ОБ АНАЛИЗЕ ТРАСС ПРИ ТЕСТИРОВАНИИ ПРОГРАММ МЕТОДОМ ФАЗИНГА

Макаров А.Н.
УМО ИБ

* Переход от порождающего и мутационного фаззинга к «умному» требует решения следующей задачи:
  ГЕНЕРАЦИЯ ВХОДНЫХ ДАННЫХ, КОТОРЫЕ ПРИВОДЯТ К ВЫПОЛНЕНИЮ ВЕТВИ КОДА, НЕ ВЫПОЛНЯВШЕЙСЯ РАНЕЕ

* Генерация входных данных основывается на предыдущих итерациях тестирования, таким образом обеспечивается обратная связь между процессом тестирования (фаззинга) и процессом генерации входных данных

* **Необходимая обратная связь, может быть обеспечена:**
  * Вычислением покрытия кода
  * Получением трассы выполнения программы
  * Символьными вычислениями

Существующие решения

* **Трасса является исходным материалом для различных техник анализа программ:**
  * Построение программных срезов (*slicing*)
  * Тейнтирование (*tainting*) данных, которое, как правило, предваряет символьные вычисления
  * Выполнение по трассе символьных вычислений, например:
    * Проекты BitBlaze, *S²E* (трасса –> *LLVM* –> *KLEE*)
  * «Проигрывание» трассы, пример: *PinPlay*
  * Построение формул для решателей (*solver*) в форматах *SMT-LIB* или *STP*, например:
    * Проект *fuzzgrind* / *pathgrind*

Как получить трассу?

* **Используя средства динамического бинарного анализа (*DBI*)**
  * Утилиты (инструменты) для *PinTools, DynamoRIO, Valgrind*
* **На основе виртуальных машин: *QEMU, XEN***
  * Проекты: *TEMU* из *BitBalze, Ether*
* Кроме того, возможность **детерминированного воспроизведения вычислений на заданных данных** или **возможность «проиграть» трассу** помогает в анализ ПО

Как получить трассу?

Virtual Machine	*DBI PinTools, DynamoRIO*	*Valgrind*
Возможность инструментирования всего исполняемого кода, включая ядро ОС	Высокая производительность из-за низких накладных расходов	Работает не с машинным кодом, а с промежуточным представлением *VEX IR*
Возможность работы с промежуточным представлением	Не требует исходных кодов	Работает только в *Linux* и *Mac OS*, но поддерживает *PowerPC*
Для инструментирования даже небольшой программы требуется виртуальная ОС	Работают под ОС *Windows, Linux, Mac OS*, но только с процессорами *IA-32, IA-64, Intel-64*	Исходные коды желательны при использовании некоторых инструментов
Для поддержки гостевых ОС могут потребоваться дополнительные драйвера	Работа с машинным кодом	

Скорость вычисления трассы

* **В процессе тестирования (фаззинга) требуется миллионы запусков, таким образом:**
  * невозможно тратить много времени на получение трассы
  * невозможно для каждого запуска обеспечить сбор полной информации о трассе
* **Возможны следующие варианты:**
  * получение трассы для входных данных, если число новых блоков (не покрытых ранее) больше некоторого значения;
  * получение трассы только для входных данных, которые инициируют интересующее нас событие.

Анализ трассы выполнения

* **Целесообразно представлять трассу в промежуточном коде**
  * *VEX IR*: используется в проектах с *Valgrind*
  * *LLVM* байт-код: проекты *S²E, libcpu*

* **Идеально, если трасса включает код, выполненный в ядре ОС но:**
  * производительность уменьшается
  * кроме того, анализ приложений пользовательского режима может существенно осложнится из-за асинхронных вызовов в ядре и различных механизмов *IPC*

* **Полная трасса не всегда нужна, возможно отказаться:**
  * от трасс системных потоков ядра ОС
  * от участка трасс «хорошо-известных» функции, например *ReadFile*

Анализ трассы выполнения

Число инструкций в трассе	Число блоков в трассе	Число различных инструкций	Число различных блоков
≈ 800 млн	≈ 120 млн	≈ 1,0 млн	≈ 250 тысяч
≈ 3000 млн	≈ 300 млн	≈ 1,5 млн	≈ 350 тысяч

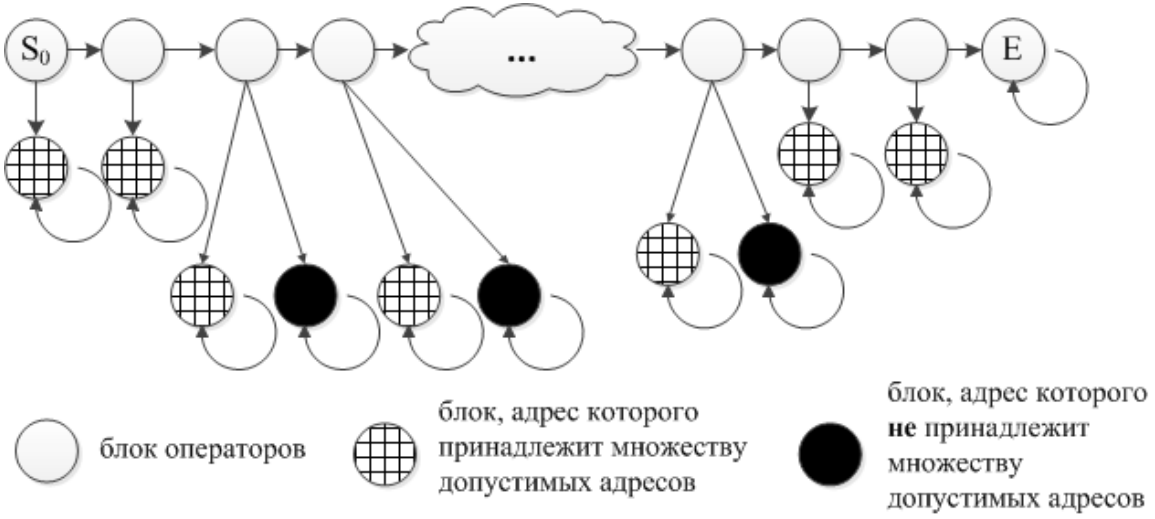
* Работа с трассой значительно упрощается, если оперировать не отдельными инструкциями, а блоками инструкций
* Каждый блок рассматривается как «неделимый» элемент

Ограничения при работе с трассой блоков

* Анализ многопоточных приложений не рассматривается
* Для приложений, обрабатывающих сложно-структурированные файлы, данное ограничение допустимо
* Практически все средства фаззинга ориентированы на поиск ошибок кодирования и не предназначены для поиска логических ошибок

Поиск логических ошибок

* Трассу, можно представить как частный случай программы *T* для конкретных входных данных
* Для программы *T* (трассы) на основе «неявных» алгоритмов строится структура Крипке
* Требования, которые должны выполняться для *T*, задаются в виде спецификаций, представленных формулами логики *CTL*



Например, пусть *p* — предикат, который истинен

$s \models \mathbf{AG}(\neg p)$ - из состояния *s* для любого пути нельзя попасть в состояние, для которого предикат *p* истинен

Спасибо за внимание