ПОЛИТЕХ
Санкт-Петербургский
политехнический университет
Петра Великого

Автоматизация поиска уязвимостей с помощью обратной трассировки графа передачи управления

Демидов Роман, СПБПУ

Решаемая задача

- Хотим находить уязвимости в бинарном коде (пока что x86).
- В идеале хотим находить все уязвимости (из известных классов), а не только те, что легко находятся.
- Хотим получить масштабируемое решение.
- Лучше false positive, чем false negative.

Ключевые проблемы статического анализа

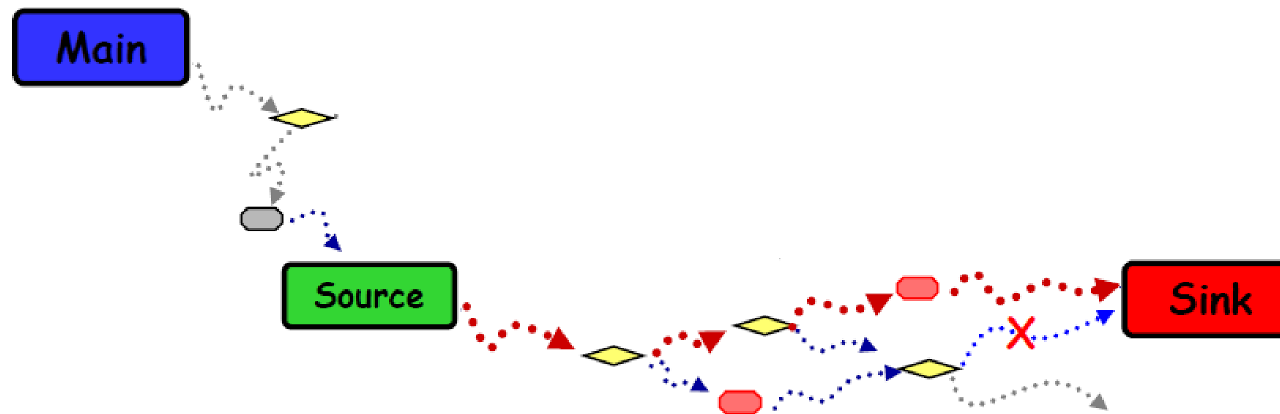
- *Сторонние эффекты* у команд.
- *Непрямые переходы/вызовы* функций. Пример: call [eax]. Граф вызовов без них неполный.
- *Косвенная адресация*.
- *Стек,* семантика push/pop. Наличие “памяти” - результат зависит от предыдущих вызовов.
- *Циклы*. Проблема останова.
- *Параметры функций* кладутся на стек. Отсутствие точной границы между параметрами функции и прочими величинами.

Существующие решения

- Evarista[1] – экспериментальный инструмент статического анализа для архитектуры SPARC.
- IntScope[2] – программа для поиска целочисленного переполнения в бинарном коде для x86.
- CodeSurfer/CodeSonar[3] – статический анализ кода и поиск различных уязвимостей.

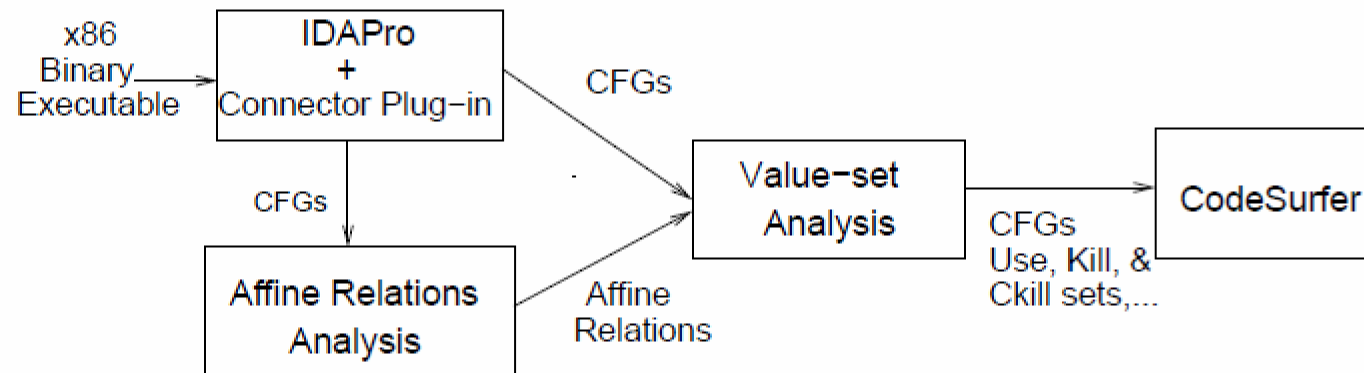
IntScope

- Ищутся пути в CFG от источников входных данных (fread, recv, argv[] и.т.д.) к потенциально уязвимым операциям (malloc, new, array[i]).
- На каждом из найденных путей производится символьные преобразования входных данных к целевым параметрам в конце путей (размер буфера, индекс массива, сдвиг указателя).
- Проверяется возможность целочисленного переполнения целевых параметров на выбранном пути.



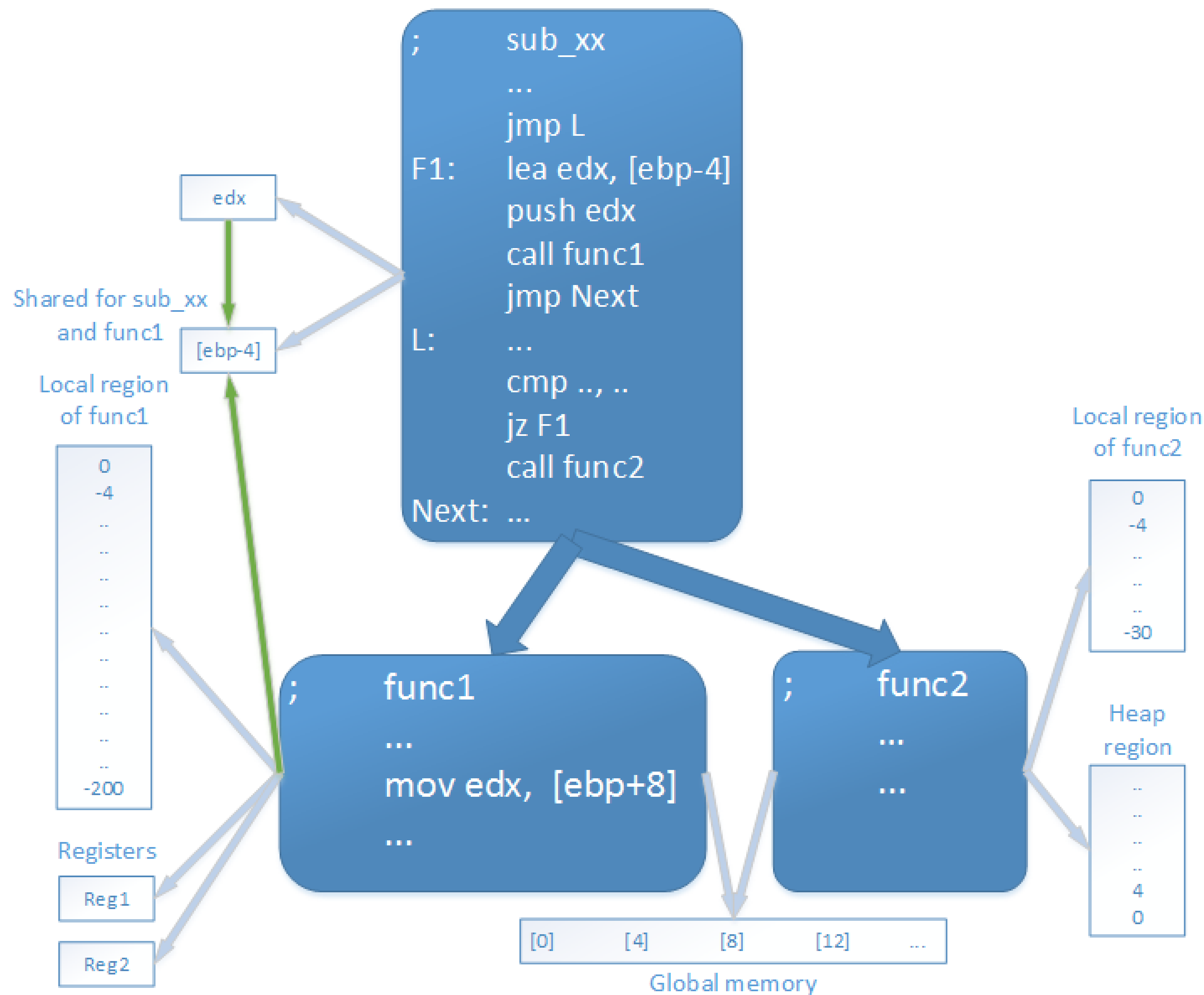
CodeSurfer

- Абстрактные ячейки вместо реальных адресов в памяти
- Подсчет наборов валидных значений в ячейках после каждой инструкции программы (Value-Set Analysis).
- Подбор возможных значений для косвенных переходов/вызовов функций.



Наш подход

- Изучаемый код предполагается однопоточным и соответствующим стандартной модели компиляции.
- Графовое представление кода и данных.
- Регионы памяти: глобальная память, регионы с данными в куче, изолированные локальные регионы на стеке (для каждой процедуры).



Наш подход

- Пользовательский ввод – символьные переменные.
- Условные переходы – система ограничений на символьные переменные.
- Абстрактные переменные, SSA-представление.
- “Слои” кода внутри функций с равным esp.
- Развертка циклов со статически вычисляемыми границами.
- Развёртка циклов с символьными границами “в обратном порядке”, если в теле цикла есть чувствительные операции, зависящие от величины границы (косвенная адресация, выделение/освобождение памяти, и.т.д.).
- Обратная трассировка по графу кода/данных.

Code-Data Graph

- *Бинарный код как один большой граф* – присутствует код, абстрактные переменные, задействованные регионы памяти.
- Абстрактным переменным присваиваем значение только 1 раз – много версий одной и той же ячейки памяти.
- Несколько типов ребер:

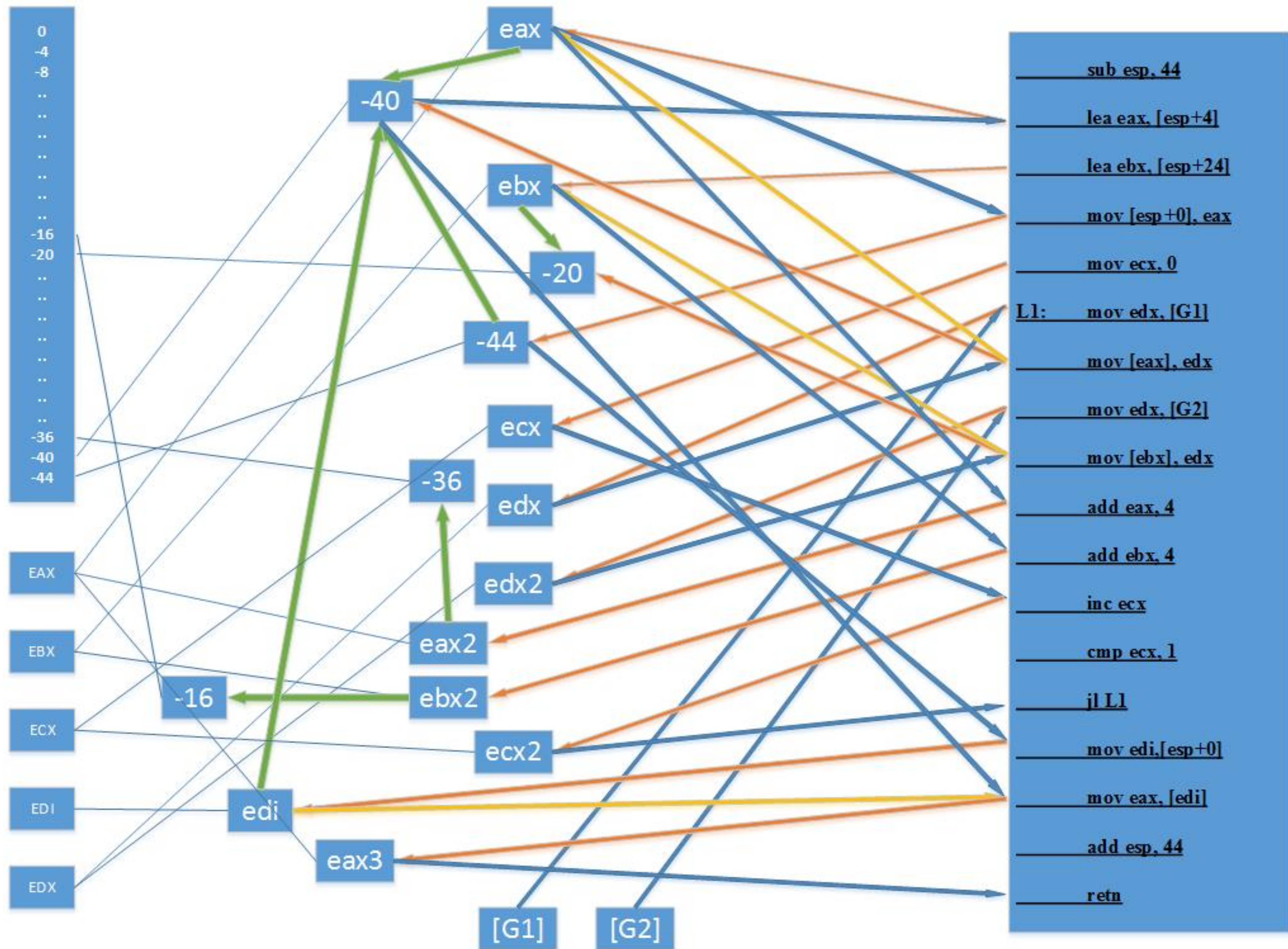
- от переменной-источника к соответствующей машинной инструкции, от той же инструкции – к переменной-приемнику (тип A1 и A2).

- от переменной, хранящей эффективный адрес - к переменной, связанной с этим адресом (тип Б).

- между инструкцией с разыменованием адреса, лежащего в переменной, и соответствующей переменной (тип В).

- между переменными и соответствующими ячейками памяти (тип Д).

- Типы А1, А2 – красные и синие ребра. Связь между переменными.
- Тип Б - зеленые ребра. Работа с указателями.
- Тип В – желтые ребра. Разыменования адресов.
- Тип Д – тонкие ребра. Для косвенной адресации



Обратная трассировка

- *Потенциально уязвимые операции* – вычисление параметров для malloc, косвенное разыменование памяти, и т. д., с участием символьных переменных.
- Накладываются заведомо приводящие к уязвимой ситуации ограничения на переменные, участвующие в таких операциях. От таких переменных осуществляется *движение в обратную сторону* по ребрам в построенном графе.
- При движении осуществляется *символьные вычисления систем ограничений*.
- Движение останавливается либо при достижении *противоречия* системы ограничений, либо при получении *нетривиального решения такой системы* в источниках символьных переменных (это сигнализирует об уязвимости).

Типы уязвимостей

- *Разыменование памяти возле нулевого адреса*: накладывается нулевое ограничение *на переменную-базовый адрес* при косвенной адресации.
- *Переполнение стека/кучи*: ограничение накладывается *на переменную-смещение* при косвенной адресации. При необходимости осуществляется обратная раскрутка цикла.
- *Целочисленное переполнение*: накладывается ограничение *на переменную*, формирующую *размер* буфера перед *выделением памяти*.

Инструменты реализации

- IDA Pro: базовое промежуточное представление – сам машинный код, локальные переменные на стеке, идентификация библиотечных процедур.
- Построение и анализ графа: Parallel Boost Graph Library[4]
- Символьные вычисления: GiNaC[5]

Спасибо за внимание!

Ссылки:

[1] - *The Evarista program transformer*, *http://www.eresi-project.org/wiki/Evarista*

[2] - *IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution*, Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou

[3] - *Analyzing Memory Accesses in x86 Executables,* Gogul Balakrishnan and Thomas Reps, Comp. Sci. Dept., University of Wisconsin

[4] - *Parallel Boost Graph Library, http://www.boost.org/doc/libs/master/libs/graph_parallel/doc/html/index.html*

[5] – *GiNaC Library*, http://www.ginac.de/