НАПРАВЛЕННЫЙ ФАЗЗИНГ НА ОСНОВЕ ДИНАМИЧЕСКОЙ ИНСТРУМЕНТАЦИИ

Сережа Асрян
asryan@ispras.ru
Лаборатория системного программирования ЕГУ, ИСПРАН

Введение

- В современном мире разработка надежного программного обеспечения является одной из важнейших задач в области информационных технологий
- С ростом сложности и объёма нынешних программных комплексов все труднее становится разработка эффективных методов анализа качества и надежности программного обеспечения (ПО)
- Не существует единственного метода автоматического анализа позволяющий обнаружить все типы дефектов

Метод динамического анализа

- Динамический анализ проводится *в среде выполнения самой* программы во время ее *исполнения*, что позволяет исследовать программы, если инструменту анализа доступен исполняемый код программы, но не доступен её исходный код
- Анализ проводится на конкретных входных данных, что почти полностью устраняет ложные срабатывания. В то же время исследуются только те участки программы которые были реально выполнены
- Основными методами являются ***фаззинг*** и ***динамическое символьное выполнение***

Фаззинг

- Многократный запуск ПО на модифицированных (*неправильных, неожиданных или случайных*) данных и отслеживание поведения программы (*падения, зависания и т.д.*)
- Разделяются несколько типов фаззинга
  - **Глупый** фаззинг (*dumb, black-box fuzzing*)
    - Отсутствует представление о внутренней структуре ПО
  - **Умный** фаззинг (*smart, white-box, gray-box fuzzing*)
    - При помощи инструментации (*DynamoRio DBI, QEMU*) программы, генерирует входные данные учитывая покрытие кода ПО
- Одни из самых популярных инструментов фаззинга – *AFL* (*American Fuzzy Lop*), oss-fuzzer, libfuzzer и т.д.

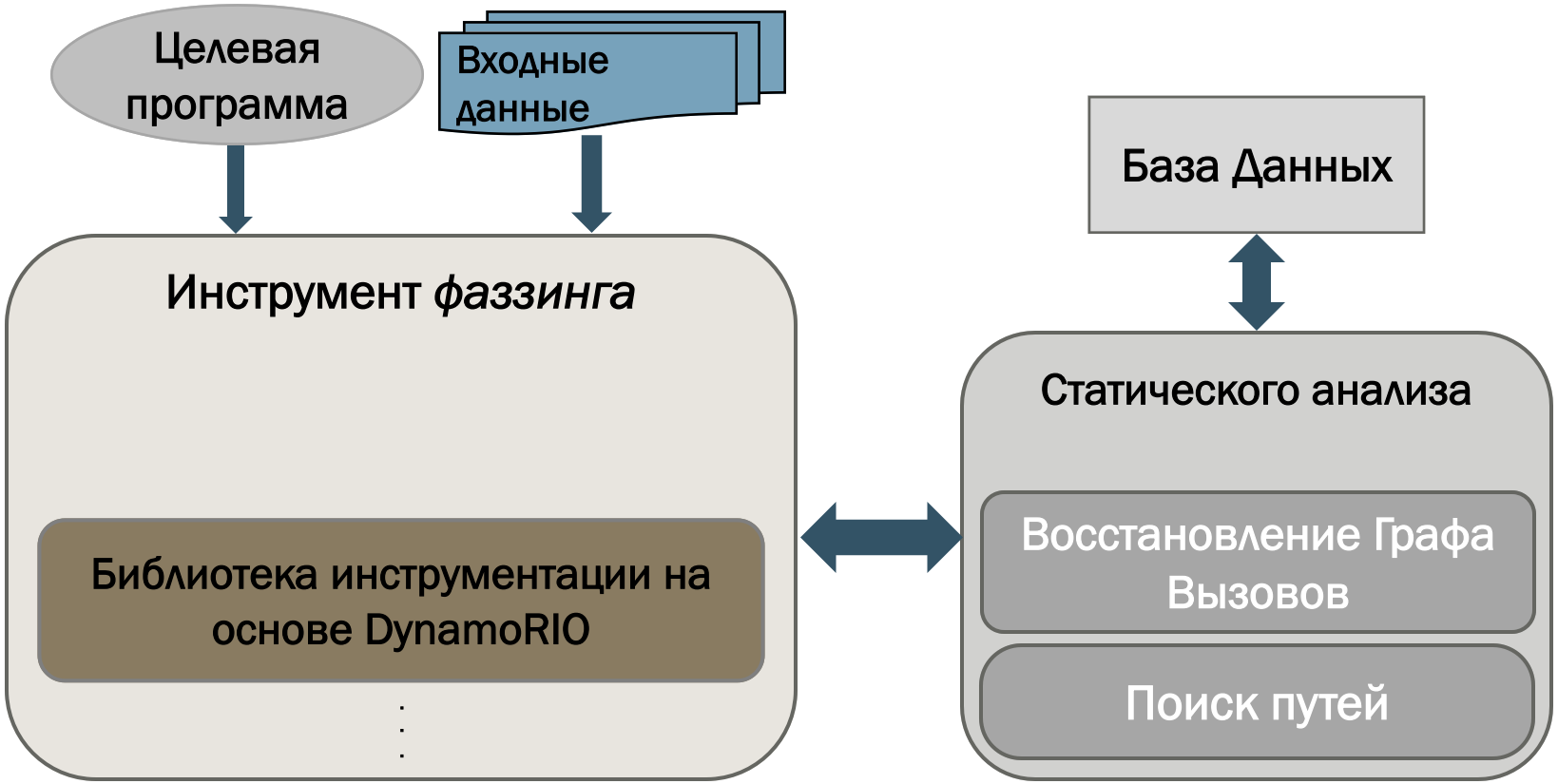
Направленный динамический анализ ПО

- С помощью динамического анализа практически невозможно исследовать все пути выполнения
- Разработаны и реализованы два метода направленного фаззинга, которые основаны на ограничении анализируемой части программного кода, на основе информации о *интересующих участков кода*, потенциально содержащие дефекты:
  - *Выполнение частичной инструментации исполняемого кода (**Direct**)*
  - *Вставка инструкций завершения выполнения (exit (0)), в определенных базовых блоках ПО (**Direct Fast**)*

Комбинированный подход двух основных типов анализа

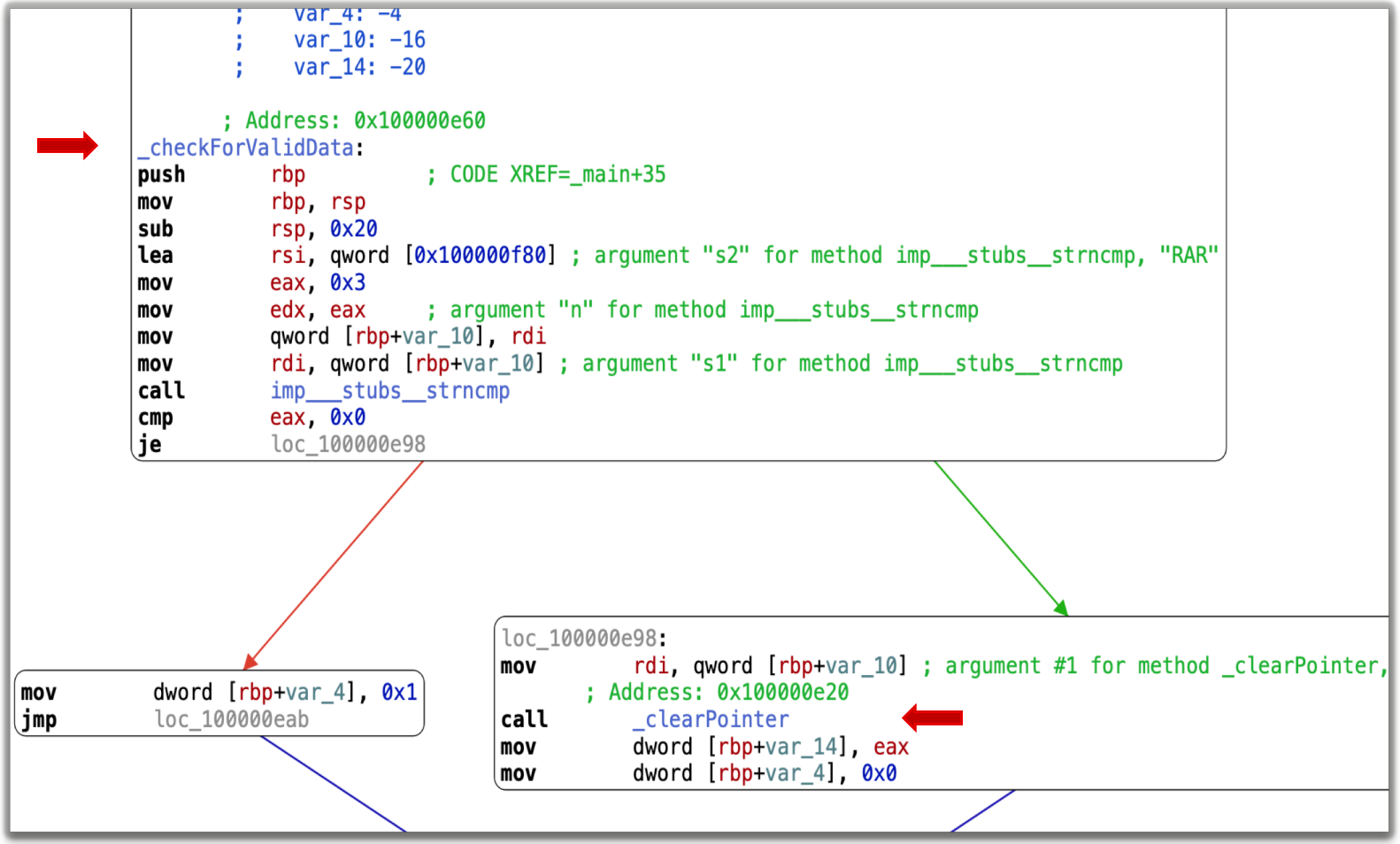
- Чтобы решить эту проблему нами был разработан и реализован метод позволяющий комбинировать возможности динамического и статического анализа
- Метод можно разделить на следующие этапы
  - ***Этап 1 – Поиск точек в ПО являвшимися потенциальными дефектами***
    - Получить интересующие точки (адреса функций) в ПО выполнив статический анализ
  - ***Этап 2 - Статическое построение путей в программе***
    - Использовать полученные точки для построения путей в ПО, достигающие заданным точкам (*проходящие по этим адресам*), с помощью разработанного нами инструмента ***SPD (Static Path Detection)***
  - ***Этап 3 - Выполнение направленного фаззинга на основе сгенерированных путей***
  - ***Этап 4 - Итеративное обновление статической базы данных на основе трасс (****execution trace****) выполнения ПО***

Архитектура инструмента



Комбинированный подход двух основных типов: *этап 1*

- Адреса полученные из статического анализа – это адреса вызовов функций с дефектами и точка их вызова

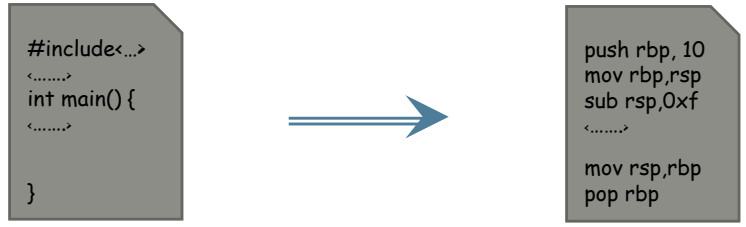


clearPointer в точке вызова *0x100000e20* где присутствует дефект повторного освобождения памяти

В функции *checkForValidData* с адресом *0x100000ed1*
откуда и вызвана функция *clearPointer*

Комбинированный подход двух основных типов: *этап 2*

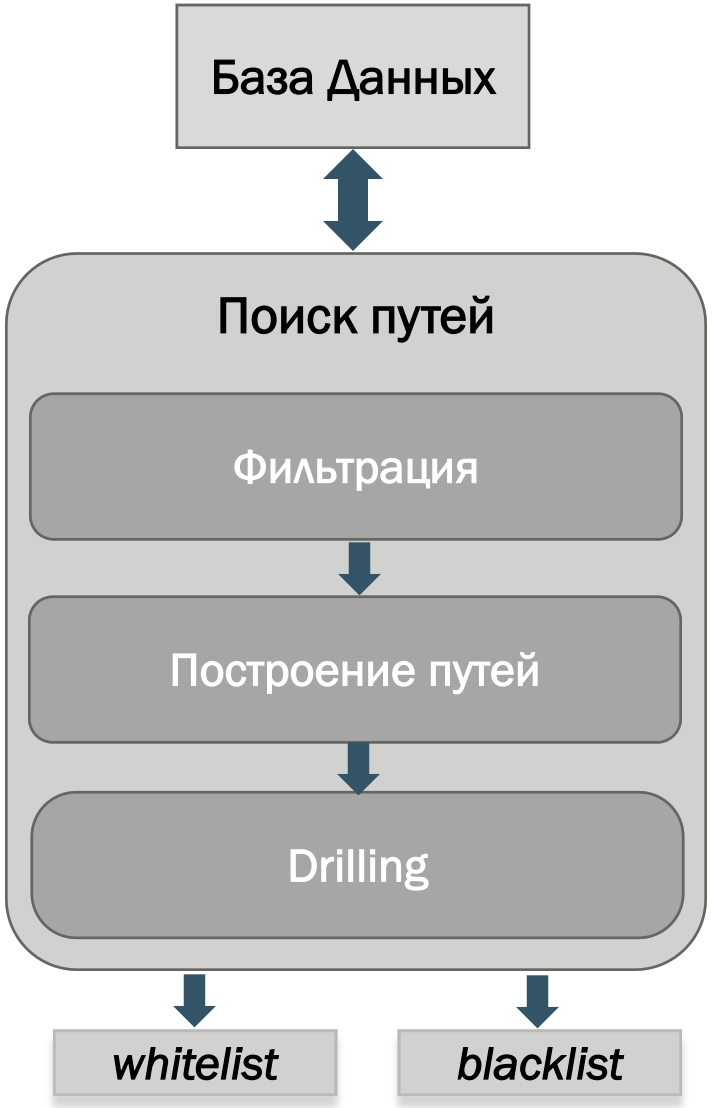
- Инструмента статического поиска путей ***SPD***



Исполняемый код

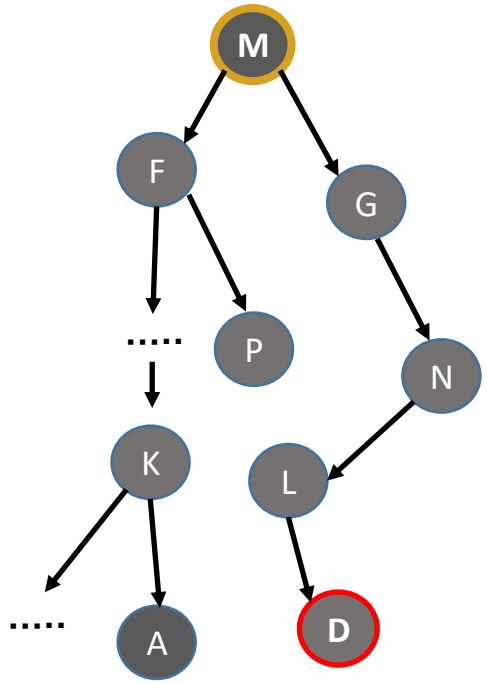
Дизассемблированный код

- Основные этапы построения путей
  - *Обход (BFS) по графу вызовов функций (Call Graph)*
  - *Обход (DFS) по графу потока управления (Control Flow Graph)*



Этап 2 - Статическое построение путей в программе

- Нужно найти путь от базового блока ***M*** до точки дефекта в базовом блоке ***D***



- Построенный путь от ***M*** до ***D*** будет содержать начальные и конечные адреса базовых блоков
- В случаи когда в базовых блоках присутствует вызов функции (*call func*) все блоки этой функции включаются в результат (*Drilling*)

Этап 3 - Направленный динамический анализ ПО

- Разработан и реализован метод направленного фаззинга путем частичной инструментации исполняемого кода
- Целью данного компонента является локализация той части исполняемого программного кода, который необходимо анализировать
- На этапе выполнения динамической инструментации программы, инструментируются только базовые блоки, принадлежащие построенным путям
- В следствии этого все ***мутации над входными данными*** выполняются только по отношению ***этого участка кода***. ***Повышается эффективность и скорость*** выполнения анализа

Этап 4 - Итеративное обновление статической базы данных

- Иногда во время статического поиска путей отсутствует информация о некоторых вызываемых функциях
- В языке программирования C++, поддерживается механизм виртуальных функций
- В программах, написанных на языке C++, могут быть два типа виртуальных вызовов функций:
  - *классические вызовы по указателю на функцию*
  - *вызовы виртуальных методов классов*
- Во время компиляции невозможно определить, какая функция будет вызываться, следовательно появляется проблема ***восстановлении*** *виртуальных функций*

Этап 4 - Итеративное обновление статической базы данных, Восстановлении виртуальных функций

- Разработан и реализован метод CgRec (*Call Graph recovery*) для восстановления неявных вызовов функций
- Для восстановлении виртуальных функций, ***во время фаззинга***, сохраняется трасса выполнения (*execution trace*) ПО
- Трасса представляющая последовательность всех выполненных базовых блоков, в виде двух последующих базовых блока с их начальными и конечными адресами
- Используя полученную трассу и представление в базе данных (*Binnavi*), выполняется восстановление виртуальных вызовов, путем обновления соответствующих таблиц базы данных
- После восстановления нужных данных переходим на **этап 2**

ИСП РАН

Результаты

- Метод был тестирован на программах из набора *DARPA challenges*. Время выполнение каждого тесто - 10 часов. В таблице количество найденных падений и зависаний программ

Имя программы	*Direct fuzz*	*Direct Fast*	*AFL*
Personal_Fitness_Manager	2	0	0
Humaninterface	1	0	0
H2OFlowInc	0	1	0
One_Vote	1	0	0
Middleout	1	0	0
Particle_Simulator	0	1	0
Single-Sign-On	2	0	0
Stream_vm2	0	0	2
ASL6parse	1	0	0
Network_Queuing_Simulator	0	0	27

Результаты

Имя программы	*Direct fuzz*	*Direct Fast*	*AFL*
3D_Image_Toolk it	5	0	0
ECM_TCM_Simulator	2	2	0
XStore	1	0	0
HackMan	1	1	0
SAuth	1	0	1
CGC_Board	2	1	0
Multipass3	0	2	0
Flash_File_System	13	0	43

В следствии тестирования получена оценка по которой метод *Direct Fast* может быть в **3** раза быстрее чем *Direct fuzz*

ИСП РАН

СПАСИБО !