданян, ORCID: 0000-0002-4899-2999 <vaag@ispras.ru>
[4] А.С. Арутюнян, ORCID: 0000-0002-2420-7968 <arutunian@ispras.ru>
[4] Г.С. Керопян, ORCID: 0000-0003-2150-0461 <goqor@ispras.ru>

[1] Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2] Московский авиационный институт (национальный исследовательский университет)
125993, Россия, г. Москва, Волоколамское шоссе, д. 4
[3] Московский Государственный Технический Университет имени Н.Э.Баумана,
105005, Россия, г. Москва, 2-я Бауманская ул., д. 5, стр. 1
[4] Российско-Армянский Университет,
0051, Республика Армения, г. Ереван, ул. Овсепа Эмина 123

Аннотация. В последнее время всё больше компаний, производящих программное обеспечение, заинтересованы в инструментах повышения стабильности и безопасности их продукта. Используемые разработчиками закрытые библиотеки и сторонние приложения могут содержать дефекты, использование которых злоумышленником или пользователем может привести к нарушению стабильности и безопасности работы приложения. В ряде случаев исходный код проблемных участков может отсутствовать. Приобретают популярность статические методы поиска дефектов в коде, позволяющие находить дефекты, недостижимые для динамических методов. Статические методы представляют собой алгоритмы исследования статической модели программы, в том числе графа вызовов, потока управления, потока данных. Исследование бинарного кода предполагает восстановление статической модели программы из бинарного файла путём дизассемблирования, восстановления границ функций, трансляцию в промежуточное представление и восстановление графа вызовов. Дефекты в современных кодовых базах, как правило, проявляются лишь на определённом множестве путей в графе вызовов, что требует межпроцедурных алгоритмов поиска дефектов. Целью данной работы является разработка методов межпроцедурных алгоритмов поиска дефектов в бинарном коде, обладающих хорошей масштабируемостью, набором поддерживаемых архитектур и приемлемой точностью. Алгоритмы построены на базе инструмента ИСП РАН Binside.

Ключевые слова: статический анализ кода; поиск ошибок; анализ исполняемого кода

Для цитирования: Иванов Г.С., Пальчиков П.М., Тарасов А.Ю., Акимов Г.С., Асланян А.К., Варданян В.Г., Арутюнян А.С., Керопян Г.С. Исследование и разработка межпроцедурных алгоритмов поиска дефектов в исполняемом коде программ. Труды ИСП РАН, том 31, вып. 6, 2019 г., стр. 89–98. DOI: 10.15514/ISPRAS–2019–31(6)–5

Благодарности: Работа поддержана грантом РФФИ 18-07-01154А

Research and development of interprocedural algorithms for defect searching in executable program code

[1,2] G.S. Ivanov, ORCID: 0000-0002-0544-6816 <gregory@ispras.ru>
[3] P.M. Palchickov, ORCID: 0000-0003-0449-2447<pavel98@ispras.ru>
[3] A.Y. Tarasov, ORCID: 0000-0002-6212-8144 <tematarasov48rus@ispras.ru>
[3] G.S. Akimov, ORCID: 0000-0002-6601-4312 <akimg@ispras.ru>
[4] A.K. Aslanyan, ORCID: 0000-0002-7320-4835 <hayk@ispras.ru>
[4] V.G. Vardanyan, ORCID: 0000-0002-4899-2999 <vaag@ispras.ru>
[4] M.S. Arutunian, ORCID: 0000-0002-2420-7968 <arutunian@ispras.ru>
[4] G.S. Keropyan, ORCID: 0000-0003-2150-0461 <goqor@ispras.ru>

[1] Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[2] Moscow Aviation Institute (National Research University)
4, Volokolamskoe shosse, Moscow,125993, Russia
[3] Bauman Moscow State Technical University,
5/1, 2nd Baumanskaya, Moscow, 10500, Russia
[4] Russian-Armenian University,
123 Hovsep Emin str., Yerevan, 0051, Armenia

Abstract: Recently, more and more software companies are interested in tools to improve the stability and security of their product. The closed libraries and third-party applications used by developers may contain defects, the use of which by an attacker or by a user may lead to a violation of the stability and security of the application. In some cases, the source code of the problem areas may be missing. At the moment, static methods for finding defects in code are gaining popularity, which allow finding defects that are unattainable for dynamic methods. Static methods are algorithms for studying a static model of a program, including a call graph, control flow, data flow. Studying binary code involves restoring a static model of a program from a binary file by disassembling, restoring function boundaries, translating it into an intermediate representation, and restoring a call graph. Defects in modern code bases, as a rule, appear only on a certain set of paths in the call graph, which requires interprocedural algorithms for finding defects. The aim of this work is to develop methods of interprocedural algorithms for finding defects in binary code that have good scalability, a set of supported architectures, and acceptable accuracy. Algorithms are developed based on ISP RAS Binside tool.

Keywords: staic code analysis; defect searching; executable code analysis

For citation: Ivanov G.S., Palchickov P.M., Tarasov A.Y., Akimov G.S., Aslanyan A.K., Vardanyan V.G., Arutunian M.S., Keropyan G.S. Research and development of interprocedural algorithms for defect searching in executable program code. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 6, 2019. pp. 89-98 (in Russian). DOI: 10.15514/ISPRAS-2019-31(6)-5

Acknowledgements. The work is supported by RFBR, grant 18-07-01154.

1. Введение

В последнее время доля информационных технологий во всех сферах жизнедеятельности человека стремительно увеличивается. Становится всё больше мест, где внедряются автоматизированные системы, начиная с критических информационных инфраструктур и заканчивая повседневной жизнью человека.

Закономерно, что производители программного обеспечения стараются защитить свои продукты от воздействия злоумышленников. Пользователи же хотят быть уверенными, что их персональные данные не попадут к третьим лицам, а приложение будут стабильно работать. Таким образом, встает вопрос обеспечения безопасности и стабильности приложений.

В целях обеспечения компьютерной безопасности и повышения стабильности, программу исследуют и проводят различные виды анализов, для обнаружения сбоев, дефектов, либо утечек данных. В настоящее время используют два подхода к анализу бинарного кода: статический и динамический анализ. Динамический анализ позволяет проводить анализ во время исполнения программы. Каждый дефект, найденный в процессе динамического анализа, соответствует реальному дефекту в коде программы. Кроме того, для динамического анализа не требуется наличие исходного кода исследуемой программы. Сложности возникают при решении задачи, касающейся генерации входных данных, которые могут покрыть все ошибочные точки. Статический анализ проводится без исполнения кода программы и включает в себя методы анализа потока управления, анализа потока данных, а также методы, использующие символьное выполнение. Статические методы поиска дефектов позволяют найти дефекты на всех путях исполнения программы, в то время как динамические – только на выполнившихся путях.

Статический анализ может быть выполнен как на уровне исходного, так и бинарного кода. Анализ исходного кода будет более полным, поскольку опирается на наиболее полную информацию о программе, имеющуюся в исходном коде. Компиляция отбрасывает часть информации, содержащейся в исходном коде, например, имена переменных и функций. Однако, анализ бинарного кода также имеет смысл. Например, компиляция дефектным компилятором, наличие неоднозначных конструкций в исходном коде, экспериментальных возможностей языка и ошибок в компиляторных оптимизациях могут создавать такие дефекты в бинарном коде, которые отсутствуют в исходном тексте программы. Дефекты на уровне исходного кода, которые содержатся в закрытых библиотеках, используемых при сборке программы, переносятся и в собранную программу.

К сожалению, в наше время открытые промышленные анализаторы бинарного кода не обладают хорошей масштабируемостью, набором поддерживаемых архитектур и достаточной точностью работы. В ИСП РАН разрабатывается инструмент, способный решать поставленные задачи, – Binside. В данной статье будет рассмотрены принципы работы инструмента и достигнутые на текущий момент результаты.

Дальнейшее изложение построено следующим образом. В разд. 2 рассмотрены существующие методы и инструменты поиска дефектов на бинарном коде. В разд. 3 описывается общая архитектура межпроцедурного поиска дефектов в исполняемом коде. Разд. 4 посвящён промежуточному представлению и выполняемому над ним анализу. Межпроцедурный анализ помеченных данных рассматривается в разд. 5. Разд. 6 посвящён учёту вызовов функций в анализе потока данных. Разд. 7 содержит описание детекторов ошибок и пример их работы. Разд. 8 описывает результаты работы инструмента, и разд. 9 завершает статью.

2. *Исследование существующих подходов*

BAP – статический анализатор бинарных файлов (x86 и ARM) [1]. На первом этапе бинарный код дизассемблируется на основе линейного алгоритма. Следующий этап – это транслирование бинарного кода в промежуточное представление (без побочных эффектов). Промежуточное представление приводится в форму статического единственного присваивания. На полученном представлении проводится анализ и оптимизация, восстанавливаются зависимости по данным на базе цепочек def-use и use-def, удаляется мертвый код. BAP выполняет задачу транслирования кода в промежуточное представление и восстановления графа потока управления и графа вызовов. В текущей версии [2] пользователям доступны API для написания собственных модулей детекторов. Использование в процессе анализа SMT-решателей, позволяет получать хорошую точность анализов, но при этом уменьшается масштабируемость инструмента.

BARF [3] - статический анализатор бинарных файлов с открытым исходным кодом. Архитектура фреймворка состоит из трех модулей: ядро, архитектура и анализатор. Первый разделён на следующие модули: REIL, SMT и BL, которые не зависят от платформы. Кроме того, он реализует эмулятор, парсер и трансляцию во внутреннее представление. Модуль SMT используется для создания переменных и установки допущений, после проверяет выполнимость бинарного кода на различных наборах формул. Модуль архитектуры описывает архитектуру, т.е. регистры, размер адреса памяти, наборы команд, операндов и другие данные. Модуль архитектуры содержит модуль синтаксического анализа, который получает строку с разобранными инструкциями и создает серию аннотированных объектов, которые их описывают, далее каждая инструкция преобразовывается в семантически эквивалентную последовательность команд REIL [4, 5].

BARF выполняет задачу транслирования кода в REIL и восстановления графа потока управления и графа вызовов. Анализ проводится за счет других инструментов, к примеру, в качестве SMT-решателей выступают инструменты Z3 и CVC4, которые проверяют заданные выражения и находят программные дефекты. Пользователями доступны API для написания собственных модулей детекторов.

BAP и BARF, хотя и предоставляют программный доступ к своим инструментам, требуют значительной доработки. В частности, BARF предоставляет в списке инструментов кроме алгоритмов статического представления программы SMT-решатель. SMT-решатели часто используются в инструментах статического исследования программ, в том числе и в целях поиска дефектов. Однако, как показывает практика, одних SMT-решателей недостаточно для масштабируемого поиска дефектов в коде программы.

3. *Архитектура межпроцедурного поиска дефектов*

Дефекты бинарного кода классифицированы в базе CWE [5]. Каждый элемент этой базы определяет тип дефекта, имеет свой идентификационный номер и описание. С помощью описания можно составить правило, определяющее наличие дефекта в бинарном коде. Для поиска сразу нескольких типов ошибок в коде целесообразно реализовать платформу, на основе которой можно было бы создавать правила поиска дефектов. Большинство дефектов в базе CWE могут быть вызваны последовательностью вызовов различных функций, что требует межпроцедурности платформы [6]. Платформа межпроцедурного поиска дефектов определяется обходом графа вызовов [6], внутрипроцедурным и межпроцедурным анализом потока данных, а также детекторами ошибок (так называемые чекеры), каждый из которых реализует правило поиска дефекта из базы CWE. В инструменте Binside [7], разработанном в ИСП РАН, были реализованы две платформы межпроцедурного поиска дефектов, одна обеспечивает обход графа вызовов снизу-вверх по слоям, вторая – сверху-вниз.

Для межпроцедурного анализа в Binside используется архитектура обхода графа вызовов снизу-вверх, что предполагает снижение контекстной чувствительности, но позволяет учитывать влияние вызовов функций на состояние программы с помощью аннотаций. Аннотации представляют собой структуру данных ограниченного размера, в которую записываются наиболее важные для анализа эффекты от вызова анализируемой функции в произвольном контексте. Аннотации являются структурой, достаточной для создания передаточной функции вызова процедуры. Обход функций производится в соответствии с графом вызовов таким образом, чтобы вызываемая функция анализировалась перед вызывающей. То есть анализ функций начинается с листьев графа вызова функций; при этом сам граф вызова функций разбивается на слои и анализируется послойно.

Слой представляет собой набор функций, представленный таким образом, чтобы каждая функция в графе вызовов анализировалась один раз, и сохранялся порядок снизу-вверх.

Анализ функций проводится независимо друг от друга, и функции в одном слое могут анализироваться параллельно, что позволяет увеличивать масштабируемость инструмента и снижать время анализа бинарного файла. Обход графа вызовов снизу-вверх позволяет найти больше дефектов, однако в силу своей природы теряет точность из-за замены функций в аннотации на их представления.

Качество работы детекторов ошибок можно описать двумя признаками: точностью и полнотой. Точность характеризует отношение количества корректно найденных точек к количеству всех найденных точек, а полнота – отношение корректно найденных точек к количеству всех действительных дефектов в программе. При реализации алгоритма может возникнуть неполнота и избыточность. У хорошего промышленного анализатора кода неполнота и избыточность должны быть сведены к установленному минимуму.

4. Промежуточное представление и выполняемые над ним анализы

Первичный разбор бинарного файла, дизассемблирование, восстановление графа потока управления и восстановление графа вызовов выполняются с помощью IDA Pro [8, 9]. Дизассемблерное представление транслируется в промежуточное представление REIL [4, 5]. Так, множество поддерживаемых для анализа архитектур определяется множеством трансляторов. На данный момент поддерживаются следующие архитектуры: ARM, x86, x64 и MIPS.

REIL – это промежуточное представление программного кода, которое является независимым от платформы. Это язык сходный с ассемблерным представлением, однако содержит всего 17 инструкций (против сотен и тысяч инструкций популярных архитектур) и каждая инструкция не имеет побочных эффектов. Инструкции REIL можно сгруппировать в пять категорий:

1) арифметические;
2) побитовые;
3) работа с памятью;
4) передача управления;
5) служебные (не нужны для анализа в Binside).

Каждая инструкция имеет ровно три операнда, первые два являются источниками и третий – операнд назначения. Существуют три типа операндов: целочисленные литералы, регистры и адреса. Адреса REIL состоят из двух целочисленных частей. Первое число представляет адрес инструкции из исходной архитектуры, второе – нумерация REIL-инструкций.

В Binside реализован анализ значений (Value Analysis, VA) [10]. VA восстанавливает аппроксимированную карту памяти и регистров в каждой точке программы. Все анализы потока данных, включая VA выполняются на промежуточном представлении REIL. VA - это комбинированный алгоритм числового анализа и анализа указателей, Ключевой особенностью VA является то, что он обрабатывает числовые и адресные значения однообразно [11], что имеет решающее значение для анализа исполняемых файлов, так как числовые значения и адреса неразличимы во время выполнения. VA имеет чувствительность к потоку управления.

Восстановление множества инструкций в программе, результаты которых зависят от входных данных, называется анализом помеченных значений [12]. Анализ помеченных значений имеет высокое значение для составления детекторов ошибок в коде программы и трассы, полученные с его помощью, могут характеризоваться тремя видами зависимости: функциональные адресные и по управлению [13]. Функциональная зависимость возникает, когда результат вычисления на одном шаге программы напрямую зависит от результата вычисления на другом. Адресная зависимость возникает, когда выбор ячейки, от которой функционально зависит результат вычисления на одном шаге,

зависит от результата вычисления на другом шаге. А зависимость по управлению определяет возможный порядок исполнения инструкций. Внутрипроцедурный анализ помеченных данных в Binside построен, аналогично VA, на монотонных решётках, что обеспечивает функциональную зависимость и зависимость по управлению в полученных трассах [14]. Схема работы инструмента представлена на рис. 1.

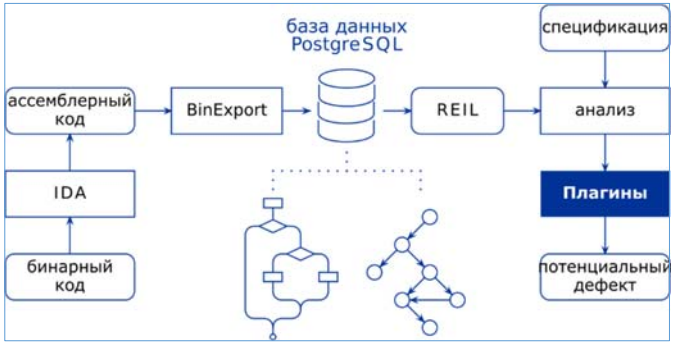

Рис. 1. Схема работы инструмента Binside
Fig. 1. Binside tool operation diagram

5. Межпроцедурный анализ помеченных данных

Платформа Binside поддерживает поиск дефектов сверху-вниз, реализованный в виде межпроцедурного анализа помеченных данных. Начиная с точки входа помеченных данных, помеченные данные распространяются вниз по графу вызовов через аргументы вызываемых функций. При таком подходе в функции ниже по графу вызовов передаются конкретные помеченные аргументы, обеспечивая контекстную чувствительность и повышая полноту анализа. Пользователь имеет возможность задавать вручную функцию, с которой начинается распространение помеченных данных (source), что позволяет дополнить анализ результатами обратной разработки программного обеспечения аналитиком.

6. Учёт вызовов функций в анализах потока данных

Полнота анализа потока данных требует учёта влияния вызываемых функций. В результате динамической линковки программ код в результирующем бинарном файле вызывает функции из других бинарных файлов – динамически подключаемых библиотек. Учёт влияния вызовов библиотечных функций на поток данных можно реализовать двумя способами. Первый способ состоит в использовании в процессе анализа бинарного файла все требуемые для его работы динамически подключаемые библиотеки [15], что требует их наличия в системе. Второй метод использует базу аннотаций для каждой неизвестной функции [16]. Аннотация определяет влияние конкретной функции на контекст вызывающей функции. Использование аннотации при анализе инструкции вызова функции позволяет учесть влияние вызываемой функции на контекст анализа потока данных не имея кода вызываемой функции.

Для аннотаций в инструменте Binside составлена база функций из наиболее часто используемых библиотек. Функции управления памятью, используемые в операторах C++ “new” и ”delete” [17], также включены в базу. Аннотация записывается в виде набора правил учёта вызова функции на контекста анализа потока данных. К примеру, аннотация “allocatesMemory” имеет семантику выделения памяти на куче и при анализе потока данных помещает в возвращаемое значение функции символический адрес новой

выделенной памяти на куче. Аннотации могут использоваться в детекторах ошибок, для поиска дефектов некорректного вызова функций.

7. Детекторы ошибок и пример их работы

Детекторы ошибок в Binside запускаются платформой в процессе межпроцедурного анализа как сверху вниз, так и снизу вверх после выполнения всех внутрипроцедурных анализов потока данных. Детекторы принимают на вход промежуточное представление программы и результаты внутрипроцедурных анализов потока данных и потока управления, в частности: восстановление карты памяти и регистров, построение цепочек определение-использование (def-use) и восстановление трассы помеченных данных. Если правило, реализованное в детекторе сработало, с учётом результатов всех выполненных анализов, платформа генерирует сообщение об ошибке.

Примером такого детектора ошибок может служить детектор переполнения буфера, основанного на некорректном вызове функции `strcpy`. Анализ помеченных данных, поддержанный анализом значений, помечает все инструкции и операнды, содержащие данные зависимые от входных. Если доказывается, что первый аргумент `strcpy`, представляющий собой указатель на буфер-источник, строится из данных пользователя – генерируется ошибка. Схожим детектором является детектор инъекции команд, также основанного на результатах анализа помеченных данных. Если доказывается, что в аргумент вызова system, передающего команду в командную оболочку, попадают данные, формируемые пользователем – платформа генерирует предупреждение.

В качестве информации, дополняющей анализ, могут служить функции, полученные вследствие ручного анализа и потенциально содержащие ошибку в работе с памятью, например, выполняющие неаккуратное копирование в ходе своей работы. При успешном доказательстве контроля пользователем данных поступающих в аргументы такой функции соответствующим детектором генерируется предупреждение об ошибке.

Рассмотрим действие алгоритма и правил на простом примере (листинг 1).

```
int main(int argc, char *argv[]) {
   char buf[10];
   strcpy(buf, argv[1]);
}
```

Листинг 1. Пример утечки помеченных данных
Listing 1. Example of tagged data leakage

На листинге 1 приведен пример программы, написанной на языке C++, содержащей переполнение буфера. В первой строке программа получает аргументы функции main которые будут являться источником помеченных данных. В третьей строке вызывается небезопасная функция strcpy, которой в качестве второго аргумента попадает помеченное значение `argv[1]`. Так как это значение полностью контролируется пользователем, а функция `strcpy` не проверяет длину исходной строки, то, если подать на вход строку, содержащую более 10 символов, произойдёт переполнение буфера. Функция `strcpy` присутствует в базе аннотаций Binside с аннотацией неаккуратного копирования данных. А условие, при котором второй аргумент этой функции помечен, генерирует предупреждение о дефекте в программе.

8. Результаты

В табл. 1 приведены результаты работы детекторов при использовании обхода графов вызовов сверху вниз.

Запуск межпроцедурной платформы поиска дефектов сверху вниз также показал неплохие результаты на бинарных файлах с дефектом переполнением буфера при специально

оформленных аргументах командной строки. Для таких бинарных файлов точкой входа помеченных данных является функция «main». Результаты работы представлены в табл. 2.

Табл. 1. Результаты работы детекторов ошибок с обходом графа вызовов сверху вниз
Table 1. The results of the operation of error detectors with the call graph top-down traversal

Имя бинарного файла	Версия	Размер	Время анализа	Количество найденных точек	Количество корректно найденных точек
accell-ppd	1.10.0	244KB	1m40sec	6	4
giflib	5.1.2	116KB	15 sec	2	2
open-slp	1.2.1	336KB	50 sec	4	3
tiff2pdf	4.0.3	124KB	3m 31sec	4	2
jasper	<1.900.2	200KB	5m 36 sec	4	2
Процент корректных срабатываний:					65%

Табл. 2. Результаты работы детекторов ошибок с обходом графа вызовов снизу вверх
Table 2. The results of the operation of error detectors with the call graph bottom-up traversal

Имя бинарного файла	Версия	Размер	Время анализа	Количество найденных точек	Количество корректно найденных точек
hsolinkcontrol	1.0.118	28KB	3s	23	23
shar	4.2.1	42.5KB	31s	2	1

Как видно, в обоих таблицах присутствуют срабатывания на точках, не являющихся действительными дефектами программного обеспечения. В общем случае невозможно разработать абсолютно строгий алгоритм поиска дефектов для применения на современных программах, поэтому в современных статических анализаторах делается ряд предположений, вносящих нестрогость в общий алгоритм и понижающих точность анализа.

9. Заключение

В ходе работы была реализована платформа поиска дефектов в исполняемом коде с поддержкой анализа значений на основе монотонных решеток. На её основе была выбрана архитектура обхода графа вызовов функций снизу-вверх с использованием аннотаций и совместно реализована архитектура межпроцедурного анализа помеченных данных сверху-вниз. Была составлена база аннотаций, которые могут использоваться в детекторах ошибок для поиска дефектов некорректного вызова функций. Также были разработаны детекторы для обнаружения дефектов в программе на основе результатов полученных анализов. Используемые решения обладают хорошей масштабируемостью, поддерживают различные архитектуры и обладают приемлемой точностью.

Список литературы / References

[1]. David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: a binary analysis platform. In Proc. of the 23rd International Conference on Computer Aided Verification, 2011, pp. 463-469.

[2]. BinaryAnalysisPlatform / bap. Available at https://github.com/BinaryAnalysisPlatform/bap, accessed 10.12.2019.

[3]. programa-stic / barf-project. Available at https://github.com/programa-stic/barf-project, accessed 10.12.2019.

[4]. Thomas Dullien, Sebastian Porst. REIL: A platform-independent intermediate representationof disassembled code for static code analysis. In Proc. of the CanSecWest Conference, 2009, 7 p.
[5]. Mitre. Available at https://cwe.mitre.org/, accessed 10.12.2019.
[6]. V. P. Ivannikov, A. A. Belevantsev, A. E. Borodin, V. N. Ignatiev, D. M. Zhurikhin, A. I. Avetisyan. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, vol. 40, issue 5. 2014, pp. 265-275. DOI: 10.1134/S0361768814050041.
[7]. BINSIDE. Инструмент обнаружения дефектов в программе методами статического анализа исполняемого кода. Режим доступа: https://www.ispras.ru/technologies/binside/, дата обращения 10.12.2019 / BINSIDE. Static binary code analysis tool. Available at https://www.ispras.ru/en/technologies/binside/, accessed 10.12.2019.
[8]. IDA Pro. Available at https://www.hex-rays.com/, accessed 10.12.2019.
[9]. Chris Eagle. The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. 2nd edition. No Starch Press, 2011, 672 p.
[10]. G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You Execute. ACM Transactions on Programming Languages and Systems, 2010, Article No. 23.
[11]. Balakrishnan G., Reps T. Analyzing Memory Accesses in x86 Executables. Lecture Notes in Computer Science, vol. 2985, 2004, pp. 5-23.
[12]. Кошелев В.К., Избышев А.О, Дудина И.А. Межпроцедурный анализ помеченных данных на базе инфраструктуры LLVM. Труды ИСП РАН, том 26, вып. 2, 2014 г., стр. 97-118 / Koshelev V.K., Izbyshev A.O., Dudina I.A. Interprocedural taint analysis for LLVM-bitcode. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 2, 2014, pp. 97-118 (in Russian). DOI: 10.15514/ISPRAS-2014-26(2)-4.
[13]. Тихонов А.Ю., Аветисян А.И. Развитие taint-анализа для решения задачи поиска программных закладок. Труды ИСП РАН, том 20, 2011 г., стр. 9-24 / Tichonov A.Y., Avetisyan A.I. Development of taint-analysis methods to solve the problem of searching of undeclared features. Trudy ISP RAN/Proc. ISP RAS, vol. 20, 2011, pp. 9-24 (in Russian).
[14]. Беляев М.В., Шимчик Н.В., Игнатьев В.Н., Белеванцев А.А. Сравнительный анализ двух подходов к статическому анализу помеченных данных. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 99-116 / M.V. Belyaev, N.V. Shimchik, V.N. Ignatyev, A.A. Belevantsev Comparative analysis of two approaches to the static taint analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017, pp. 99-116. DOI: 10.15514/ISPRAS-2017-29(3)-7.
[15]. Angr. Available at https://github.com/angr/angr, accessed 10.12.2019
[16]. Статический анализатор Svace. Промышленный поиск критических ошибок в безопасном цикле разработки программ. Режим доступа: https://www.ispras.ru/technologies/svace /, дата обращения 10.12.2019 / Svace Static Analyzer. Finding critical program errors in production within secure development lifecycles. Available at https://www.ispras.ru/en/technologies/svace/, accessed 10.12.2019.
[17]. Bjarne Stroustrup. The C++ Programming Language: Special Edition (3rd Edition). Addison-Wesley Professional, 2000, 1030 p.

Информация об авторах / Information about authors

Григорий Сергеевич ИВАНОВ – старший лаборант ИСП РАН. Студент специалитета МАИ. Научные интересы: анализ бинарного кода, компиляторные технологии, параллельное программирование.

Grigoriy Sergeevich IVANOV – senior assistant of ISP RAS, MAI student. Research interests: binary code analysis, compiler technologies, parallel programming.

Павел Михайлович ПАЛЬЧИКОВ – студент специалитета МГТУ им. Баумана. Научные интересы: анализ бинарного кода, компиляторные технологии, параллельное программирование.

Pavel Michaylovich PALCHIKOV – BMSTU student. Research interests: binary code analysis, compiler technologies, parallel programming.

Артём Юрьевич ТАРАСОВ – студент специалитета МГТУ им. Баумана. Научные интересы: анализ бинарного кода, компиляторные технологии, параллельное программирование.

Artem Yuryevich TARASOV – BMSTU student. Research interests: binary code analysis, compiler technologies, parallel programming.

Глеб Станиславович АКИМОВ – студент специалитета МГТУ им. Баумана. Научные интересы: анализ бинарного кода, компиляторные технологии, параллельное программирование.

Gleb Stanislavovich AKIMOV – BMSTU student. Research interests: binary code analysis, compiler technologies, parallel programming.

Айк Каренович АСЛАНЯН – научный сотрудник, преподаватель, кандидат физико-математических наук. Сфера научных интересов: анализ программ, статический анализ кода, поиск клонов кода.

Hayk Karenovich ASLANYAN – researcher, lecturer, Ph.D in physical and mathematical sciences. Research interests: program analysis, code static analysis, code clone detection.

Ваагн Геворгович ВАРДАНЯН – научный сотрудник, преподаватель, кандидат технических наук. Сфера научных интересов: анализ программ, статический и динамический анализ кода.

Vahagn Gevorgovich VARDANYAN – researcher, lecturer, Ph.D in technical sciences. Research interests: program analysis, code static and dynamic analysis.

Мариам Сероповна АРУТЮНЯН – научный сотрудник, преподаватель, аспирант. Сфера научных интересов: анализ программ, статический анализ кода, поиск клонов кода.

Mariam Seropovna ARUTUNIAN – researcher, lecturer, Ph.D student. Research interests: program analysis, code static analysis, code clone detection.

Григор Сосович КЕРОПЯН – научный сотрудник, магистрант. Сфера научных интересов: анализ программ, статический анализ кода.

Grigor Sosovich KEROPYAN – researcher, master student. Research interests: program analysis, code static analysis.