Методы поиска ошибок в бинарном коде *

Технический отчет 2013-1

В.В. Каушан, Ю.В. Маркин, В.А. Падарян, А.Ю. Тихонов

Отчет представляет собой обзор современных методов и инструментов поиска ошибок в бинарном коде программ. Рассматриваются различные решения, основанные на динамическом и статическом подходе, комбинирующие их. Отдельно рассматриваются методы, использующие символьную интерпретацию, а так же фаззинг. Приводятся оценки сложности повторения показанных в рассмотренных работах результатов.

Содержание

* Работа поддержана грантом РФФИ 12-01-31417 и грантом Президента РФ МК-1281.2012.9

1 Уровни проведения анализа

Различают два основных уровня, на которых происходит поиск ошибок и уязвимостей в программах: анализ исходного и бинарного (исполняемого) кода.

Каждый уровень обладает своими преимуществами и недостатками. Принципиальным ограничением является то, что все промышленные языки

программирования предполагают статичность кода. Методы анализа бинарного кода, в определенных ограничениях, применимы в случаях самомодификации кода.

Рассмотрим типовые проблемы, возникающие при анализе исходного кода.

Эффект использования компиляторов и других средств автоматического преобразования кода, обозначаемый в англоязычной литературе как WYSINWYX [1] (What You See Is Not What You eXecute). Ошибки и уязвимости возникают вследствие того, что разработчики ПО не могут учесть всех особенностей исполняемого кода программы, возникающих из-за использования оптимизирующих компиляторов и бинарных трансляторов.

Пример 1.

```
memset(password, '\0', len);
free(password);
```

Перед тем, как освободить динамически выделенный буфер памяти (в котором хранится конфиденциальная информация), его содержимое заполняется нулями. Оптимизирующий компилятор может решить, что значения, записанные в буфер в результате выполнения функции memset, нигде далее не используются и удалит соответствующий вызов. В результате в куче на протяжении какого-то времени будет храниться конфиденциальная информация, что является потенциальной уязвимостью.

При анализе исходного кода, как правило, делается (ничем не обоснованное) предположение о том, что анализируемый код соответствует стандарту, в случае языка Си таким стандартам, как ANSI C, ANSI X3.159-1989, ISO/IEC 9899. Анализатор не принимает в расчет то, как именно языковые конструкции отображаются на уровень машинных команд [2].

Пример 2.

```
int a[2], b;
memset(a, 0, 12);
```

При выполнении данного фрагмента кода на 32-разрядной машине разработчик рассчитывал на заполнение нулями переменных a и b, тогда как анализатор фиксирует ошибку переполнения буфера a. Следует отметить, что расположение переменных a и b в памяти может быть любым, как приводящим к реализации ошибки во время выполнения, так и не приводящим.

Анализ исходного кода сталкивается с проблемой языковых расширений и отклонений от стандарта, присутствующих в промышленных компиляторах, например механизм предварительной компиляции заголовочных файлов в среде MS VS. Механизм ассемблерных вставок стандартом языка Си не

оговорен, когда таковые встречаются при анализе кода, анализатор, как правило, их пропускает.

Порядок вычисления операндов у некоторых операций Си/Си++ не определен, что приводит к неопределенному поведению программы, в особенности, если вычисление операнда сопровождается побочными эффектами. При анализе исполняемого файла подобных неопределенностей не возникает.

Поведение многопоточных приложений в общем случае является неопределенным, по причине того, что поведение некоторых механизмов управления потоками выполнения существенно зависит от реализации. Анализ исходного кода дополнительной информации внести не способен, поскольку особенности данных механизмов выходят за пределы спецификаций языка программирования и используемых библиотек. В то же время, анализ бинарного кода позволяет точно определить, как именно эти механизмы были реализованы.

Пример 3

```
int (*f)(void);
int diff = (char*)&f2 - (char*)&f1; // The offset
between f1 and f2
f = &f1;
f = (int (*)())((char*)f + diff); // f now points to f2
(*f)(); // indirect call;
```

Анализ бинарного кода способен корректно обрабатывать арифметические операции над указателями, тогда как анализ исходного кода для решения данной задачи не подходит. Как правило, анализаторы исходного кода полагают, что либо потенциально вызывается любая функция, либо, игнорируют арифметические операции, считая, что вызывается функция f1.

Для работы программе могут потребоваться библиотеки, исходный код которых недоступен. В этом случае создаются заглушки, моделирующие взаимодействие с библиотеками. Поскольку заглушки создаются разработчиком, они могут содержать ошибки, которых не содержат библиотеки, используемые программой.

Последней проблемой, которую следует упомянуть, является необходимость интеграции со средой сборки, поскольку она определяет, какие именно файлы, и в каком окружении, будут компилироваться и в итоге попадут в программу.

Говоря о преимуществах анализа исходного кода, следует отметить следующее.

1. Возможность выявления ошибок на этапе кодирования, что позволяет существенно снизить стоимость всей разработки проекта. На данный момент существует несколько промышленных инструментов, рассчитанных на интеграцию в ночные сборки [2, 3, 4]. Время работы этих инструментов на значительных по объему кода проектах,

порядка нескольких миллионов строк кода, не превышает 12 часов, что позволяет использовать их на регулярной основе.

2. Исходный код подвергается, как правило, статическому анализу, что дает полное покрытие кода. Статические анализаторы проверяют даже те фрагменты кода, которые получают управление крайне редко. Такие участки кода крайне затруднительно подвергнуть динамическому анализу. Это позволяет находить дефекты, например, в обработчиках ошибок или в системе ведения журнала.
3. Наличие высокоуровневой семантики, в первую очередь информации о типах переменных, что позволяет оценивать размеры используемых буферов памяти.

Кратко перечислим основные недостатки анализа исполняемых файлов.

1. Высокая сложность. Количество инструкций весьма велико, в случае CISC-архитектуры отдельные инструкции обладают сложной семантикой.
2. Отсутствие высокоуровневой семантики.
3. Исполняемый код может быть защищен от анализа посредством запутывания и антиотладочных приемов.

Сравнение рассмотренных выше особенностей позволяет заключить следующее. Анализ исходного кода использует высокоуровневую информацию о программе, часть которой, при переходе на уровень исполняемого кода, будет безвозвратно утрачена. На уровне бинарного кода операционная семантика программы представлена в полном объеме, но уровень представления этой семантики менее удобен для автоматизированного поиска ошибок. Зачастую, причины, вызывающие затруднения в анализе на одном уровне, дают другому уровню анализа ощутимые преимущества.

1.1 Типы ошибок, обнаруживаемых в исходном коде

Рассмотрим некоторые ошибки в исходном коде, которые могут быть обнаружены средствами статического анализа.

Пример 1.

```
int x;
int y = x + 2;
```

Использование неинициализированных переменных является ошибкой. В простейшем случае подобные ошибки обнаруживаются компилятором.

Пример 2.

```
int *ptr;
*ptr = 10; // разыменование нулевого указателя
```

Ошибкой является разыменование нулевого указателя.

Пример 3.

```
void doSomething(const char* x)
{
  char s[40];
  sprintf(s, "[%s]", x); // возможно переполнение буфера
  ....
}
```

Ошибка, связанная с переполнением буфера, возникает из-за неправильной работы с данными, полученными извне, и памятью. В результате переполнения данные, расположенные следом за буфером (или перед ним), могут быть испорчены. Отметим, что переполнение буфера является наиболее популярным способом взлома компьютерных систем, так как большинство языков высокого уровня использую технологию стекового кадра – размещение данных в стеке процесса (в том числе, адрес начала стекового кадра и адрес возврата из исполняемой функции).

Ошибкой является нарушение «алгоритма» использования некоторых библиотечных функций. Например, при работе с файлами для каждого вызова функции fopen() нужен соответствующий вызов функции fclose().

Пример 4.

```
std::string s;
...
s.empty(); // код ничего не делает
```

Многие функции из стандартных библиотек не имеют «побочных эффектов», и вызывать их бессмысленно.

Пример 5.

```
std::wstring s;
printf("s is %s", s);
```

Ошибкой является несоответствие форматной строки реальному типу параметров.

Пример 6.

```
int i, destlen = 0, l, k;
for (i = 0; i < srclen; i++)
{
  ...
  for (k = i; i < srclen; k++)
  {
    if (src[k] == '>')
      break;
  }
  i = k;
}
```

В коде содержится опечатка – во вложенном цикле для сравнения используется переменная i, а не k. Это код с большой вероятностью приведет к обращению к памяти за пределами обрабатываемого массива.

Пример 7.

```
class CSize : public SIZE
{
  ...
  CSize(POINT pt) { cx = pt.x; cx = pt.y; }
  ...
}
```

Из-за опечатки одной и той же переменной последовательно присваиваются различные значения. Корректный вариант второго присваивания:

```
cy = pt.y;
```

Пример 8.

```
#define CONT_MAP_MAX 50
int _iContMap[CONT_MAP_MAX];
memset(_iContMap, -1, CONT_MAP_MAX);
```

В приведенном коде функция memset() очищает далеко не весь массив _iContMap. Корректным будет следующий вызов memset():

```
memset(_iContMap,        -1,        CONT_MAP_MAX        *
sizeof(_iContMap[0]));
```


Краткие выводы

Оптимальная ситуация, когда доступны как исходные, так и исполняемые коды программы, известен механизм сборки. Однако на практике, такая ситуация не всегда достижима.

Перечисленные сложности анализа исходного кода позволяют рассчитывать на то, что поиск ошибок на уровне бинарного кода будет давать результаты, не фиксируемые существующими промышленными инструментами статического анализа, поскольку все они работают с исходным кодом. Однако в области анализа бинарного кода на данный момент уже существует целый ряд подходов, поддержанных программными инструментами. Большинство работ находятся в стадии исследований, но некоторые подходы и инструменты уже доведены до промышленного уровня.

2 Подходы к анализу исполняемых файлов

Среди методов поиска ошибок в бинарном коде можно выделить следующие основные подходы:

1. статический подход,
2. динамический подход,
3. смешанный подход, совмещающий статический и динамический анализ,
4. символьное выполнение,
5. фаззинг.

На практике разрабатываемые решения, как правило, комбинируют в себе механизмы и приемы различных подходов, т.к. каждый из них обладает рядом ограничений. Согласованное применение различных подходов позволяет эти ограничения преодолевать полностью или частично.

Статический анализ – это анализ без реального выполнения исследуемой программы. В рамках статического подхода анализ, как правило, состоит из следующих этапов:

- построение промежуточного представления программы,
- построение графов (CFG, SDG, граф вызовов, …),
- проверка выполнимости наложенных аналитиком ограничений.

Динамический подход, напротив, предполагает анализ программ во время их выполнения. Утилиты динамического анализа могут требовать загрузки специальных библиотек, перекомпиляцию программного кода. Некоторые утилиты могут инструментировать исполняемый код в процессе выполнения или перед ним. Для большей эффективности динамического анализа требуется подача тестируемой программе достаточного количества входных данных для получения более полного покрытия кода. Также необходимо позаботиться о

минимизации воздействия внедренного инструментального кода на исполнение тестируемой программы.

Смешанный подход предполагает дополнение результатов статического анализа результатами динамического анализа (и наоборот). Например, CFG, построенный статическим анализатором, является неполным – неизвестными остаются значения операндов инструкций передачи управления с косвенной адресацией. Посредством динамического анализа соответствующие значения операндов могут быть найдены.

Символьное выполнение применяется для увеличения покрытия кода анализируемой программы. Основная идея состоит в том, чтобы заменить входные данные (а именно, конкретные значения) некоторой программы символами. Вместо конкретных значений программа будет обрабатывать символьные выражения.

Пусть результат выполнения некоторой инструкции условного перехода зависит от символьных данных. В этой точке программы происходит разделение траекторий выполнения и вместо одного множества значений символьных данных создается два – в первое добавляется ограничение на символьные значения, что бы условие перехода было истинно, во второе множество – наоборот, включают ограничение, описывающее ложность условия. Каждая точка программы на некоторой траектории выполнения характеризуется набором ограничений на значения символьных данных. Для определения достижимости интересующих точек программы необходимо разрешить систему ограничений.

Фаззинг - технология тестирования ПО, когда вместо ожидаемых входных данных программе передаются случайные или специально сформированные данные. Является одним из наиболее популярных средств выявления ошибок в коде.

Если говорить о способе манипуляции с входными данными, то фаззинг разделяется на генерацию и мутацию. Генерация – это создание входных данных случайным образом (входные данные либо не имеют никакой структуры, либо генерируются согласно заданным моделям). Мутация – это внесение изменений в существующие «образцы» входных данных.

3 Статический анализ

3.1 Система CodeSurfer/x86

Система CodeSurfer/x86 [5] предназначена для проведения статического анализа исполняемых файлов платформы x86 при отсутствии исходного кода. Она не является свободно распространяемым ПО, вследствие чего нет возможности непосредственно оценить работу системы. На рисунке 1 представлена общая схема работы.

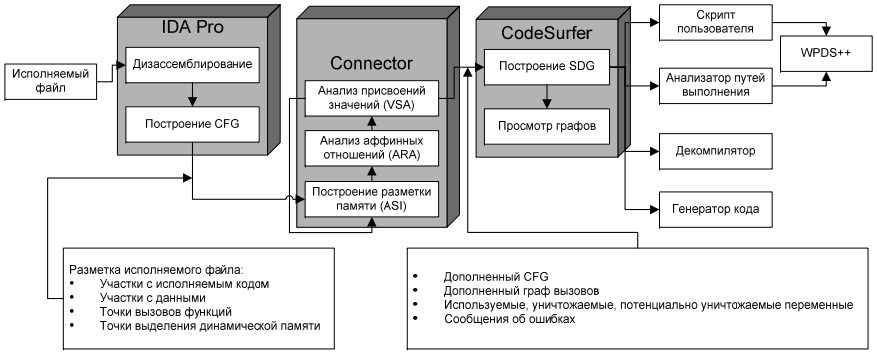

Рисунок 1 – Схема работы системы CodeSurfer/x86.

При помощи IDA Pro исполняемый файл дизассемблируется. IDA Pro предоставляет разметку кода, включающую границы процедур, вызовы библиотечных функций (используется алгоритм FLIRT), используемые области памяти (адреса и смещения), которые удается определить путем статического анализа. Строятся граф потока управления и граф вызовов. Графы являются неполными из-за наличия инструкций передачи управления с косвенной адресацией.

Адресное пространство анализируемой программы представляется посредством абстрактной модели памяти. Вводится понятие области памяти (memory region). Область памяти – это набор абстрактных локаций (a-locs, abstract locations), обладающих схожими во время выполнения анализируемой программы свойствами. Выделяют три вида областей:

- области с глобальными переменными (global regions)
- области со стековыми фреймами (AR-regions)
- области динамически выделяемой памяти (malloc-regions)

Далее используется модуль расширения IDA Pro Connector. Он пытается восстановить содержимое областей памяти и то, как эти области используются анализируемой программой. То есть осуществляется попытка восстановления структур данных. Connector проводит итерационный анализ, каждая итерация состоит из трех процедур:

- ASI (aggregate-structure identification) - идентификация составных типов данных,
- ARA (affine-relation analysis) - анализ аффинных отношений,
- VSA (value-set analysis) – анализ присвоений значений.

Для работы процедуры ASI необходимо сформулировать модель обращений к различным участкам памяти. Процедура VSA строит надмножество возможных значений всех регистров и абстрактных локаций памяти в каждой

точки анализируемой программы. Процедура ASI предназначена для уточнения (разметки) области памяти. Процедура ARA используется для восстановления зависимостей между регистрами и/или ячейками памяти в местах условных переходов. Подробнее работа процедур описана в работах [6], [7] и [8]. Полученная на данном этапе анализа информация уточняет граф потока управления и граф вызовов[1].

Далее полученные надмножества локаций памяти используются для построения в каждой точке программы наборов используемых, уничтожаемых и потенциально уничтожаемых локаций. Вся информация передается в CodeSurfer. Посредством CodeSurfer строится набор промежуточных представлений анализируемой программы. Этот набор включает синтаксическое дерево, граф потока управления, граф вызовов, граф зависимостей (SDG), результаты VSA, надмножества используемых, уничтожаемых и потенциально уничтожаемых локаций для каждой инструкции, разметка областей, содержащих глобальные переменные, стековые фреймы, динамически выделяемую память. В CodeSurfer имеется API и поддержка скриптового языка, посредством которых осуществляется доступ к этой информации. На рисунке 2 изображен граф зависимости (SDG) для следующей модельной программы:

```
void main(){
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = add(sum,i);
            i = add(i, 1);
      }
      printf("sum = %d\n", sum);
      printf("i = %d\n", i);
}
static int add(int a, int b){
      return (a + b);
}
```

Знаковая целочисленная переменная sum инициализируется нулем, знаковая целочисленная переменная i инициализируется единицей. Далее в цикле вычисляется сумма переменных sum и i, результат записывается в sum,

[1] Следует отметить, что VSA является базовым средством, она используется во многих работах, связанных с анализом бинарного кода, как данного научного коллектива, так и других.

значение переменной i увеличивается на единицу. По завершению цикла (i принимает значение 11) значения переменных sum и i распечатываются.

Стрелки с черными концами отображают зависимости по управлению, с белыми – зависимости по данным. Внутрипроцедурные зависимости изображаются сплошными линиями, межпроцедурные зависимости – прерывистыми.

Рассмотрим вершину с пометкой «while i < 11» (эта вершина соответствует проверке условия цикла). Необходимым условием для выполнения кода проверки является попадание управления в функцию main («черная» стрелка из «entry main» в «while i < 11»). Код проверки может быть выполнен больше одного раза, если условие i < 11 было истинным при первом выполнении («черная» стрелка из «while i < 11» в «while i < 11»). Необходимым условием для выполнения функции add является попадание управления в тело цикла («черные» стрелки из «while i < 11» в «call add»). Истинность условия цикла зависит от того, каким значением была проинициализирована переменная i («белая» стрелка из «i = 1» в «while i < 11»), а также от того, как изменялось значение этой переменной («белая» стрелка из «i = ret» в «while i < 11»).

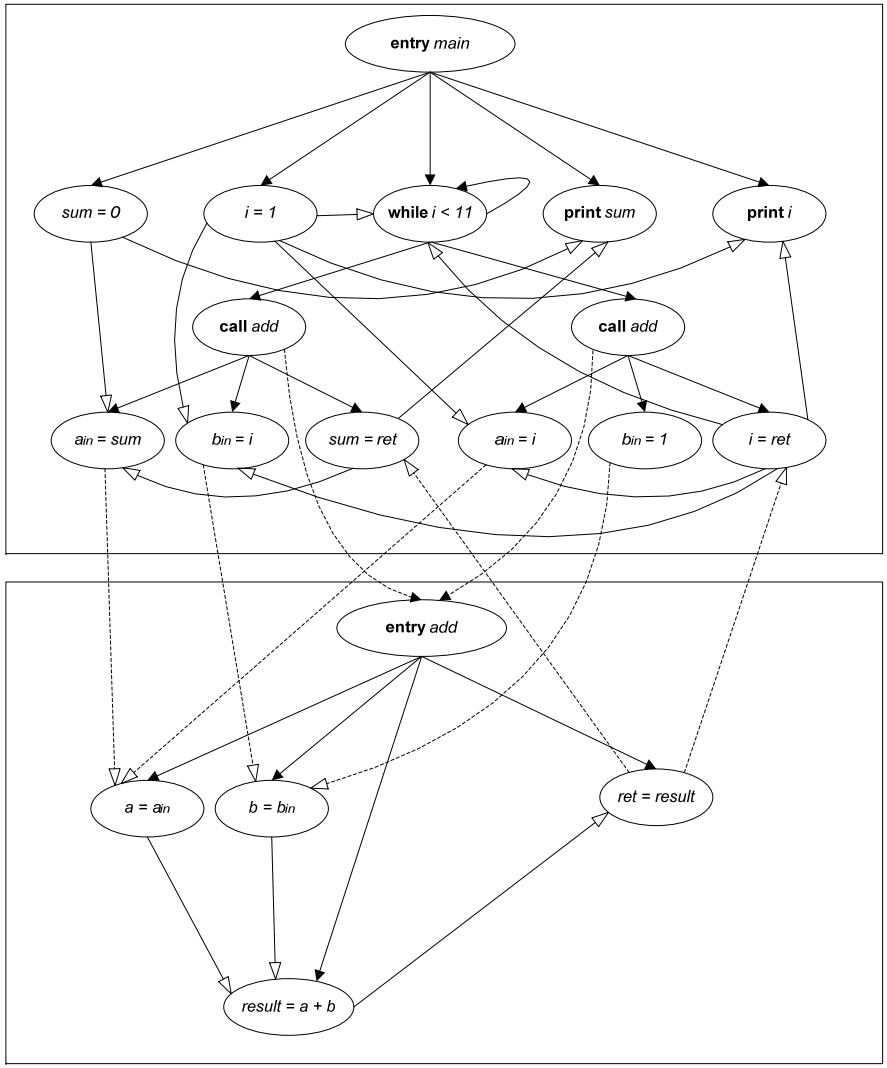

Рисунок 2 – SDG модельной программы.

SDG моделируется в терминах weighted pushdown system (WPDS, [9]), после чего могут быть использованы алгоритмы библиотеки WPDS++ [10]. Эти алгоритмы применяются для определения достижимости заданных (в терминах WPDS) пользователем конфигураций. Утилита Path Inspector получает на вход конфигурацию и WPDS представление для SDG и определяет, достижима ли заданная конфигурация.

3.1.1 Магазинный автомат

PDS (PushDown System, [11]) – это четверка $S = (\mathrm{P}, \Gamma, \Delta, \mathrm{c}_0)$, где P – конечное множество состояний, Γ – конечное множество символов стека, Δ – конечное множество правил: $\Delta \subseteq (\mathrm{P} \times \Gamma) \times (\mathrm{P} \times \Gamma^*)$. Конфигурация PDS – это пара $\langle p, \omega \rangle$, где $p \in \mathrm{P}$ и $\omega \in \Gamma^*$. Множество всех конфигураций обозначается $Conf(S)$. PDS характеризуется системой переходов $\tau_S = (Conf(S), \Rightarrow_S, c_0)$, где c_0 – начальная конфигурация. Каждое правило $((p,\gamma),(q,\omega)) \in \Delta$ можно записать в виде $\langle p,\gamma \rangle \rightarrow_S \langle q,\omega \rangle$. Система переходов PDS определяется следующим образом: если $\langle p,\gamma \rangle \rightarrow_S \langle q,\omega \rangle$, то $\langle p,\gamma x \rangle \Rightarrow_S \langle q,\omega x \rangle \ \forall x \in \Gamma^*$. Другими словами, при переходе может измениться состояние, а также верхний (то есть, самый левый) символ стека. В общем случае, верхний символ стека заменяется строкой символов (строка может быть пустой).

Построение PDS состоит из двух этапов. На первом этапе для каждой процедуры строится SDG. Далее приведен псевдокод модельной программы и SDG ее процедур (рисунок 3).

```
void main(){
      s();
}
void s(){
      if (?) return;
      up(); m(); down();
}
void m(){
      if (?) {
            s(); right();
            if (?) m();
      }
      else {
            up(); m(); down();
      }
}
```

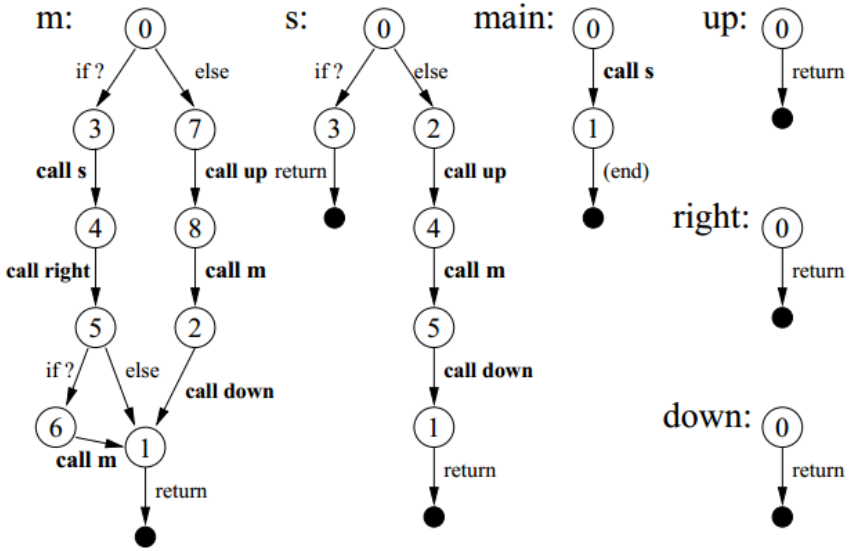

Рисунок 3 – SDG процедур модельной программы.

Данная программа предназначена для построения гистограмм.

На втором этапе построения происходит преобразование SDG в PDS. Пусть N – множество вершин в SDG всех процедур. Тогда искомая PDS – это $(\{.\}, N, \Delta, \langle ., main_0 \rangle)$. Множество состояний PDS содержит один элемент. Алфавит стека состоит из меток вершин SDG всех процедур. Правила перехода определяются следующим образом:

- если управление попадает из вершины *m* в вершину *n*, и при этом не происходит вызова какой-либо процедуры, необходимо добавить правило $\langle \cdot, m \rangle \to \langle \cdot, n \rangle$,
- если управление попадает из вершины *m* в вершину *n*, и при этом происходит вызов процедуры *f*, необходимо добавить правило $\langle \cdot, m \rangle \to \langle \cdot, f_0 n \rangle$ (здесь f_0 – точка входа f, n может рассматриваться как адрес возврата),
- если ребро представляет оператор возврата, необходимо добавить правило $\langle \cdot, m \rangle \to \langle \cdot, \varepsilon \rangle$.

Таким образом, конфигурация $\langle \cdot, n\omega \rangle$ соответствует тому, что управление находится в вершине *n*, а ω – адрес возврата вызывающей процедуры. Приведем полный набор правил для модельной программы.

$$\langle \cdot, m_0 \rangle \to \langle \cdot, m_3 \rangle \qquad \langle \cdot, m_0 \rangle \to \langle \cdot, m_7 \rangle$$
$$\langle \cdot, m_3 \rangle \to \langle \cdot, s_0 m_4 \rangle \qquad \langle \cdot, m_4 \rangle \to \langle \cdot, right_0 m_5 \rangle$$
$$\langle \cdot, m_5 \rangle \to \langle \cdot, m_1 \rangle \qquad \langle \cdot, m_5 \rangle \to \langle \cdot, m_6 \rangle$$
$$\langle \cdot, m_6 \rangle \to \langle \cdot, m_0 m_1 \rangle \qquad \langle \cdot, m_7 \rangle \to \langle \cdot, up_0 m_8 \rangle$$
$$\langle \cdot, m_8 \rangle \to \langle \cdot, m_0 m_2 \rangle \qquad \langle \cdot, m_2 \rangle \to \langle \cdot, down_0 m_1 \rangle$$
$$\langle \cdot, m_1 \rangle \to \langle \cdot, \varepsilon \rangle$$

$$\langle \cdot, s_0 \rangle \to \langle \cdot, s_2 \rangle \qquad \langle \cdot, s_0 \rangle \to \langle \cdot, s_3 \rangle$$
$$\langle \cdot, s_2 \rangle \to \langle \cdot, up_0 s_4 \rangle \qquad \langle \cdot, s_3 \rangle \to \langle \cdot, \varepsilon \rangle$$
$$\langle \cdot, s_4 \rangle \to \langle \cdot, m_0 s_5 \rangle \qquad \langle \cdot, s_5 \rangle \to \langle \cdot, down_0 s_1 \rangle$$

$$\langle \cdot, main_0 \rangle \to \langle \cdot, s_0 main_1 \rangle \qquad \langle \cdot, main_1 \rangle \to \langle \cdot, \varepsilon \rangle$$

$$\langle \cdot, up_0 \rangle \to \langle \cdot, \varepsilon \rangle$$
$$\langle \cdot, down_0 \rangle \to \langle \cdot, \varepsilon \rangle$$
$$\langle \cdot, right_0 \rangle \to \langle \cdot, \varepsilon \rangle$$

В модельной программе нет переменных, поэтому невозможно вычислить условия переходов. В результате, построенная система является недетерминированной. PDS расширяется добавлением информации о глобальных и локальных переменных. При этом состояние системы превращается в кортеж (он содержит значения глобальных переменных). Алфавит стека теперь содержит пары вида (n, l), где *n* – метка вершины SDG (как и прежде), *l* – кортеж, содержащий значения локальных переменных той процедуры, которой принадлежит вершина с меткой *n*.

Пусть *G* – область определения глобальных переменных (то есть, прямое произведение множеств значений глобальных переменных). Пусть в

программе m процедур и L_i – область определения локальных переменных процедуры i, $1 \le i \le m$. Пусть L – такое множество, что $|L| \ge |L_i|$, $1 \le i \le m$, и все значения из множества L_i, $1 \le i \le m$ могут быть представлены посредством элементов множества L. Расширенная PDS – это $\left(G,\ N \times L,\ \Delta',\ \left\langle g_0, (main_0, l_0)\right\rangle\right)$.

Глобальные переменные инициализируются набором значений g_0, локальные переменные – набором значений l_0. Изменяется процедура построения правил перехода:

- если ребро $n_1 \to n_2$ не является вызовом функции или оператором возврата, оно представляется в виде правила $\left\langle g, (n_1, l)\right\rangle \to \left\langle g', (n_2, l')\right\rangle$.
- если ребро $n_1 \to n_2$ представляет собой вызов функции m (с точкой входа m_0), оно представляется в виде правила $\left\langle g, (n_1, l)\right\rangle \to \left\langle g, (m_0, l')(n_2, l)\right\rangle$. Глобальные переменные не изменяются. l' – кортеж начальных значений локальных переменных процедуры m (а также значений аргументов функции). Локальные переменные вызывающей процедуры сохраняются на стеке вместе с адресом возврата n_2.
- оператор возврата представляется в виде правила $\left\langle g, (n, l)\right\rangle \to \left\langle g', \varepsilon\right\rangle$. Для того, чтобы учесть возвращаемое процедурой значение, необходимо соответствующим образом изменить глобальные переменные (возможно, ввести новые).

3.1.2 Взвешенный магазинный автомат

WPDS (PushDown System) – это PDS, в которой каждому переходу сопоставлен элемент из некоторого множества – вес перехода. В качестве множества весов используется ограниченное идемпотентное полукольцо. По определению, ограниченное идемпотентное кольцо – это пятерка $S = (D, \oplus, \otimes, 0, 1)$, где D – некоторое множество, содержащее 0 и 1, а операции $\oplus$ и $\otimes$ удовлетворяют следующим свойствам:

1. $(D, \oplus)$ – коммутативный моноид (с нулевым элементом 0), причем операция $\oplus$ идемпотентна ($\forall a \in D \to a \oplus a = a$),
2. $(D, \otimes)$ – моноид с единичным элементом 1,
3. Операция $\otimes$ дистрибутивна по отношению к операции $\oplus$, то есть $a \otimes (b \oplus c) = (a \otimes b) \oplus (b \otimes c)$ и $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$,
4. $\forall a \in D \to a \otimes 0 = 0 = 0 \otimes a$,
5. В отношении частичного порядка $\subseteq$ ($\forall a, b \in D \to a \subseteq b \ iff \ a \oplus b = a$) на множестве D не существует бесконечно убывающих последовательностей.

WPDS – это тройка $W = (Q, S, F)$, где $Q = (P, \Gamma, \Delta)$ – PDS, $S = (D, \oplus, \otimes, 0, 1)$ – ограниченное идемпотентное полукольцо, $f : \Delta \to D$ – отображение множества правил перехода на множество элементов полукольца.

Пусть $\sigma \in \Delta^*$ – последовательность правил $r_1, r_2, ..., r_k$. Определим $v(\sigma) =^{def} f(r_1) \otimes ... \otimes f(r_k)$. Более того, для двух конфигураций c и c' системы Q определим $path(c, c')$ как множество последовательностей правил $[r_1, ..., r_k]$, преобразующих конфигурацию c в c', то есть $c \Rightarrow^{<r_1>} ... \Rightarrow^{<r_k>} c'$.

Сформулируем общую задачу достижимости (generalized pushdown reachability problem). Пусть заданы WPDS $W = (Q, S, F)$, где $Q = (P, \Gamma, \Delta)$, и регулярное множество $C \subseteq P \times \Gamma^*$. Необходимо для каждой конфигурации $c \in P \times \Gamma^*$ найти:

- $\delta(c) := \oplus \{ v(\sigma) \mid \sigma \in path(c, c'), c' \in C \}$,
- «подтверждающее» множество $\omega(c) \subseteq \bigcup_{c' \in C} path(c, c') : \oplus_{\sigma \in \omega(c)} v(\sigma) = \delta(c)$.

Краткие выводы

Алгоритмы достижимости библиотеки WPDS++ не рассматривались. Не до конца понятно, как работает процедура VSA и сопутствующие алгоритмы (ASI, ARA).

3.2 Инструмент PREfix

В статье [12] описан метод поиска ошибок путем анализа исходного кода программ, написанных на языках Си/Си++. Предметом анализа является отдельно взятая функция. Для каждой анализируемой функции строятся граф вызовов, модель и набор ограничений. Процедура анализа состоит в проверке ограничений для допустимых траекторий выполнения. Обход графа вызовов анализируемой функции начинается в листовых вершинах и заканчивается в корневой вершине (то есть, сначала анализируются вложенные вызовы). Анализируются только те траектории выполнения, которые могут реализоваться на практике (а не все траектории CFG). Максимальное число анализируемых траекторий является параметром. При анализе каждой траектории выполнения отслеживаются изменения, происходящие с используемой памятью. После того, как траектории проанализированы по отдельности, результаты анализа обобщаются, и формируется модель. Приводится алгоритм построения модели:

```
пока (не все траектории выполнения проанализированы)
{
      выбор траектории для анализа;
      инициализация состояния памяти;
      анализ траектории, выявление несоответствий,
      внесение соответствующих изменений в состояние
      памяти;
      анализ траектории с учетом результирующего
      состояния памяти;
}
объединение информации, полученной при анализе отдельных
траекторий
```

Анализируется память, используемая в функции. С каждой используемой областью памяти связывают ее статус (значение каждого байта области известно; известно, что область проинициализирована; известно, что область не проинициализирована) и набор предикатов.

Выбор анализируемой траектории в точке ветвления (для `if` или `switch`) осуществляется случайным образом. Для выбора траектории вызываемой функции используются эвристики (вызываемая функция к этому моменту уже

проанализирована, поскольку при построении модели граф вызовов обходится в направлении от листовых вершин к корню).

Модель функции включает:

- набор *исходов* (изменений состояния памяти по сравнению с состоянием на входе)
- набор *внешних* объектов (внешними по отношению к данной функции объектами, например, являются ее параметры, возвращаемое значение, глобальные и статические переменные (при условии, что они используются внутри функции))

Каждый *исход*, в свою очередь, состоит из *операций*. Под *операцией* в данном случае понимается оператор и набор операндов. Операндом может быть *внешний* объект, константа, временная переменная, созданная при моделировании функции, а затем удаленная. *Операции* делятся на *проверки*, *ограничения* и *результаты*.

- *Ограничение* – это предикат, который должен быть истинным в момент вызова функции (предусловие).
- *Результат* – это постусловие.
- *Проверка* – это тест, доказывающий (или опровергающий) тот факт, что конкретный исход зависит от входных параметров функции. По сути, это попытка сопоставить исходы и значения входных данных.

В качестве примера, рассматривается функция `fopen` стандартной библиотеки языка С:

```
FILE* fopen (const char *filename, const char *mode);
```

Простая модель данной функции содержит два исхода: положительный (возвращается корректный указатель на `FILE`) и отрицательный (возвращается значение `NULL`). Внешними объектами являются входные параметры и области памяти, на которые указывают входные параметры, возвращаемое значение, глобальная переменная `errno`. Ограничения: указатели `filename` и `mode` должны быть корректными; соответствующие области памяти должны быть проинициализированы. Результаты: один для положительного исхода – возвращается корректный указатель на `FILE`, два для отрицательного исхода – возвращается указатель на `NULL` и глобальная переменная `errno` принимает некоторое положительное значение.

Рассматривается еще один пример:

```
int deref(int *p)
{
      if (p == NULL)
            return NULL;
      return *p;
}
```

Внешние объекты: входной параметр `p` (указатель на `int`); область памяти, на которую указывает `p`; возвращаемое значение. Два исхода:

- Исход #1:
- Проверка: указатель `p` равен `NULL`
- Ограничение: указатель `p` должен быть проинициализирован
- Результат: возвращаемое значение равно `NULL`
- Исход #2:
- Проверка: указатель `p` не равен `NULL`
- Ограничение: указатель `p` должен быть проинициализирован
- Ограничение: указатель `p` должен быть корректным
- Ограничение: область памяти, адресуемая указателем `p`, должна быть проинициализирована
- Результат: возвращаемое значение равно значению, хранящемуся в области памяти, адресуемой указателем `p`

Далее приводится модель функции *deref*:

```
(deref
            (param p)
      (alternate return_0
            (guard peq p NULL)
            (constraint memory_initialized p)
            (result peq return NULL)
      )
      (alternate return_X
            (guard pne p NULL)
            (constraint memory_initialized p)
            (constraint memory_valid_pointer p)
            (constraint memory_initialized *p)
            (result peq return *p)
      )
)
```

После того, как модель функции построена, начинается анализ. Выполняются проверки (проверка может быть истинной, ложной, или об ее истинности

ничего сказать нельзя). Далее анализируются только допустимые исходы (исходы, проверки которых не противоречат друг другу). Для каждого допустимого исхода выполняется проверка всех ограничений.

Авторы публикации отмечают, что с помощью разработанного языка описания моделей нельзя сформулировать ряд важных ограничений. В качестве примера рассматривается функция `strcpy` и ее модель.

```
/* char *strcpy(char *s1, const char *s2); */
(strcpy
            (param s1 s2)
      (normal
            (constraint memory_valid_pointer s1)
            (constraint memory_valid_pointer s2)
            (constraint memory_initialized *s2)
            (result memory_initialized *s1)
            (result peq return s1)
      )
)
```

Не могут быть описаны следующие характеристики:

1. Копируемая строка должна заканчиваться нулем (‘\0’),
2. Строка, в которую осуществляется копирование, должна заканчиваться нулем (‘\0’),
3. Длины строк должны совпадать,
4. Области памяти, выделенные под строки, не должны пересекаться,
5. Сумма размеров строк не должна превышать размер доступной памяти.

Детальное описание конструкций языка представлено в статье.

Краткие выводы

Авторы отмечают, что при анализе функции отслеживаются изменения, происходящие с используемой областью памяти. Понятие используемой области не формализовано. Непонятно, как анализируются вызовы функций по указателям.

4 Динамический анализ

Инструменты динамического анализа обнаруживают программные ошибки в коде, запущенном на исполнение. При этом аналитик имеет возможность наблюдать или диагностировать поведение приложения во время его исполнения, в идеальном случае – непосредственно в целевой среде.

Во многих случаях в инструменте динамического анализа производится модификация исходного или бинарного кода приложения с целью установления процедур-перехватчиков для проведения инструментальных измерений. С помощью таких «ловушек» можно обнаружить программные ошибки на этапе выполнения, проанализировать использование памяти, покрытие кода и проверить другие условия. Инструменты динамического анализа могут генерировать точную информацию о состоянии стека, что позволяет отладчикам отыскать причину ошибки. Поэтому, когда инструменты динамического анализа находят ошибку, то, скорее всего, это настоящая ошибка, которую программист может быстро идентифицировать и исправить. Следует заметить, что для создания ошибочной ситуации на этапе выполнения должны существовать точно необходимые условия, при которых проявляется программная ошибка. Соответственно, разработчики должны создать некоторый контрольный пример для реализации конкретного сценария.

4.1 Среда анализа Valgrind

Valgrind [13], [14] – это система динамического профилирования с последующим анализом для платформы x86/Linux. Является свободно распространяемым ПО.

Анализ проводится по следующей схеме. Анализируемая программа не выполняется непосредственно. Вместо этого, она динамически транслируется в промежуточное представление Valgrind VEX (далее IR). Промежуточное представление не зависит от целевой платформы и представлено в SSA-виде. Разработанные на базе Valgrind инструменты выполняют необходимые преобразования над IR, после чего Valgrind транслирует IR обратно в машинный код. Концептуальная схема работы Valgrind представлена на рисунке 4.

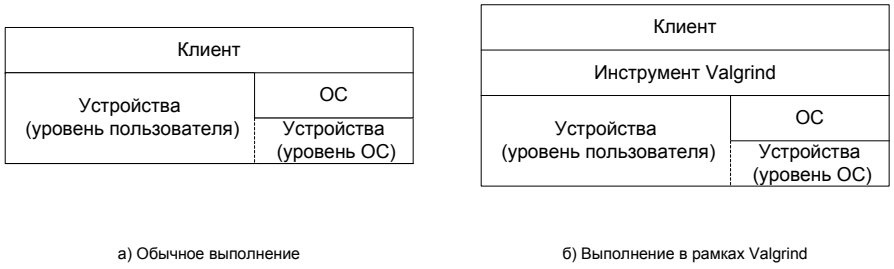

Рисунок 4 – Выполнение программы в рамках Valgrind.

Каждый исполняемый базовый блок транслируется в IR. Трансляция разбивается на пять этапов.

1. Исполняемый код представляется в виде двухадресных инструкций Valgrind. При этом используются виртуальные регистры.
2. Оптимизация IR.
3. Внесение инструментального кода. Инструменты, выбранные для анализа, добавляют необходимый код в промежуточное представление программы.
4. Распределение регистров. Виртуальные регистры, используемые в коде IR, отображаются на 6 аппаратных регистров: eax, ebx, ecx, edx, esi, edi. Регистры ebp и esp зарезервированы: на ebp хранится адрес текущего базового блока, на esp – адрес стека Valgrind.
5. Генерация кода. Каждая инструкция IR независимо от других конвертируется в одну или несколько инструкций целевой архитектуры. При этом счетчик инструкций должным образом модифицируется.

Отметим, что Valgrind поддерживает FP- и SIMD-инструкции.

Valgrind не может транслировать код ядра ОС, поэтому системные функции выполняются непосредственно, без внесения в код каких-либо изменений. Используется следующий алгоритм:

1. Сохранение стека анализируемой программы
2. Копирование значений из памяти на аппаратные регистры (за исключением счетчика команд)
3. Исполнение системной функции
4. Копирование значений аппаратных регистров после системного вызова в память (за исключением счетчика команд)
5. Восстановление стека анализируемой программы

Заметим, что Valgrind поддерживает обработку сигналов. Valgrind не позволяет анализировать самомодифицирующийся код.

Собственно анализ кода выполняется подключаемыми к Valgrind модулями-расширениями. В каждом модуле должны определяться как минимум четыре функции.

```
1. void SK_(pre_clo_init)(void)
2. void SK_(post_clo_init)(void)
```

Эти функции вызываются при обработке параметров командной строки, запускающей анализ с использованием данного модуля (первая выполняется до обработки, вторая – после). Посредством данных функций модуль расширения передает ядру Valgrind информацию о том, какой функционал ядра будет использоваться при анализе.

3. `UCodeBlock* SK_(instrument)(UCodeBlock *cb, Addr orig_addr)`

Эта функция вызывается при внесении инструментального кода в базовый блок. Входные параметры – адрес базового блока IR, адрес соответствующего базового блока исполняемого файла. Возвращаемое значение – адрес базового блока IR с внесенным инструментальным кодом. Код для проведения анализа может быть добавлен тремя способами. Первый способ – встроить анализирующий код непосредственно в промежуточное представление. Второй способ – добавить вызов ассемблерной подпрограммы, определенной в коде инструмента, посредством инструкции IR CALLM. Третий способ – добавить вызов СИ-функции, определенной в коде инструмента, посредством инструкции IR CCALL.

4. `void SK_(fini)(Int exitcode)`

Данная функция предназначена для проведения заключительных этапов анализа (сохранение результатов, создание записей в лог-файле и т. д.). Входной параметр – код завершения анализируемой программы.

Наиболее популярным инструментом Valgrind является модуль memcheck. Он предназначен для выявления ошибок при работе с памятью в программах языков СИ/СИ++.

- С каждым байтом памяти связывается бит *адресуемости* (addressability bit, A). Он определяет возможность обращения к соответствующему байту.
- С каждым байтом регистров и каждым байтом памяти связываются 8 бит *корректности* (validity bits, V). Определяют, были ли проинициализированы соответствующие биты в байте.
- Для каждого блока динамически выделенной памяти сохраняется его адрес, а также функция (malloc()/calloc()/realloc(), new, new[]), при помощи которой блок был выделен.

По ходу выполнения программы информация об используемой памяти модифицируется: Valgrind заменяет вызовы функций malloc(), memcpy(), strcpy(), strcat() на вызовы аналогичных функций, сохраняющих информацию об используемых участках памяти.

С помощью модуля memcheck выявляются следующие ошибки:

- попытка использования неинициализированной памяти,
- чтение/запись в память после её освобождения,
- чтение/запись с конца выделенного блока,
- утечки памяти.

Memcheck неспособен обнаруживать ошибки при использовании статических или помещенных на стек данных. Подобные ошибки могут быть обнаружены модулем Ptrcheck для Valgrind.

Прочие модули Valgrind:

- Massif - профилировщик кучи памяти (детальное описание изменений кучи за счет создания снапшотов; построение графа, демонстрирующего использование кучи в процессе выполнения анализируемой программы),
- Helgrind, DRD – инструменты для детектирования ошибок синхронизации в многопоточных приложениях (языки Си/Си++),
- Cachegrind - профилировщик кэш памяти (L1, D1, L2; указываются участки кода, в которых обнаружены кэш-промахи),
- Redux [15] – построение динамических графов потоков данных (DDFG), отражающих процесс получения любого значения в каждой точке программы,
- TaintCheck [16] – анализ «помеченных» данных, в частности, отслеживание потоков данных, полученных из внешних источников (сетевые пакеты, ввод с клавиатуры).

4.2 Среда анализа Pin

Pin [17, 18] – это система для проведения динамического анализа посредством добавления инструментального кода в исполняемые файлы. Поддерживаются следующие архитектуры – IA-32, Intel(R) 64, IA-64. Pin предоставляет пользователю API для разработки собственных инструментов анализа. Инструментальный код добавляется в запущенное приложение посредством JIT-компиляции – блоки инструкций перекомпилируются и инструментируются непосредственно перед выполнением. Модифицированные инструкции записываются в программный кэш. Перекомпиляция вносит существенное замедление в работу анализируемой программы (время выполнения тестов из набора SPEC (целочисленная арифметика) в среднем увеличилось в два раза). Существует возможность внесения инструментального кода непосредственно в исполняемый файл, загруженный в память (probe mode) – при этом доступна лишь часть API, однако накладных расходов практически не возникает.

На рисунке 5 представлена архитектура системы Pin.

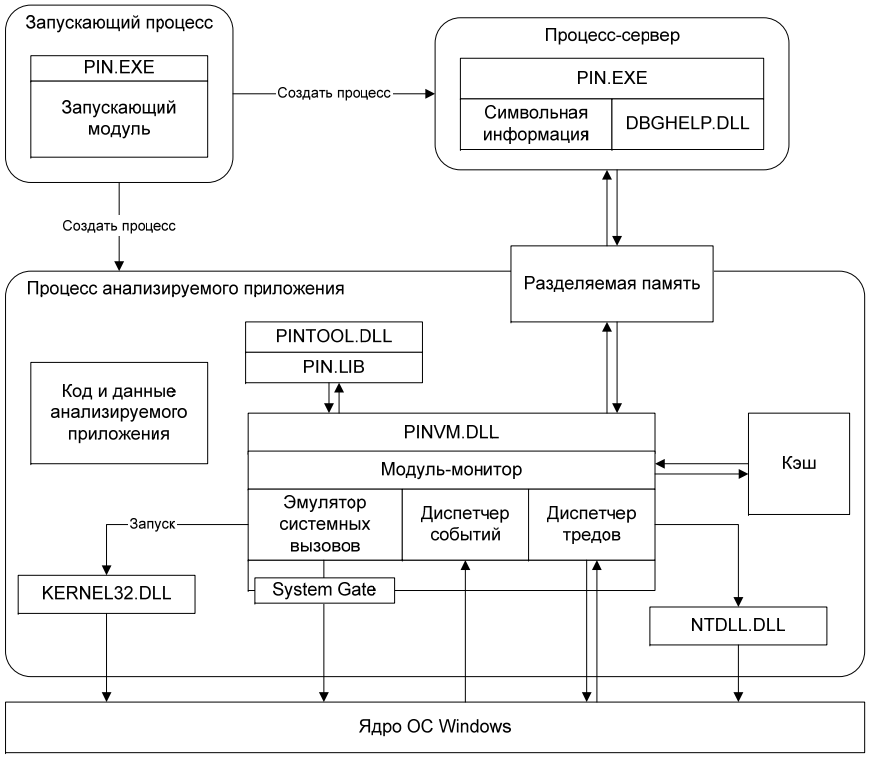

Рисунок 5 – Архитектура системы Pin

Запускающий модуль создает процесс анализируемой программы и процесс сервер, а также производит внедрение библиотек Pin. Процесс-сервер осуществляет взаимодействие с анализируемой программой посредством механизма разделяемой памяти. В адресное пространство процесса анализируемой программы дополнительно загружаются библиотеки инструментов Pin, а также pinvm.dll. Pinvm.dll включает в себя модуль-монитор (Virtual Machine Monitor, VMM), а также API для инструментов Pin. Модуль-монитор (JIT-компилятор, диспетчеры, компонента взаимодействия с ОС) управляет выполнением анализируемой программы и добавляет инструментальный код.

Рассмотрим процесс внедрения более подробно. Под внедрением в данном случае понимается загрузка динамической библиотеки pinvm.dll в адресное пространство процесса анализируемого приложения. Pin создает процесс приложения в состоянии приостановки, затем присоединяется к нему посредством API отладчика для Win32. Отладчик ждет завершения инициализации процесса (kernel32.dll), после чего соединение разрывается. Pin сохраняет контекст программы и заменяет значение счетчика команд, что

приводит к выполнению процедуры, загружающей pinvm.dll и библиотеки инструментов.

Pin контролирует выполнение программы в режиме пользователя, но не в режиме ядра ОС. Для того, чтобы по завершении системного вызова управление вновь попало в модуль-монитор, необходим специальный обработчик. Перехват системного вызова осуществляется при выполнении инструкции, передающей управление из режима пользователя в режим ядра. В 32-битной ОС Windows используются инструкции *sysenter* и *int 2e*, в 64-битной – *syscall*. В 32-битных приложениях, запущенных в 64-битной ОС Windows, для перехода управления в режим ядра выполняется инструкция *jmp far*. Для каждого отслеживаемого Pin системного вызова в ntdll.dll существует функция-обертка; она записывает на регистр номер вызова и осуществляет сам вызов. Pin сохраняет номер системного вызова, а также расположение его аргументов относительно вершины стека.

Некоторые системные вызовы требуют дополнительной обработки Pin (например, если анализируемое приложение запускает процесс-потомок, Pin необходимо создать соответствующий объект-описатель). Приведем список некоторых таких вызовов:

```
NtAllocateVirtualMemory,
NtFreeVirtualMemory,
NtProtectVirtualMemory,
NtMapViewOfSection,
NtUnmapViewOfSection,
NtCreateProcess/Ex,
NtCreateUserProcess,
NtCreateThread/Ex,
NtResumeThread,
NtSuspendThread,
NtGetContextThread,
NtSetContextThread,
NtContinue,
NtCallbackReturn,
NtTerminateThread,
NtTerminateProcess.
```

Другие системные вызовы необходимо выполнять без какой-либо дополнительной обработки. Схематично этот процесс изображен на рисунке 6.

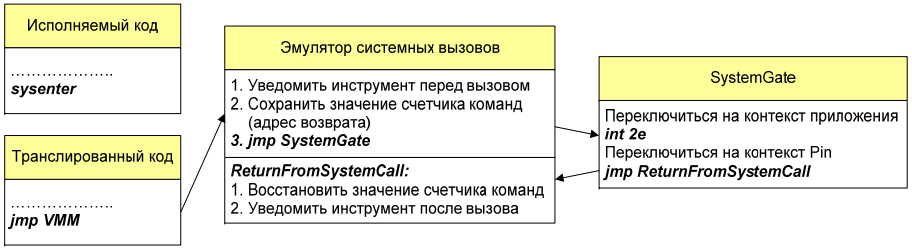

Рисунок 6 – Обработка системного вызова.

При обнаружении системного вызова управление попадает в эмулятор системных вызовов. Если вызов не требует дополнительной обработки Pin, эмулятор направляет его в System Gate. System Gate – это сгенерированная модулем VMM процедура, осуществляющая выполнение системной функции в исходном, неизмененном контексте памяти (за исключением счетчика команд).

Рассмотрим подробнее обработку прерываемых системных вызовов. В ОС Windows асинхронное событие в режиме пользователя возникает в двух случаях – асинхронный вызов процедуры (Asynchronous procedure call, APC) или непрямой переход (callback). Управление механизмом асинхронных вызовов и непрямых переходов осуществляется посредством библиотеки ntdll.dll. Вызовы функций ntdll.dll перехватываются, после чего управление попадает в соответствующий обработчик Pin.

При перехвате асинхронного вызова процедуры проверяется, в каком потоке возник этот вызов. Если вызов возник в потоке, выполнение которого не контролируется, Pin регистрирует новый поток, и управление попадает в транслированный код обработчика KiUserApcDispatcher. В противном случае (асинхронный вызов возник в потоке, выполнение которого контролируется) контекст памяти прерываемого системного вызова сохраняется на стеке. Далее восстанавливается контекст памяти анализируемого приложения, после чего управление попадает в транслированный код обработчика KiUserApcDispatcher. Возврат из процедуры осуществляется посредством системного вызова NtContinue.

Для обработки непрямых переходов в Pin организован стек, в котором хранятся адреса возврата соответствующих прерываемых системных вызовов. То есть, при перехвате непрямого перехода в стек помещается адрес возврата прерываемого системного вызова (соответствующий адрес возврата сохраняется эмулятором системных вызовов). Далее управление попадает в транслированный код обработчика KiUserCallbackDispatcher. Когда эмулятор системных вызовов перехватывает вызов NtCallbackReturn, считывается адрес возврата из вершины стека и записывается вместо адреса возврата текущего NtCallbackReturn. Вершина стека выталкивается, после чего управление

передается в System Gate для исполнения NtCallbackReturn. Поскольку адрес возврата был изменен, по завершении NtCallbackReturn управление попадает обратно в System Gate, после чего восстанавливается значения счетчика команд, сохраненное перед вызовом NtCallbackReturn – адрес возврата системного вызова, прерванного непрямым переходом.

Для обработки исключений, возникших в режиме пользователя, в ОС Windows используется процедура KiUserExceptionDispatcher библиотеки ntdll.dll. Перехватив вызов этой функции, Pin вызывает собственный обработчик. В первую очередь выполняется проверка того, где возникло исключение – в коде анализируемой программы или в коде Pin. Для аппаратного исключения восстанавливается исходный контекст памяти, после чего выполняется транслированный код обработчика. Для программного исключения восстанавливать контекст памяти не нужно, поскольку параметры исключения были установлены в режиме пользователя.

Схема обработки прерываемых системных вызовов и исключений представлена на рисунке 7.

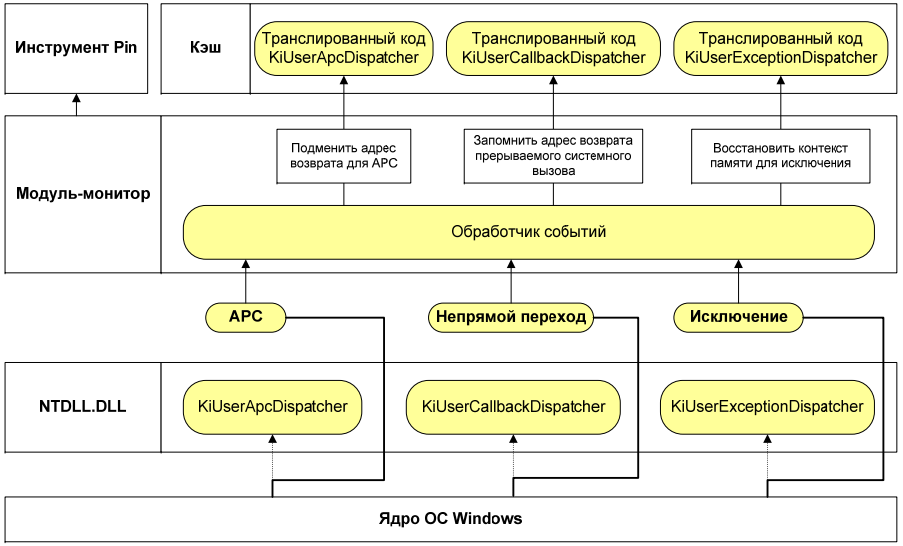

Рисунок 7 – Обработка прерываемых системных вызовов и исключений.

В Pin поддерживается анализ многопоточных приложений, в рамках которого отслеживается создание потоков и их взаимодействие. Реализованы функции-обертки для соответствующих механизмов синхронизации ОС Windows.

Краткие выводы

Pin предоставляет аналитику возможность использовать базовые инструменты анализа, а также создавать свои собственные (посредством Pin API). Отличительная особенность системы – поддержка ОС Windows. На сайте разработчиков доступен исходный код инструмента для следующих архитектур и ОС:

- IA32 и intel64 (x86 32 bit и 64 bit) – Windows
- IA32 и intel64 (x86 32 bit и 64 bit) – Linux
- IA64 (Itanium) – Linux
- ARM – Linux
- IA32 (x86) – MacOs

В ОС Windows были проведены практические эксперименты для базовых инструментов. С помощью инструмента imageload получен список модулей, используемых веб-браузером Google Chrome (приведен фрагмент списка):

```
Loading
C:\Users\ustas.INTRA\AppData\Local\Google\Chrome\Applica
tion\chrome.exe, Image id = 1
Loading C:\Windows\system32\apphelp.dll, Image id = 2
Loading C:\Windows\syswow64\KERNELBASE.dll, Image id = 3
Loading C:\Windows\syswow64\kernel32.dll, Image id = 4
Loading C:\Windows\SysWOW64\ntdll.dll, Image id = 5
Loading C:\Program Files\AVAST Software\Avast\snxhk.dll,
Image id = 6
Loading C:\Windows\syswow64\SHLWAPI.dll, Image id = 7
Loading C:\Windows\syswow64\GDI32.dll, Image id = 8
……
Loading C:\Windows\system32\KBDUS.DLL, Image id = 41
Unloading C:\Windows\system32\KBDUS.DLL
……
Loading C:\Windows\SysWOW64\oleacc.dll, Image id = 63
Unloading C:\Windows\SysWOW64\oleacc.dll
……
```


Инструмент inscount0 осуществляет подсчет выполненных инструкций. С помощью инструмента proccount можно получить список вызываемых процедур. Дополнительно для каждой процедуры определяются модуль, которому принадлежит процедура, абсолютный адрес, количество вызовов, количество выполненных инструкций. Фрагмент списка для тестового запуска Google Chrome представлен в таблице 1.

Procedure	Image	Address	Calls	Instructions
unnamedImageEntryPoint	C:\Windows\system32\netutils.dll	739015a6	1	126
NetApiBufferAllocate	C:\Windows\system32\netutils.dll	73901415	0	97
NetApiBufferFree	C:\Windows\system32\netutils.dll	739013d2	350	6650
unnamedImageEntryPoint	C:\Windows\system32\MSVCP100.dll	67eb3cf4	1	2153
_Mtxunlock	C:\Windows\system32\MSVCP100.dll	67eb31c3	71	497
_Mtxlock	C:\Windows\system32\MSVCP100.dll	67eb31ae	71	497
_Mtxinit	C:\Windows\system32\MSVCP100.dll	67eb3184	16	112
??0_Mutex@std@@QAE@XZ	C:\Windows\system32\MSVCP100.dll	67eabae8	12	156
?_Init@locale@std@@CAPAV_Locimp@12@XZ	C:\Windows\system32\MSVCP100.dll	67eab72f	24	296
?_Locinfo_ctor@_Locinfo@std@@SAXPAV12@PBD@Z	C:\Windows\system32\MSVCP100.dll	67eab6db	3	99
?_Locinfo_dtor@_Locinfo@std@@SAXPAV12@@Z	C:\Windows\system32\MSVCP100.dll	67eab6ba	3	42
?_Setgloballocale@locale@std@@CAXPAX@Z	C:\Windows\system32\MSVCP100.dll	67eab5d2	1	13
?_Getgloballocale@locale@std@@CAPAV_Locimp@12@XZ	C:\Windows\system32\MSVCP100.dll	67eab5c7	24	48
??1_Lockit@std@@QAE@XZ	C:\Windows\system32\MSVCP100.dll	67eab472	71	639
??0_Lockit@std@@QAE@H@Z	C:\Windows\system32\MSVCP100.dll	67eab445	71	1278
??0_Init_locks@std@@QAE@XZ	C:\Windows\system32\MSVCP100.dll	67eab397	14	167
_Getctype	C:\Windows\system32\MSVCP100.dll	67eab2d8	3	90
_Getcvt	C:\Windows\system32\MSVCP100.dll	67ead52	2	18
_Mbrtowc	C:\Windows\system32\MSVCP100.dll	67eaa9d4	8	280
?uncaught_exception@std@@YA_NXZ	C:\Windows\system32\MSVCP100.dll	67eaa1f4	0	119
??0?$basic_streambuf@GU?$char_traits@G@std@@@std@@IAE@XZ	C:\Windows\system32\MSVCP100.dll	67e93f60	4	92

Таблица 1 – Результаты анализа посредством инструмента proccount.dll для Google Chrome.

4.3 Инструмент TEMU

TEMU – инструмент динамического анализа на базе полносистемного эмулятора QEMU, разрабатываемый в университете Беркли (исследовательский центр CyLab), в рамках проекта BitBlaze [19]. Инструмент анализирует состояние программ и операционной системы непосредственно во время их работы на виртуальной машине. Во время анализа дополнительно извлекается семантика уровня ОС – данные о процессах, нитях, загруженных модулях и именах функций. Схема работы изображена на рисунке 8.

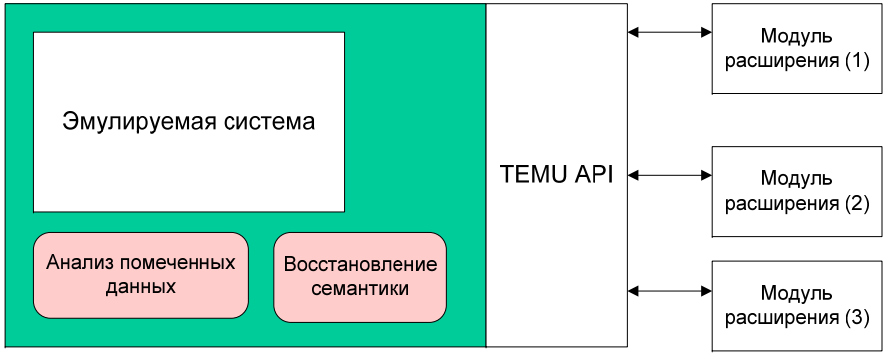

Рисунок 8 – Схема компонентов TEMU.

TEMU предоставляет следующие возможности:

- снятие бинарной трассы уровня машинных инструкций,
- интерфейс для создания модулей дополнительного анализа.

Для Microsoft Windows создан модуль ядра (module notifier), загружаемый в гостевой ОС во время трассировки. Этот плагин отслеживает создание/уничтожение процессов, а также загрузку модулей (для каждого модуля сохраняется диапазон адресов в виртуальном адресном пространстве процесса, загрузившего модуль). Кроме того, для каждого процесса сохраняется значение регистра CR3. Также для ОС Windows сохраняются идентификаторы потоков. Для исполняемых файлов формата PE сохраняется таблица экспортируемых символов.

Для ОС Linux анализируется содержимое файла /proc/kallsyms. В нем находится символьная таблица адресов функций и процедур, используемых ядром ОС. Кроме того, во время трассировки осуществляется перехват вызовов для некоторых функций ядра, например, do_fork и do_exec.

Одно из ключевых возможностей TEMU - анализ помеченных данных, выполняющийся непосредственно во время работы системы. Для каждого помеченного байта создается структура данных, в которой хранится информация о том, где этот байт используется. В качестве источников помеченных данных в TEMU рассматриваются:

- пользовательский ввод,
- сетевые пакеты,
- данные жесткого диска.

Кроме того, можно «помечать» абстрактные высокоуровневые объекты (например, значение, возвращаемое некоторой функцией).

В статье [20] описан метод поиска уязвимостей на основе анализа полученных трасс. Схема предлагаемой методики приведена на рисунке 9.

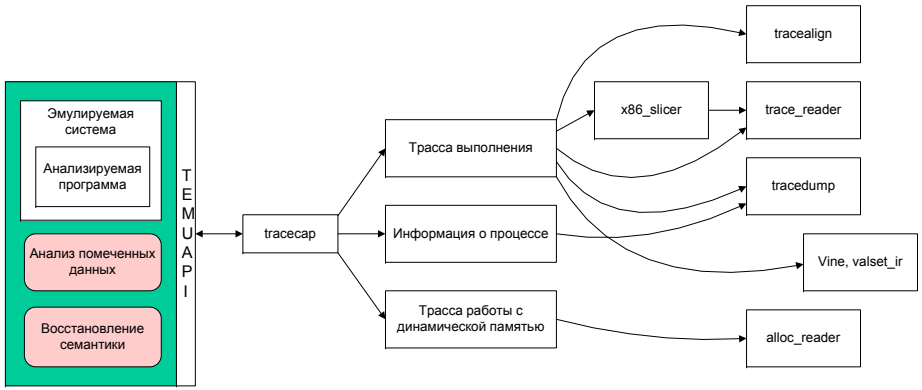

Рисунок 9. Анализ уязвимостей на основе трасс исполнения.

Первым шагом является получение набора трасс для разных входных данных с помощью плагина tracecap. Вместе с кодом инструкции в трассу попадает контекст ее выполнения. Например, для инструкции:

```
rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi]
```

в трассу записывается следующая информация:

993850	номер шага трассы
7814507a	адрес инструкции
rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi]	
M@0x0267ae7c[0xed012800][4](CR)	операнд в памяти (4 байта)
T1 {12 ()()(1111, 5) (1111, 5) }	2 старших байта помечены
R@ecx[0x000007f3][4](RCW)	операнд-регистр
T0	байты не помечены
M@0x0267f0e0[0x0cc00b2f][4](CW)	операнд в памяти (4 байта)
T1 {15 (1111, 5) (1111, 5) (1111, 5) (1111, 5) }	все байты помечены
ESP:	регистр ESP не используется
NUM_OP: 5	количество операндов
TID: 1756	идентификатор нити
TP: TPSrc	инструкция считывает помеченные данные
EFLAGS: 0x00000202	регистр EFLAGS
CC_OP: 0x00000010	
DF: 0x00000001	
RAW: 0xf3a5	код инструкции
R@edi[0x0267f0e0][4](R) T0	операнд EDI не помечен
R@esi[0x0267ae7c][4](R) T0	операнд ESI не помечен

Помимо трасс сохраняется информация о распространении интересующих помеченных данных (tainting information). Далее используются инструменты для чтения и анализа трасс. В частности, применяются следующие алгоритмы:

1. Обратный слайсинг трасс (плагин x86_slicer),
2. Выравнивание пар трасс по отношению друг к другу для их сопоставления и совместного анализа (плагин tracealign),

3. Совмещение информации о распространении помеченных данных с дампами, сделанными в момент падения анализируемой программы, для определения влияния входных данных на некорректное поведение (плагины tracecap, tracedump),
4. Отслеживание работы программы с динамической памятью (плагины tracecap, alloc_reader),
5. Получение количественной оценки влияния отдельных переменных на поведение программы,
6. Получение множеств допустимых значений переменных программы.

Далее приводится описание некоторых плагинов.

1. trace-reader - инструмент извлечения информации об инструкциях, попавших в трассу (адрес, код инструкции, операнды и т. д.),
2. x86_slicer ─ реализация обратного динамического слайсинга по трассе. Отслеживаются только зависимости по данным (не по управлению),
3. tracealign ─ инструмент "выравнивания" трасс ─ сопоставляет шаги одной трассы соответствующим шагам другой трассы. Компонент использует методику "индексирования трасс", описанную в статье [21]. Особенно интересны выявляемые этим компонентом, так называемые, "точки дивергенции" ─ точки ветвления, в которых выполнение разных трасс пошло по разным ветвям. Данный модуль может использоваться отдельно от taint-анализа, однако совместное использование может улучшить результаты по крайне мере двумя способами:
   - taint-анализ показывает, условия каких точек ветвления зависят от помеченных данных, выявляя "точки дивергенции",
   - если при выравнивании двух трасс выполненный на них taint-анализ отличается в исходных данных только некоторым подмножеством ячеек (и их значений) – в одной трассе они помечены, в другой – нет, – и при этом некоторые значения, помеченные в одной из трасс, совпадают со значениями в другой, то, по-видимому, их значения на самом деле не зависят от значений входных данных, и пометку можно снять. Данный метод позволяет уточнить результаты taint-анализа,
4. tracedump – извлечение и отображение информации о дампах, которые могут сниматься в произвольных точках (например, в точке падения программы) плагином tracecap. В частности, такие дампы могут хранить информацию о помеченных участках памяти. Для

точки снятия дампа хранится дополнительная информация – стек вызовов, значения регистров общего назначения,

5. alloc_reader – получение информации из трассы динамического выделения/освобождения памяти. Плагин позволяет анализировать уязвимости, связанные с переполнениями буферов и некорректной работой с указателями (двойное освобождение и т.д.). Для заданной точки в трассе и адреса памяти можно определить область выделения, к которой адрес относится (если есть), а также показать ближайшие к адресу выделенные области,
6. valset_ir – компонента, позволяющая оценивать для заданной точки программы множества возможных значений, принимаемых различными участками памяти (и регистрами). Также valset_ir используется для количественной оценки влияния переменной на поведение программы – эта методика описывается в статье [22].

Краткие выводы

Вопрос поиска источника уязвимости (например, «плохого» входного файла) остаётся за рамками статьи [20]. Для исследования требуется построение трассы – разности трассы с реализацией уязвимости и трассы без нее. Также требуется файл соответствия между «разностной» и «хорошей» трассой. Методы получения «разностной» трассы и файла соответствий не описаны.

На новых версиях и дистрибутивах Linux существует проблема сборки TEMU, так как для сборки требуется gcc-3.4. TEMU поддерживает достаточно старые гостевые ядра, в коде для них прописаны адреса и смещения в файле shared/read_linux.c. Список поддерживаемых ядер:

- 2.4.20-24.7,
- 2.6.15-1.2054_FC5 (Fedora Core 5),
- 2.6.26-1-686 (Debian 2.6.26-1-686 2.6.26-4),
- 2.6.24-19-generic (Ubuntu 2.6.24-19.41),
- 2.6.28-11-generic (Ubuntu 2.6.28-11.42),
- 2.6.28-14-generic (Ubuntu 2.6.28-14.47),
- 2.6.28-15-generic (Ubuntu 2.6.28-15.49),
- CentOS_5_2.6.18_92.1.10.el5.

Для неподдерживаемых ядер (например, 3.3.5) TEMU не может получить список процессов и поэтому не может снимать трассы с нужных процессов. Для получения нужных смещений для других версий ядра в комплекте TEMU есть драйвер ядра procinfo.

В проведенных экспериментах было обнаружено, что TEMU (даже без плагинов) вносит ощутимое замедление в работу гостевой ОС по сравнению с оригинальной версией QEMU.

4.4 Фаззинг

Фаззинг является универсальным методом тестирования программ, который можно отнести к динамическому анализу. Идея подхода заключается в многократном запуске анализируемой программы на различных начальных данных (рисунок 10). В качестве анализатора, фиксирующего ошибочное состояние программы во время ее работы, могут выступать рассмотренные в предыдущем разделе системы, такие как Valgrind, Pin и т.п. Сам процесс фаззинга осуществляется т.н. драйвером, компонентой системы, которая оценивает результаты запуска, полученные от анализатора, и запускает очередную итерацию.

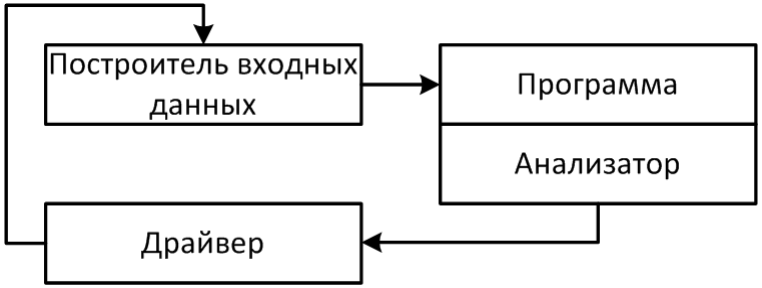

Рисунок 10 – Упрощенная схема фаззинга.

Существенно, каким образом строится очередной набор входных данных: если система никак не использует знания о внутренней организации анализируемой программы, то такой подход является тестированием «черного» ящика. Существует множество программных решений, например *MiniFuzz* [23] (фаззер формата файла), *Peach* [24] (фаззер сетевых протоколов), *IOCTL Fuzzer* [25] (фаззер драйверов устройств). Тем не менее, каждая программа обладает рядом особенностей, вследствие чего тот или иной фаззер может оказаться непригодным для поиска дефектов.

В противном случае, когда входные данные строятся, исходя из статического представления программы и результатов предыдущих запусков, происходит тестирование «белого» ящика. Современные системы поиска ошибок, использующие такой подход, рассматриваются в следующем разделе - «Символьное выполнение».

4.4.1 Применение симулятора для анализа драйвера Wi-Fi устройства

В статье [26] описан метод поиска уязвимостей в драйвере сетевого устройства, отвечающего стандарту IEEE 802.11 (Wi-Fi). Авторы отмечают следующие трудности, возникающие при фаззинге сетевых протоколов:

- Ограничения по времени (timing) – сетевой пакет будет отброшен, если придет слишком поздно.
- Перегрузка (overload) – драйвер устройства может отбросить сетевой пакет вследствие переполнения буфера, обусловленного большим количеством входящих пакетов.
- Нарушение целостности (corruption) – содержимое сетевого пакета может измениться из-за сбоев в канале передачи данных. Драйвер устройства не будет обрабатывать сетевой пакет с неправильным значением контрольной суммы.
- Неправильное состояние протокола (state mismatch) – в момент получения сетевого пакета, содержимое которого намеренно изменено, протокол анализируемого приложения должен находиться в некотором определенном состоянии.
- Временные ограничения состояния протокола (state timeout) – если протокол анализируемого приложения находится в правильном состоянии, но в течение заданного промежутка времени ожидаемый сетевой пакет не был получен, состояние протокола может измениться.

Для преодоления указанных трудностей предлагается запускать анализируемое приложение на виртуальной машине и организовывать непосредственное взаимодействие фаззера с виртуальным сетевым интерфейсом. При этом появляется возможность эффективного анализа результатов фаззинга:

- анализ исключительных ситуаций, перезапусков ЦПУ виртуальной машины,
- получение дампа произвольной области памяти в любой момент,
- трассировка счетчика команд для ЦПУ виртуальной машины (позволяет выявить случаи попадания управления на определенные ветки кода межпроцедурного CFG).

Компоненты предлагаемой системы анализа представлены на рисунке 11.

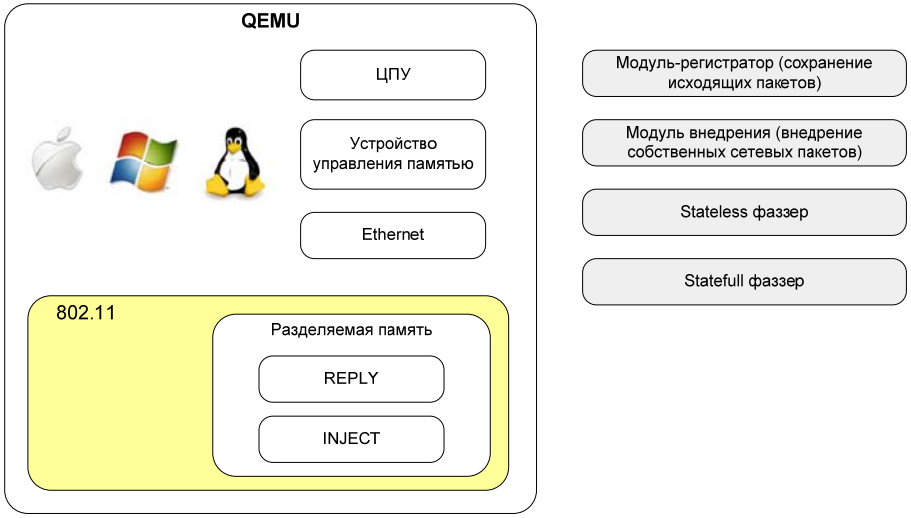

Рисунок 11 – Архитектура системы фаззинга в рамках Qemu.

Система имеет модульную структуру. Каждый модуль взаимодействует с виртуальным сетевым интерфейсом посредством механизма разделяемой памяти.

4.4.2 Инструмент TaintScope

В статье [27] рассматривается мутация сетевых пакетов с возможностью пересчета контрольной суммы (система TaintScope). Для локализации участков кода, проверяющих значение контрольной суммы, проводится трассировка анализируемой программы с последующим профилированием. Строится четыре набора предикатов: P_1, P_0, Q_1, Q_0. В набор P_1/P_0 попадают предикаты, обращающиеся только в истину/только в ложь на *корректных* входных данных. В набор Q_1/Q_0 попадают предикаты, обращающиеся только в истину/только в ложь на *некорректных* входных данных. Набор предикатов $(P_1 \cap Q_0) \cup (P_0 \cap Q_1)$ обычно соответствует проверке контрольной суммы.

Каждому байту модифицируемых сетевых пакетов ставится в соответствие его индекс. Во время трассировки отслеживаются операции обращения к байтам и их использование. TaintScope определяет, какие именно байты сетевого пакета оказывают влияние на потенциально уязвимые функции (например, malloc и strcpy).

На рисунке 12 представлена схема анализа TaintScope.

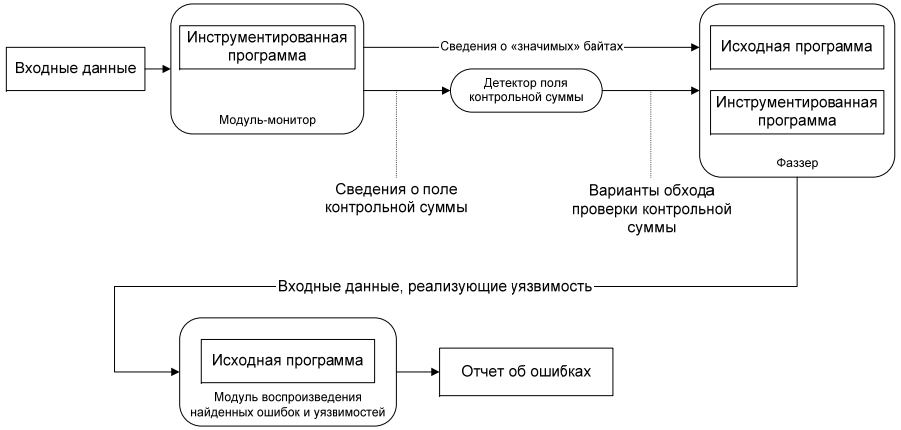

Рисунок 12 – Анализ в рамках TaintScope.

Первый этап анализа состоит в запуске приложения на «хороших» входных данных (т. е. на тех входах, для которых приложение работает корректно). Модуль-монитор следит за тем, какие байты входных данных передаются в качестве аргументов в функции API («значимые» байты), а также какие байты оказывают влияние на условия переходов (сведения о поле контрольной суммы). Далее для того, чтобы выявить участок кода, соответствующий проверке поля контрольной суммы, строятся наборы предикатов P_1, P_0, Q_1, Q_0. Если участок найден, то выполняется следующий этап анализа – фаззер генерирует «плохие» входные данные и передает их программе. При этом внесенный ранее в программу инструментальный код «пытается» обойти проверку контрольной суммы (в соответствии с информацией, полученной на предыдущем этапе). Заключительный этап анализа – восстановление входных данных, воспроизводящих ошибку в исходной программе.

С помощью TaintScope авторам публикации удалось выявить ошибки в таких приложениях, как Microsoft Paint, Adobe Acrobat, MPlayer, WinAmp.

5 Символьное выполнение

Идея символьной интерпретации программ появилась в середине 70-х годов применительно к задаче тестирования программ [28]. Эта техника позволяет автоматически получать тестовое покрытие, однако на практике ее применение сталкивается с рядом ограничений. Каждое ветвление добавляет новое уравнение (ограничение) над символьными переменными и требует отдельного рассмотрения каждого пути, что достаточно быстро приводит к «экспоненциальному взрыву» количества рассматриваемых путей выполнения. Переход от обработки конкретных значений на аппаратуре к обработке символьных данных программным интерпретатором вызывает ощутимые накладные расходы, время выполнения отдельно взятого пути

существенно увеличивается. Регулярное решение систем логических уравнений, требует достаточной эффективности от программы-решателя. Перечисленные ограничения не позволяют применять эту технику для больших программных систем «в лоб», без применения эвристик, отсекающих пути выполнения, и сокращения количества интерпретируемых переменных.

В результате развития компиляторных технологий и анализа помеченных данных в частности, возникла идея смешанного выполнения[2] программы, когда только ограниченное подмножество входных данных рассматривается в качестве символьных переменных, все остальные переменные программы получают конкретные значения. Большое количество публикаций по этой тематике было сделано, начиная с 2005 года; к числу первых значимых работ можно отнести, например, систему Sage [29]. Рассматриваемый подход к анализу фактически является динамическим, но значимость показанных результатов позволяет выделить символьную интерпретацию в самостоятельный раздел, поскольку одним из успешных практических применений смешанного выполнения является поиск ошибок в бинарном коде программ.

В качестве источника символьных данных, как правило, выступают функции приема сетевых пакетов и чтения файлов, пользовательский ввод. Исполнитель программы содержит в себе механизмы конкретного и символьного выполнения, механизм отслеживания помеченных данных. Если вычисления в программе не используют символьные переменные, то они проходят в обычном режиме, в противном случае происходит символьная интерпретация, результат этих вычислений рассматривается как новая символьная переменная. Обработка условного перехода в программе, зависящего от символьных переменных, приводит к созданию нового экземпляра исполнителя. В момент «разделения» исполнители содержат одинаковый контекст конкретного исполнения и контекст символьного исполнения, различающийся одним уравнением: оно описывает выполнение ветки «true» в одном исполнителе и «false» в другом.

В процессе смешанного выполнения набор логических уравнений можно дополнять, описывая тем самым интересующие состояния программы, например, являющиеся ошибочными: разыменование нулевого указателя, переполнение буфера памяти и т.п.

Следует упомянуть серию публикаций коллектива Стенфордского университета, в состав которого входят специалисты компании Coverity [2]. В 2006 году был представлен инструмент EXE [30], предназначенный для поиска ошибок в коде языка Си, который два года спустя (2008) получил

[2] В англоязычной литературе достаточно часто используется термин concolic execution, сочетающий в себе слова concrete и symbolic. Адекватный перевод термина concolic на русский язык никем еще предложен не был.

развитие в символьном интерпретаторе KLEE [31]. И в том и в другом проекте был реализован механизм смешанного выполнения, но в более поздней работе авторы применили трансляцию Си-кода в llvm [32], что позволило перейти от подтверждения концепции к созданию инструмента промышленного уровня.

В области анализа бинарного кода наиболее значимые результаты были получены в работах CyLab [33] и Dependable Systems Lab [34].

Многие инструменты символьного анализа включают в себя следующие компоненты:

- Модуль, осуществляющий taint-анализ.
- Модуль работы с промежуточным представлением.
- Модуль символьной интерпретации.

Качество работы инструмента символьного анализа достаточно сильно зависит от эффективности каждой из этих компонент.

5.1 Семейство инструментов проекта BitBlaze

Первые публикации коллектива CyLab, связанные с работами по смешанному выполнению бинарного кода, относятся к 2007 году [35, 36]. Они проводились в рамках более объемного проекта BitBlaze [19]. Ставилась задача автоматического выделения вредоносного кода и поддержка типовых действий аналитика, изучающего этот выделенный код. Решить эту задачу методами статического анализа крайне затруднительно, поскольку срабатывание вредоносного кода происходит только в определенных ситуациях, на специальных наборах входных данных. Символьное выполнение способно помочь в классификации возможных входных данных и последующей оценке влияния выделенных классов на ход работы программы. Основным недостатком этого подхода является сильное замедление работы программы при символьной интерпретации и экспоненциальный рост количества путей выполнения, которые необходимо анализировать.

Поскольку в рамках данной постановки задачи предметом исследования является относительно небольшой (порядка нескольких тысяч машинных команд) фрагмент кода, авторы работы предложили использовать механизм смешанного выполнения, способный значительно уменьшить длительность анализа. Код окружения выполняется непосредственно, но когда управление попадает в анализируемый код, происходит автоматическое переключение на режим символьного выполнения.

В первой работе [35] был представлен инструмент MineSweeper, позволяющий выявлять поведение программы, активируемое триггерами. От пользователя инструмента требуется заранее знать, какой вид внешних данных (сетевые пакеты, системное время, пользовательский ввод и т.д.) способен активировать в программе исследуемую функциональность. В ходе работы инструмента отслеживаются все данные, вводимые через указанные интерфейсы, обработка этих данных ведется в символьном режиме. В

частности, отслеживается влияние этих данных на условные переходы, что позволяет выделять фрагменты кода, выполнение которых (активация функциональности) зависит от значений введенных символьных данных. Для найденных фрагментов инструмент решает систему уравнений, определяя конкретные значения, для которых будет происходить срабатывание кода.

Реализация подхода опирается на две компоненты, являющиеся программами с открытым исходным кодом: эмулятор QEMU и решатель уравнений STP. Первый используется для конкретного выполнения исследуемой программы, второй – для решения систем уравнений, получаемых при символьной обработке данных-триггеров. В частности, совместность системы уравнений, полученной по условным переходам, позволяет оценить возможность реализации заданного пути выполнения. Следует отметить, что символьная интерпретация осуществляется для низкоуровневого промежуточного представления (RISC-машина), разработанного авторами статьи. Такой подход потребовал от них разработать и реализовать как транслятор кода x86 в это представление, так и символьных интерпретатор.

Вторая статья [36] датирована тем же годом, что и рассмотренная выше, но по сути представленных результатов является более поздней работой: механизм смешанного выполнения был обособлен в отдельную компоненту и получил название Rudder. Более того, в статье ставятся более общие цели относительно анализа вредоносного кода. Предлагаются методы, позволяющие не только выделять вредоносный код, но и помогающие понять принципы работы выделенного кода. Более конкретно, автоматизируется решение следующих практических задач.

- Влияние заданного API на работу кода (проход по графу потока управления).
- Обратная задача – поиск допустимых последовательностей вызовов функций API, в рамках выявленного графа потока управления.
- Поиск начальных данных, обеспечивающих попадание в заданную точку программы. Поскольку в общем случае задача решается полным перебором, авторами предложена эвристика: из графа потока управления выделяется подграф (*чоп*) меньшего размера. Он строится путем пересечения двух подмножеств вершин исходного графа. Первое подмножество составляют те вершины, в которые можно попасть из указанного входа, второе – те, из которых можно попасть в заданную точку программы. В выделенном подграфе ищется путь минимальной длины и по всем попавшим него условным переходам строится система уравнений, решение которой дает требуемые начальные данные. Если система несовместна – рассматривается следующий наикратчайший путь.
- Поиск константных значений в выходных данных. Учитываются зависимости по данным, позволяющие отследить распространение констант по коду. Если все входные операнды машинной команды

помечены как константные, то и результат считается константным значением. Анализ производится по всем допустимым путям выполнения.
- Построение обратного динамического «слайса» для заданной трассы выполнения. Следует отметить, что получаемое подмножество машинных команд не является слайсом в классическом понимании, как минимум из-за того, что авторами работы учитываются только зависимости по данным, зависимости по управлению в расчет не берутся.

Как и результаты предыдущей статьи, описанный инструмент BitScope не является свободно распространяемым ПО. На рисунке 13 приводится схема его устройства.

Компонента Path Selector осуществляет выбор траекторий выполнения в случаях, когда условный переход зависит от символьных данных. Реализован интерфейс для введения собственных функций, определяющих порядок перебора траекторий выполнения.

Компонента Solver выясняет совместность ограничений, наложенных на данную траекторию (используется решатель уравнений STP [37]), а также определяет область памяти, соответствующую символьным адресам.

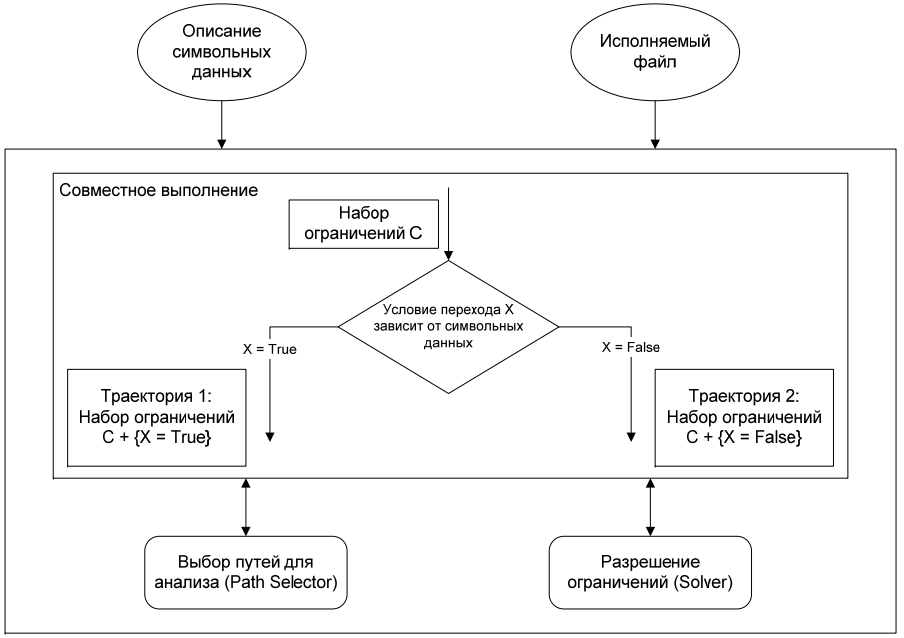

Рисунок 13 – Архитектура инструмента BitScope.

Механизм смешанного выполнения непосредственно отвечает за переключение между режимами работы. Для каждой инструкции выполняется проверка – являются ли операнды-источники символами или имеют конкретные значения. Если операнды-источники представляют собой конкретные значения, то инструкция выполняется. Если операнды-источники являются символами, то операнду-приемнику присваивается символьное выражение, получающееся в результате выполнения инструкции. Для распространения символьных данных используется та же функциональность, что и в TEMU для проведения динамического анализа помеченных данных.

Совместное выполнение подразумевает комбинацию символьного выполнения и выполнения на конкретных данных. Во время совместного выполнения «на лету» составляется таблица, которая символам ставит в соответствие конкретные значения. Когда результат вычисления символьного выражения непосредственно влияет на выбор траектории в графе потока управления, его возможные значения вычисляются в соответствии с текущими значениями символов в таблице, то есть осуществляется переход к конкретным значениям. Данное выражение перестает быть символьным и рассматривается отныне как константное (с точки зрения символьного выполнения). В том числе, если адрес, по которому происходит обращение, представляет собой символьное выражение, используется Solver (для определения возможных значений данного символьного выражения).

Основным отличием BitScope от результатов предыдущей публикации является компонент Extractor, в котором реализуется решение перечисленных выше практических задач.

В более поздней работе этого же коллектива авторов [20] упоминаются инструменты BitFuzz [38] и FuzzBall [39], также применяемые при поиске ошибок, но в несколько нестандартной постановке задачи.

Первый инструмент используется для поиска ошибок в бинарном коде вредоносного ПО. Особенностью такого ПО является то, что входные данные проходят криптографическую обработку. Часть системы ограничений для символьных данных описывает эту обработку, автоматически получить решение для этой части системы (т.е. построить обратную функцию) крайне затруднительно. В работе [38] предложен подход, позволяющий преодолевать это затруднение: в бинарном коде идентифицируются функции, «сложные» для символьного анализа (криптографические функции попадают в эту категорию), производится декомпозиция пути выполнения, для «сложных» функций ищутся обратные, для анализа остальных функций решается система соответствующих ограничений. Следует отметить, что поиск обратных функций выполняется путем тестирования на произвольных начальных данных функций, имеющих сходный интерфейс, для определения которого применяется инструмент BCR [40], разработанный этим же коллективом авторов.

Второй инструмент [39] предназначен для поиска ошибок в программных симуляторах. Метод анализа предполагает, что доступны два программных симулятора, корректность одного их них оценивается, второй выступает в качестве образца. Используется предположение, что в симуляторе-образце интерпретация машинных команд организована следующим образом: результаты декодирования команды определяют выбор ветви оператора switch, в теле каждой ветви расположен относительно компактный код интерпретирующий команду, причем характер работы машинных команд позволяет рассчитывать на отсутствие циклов. Тело каждой ветви независимо от других подвергается символьной интерпретации, которая позволяет построить набор тестов. Корректность работы первого симулятора оценивается сравнением результатов работы тестов, которые можно выполнять не только в симуляторах, но и непосредственно на аппаратуре.

Краткие выводы

Без описания реализации невозможно оценить BitScope и критерии его применимости. По представленным публикациям можно заключить, что Rudder обеспечивает функциональность схожую с S2E [41], реализуя механизм смешанного выполнения. Различными в этих проектах является инфраструктура, на которую смешанное выполнение опирается. В случае S2E используется симулятор QEMU для непосредственного (конкретного) выполнения и виртуальная машина KLEE для символьного. Связь между этими компонентами обеспечивается разработанным авторами S2E бинарным транслятором, который переводит внутреннее представление QEMU в бит-код LLVM.

5.2 Инструмент Avalanche

Avalanche [42] – это инструмент для динамического поиска дефектов в бинарном коде, разработанный в ИСП РАН. Avalanche использует возможности динамического профилирования программы, предоставляемые Valgrind, для сбора и анализа трассы выполнения программы. Результатом такого анализа становится либо набор входных данных, на которых в программе возникает ошибка, либо набор новых тестовых данных, позволяющий обойти ранее не выполнявшиеся и, соответственно, еще не проверенные фрагменты программы. Таким образом, имея единственных набор тестовых данных, Avalanche реализует итеративный динамический анализ, при котором программа многократно выполняется на различных, автоматически сгенерированных тестовых данных, при этом каждый новый запуск увеличивает покрытие кода программы.

На начальном этапе разработки область применимости инструмента ограничена определенным классом программ – входные данные программа

получает из одного файла. В настоящий момент реализовано получение входных данных посредством сетевого интерфейса (TCP, UDP).

Общая схема работы Avalanche представлена на рисунке 14.

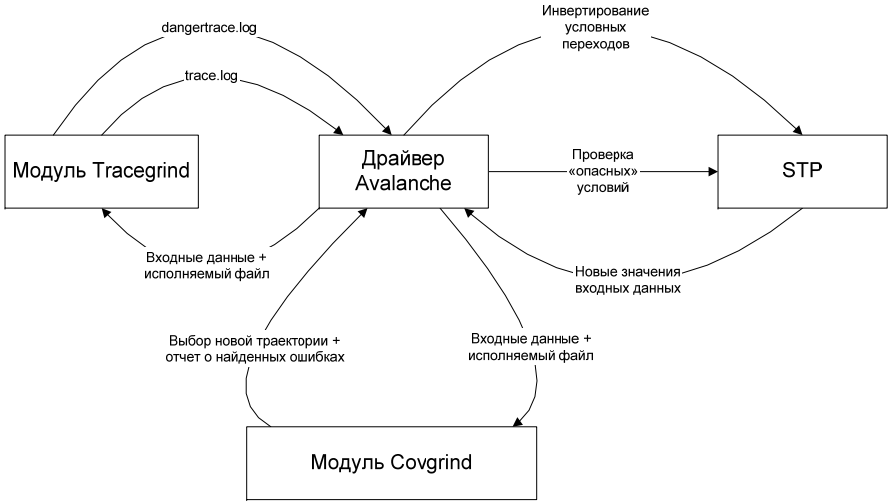

Рисунок 14 – Схема работы Avalanche.

Инструмент Avalanche состоит из четырех основных компонентов: двух модулей расширения Valgrind – Tracegrind и Covgrind, инструмента проверки выполнимости ограничений STP и управляющего модуля. Tracegrind динамически отслеживает поток помеченных данных в анализируемой программе и накапливает условия для обхода еще не пройденных частей и для срабатывания опасных операций. Эти условия при помощи управляющего модуля передаются STP для исследования их выполнимости. Если какое-то из условий выполнимо, то STP определяет те значения всех входящих в условия переменных (в том числе и значения байтов входного файла), которые обращают условие в истину.

В случае выполнимости условий для срабатывания опасных операций программа запускается управляющим модулем повторно (на этот раз без профилирования) с соответствующим входным файлом для подтверждения найденной ошибки.

Выполнимые условия для обхода еще не пройденных частей программы определяют набор возможных входных файлов для новых запусков программы. Таким образом, после каждого запуска программы инструментом STP автоматически генерируется множество входных файлов для последующих запусков анализа. Далее встает задача выбора из этого множества наиболее «интересных» входных данных – в первую очередь

должны обрабатываться входные данные, на которых наиболее вероятно возникновение ошибки. Для решения этой задачи используется эвристическая метрика – количество ранее не обойденных базовых блоков в программе. Для измерения значения эвристики используется компонент Covgrind, в функции которого входит также фиксация возможных ошибок выполнения.

Существенным ограничением Avalanche является экспоненциальная сложность решаемых задач – проверки выполнимости булевских форму и обхода дерева условных переходов программы. Наличие подобного ограничения приводит к тому, что эффективно обнаруживаются лишь те ошибки, которые находятся близко к началу пути выполнения программы.

5.3 Поиск уязвимостей через анализ кода обновления ПО

В работе CyLab 2008 года [43] была поставлена задача, схожая с представленной выше, но решаемая в совершенно других условиях. Предлагается метод автоматического поиска уязвимостей, присутствующих в программе *P*, и исправленных в *P'*. Программы *P* и *P'* представлены бинарным кодом, какой-либо дополнительной отладочной информации не требуется. Результат поиска – место в бинарном коде, содержащее ошибку, и набор начальных данных, позволяющих эту ошибку реализовать.

Основная идея работы заключается в том, что в изменения исправленной версии программы входят дополнительные проверки состояния, которые не позволяют реализоваться обнаруженным ошибкам. В результате сопоставления исходной и исправленной версий программы выявляются места в коде, которые «защищаются» этими дополнительными проверками. Для каждого такого места в коде строится система логических уравнений, описывающих условие попадания в это место программы и дополнительное условие, являющееся отрицанием условия добавленной «защитной» проверки. Если построенная система совместна, ее решение может активировать некоторую (неизвестную для исследователя) ошибку. Нет гарантий, что полученный набор входных данных приведет к срабатыванию ошибки, поскольку неизвестен смысл внесенных в программу изменений. Поэтому заключительным этапом анализа является тестирование полученного решения.

Для описанного подхода было предложено название APEG, являющееся сокращением от automatic patch-based exploit generation. Это же название используется и для программной реализации, которая базируется как на собственных разработках коллектива, так и на сторонних программных инструментах. Для сравнения двух версий программы используется сторонняя утилита eEyE's Binary Diffing Suite (EBDS) [44]; построение системы уравнений осуществляется после трансляции команд x86 во внутреннее представление Vine [45], эта же система используется для построения графа потока управления при статическом анализе бинарного кода программы; для

решения системы уравнений используется STP [37]; проверка полученного решения проводится в симуляторе TEMU [19].

Авторы работы различают три подхода к построению системы уравнений: динамический, статический, комбинированный.

В первом случае имеется трасса выполнения, полученная для некоторого набора входных данных. Трасса работы программы *P'* строится на уровне потока машинных команд, для экономии места в нее записываются только те команды, которые зависят от помеченных входных данных. Причем трасса должна проходить через две точки в программе: ввод каких-либо помеченных данных и место расположения «защитной» проверки. Задача получения такой трассы выходит за рамки рассматриваемой работы, авторы предполагают, что «типовой» набор входных данных должен приводить к прохождению выполнения через ввод помеченных данных и какую-либо введенную проверку. После трансляции кода в Vine, для условия проверки строится система логических уравнений, описывающая слабейшее предусловие. Техника построения слабейшего предусловия для машинных команд была рассмотрена авторами в предыдущей работе [46]. Правила вычисления слабейшего предусловия (Рис. 15) последовательно применяются на каждом шаге трассы при обратном проходе по ней, от места срабатывания до места ввода помеченных данных. Полученная формула предается в решатель STP. Основной недостаток подхода – исходная трасса способна неявно задать ограничения на состояние программы, которые сделают невозможным реализацию уязвимости.

$$\frac{}{wp(x := e, Q) \vdash let\ x = e\ in\ Q}\ Assign$$

$$\frac{}{wp(assert\ e, Q) \vdash e \wedge Q}\ Assert$$

$$\frac{wp(s_1, wp(s_2, Q)) \vdash Q_1}{wp(s_1; s_2, Q) \vdash Q_1}\ Seq$$

$$\frac{wp(s_1, Q) \vdash Q_1 \;\; wp(s_2, Q) \vdash Q_2}{wp(if\ e\ then\ s_1\ else\ s_2, Q) \vdash (e \Rightarrow Q_1) \wedge (\neg e \Rightarrow Q_2)}\ Choice$$

Рисунок 15 – Правила вычисления слабейшего предусловия для внутреннего представления Vine.

Во втором случае (статический подход) рассматривается не один путь, а все найденные в CFG пути, выходящие из точки ввода помеченных данных и приходящие в точку размещения «защитной» проверки, что снимает ограничение динамического подхода. Построение CFG осуществляется в

инструменте Vine, он же используется для сокращения рассматриваемого кода: строится чоп между точкой ввода данных и «защитной» проверкой. Циклы и рекурсивные процедуры разворачиваются фиксированное число раз, с целью получить ациклический граф, который затем используется для построения логической формулы. Сложность полученной формулы $O(n^2)$, где n – число Vine-операторов в ациклическом CFG принципиально меньше, нежели суммарная сложность формул в итеративном динамическом анализе $O(2^b)$, где b – количество ветвлений. Однако ее решение для достаточно больших чопов длится неприемлемо долго.

Для компенсации указанных недостатков авторы предложили комбинированный подход: в трассе выполнения выбирается некоторый шаг *i*, шаги 0 – *i* используются для получения ограничений в рамках динамического подхода, а между шагом *i* и концом трассы строится чоп и применяется статический подход. Подбор номера шага предлагается делать итеративно, изначально выбирая *i* максимально близко к концу трассы. Если не удалось обнаружить уязвимость, вместо шага *i* берется *i-1* и т.д.

Практическая применимость работы была показана на модельных примерах, в качестве которых выступали системные и прикладные библиотеки ОС Windows: comctl32.dll, aspnet_filter.dll и др. В результате анализа были получены наборы входных данных, приводящие либо к переполнению буфера, либо к аварийному завершению работавшего процесса. Поиск уязвимости занимал, в зависимости от примера, от нескольких секунд до нескольких минут. В случае неудачи работа анализатора завершалась либо с отсутствием решения, либо остановкой из-за нехватки памяти.

Краткие выводы

У рассмотренного подхода есть как положительные, так и отрицательные стороны.

Положительным является то, что результат был показан на реальных, а не синтетических примерах. Обеспечивается это тем, что рассматривается относительно небольшой объем машинных команд, а именно подпоследовательность шагов трассы между вводом помеченных данных и внесенной в новой версии кода проверки, благодаря чему можно рассчитывать на достаточно невысокую сложность ограничений для символьных переменных.

Отрицательным является то, что невозможно целенаправленно искать уязвимости заданного типа. Более того, нет возможности как-либо автоматически интерпретировать смысл внесенных в код проверок, условие отдельно взятой проверки может быть никак не связано с найденными и исправленными разработчиками программы ошибками.

В свою очередь, точность статического анализа страдает от невозможности в полном объеме построить потоки управления и данных, что снижает вероятность обнаружения уязвимости.

Тем не менее, рассмотренная постановка задачи и метод ее решения представляют определенный интерес, поскольку позволяют оперативно, в течение нескольких минут, в полуавтоматическом режиме получать начальные данные, реализующие ошибки в бинарном коде (в случае, если анализ прошел успешно).

Для повторения результатов необходимо располагать: средством получения трасс машинных команд, анализом зависимостей по данным и управлению в машинном коде, RISC-подобным внутренним представлением и транслятором в это представление, разработанным методом построения ограничений, в данном случае – слабейшее предусловие, для данного внутреннего представления. Кроме того, потребуется решение технической задачи: использования результатов сравнения двух версий бинарного кода, полученных из сторонней утилиты.

5.4 Автоматический поиск уязвимостей с использованием исходного и бинарного кода

В работе [47] ставится задача автоматического получения эксплоита: набора начальных данных, обработка которых приводит к несанкционированному запуску программой шелл-кода.

Авторами был отмечен существенный принципиальный недостаток анализа программ на уровне исходного кода: эксплуатация уязвимости невозможна без знания особенностей инструментов компиляции. Анализ кода на языке высокого уровня может показать потенциальную уязвимость кода, но ее эксплуатация в работающей программе требует выполнения дополнительных условий, таких как специфическое расположение переменных в памяти, что, как правило, выходит за рамки языковых спецификаций. В качестве примера в данной статье рассматривается следующий фрагмент кода:

```
char src [12], dst [10];
strncpy (dst, src, sizeof(src));
```

На практике переполнение буфера может быть недостижимо, т.к. большинство современных компиляторов выравнивают данные по 16 байт.

В тоже время анализ одного только бинарного кода трудно масштабировать, его применение ограничивается приложениями и фрагментами кода в несколько тысяч машинных команд. Другой сложность является отсутствие информации о размере буферов памяти, ее приходится восстанавливать по бинарному коду, обрабатывающему эти буферы. На практике такое восстановление дает большую погрешность.

Сформулированная постановка задачи более жесткая, нежели выявление потенциальной ошибки: требуется автоматически построить эксплоит. Однако входными данными для анализа является Си-код, что формально не соответствует теме данного обзора. Тем не менее, предлагаемые технические приемы и эвристики стоит рассмотреть, поскольку они затрагивают общие проблемы поиска ошибок путем символьной интерпретации.

Основной идеей, предлагаемой авторами является комбинация анализа исходного и бинарного кода: поиск потенциальных ошибок происходит на уровне исходного кода, путем символьной интерпретации, полученные условия срабатывания каждой потенциальной ошибки дополняются условиями, полученными при анализе бинарного кода, если совокупность полученных условий разрешима, она позволяет получить набор входных данных, на которых эта уязвимость реализуется. Описанный подход последовательно выполняется в шесть этапов, итоговым результатом является работоспособный эксплоит.

- **Предварительная обработка.** Исходный код на языке Си компилируется в (1) исполнимый код B_{gcc}, используемый для построения эксплоита, и (2) LLVM-код B_{llvm}, используемый для поиска потенциальных ошибок.
- **Анализ размера буферов.** Одной из основных проблем символьной интерпретации является размер входных символьных данных. Неопределенный (произвольный) размер входных буферов приводит к быстрому «экспоненциальному взрыву» количества анализируемый путей. В рамках данной работы эта проблема решается эвристически: находится размер в байтах самого большого статического буфера в B_{llvm}, размер входных символьных данных фиксируется таким образом, что он превышает это значение на 10%, в предположении, что этого будет достаточно для срабатывания ошибок переполнения.
- **Поиск потенциальных ошибок.** Для каждого поддерживаемого типа ошибок (переполнение буфера, небезопасная форматная строка и т.п.) символьный интерпретатор B_{llvm} строит набор ограничений на траекторию выполнения и описание найденной потенциальной ошибки: в какой функции она срабатывает (функция срабатывания) и какие переменные она затрагивает.
- **Динамический анализ бинарного кода.** Для каждой потенциальной ошибки по полученным ограничениям строятся начальные данные, обеспечивающие попадание в точку срабатывания ошибки. B_{gcc} запускается на полученных входных данных; в момент достижения функции срабатывания, извлекается низкоуровневая информация: адреса размещения участвующих в реализации ошибки переменных, содержимое стека и текущего фрейма в частности, размещение адреса возврата.

– **Построение эксплоита.** Полученные на предыдущем шаге данные используются для составления дополнительных условий. Решение совокупной системы ограничений – эксплоит данной ошибки, позволяющий запустить шелл-код.
– **Проверка эксплоита.** Полученный эксплоит проверяется: происходит выполнение программы с динамическим онлайн-анализом на срабатывание ошибки, т.е. отслеживается запуск шелл-кода.

На этапе символьного выполнения авторы работы используют ряд эвристик, направленных на сокращение отслеживаемых путей выполнения. Помимо фиксированного размера входных данных, может использоваться фиксированный (конкретный) префикс, сокращающий количество символьный байт. Крайней формой этой эвристики выступает случай, когда известны данные, на которых программа аварийно завершается, AEG использует их для эффективного построения ограничений траектории и последующего построения работоспособного эксплоита.

Другой оригинальной эвристикой является обработка циклов. Несмотря на эвристики, связанные с входными данными, циклы способны порождать огромное количество анализируемых путей. Авторы предлагают давать больший приоритет тем траекториям, которые выполняют большее количество итераций, т.е. *вырабатывают цикл* (exhaust the loop) до конца. В основе этой эвристики лежит идея, что ошибка происходит на последних итерациях, когда программист ошибается в размерах буфера.

Еще одной важной проблемой, на которую указали авторы, является то, что из-за особенностей размещения данных в памяти эксплоит может оказаться нефункциональным, т.к. прежде его реализации произойдет аварийное завершение программы.

```
char *ptr = malloc(100);
char buf[100];
strcpy(buf, input); // переполнение
strcpy(ptr, buf); // использование ptr
return;
```

В приведенном примере переполнение буфера в третьей строке «испортит» адрес, хранимый переменной ptr. Следующая команда приведет к аварийной остановке программы и оператор return не будет выполнен. Избежать таких ситуаций можно путем добавления ограничений к общей системе уравнений: они должны уточнять содержимое входных данных так, что бы перезапись других переменных не приводила к преждевременному падению программы.

В качестве механизма эксплоита авторы используют два подхода: (1) выполнение кода, размещенного на стеке, (2) вызов функции из стандартной библиотеки языка Си.

Реализация описанного в статье подхода использовала доработанный авторами интерпретатор KLEE (реализованы собственные эвристики), для решения систем ограничений использовался решатель STP, анализ содержимого памяти во время динамического анализа использует отладчик gdb.

Работоспособность подхода демонстрируется на программах с открытым исходным кодом. Среди их числа присутствовали программы с достаточно большим размером: почтовый сервер exim и интерпретатор expect. Их размер составляет 241 и 458 тысяч LLVM-команд соответственно. Основным недостатком экспериментальных данных является то, что только два примера эксплоита получали символьные данные из сокета, в остальных 14 случаях источником символьных данных являлись локальные данные: главным образом, аргументы командной строки и переменные окружения.

5.5 Проект Mayhem

В работе [48], вышедшей в 2012 году, представлена методика поиска ошибок в бинарном коде и соответствующий программный инструмент. Ключевой особенностью предлагаемого подхода является то, что задача поиска ошибок решается исключительно смешанной интерпретацией бинарного кода анализируемой программы. Помимо того, в данной работе предложен работоспособный подход к анализу машинных команд, в которых символьные значения являются адресами («символьная» память).

Разработанный подход реализован в интерпретаторе, имеющем типовую архитектуру. Входными данными являются исполняемый код программы и спецификация, какие входные данные следует рассматривать как символьные. Все остальные входные данные получат конкретные значения, что положительно сказывается на скорости работы интерпретатора. Например, допустимо ограничить символьные данные, считая, что они поступают только из некоторого фиксированного порта.

Конкретное выполнение программы происходит под управлением системы Pin [17, 18], средствами которой перехватываются функции, осуществляющие ввод помеченных символьных данных, и отслеживается распространение этих данных на уровне бинарного кода. Символьное выполнение происходит в разработанном этими же авторами интерпретаторе, использующем внутреннее представление низкого уровня системы BAP [49]. В качестве решателя используется система Z3 [50].

Особенности подхода, представленные в данной публикации, затрагивают 3 вопроса: (1) эффективная поддержка достаточного количества анализируемых траекторий во время анализа, (2) модель «символьной» памяти, (3) эвристики назначения приоритетов анализируемым путям.

Одной из характерных проблем символьной интерпретации является то, что количество анализируемых путей растет экспоненциально. Каждый анализируемый путь требует поддержки в памяти контекста выполнения. В случае если интерпретатор использует *онлайн*-подход, все пути выполнения анализируются одновременно; в определенный момент доступная память заканчивается, и интерпретатор вынужден отбрасывать новые пути, не выполняя их анализ. *Оффлайн*-подход предполагает последовательный анализ путей: программа итеративно запускается, каждый новый запуск покрывает новый путь выполнения. Нехватки памяти не возникает, но время анализа значительно возрастает, поскольку каждый запуск требует повторного выполнения общего начального фрагмента пути. *Онлайн*-подход в качестве накладных расходов содержит время переключения между контекстами, поскольку перебор путей в глубину на практике несостоятелен – интерпретатор вынужден распределять процессорное время между анализируемыми путями. Авторы предлагают комбинировать подходы: при достижении порогового значения расхода памяти создается контрольная точка, интерпретация некоторых путей прекращается, их контексты символьного выполнения и конкретные входные данные сохраняются на диске. Если в процессе работы интерпретатора количество анализируемых путей сократилось, восстанавливается одна из контрольных точек. Восстановление заключается в воспроизведении конкретного выполнения по сохраненным водным данным до места создания контрольной точки и загрузка с диска в этот момент контекста символьного выполнения. Выгода такого подхода заключается в том, что при воспроизведении не происходит дорогостоящей символьной интерпретации.

Второй ключевой особенностью является поддержка символьных адресов памяти. Память представляется как отображение $\mu: I \to E$ 32-разрядного индекса i в выражение e. Таким образом, загрузка load(μ, i) вырабатывает символьное выражение e, расположенное по адресу i; выгрузка данных store(μ, i, e) в качестве результата даст новое состояние памяти $\mu[i \leftarrow e]$ в котором i отображено на выражение e.

Существует два крайних варианта работы с такой моделью памяти: либо конкретизировать (получать из решателя допустимые конкретные значения) символьные выражения в момент обращения к памяти, либо рассматривать всю память как символьную, что приводит к необходимости работы с большим контекстом символьного выполнения.

Подход, принятый в проекте Mayhem, предполагает, что загрузка load(μ, i) возвращает символьное выражение $\mu[i]$. Его конкретизация в общем случае будет представлять 2^{32} запросов к решателю, что неприемлемо трудоемко. Решение этой проблемы заключается в консервативном определении границ [L, U] в рамках которых может находиться адрес памяти, методом дихотомии.

Для случая записи данных в память по символьному адресу конкретизация производится незамедлительно.

Однако каждое обращение к символьной памяти остается трудоемкой операцией и может потребовать до 54 обращений к решателю. Авторы работы предлагают применять 5 дополнительных техник, направленных на улучшение производительности.

1. Авторами был реализован алгоритм VSA [6] для применения непосредственно во время интерпретации. Результат работы алгоритма – интервал с шагом S[L, U]. Применение VSA позволило сократить количество обращений к решателю на 70% при определении границ символьных адресов.
2. После получения интервала из VSA границы улучшаются через запросы к решателю. Поддерживается кэш потенциальных улучшений для интервалов. В случае попадания используется уже готовое решение.
3. Используемый решатель Z3 [50] позволяет инкрементально строить решение системы ограничений, что дает возможность оптимизировать многократно выполняющиеся запросы: постоянная часть системы оформляется в виде леммы, решение которой комбинируется с меняющимися условиями.
4. Запрос к решателю кодирует перебор допустимых адресов в виде дерева поиска.
5. Данные, расположенные в памяти, группируются таким образом, что их адреса описываются линейной функцией $\alpha n_i+\beta$, где n_i – номер элемента в группе. Выгода такой группировки объясняется тем, что на уровне бинарного кода активно используются таблицы: таблица переходов оператора switch, таблица трансляции символов при обработке строки и т.п.

При выборе путей (восстановление контрольных точек, предоставление процессорного времени) Mayhem использует приоритеты, назначаемые по трем эвристическим правилам: (1) больший приоритет дается путям обрабатывающим новый код, а не итерирующим циклы, (2) пути, в которых присутствуют символьные адреса памяти, имеют больший приоритет, (3) пути, в которых счетчик команд (регистр EIP) стал символьным, имеют наивысший приоритет.

Оценка полученных результатов показана на 29 реальных программах, работающих под управлением ОС Windows и Linux; одна из программ защищена запаковкой. Для всех программ были найдены критические уязвимости, в большинстве случаев – ранее известные, а также две уязвимости «нулевого дня». В среднем поиск эксплуатируемой уязвимости занимал несколько минут.

Поскольку для некоторых тестовых программ был доступен исходный код, было проведено сравнение производительности инструментов AEG и Mayhem: в среднем Mayhem проигрывает по скорости в 3,4 раза из-за того, что весь анализ происходит на уровне бинарного кода.

Уникальной особенностью данного инструмента является то, он успешно анализировал реальные программы достаточно большого размера: состоящие из более чем 500 000 машинных команд. Символьная интерпретация требовалась (и проводилась в приемлемое время) для тысяч машинных команд, максимальное количество таких команд было у программы aspell, оно составило 26 647, что примерно 3,83% от общего числа (696 275).

Последнее, что следует отметить – в данной работе не рассматривалась задача построения эффективного шелл-кода, способного преодолевать защиту современных операционных систем: рандомизацию адресного пространства и неисполняемый стек. Способы преодоления этих механизмов рассматриваются в другой работе этого же коллектива авторов [51].

5.6 Поиск уязвимостей в бинарном коде на основе полносистемной эмуляции

Система предназначена для проведения динамического анализа исполняемых файлов. Является свободно распространяемым ПО, что позволяет протестировать работу системы. На рисунке 16 представлена схема работы.

Анализируемая программа выполняется на эмуляторе QEMU. Виртуальная машина KLEE [31] отвечает за символьное исполнение. Некоторые данные, используемые программой, помечаются как символьные. Инструкции, результат выполнения которых зависит от символьных данных, транслируются в представление LLVM и выполняются на виртуальной машине KLEE. Если символьные данные влияют на результат инструкции перехода, состояние системы (состояние виртуального ЦПУ, состояние виртуальных устройств, состояние физической памяти) дублируется. Заметим, что размер одного состояния для ОС Ubuntu 10.10 64-bit составляет около 10 Мб. При этом вводятся ограничения на диапазон возможных значений символьных данных в зависимости от того, по какой ветке пошло выполнение. Далее каждая ветка анализируется отдельно.

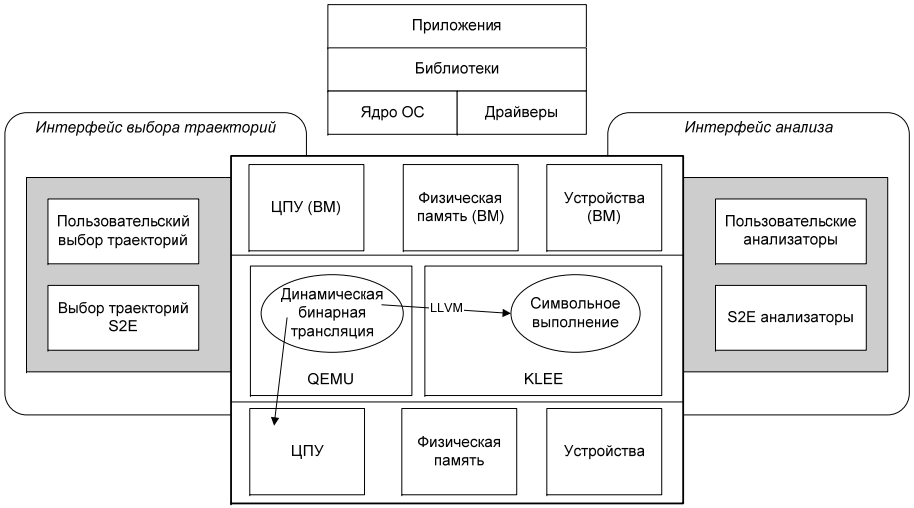

Рисунок 16 – Схема работы S2E

Ключевые особенности S2E:

- избирательное символьное исполнение (*selective symbolic execution*),
- возможность выбора уровня согласованности при выполнении (*execution consistency models*).

Программа рассматривается как суперпозиция возможных путей исполнения. Основная идея поясняется на примере. Пусть в программе есть лишь одно ветвление (if (x > 0) then ... else ...), а весь оставшийся код линеен. Такую программу можно рассматривать как суперпозицию двух путей исполнения: первый, где условие (x > 0) истинно, и второй, где это условие ложно (то есть, при $x \leq 0$). Для того чтобы проанализировать все пути, не нужно рассматривать всевозможные значения x, достаточно взять одно положительное и одно отрицательное.

Для выбора уровня согласованности вводятся следующие определения:

1. СИСТЕМА (system) – ПО компьютера (ОС, библиотеки, установленные приложения)
2. АНАЛИЗИРУЕМЫЙ УЧАСТОК (unit) – часть системы, подвергаемая анализу. Например, анализируемым участком может быть функция, библиотека, часть кода ОС.
3. ОКРУЖЕНИЕ (environment) – система без анализируемого участка.

Таким образом, СИСТЕМА = АНАЛИЗИРУЕМЫЙ УЧАСТОК + ОКРУЖЕНИЕ. Уровень согласованности определяет, какие ограничения накладываются на значения символьных данных при переходе к конкретным

значениям и наоборот (то есть при переходе от исполнения на конкретных значениях к символьному выполнению). Рассмотрим пример.

```
//unit code
int send_packet(buffer, size) {
      packet *pkt;
      status = alloc(&pkt, size);
      if (status==FAIL) {
            assert(pkt==NULL);
            return;
      }
...
      if (read_port(STATUS)==READY)
            if (!write_usb(pkt))
                  return FAIL;
}

//environment code
int write_usb(pkt) {
      if (usb_ready())
            return do_send(pkt);
      return 0;
}

int alloc(*ptr, size) {
...
}
```

В данном примере анализируется работа драйвера (unit code). Функция send_packet содержит вызов функции alloc ядра ОС и вызов функции write_usb некоторой библиотеки.

Если задана *полная* согласованность (согласованность данных на уровне АНАЛИЗИРУЕМОГО УЧАСТКА и ОКРУЖЕНИЯ – *strictly consistent concrete execution*), то S2E рассматривает только выполнение на конкретных данных (никакие данные не помечаются как символьные). Пусть задана *локальная* согласованность (*local consistency*). Данные, для которых ОКРУЖЕНИЕ генерирует конкретные значения, становятся символьными в случае попадания в АНАЛИЗИРУЕМЫЙ УЧАСТОК. Возвращаемое функцией alloc значение а также указатель pkt становятся символьными. При этом должны выполняться ограничения, наложенные ОКРУЖЕНИЕМ:

$$\text{if } (!\text{status}) \text{ then } \text{pkt} = \text{NULL}$$

Если символьные данные попадают из АНАЛИЗИРУЕМОГО УЧАСТКА в ОКРУЖЕНИЕ, то S2E отслеживает их распространение. *Приближенная*

согласованность (*overapproximate consistency*) отличается от локальной тем, что не содержит ограничений со стороны ОКРУЖЕНИЯ. Таким образом, могут быть проанализированы «несуществующие» пути исполнения.

Для того чтобы начать символьное выполнение, нужно либо пометить данные как символьные (*data-based selection*), либо указать участок кода (*code-based selection*, когда счетчик инструкций попадает в отмеченную область кода, данные этого участка помечаются как символьные).

Существует два способа тестирования: можно либо встроить вызовы S2E в код программы (эти вызовы транслируются в машинные инструкции), либо создать lua-скрипт, который будет запускать программу и нужные вызовы S2E при помощи гостевых утилит. В тексте скрипта нужно указать модули S2E, которые будут использоваться во время анализа. S2E автоматически детектирует загрузку/выгрузку модулей для ОС Windows (поддерживаются версии XPSP2, XPSP3, XPSP2-CHK, XPSP3-CHK, SRV2008SP2), детектирует появление BSOD, позволяет указать точки загрузки модулей для ОС Linux. S2E позволяет встраивать собственные обработчики событий (например, можно вызвать какую-нибудь функцию, когда счетчик инструкций виртуального процессора попадет на заданный адрес). В QEMU добавлена возможность подключения модельного PCI-устройства в гостевую систему.

Скорость работы S2E, несмотря на внесенные разработчиками изменения, сравнима со скоростью оригинальной версии QEMU. В тестах приложений ОС Windows наблюдалось аварийное завершение работы S2E вследствие нехватки оперативной памяти. Эта проблема решается путем увеличения размера файла подкачки. S2E теоретически позволяет распараллелить перебор путей на несколько ядер, но воспользоваться этой особенностью не удалось – через несколько секунд после начала анализа произошло падение S2E. Тест с анализом драйвера модельной сетевой карты не принес результатов – символьные данные не генерируются.

5.6.1 Виртуальная машина KLEE

KLEE – это виртуальная машина на базе компиляторной инфраструктуры LLVM с возможностью смешанного выполнения. Цель – обеспечить максимальное покрытие кода приложения и выявить возможные ошибки и уязвимости. В рамках проводимого анализа параллельно выполняются символические процессы (states), в каждом из которых реализуется одна траектория CFG программы. Каждая траектория характеризуется уникальным набором входных данных. Эффективная реализация создания процессов-потомков на каждом ветвлении программы позволяет анализировать большое количество путей одновременно.

Большая часть инструкций выполняется «естественным» образом. В качестве примера рассмотрим команду add. Если хотя бы один из операндов является символьным выражением, создается символьное выражение для суммы, и результат записывается на регистр.

Результат выполнения инструкции-ветвления зависит от истинности условного перехода. Проверка истинности осуществляется посредством STP [37]. Если условие перехода зависит от символьных данных, процесс клонируется, после чего анализируются обе траектории. При этом соответствующим образом изменяются наборы наложенных на траектории ограничений.

Если значение указателя является символьным выражением, и выполняется его разыменование, то соответствующий процесс клонируется столько раз, сколько конкретных значений может принимать символьное выражение.

Реализовано два алгоритма выбора траекторий для дальнейшего анализа. Первый задает траекторию случайным образом. Второй алгоритм выбирает траекторию, которая с наибольшей вероятностью покроет ранее не исполнявшийся код (используются эвристики).

В KLEE к STP-запросам применяются оптимизации, позволяющие уменьшить время обработки. Первая – это декомпозиция запроса на непересекающиеся подзапросы (в соответствии с используемыми символьными выражениями). Вторая оптимизация заключается в кэшировании карты, которая каждому выполненному запросу ставит в соответствие его результат. В карту попадают только те запросы, для которых STP генерирует контрпример. При обработке очередного запроса в карте ищутся подмножества и надмножества данного набора ограничений. Используются следующие соображения:

1. Если некоторое подмножество набора ограничений противоречиво, этот набор противоречив,
2. Если некоторое надмножество набора ограничений имеет решение, набор разрешим,
3. Если некоторое подмножество набора ограничений имеет решение, велика вероятность того, что набор разрешим.

Взаимодействие программы с окружением (ОС, пользователь) осуществляется разными способами – чтение из файла и запись в файл, отправка и получение сетевых пакетов, использование командной строки и системных переменных. В результате чтения данных из файла появляются символьные данные, для которых нет никаких ограничений. Для обработки подобных случаев аналитик добавляет свой код (на языке Си) в модель KLEE.

Приложения получают большое количество информации из файловой системы – данные и размеры файлов, права доступа к ним и т. д. Если данные считываются из конкретных файлов и директорий, то возвращаются конкретные значения. Если же осуществляется чтение из файла, который

потенциально содержит произвольные данные (например, файл с входными данными программы), то возвращаются символьные данные. В KLEE поддерживается символьная файловая система, состоящая из одной директории и N (входной параметр анализа) «символьных» файлов. Также необходимо задать максимальный размер символьного файла. В качестве примера рассмотрим упрощенную модель KLEE для функции read:

```
1 : ssize_t read(int fd, void *buf, size t count) {
2 :   if (is invalid(fd)) {
3 :         errno = EBADF;
4 :         return -1;
5 :   }
6 :   struct klee fd *f = &fds[fd];
7 :   if (is concrete file(f)) {
8 :         int r = pread(f->real fd, buf, count,
f->off);
9 :         if (r != -1)
10:               f->off += r;
11:         return r;
12:   } else {
13:   /* sym files are fixed size: don't read beyond the
end. */
14:         if (f->off >= f->size)
15:               return 0;
16:         count = min(count, f->size - f->off);
17:         memcpy(buf, f->file data + f->off, count);
18:         f->off += count;
19:         return count;
20:   }
21: }
```

Если значение файлового дескриптора fd не является символьным, то программе предоставляется доступ к реально существующему файлу. Если же fd – символьная переменная, то производится копирование данных из соответствующего символьного файла в буфер. Если на данные, считанные из символьного файла, впоследствии будут наложены ограничения, то при повторном вызове read с тем же файловым дескриптором, эти ограничения аннулируются (при условии, что в символьный файл ничего не записывалось).

Виртуальная машина KLEE тестировалась на наборе инструментов Coreutils. Результаты приведены в таблице 2. Заметим, что при вычислении полноты покрытия код библиотек не учитывается.

Покрытие кода	Количество инструментов
100 %	16
90 - 100 %	38
80 - 90 %	22
70 - 80 %	8
60 - 70 %	6

Таблица 2 – Покрытие кода инструментов Coreutils при анализе на ВМ KLEE.

6 Комбинированный анализ

Каждый из инструментов статического и динамического анализа имеет свои сильные и слабые стороны. Соответственно, аналитик должен использовать эти инструменты в тандеме. Например, инструменты статического анализа способны обнаруживать ошибки, которые пропускаются инструментами динамического анализа, потому что инструменты динамического анализа фиксируют ошибку лишь в случае, если во время тестирования ошибочный фрагмент кода исполняется. С другой стороны, инструменты динамического анализа обнаруживают программные ошибки конечного работающего процесса. Динамический подход может использоваться для подтверждения ошибок, найденных статическими анализаторами. В динамике известны значения операндов инструкций передачи управления с косвенной адресацией. Эта информация позволяет дополнить CFG, построенный статическим анализатором.

6.1 BitBlaze

Проект BitBlaze ведется на протяжении нескольких лет в Университете Беркли, США. В рамках проекта разрабатываются различные инструменты для анализа кода исполняемых файлов. В работе [19] предложен подход к согласованному применению этих инструментов на основе трех главных компонент.

- Vine – поддержка статического анализа (система с открытым исходным кодом),
- TEMU – динамический анализ (система с открытым исходным кодом, рассмотрена в разделе 4.3),
- Rudder – интерпретатор смешанного выполнения (см. раздел 6.1 BitBlaze: Rudder).

6.1.1 Vine

Компонент VINE обеспечивает инфраструктуру для проведения статического анализа исполняемого кода. Осуществляет трансляцию бинарного кода в язык промежуточного представления (IL) и предоставляет набор утилит для анализа программы на уровне этого представления. Схема компонентов изображена на рисунке 17.

Рисунок 17 – Схема компонентов Vine.

Трансляция бинарного кода в промежуточное представление Vine происходит в два этапа. Исполняемый код дизассемблируется, результат переводится в промежуточное представление VEX (см. раздел 4.1 Valgrind). Количество инструкций сокращается благодаря декомпозиции на базовые конструкции промежуточного представления. Затем промежуточное представление VEX транслируется в промежуточное представление Vine, содержащее явное описание побочных эффектов. Например, для инструкции:

```
add eax,0x2
```

генерируется следующее промежуточное представление Vine:

```
tmp1 = EAX; EAX = EAX + 2;
//eflags calculation
CF:reg1_t = (EAX<tmp1);
tmp2 = cast(low, EAX, reg8_t);
PF =(!cast(low,
((((tmp2>>7)^(tmp2>>6))^((tmp2>>5)^(tmp2>>4)))^
(((tmp2>>3)^(tmp2>>2))^((tmp2>>1)^tmp2)))), reg1_t);
AF = (1==(16&(EAX^(tmp1^2))));
ZF = (EAX==0);
SF = (1==(1&(EAX>>31)));
OF = (1==(1&(((tmp1^(2^0xFFFFFFFF))&(tmp1^EAX))>>31)));
```

Кроме того, с помощью Vine можно построить промежуточное представление по трассам выполнения, полученным в симуляторе TEMU.

Построенное внутренне представление используется различными способами:

- доступна интерпретация,
- построение CFG, для разрешения целевых адресов переходов в инструкциях с косвенной адресацией применяется VSA,
- построение графа зависимостей (PDG),
- анализ потока данных, а именно распространение констант, удаление мертвого кода, анализ живых переменных (применяется VSA),
- возможность *символьного выполнения* программы,
- вычисление слабейших предусловий (WP),
- генерация Си-кода на основе промежуточного представления Vine.

6.1.2 Согласованное применение инструментов

В статье [52] описан подход согласованного применения нескольких программных инструментов, реализующих разные подходы к анализу, для поиска ошибок в бинарном коде. Наличие каких-либо отладочных данных не предполагается, подход работоспособен для «чистого» бинарного кода.

Основная идея заключается в последовательном применении динамического анализа, статического анализа и смешанного символьного выполнения (Рис 18).

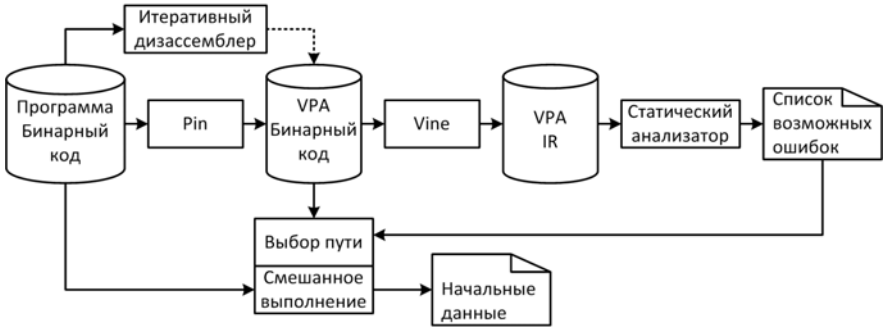

Рисунок 18 – Схема применения инструментов, предложенная авторами BitBlaze.

На первом этапе строится статическое представление программы, полученное по ее бинарному коду. Главной проблемой, препятствующей построению, является наличие в типичном бинарном коде значительного количества переходов с косвенной адресацией. Такие переходы не позволяют в статике восстановить структуру программы. Для решения этой проблемы авторы статьи предлагают строить статическое представление по результатам выполнения программы на типовых начальных данных. Трассы выполнения суммарно дают некоторое начальное покрытие, которое затем итеративно улучшается путем статического дизассемблирования кода в местах незадействованных условных переходов с явной адресацией. Такое дизассемблирование работает в определенных ограничениях, но, как утверждается в статье, в достаточном количестве случаев заканчивается успешно, т.к. выполнение возвращается в уже покрытый код прежде, чем встречается машинная команда, дизассемблирование которой в статике невозможно или затруднительно.

Для трассировки программы используется уже рассмотренный инструмент Pin, с помощью которого отслеживают целевые адреса переходов; итеративный дизассемблер разрабатывался непосредственно авторами статьи.

Восстановленная структура программы выражается в виде явного МП-автомата (VPA) [53]. Состояния такого автомата соответствуют базовым блокам и содержат соответствующий код, переходы показывают поток управления, включая явно выделенные вызовы и возвраты процедур (функций). Такая форма представления оказывается удобной для последующего статического анализа, который выполняется после перевода машинных команд во промежуточное представление с помощью Vine.

На втором этапе анализа статический анализатор строит консервативные оценки для используемых в программе буферов памяти. Для отслеживания работы с памятью используется распространенный поход моделирования абстрактных ячеек (*Aloc*s). Одной из особенностей предложенного подхода является моделирование стека в виде единого пространства, а не в виде

отдельных, никак не связанных между собой фреймов. Такой подход позволят более точно оценивать последствия выхода за пределы буферов памяти, расположенных на стеке.

К сожалению, чекеры, ищущие ошибки в таком промежуточном представлении, в публикациях освещены не были. Исходя из описанных абстрактных доменов, можно заключить, что данный подход позволяет обнаруживать такие ошибки, как выход за пределы буфера и разыменование нулевого указателя.

После того, как в результате статического анализа был получен список потенциальных ошибок, в символьном интерпретаторе можно попытаться получить набор начальных данных, на котором будет срабатывать истинная ошибка. Интерпретатор реализует смешанный механизм выполнения: символьная интерпретация затрагивает только те вычисления, в которых участвуют символьные данные, остальные команды выполняются непосредственно. Можно предположить, что реализация интерпретатора базируется на предыдущих разработках коллектива BitBlaze, таких как MineSweeper и Rudder.

Предложенный подход не позволяет использовать третий этап анализа для фильтрации ошибок, т.к. ложные срабатывания статического анализатора будут бесконечно долго обрабатываться. Однако для истинных ошибок, корректно ставить задачу получения начальных данных. Авторами BitBlaze предлагается ряд эвристик, позволяющих ускорить интерпретацию за счет стратегии выбора анализируемых путей.

В случае, когда происходит ветвление, для каждого базового блока, получающего управления, вычисляются две численные характеристики *(d, s)*. Первая – длина наикратчайшего пути в явном МП-автомате, от данного базового блока до блока, в котором реализуется ошибка. Вторая характеристика – совокупное количество машинных команд в базовых блоках обратного слайса (критерий строится для рассматриваемой ошибки), в которые можно попасть из данного блока.

Стратегия заключается в выборе базового блока с более коротким путем, но с большим количеством команд. Как можно легко заметить, эти две численные характеристики противопоставлены. Эвристика предлагает формулу, уравновешивающую влияния этих двух характеристик. Выбор между блоками с характеристиками (d_1, s_1) и (d_2, s_2) происходит, исходя из знака выражения

$$x = d_1(1 + ln(1+s_2)) - d_2(1 + ln(1 + s_1))$$

В случае отрицательного значения выбирается первый блок, в случае положительного – второй. Помимо того, в окончательное решение целенаправленно вносится элемент случайности. В каких именно ситуациях это происходит, в статье описано не было. Вычисляется функция

$$r(x) = \frac{1}{1 + e^{-kx}},$$

ее значение сравнивается со случайным числом, равномерно распределенным на отрезке [0, 1]. Выбор происходит исходя из того, больше или меньше это случайное значения *r(x)*. Параметр *k* был экспериментально выбран равным *ln(1.001)*. Фактический смысл такого подхода в том, что значение *r(x)* отличается от ½ только при достаточно больших значениях *x*, в остальных случаях происходит случайный выбор пути с равной вероятностью.

Данная стратегия выбора используется в тех случаях, когда выполняются переходы по прямым ребрам. Когда встречается ветвление с обратным ребром, то применяется другая стратегия, представляющая собой случайный перебор коротких шаблонов выполнения цикла. Шаблоны строятся следующим образом. Для данного обратного ребра ищется сильно связанный подграф, все ацикличные пути в этом графе нумеруются. Таким образом, любое выполнение цикла можно задать как последовательность номеров этих ацикличных путей. После пяти «пробных» выполнений цикла используются полученные шаблоны: зафиксированные последовательности допустимых ацикличных путей. Длина шаблона – количество итераций цикла. В последующих выполнениях цикла в половине случаев путь выбирается случайно, четверть случаев использует шаблоны единичной длины, восьмая часть случаев – шаблоны длины два и т.д. Таким образом, чем больше длина шаблона, тем меньше вероятность его применения.

Экспериментальные результаты показали условную применимость предложенного подхода: пример максимального размера состоял из 4015 машинных команд. В их число входит код самого модельного примера и код стандартной библиотеки, в качестве которой специально была использована компактная библиотека dietlibc. Сами примеры представляют собой вручную укороченный код из набора тестов Зистера [54], содержащего типичные уязвимости сетевых программных серверов.

Время работы статического анализатора во всех случаях не превышало нескольких секунд, время работы символьного интерпретатора было ощутимо улучшено: вместо четырех случаев превышения порогового времени ожидания (6 часов) был получен только один такой случай при целенаправленном выборе путей. В среднем, выдачи результата, набора входных данных, на котором реализуется ошибка, приходилось ждать от 2 до 30 секунд при направленной символьной интерпретации и от 2 до 60 секунд при ненаправленной. Несмотря на то, что в среднем на анализ одного пути выполнения стало уходить несколько большее время, количество рассматриваемых путей сократилось. В зависимости от примера это сокращение составляло от 2 до 30 раз.

Результатом описанной работы можно считать количественное улучшение анализа, но не расширение области применимости. Для достаточно больших

объемов машинного кода работоспособность подхода представляется сомнительной.

6.2 ROSE

В статье [55] рассматриваются два инструмента анализа кода на базе инфраструктуры компилятора ROSE – Compass (анализ исходного кода) и BinQ (анализ бинарного кода). Схема проведения анализа представлена на рисунке 19.

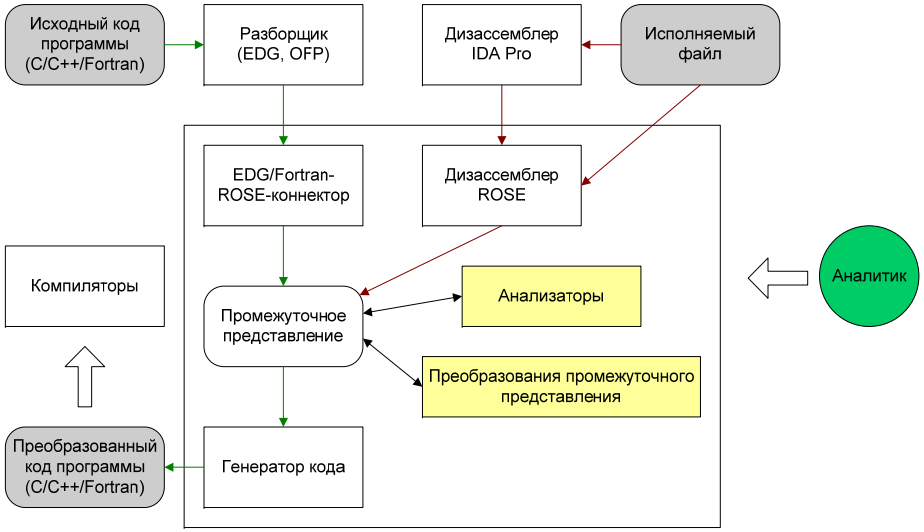

Рисунок 19 – Инфраструктура компилятора ROSE.

ROSE представляет собой компиляторную инфраструктуру, в рамках которой проводится анализ кода. Необходимо построить промежуточное представление программы. В случае исходного кода применяется разборщик (Edison Design Group (EDG, [56]) C++ front-end для Си/Си++, Open Fortran Parser (OFP) для Fortran), в случае исполняемого файла – дизассемблер (IDA Pro, а также собственный дизассемблер ROSE). В рамках анализа бинарного кода поддерживаются архитектуры x86, PowerPC, ARM, а также форматы ELF, PE, LE, NE, MS-DOS.Информация, полученная при разборе исходного (и/или бинарного) кода программы, используется для построения абстрактного синтаксического дерева (abstract syntax tree, AST). Пример AST изображен на рисунке 20. Промежуточное представление ROSE включает AST, таблицы символов, директивы препроцессора, комментарии и т. д. Пользователь включает в промежуточное представление свой анализатор, после чего по модифицированному промежуточному представлению строится код.

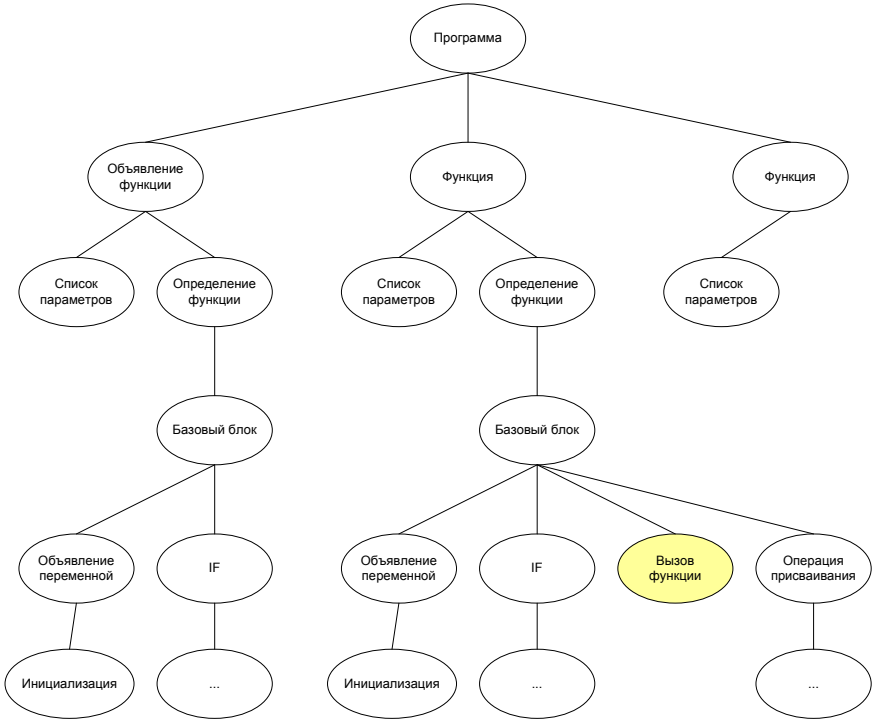

Рисунок 20 – Пример AST.

Компонента Compass (инструмент для анализа исходного кода) позволят создавать свои собственные инструменты проверки (checkers), а также использовать уже готовые (их около 100). Инструмент проверки представляет собой набор ограничений в терминах AST. Каждое ограничение проверяется отдельно. Также можно задавать ограничения в терминах CFG, SDG, графа вызовов, графа потока данных. В качестве примера приводится следующий инструмент проверки:

```
void visit(SgNode* node)
{
      SgCastExp* cast = isSgCastExp(node);
      if (cast && cast->get_cast_type() ==
SgCastExp::e_C_style_cast)
            output->addOutput(new DetectorOutput(node));
}
```

В данном примере проверяется следующее правило – не использовать приведение типов языка Си. Функция visit вызывается для каждого узла AST

анализируемой программы. Если узел синтаксического дерева задает приведение типов e_C_style_cast, то генерируется сообщение о найденном дефекте. Поскольку промежуточное представление ROSE содержит информацию о типах, будут найдены явные и неявные преобразования. Авторы публикации утверждают, что разработаны инструменты проверки для поиска ошибок, связанных с переполнением буфера, разыменованием нулевого указателя, использованием «небезопасных» функций (например, sprintf()).

Compass сохраняет сообщения об ошибках в текстовый файл. Кроме того, созданы модули расширения для Eclipse, QRose, Vim, Emacs.

Компонента BinQ (инструмент для анализа исполняемых файлов) позволяет использовать те же инструменты проверки, что и Compass – отличие лишь в том, что промежуточное представление генерируется при помощи дизассемблера. Дизассемблер ROSE в настоящий момент поддерживает архитектуры x86 и ARM.

Авторы публикации упоминают о том, что в ROSE добавлена возможность проведения динамического анализа исполняемых файлов (в основу анализа положена инфраструктура Intel Pin [17, 18]).

Краткие выводы

Сайт, на который ссылаются авторы и с которого можно скачать описанные инструменты (www.roseCompiler.org), недоступен, из-за чего практических исследований провести не удалось. Непонятно, какого рода ограничения можно описывать при создании инструментов проверки.

7 Заключение

В открытом доступе нет готовых программных сред и инструментов, позволяющих искать ошибки реализации в бинарном коде достаточно больших систем. Среди рассмотренных работ наилучшие результаты показывают системы S2E и Avalanche, использующие современную программную инфраструктуру: бинарный транслятор Valgrind, симулятор QEMU, интерпретатор KLEE.

Не решенной по-прежнему остается задача экспоненциального роста возможных путей выполнения, возникающая при символьной интерпретации кода. Анализ достаточно больших фрагментов бинарного кода, рассмотренными инструментами либо крайне затруднителен, либо невозможен.

Перспективными направлениями дальнейших исследований представляются следующие подходы к поиску ошибок.

Интеграция среды анализа бинарного кода TrEx и статического анализатора исходного кода Svace, аналогично тому, организован инструмент

Codesurfer/x86. Инструмент Svace осуществляет автоматический поиск ошибок в программах, написанных на Си/Си++; в качестве внутреннего представления, над которым работают анализаторы, является LLVM, получаемый с помощью транслятора clang [57]. Это внутреннее представление будет выступать связующим звеном (Рис 21) между двумя системами. Восстанавливаемое в среде TrEx статическое представление основано на расширенном межпроцедурном графе потока управления. Вершины графа содержат линейные последовательности машинных команд, которыми могут являться как команды целевой машины, так и внутреннее низкоуровневое представление Pivot. Предметом исследований будет являться способ трансляции, позволяющий получать код LLVM, максимально близкий к получаемому из исходного кода. Риски заключаются в том, что не устраненные артефакты трансляции могут мешать работе анализаторов Svace. Целесообразным представляется вести разработку транслятора на безе RevGen – транслятора с открытым исходным кодом, являющегося частью системы S2E.

Рисунок 21 – Интеграция инструментов динамического и статического анализа с целью поиска ошибок в бинарном коде.

Второе перспективное направление исследований связано с механизмом смешанного, символьно-конкретного выполнения, применяемого для поиска ошибок в исполняемом коде.

Интерес к задачам поиска ошибок, развитие компиляторных технологий и рост вычислительных мощностей привели к тому, что было разработано большое количество программных инструментов, основанных на механизме смешанного выполнения: KLEE, Mayhem, Avalanche и др.

Получение качественных результатов известных публикаций обеспечено развитыми инфраструктурными средствами, развиваемыми, как правило, в рамках отдельных проектов. Показателен пример проекта BitBlaze, где на основе компонент статического (Vine) и динамического (Temu) анализа был разработан целый набор инструментов, решающих различные прикладные задачи: Rudder, Minesweeper, Bitfuzz, Fuzzball.

Опыт успешных проектов позволяет перечислить их основные компоненты. Виртуальная машина, как правило, QEMU, используется в качестве среды конкретного выполнения. В нее интегрируется механизм символьной интерпретации, который задействуется для обработки символьных данных. Разделение символьного и конкретного выполнения осуществляется через механизм анализа помеченных данных. Построение системы уравнений происходит над промежуточным низкоуровневым представлением, в которое транслируется целевой машинный код. Для решения систем уравнений используется SMT-решатель, наиболее популярные – STP и Z3. Исследовательская деятельность сконцентрирована в следующих вопросах: построение системы уравнений (в том числе в отражении в этой системе критерия ошибочного состояния), и способ решения проблемы «экспоненциального взрыва».

В настоящий момент среда TrEx уже содержит инфраструктурные средства для эффективной организации символьной интерпретации, что позволяет незамедлительно начать исследовательские работы.

Литература

[1] G. Balakrishnan, T. Reps, D. Melski, T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. // In Verified Software: Theories, Tools, Experiments, Springer-Verlag, 2007.

[2] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, Dawson Engler. A Few Billion Lines of Code Later. // Communications of the ACM, February 2010, vol. 53, no. 2, pp. 66 – 75.

[3] Klocwork Insight http://www.klocwork.com/products/insight/index.php

[4] Арутюн Аветисян, Алексей Бородин. Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок. // Труды Института системного программирования РАН, том 21, 2011 г. Стр. 39-54.

[5] Gogul Balakrishnan, Radu Gruian, Thomas Reps, Tim Teitelbaum. CodeSurfer/x86 – A Platform for Analyzing x86 Executables. // Compiler Construction, 14th International Conference, 2005, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh. LNCS 3443, pp. 250-254.

[6] Gogul Balakrishnan, Thomas Reps. Analyzing Memory Accesses in x86 Executables. // Compiler Construction, 13th International Conference, 2004, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona. LNCS 2985, pp. 5-23.

[7] Gogul Balakrishnan, Thomas Reps. DIVINE: DIscovering Variables IN Executables. // Proceedings of Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, pp. 1-28.

[8] Thomas Reps, Gogul Balakrishnan, Junghee Lim. Intermediate-Representation Recovery from Low-Level Code. // Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina. pp. 100-111

[9] Thomas Reps, Stefan Schwoon, Somesh Jha. Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis. //

[10] «WPDS++ User Manual», October 5, 2005, http://research.cs.wisc.edu/wpis/wpds/wpds++/manual.pdf

[11] Stefan Schwoon. Model-Checking Pushdown Systems. // Диссертационная работа PhD, Lehrstuhl fur Informatik VII der Technischen Universitat Munchen.

[12] William R. Bush, Jonathan D. Pincus, David J. Sielaff. A static analyzer for finding dynamic programming errors. // Software: Practice and Experience, 2000, 30, pp. 775–802

[13] Nicholas Nethercote. Dynamic Binary Analysis and Instrumentation or Building Tools is Easy. // A dissertation submitted for the degree of Doctor of Philosophy at the University of Cambridge, 2004.

[14] Valgrind, http://valgrind.org/

[15] Nicholas Nethercote, Alan Mycroft. Redux: A dynamic dataflow tracer. // In Proceedings of the Third Workshop on Runtime Verification (RV'03), Boulder, Colorado, USA, July 2003.

[16] J. Newsome, D.Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. // In Proceedings of the Network and Distributed Systems Security Symposium, Feb 2005.

[17] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, Moshe Bach. Dynamic Program Analysis of Microsoft Windows Applications. // International Symposium on Performance Analysis of Software and Systems (ISPASS). White Plains, NY. April 2010.

[18] «Pin 2.11 User Guide», http://software.intel.com/sites/landingpage/pintool/docs/49306/Pin/html/index.html

[19] Dawn Song, David Brumley, HengYin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam,

Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. // International Conference on Information Systems Security, 2008, LNCS 5352, pp. 1–25.
[20] Charlie Miller. Crash analysis with BitBlaze. Independent Security Evaluators, July 26, 2010.
[21] Bin Xin, William N. Summer, and Xiangyu Zhang. Efficent program execution indexing. // Proceedings of the ACM Conference on Programming Language Design and Implementation, pages 238–248, 2008.
[22] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. // In Proceedings of the Fourth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS), Dublin, Ireland, June 2009.
[23] SDL MiniFuzz File Fuzzer, http://www.microsoft.com/en-us/download/details.aspx?id=21769
[24] Peach Fuzzing Platform, http://peachfuzzer.com/
[25] IOCTL Fuzzer, http://code.google.com/p/ioctlfuzzer/
[26] Sylvester Keil, Clemens Kolbitsch. Stateful Fuzzing of Wireless Device Drivers in an Emulated Environment. // White Paper, Secure Systems Lab, http://www.iseclab.org/papers/fuzz_qemu.pdf, (2009/05/17), September 2007.
[27] Tielei Wang, Tao Wei, Guofei Gu, Wei Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. // Proceedings of the 31st IEEE Symposium on Security and Privacy, Oakland, California, USA, May 2010. pp. 497—512.
[28] James C. King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), pp. 385-394.
[29] Patrice Godefroid, Michael Y. Levin, David Molnar. Automated whitebox fuzz testing. // In Proceedings of Network and Distributed Systems Security (NDSS 2008).
[30] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. // Proceedings of the ACM Conference on Computer and Communications Security, Oct. 2006.
[31] Cristian Cadar, Daniel Dunbar, Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. // USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008) December 8-10, 2008, San Diego, CA, USA.
[32] The LLVM Compiler Infrastructure Project.
http://llvm.org
[33] Carnegie Mellon CyLab
http://www.cylab.cmu.edu/
[34] Dependable Systems Lab
http://dslab.epfl.ch/

[35] David Brumley, Cody Hartwig, Zhenkai Liang James Newsome, Dawn Song, Heng Yin. Automatically Identifying Trigger-based Behavior in Malware. //Book chapter in "Botnet Analysis and Defense", Editors Wenke Lee et. al., 2007.
[36] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang James Newsome, Pongsin Poosankam, Dawn Song, Heng Yin. BitScope: Automatically Dissecting Malicious Binaries. // CS-07-133, School of Computer Science, Carnegie Mellon University, March 18, 2007.
[37] STP Constraint Solver, https://sites.google.com/site/stpfastprover/
[38] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song, Input generation via decomposition and re-stitching: Finding bugs in malware. // In proc. of the ACM Conference on Computer and Communications Security, Chicago, IL, October 2010
[39] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. // In proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems, London, UK, Mar. 2012.
[40] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. // In NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium, pages 391–408, San Diego, California, USA, Mar. 2010.
[41] Vitaly Chipounov, Volodymyr Kuznetsov, George Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. // Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, March 5–11, 2011, Newport Beach, California, USA.
[42] Ildar Isaev, Denis Sidorov. The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs. // Programming and Computer Software, 2010, Vol. 36, No. 4, pp. 225–236.
[43] Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: Techniques and implications. // Proceedings of the 2008 IEEE Symposium on Security and Privacy (2008).
[44] eEyE's Binary Diffing Suite.
http://www.eeye.com/resources/security-center/research/tools/eeye-binary-diffing-suite-ebds
[45] The BitBlaze binary analysis project.
http://bitblaze.cs.berkeley.edu, 2007.
[46] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest pre-conditions. // In Proceedings of the IEEE Computer Security Foundations Symposium, 2007.
[47] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. // Proceedings of the Network and Distributed System Security Symposium, Feb. 2011.

[48] Alexandre Rebert Sang Kil Cha, Thanassis Avgerinos and David Brumley. Unleashing mayhem on binary code. // IEEE Symposium on Security and Privacy, May 2012.
[49] I. Jager, T. Avgerinos, E. J. Schwartz, and D. Brumley. BAP: A binary analysis platform. // In Proc. of the Conference on Computer Aided Verification, 2011.
[50] De Moura, Leonardo, and Nikolaj Bjørner. Z3: An efficient SMT solver. // Tools and Algorithms for the Construction and Analysis of Systems (2008): 337-340.
[51] Edward J. Schwartz , Thanassis Avgerinos , David Brumley. Q: Exploit Hardening Made Easy // Proceedings of the 20th USENIX conference on Security, p.25-25, August 08-12, 2011, San Francisco, CA
[52] Domagoj Babic, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-Directed Dynamic Automated Test Generation. // Proceedings of the International Symposium on Software Testing and Analysis, 2011. pp. 12—22.
[53] Rajeev Alur, P. Madhusudan. Adding nesting structure to words. // Journal of the ACM, Volume 56 Issue 3, May 2009, Article No. 16. pp. 1—43.
[54] M. Zister, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. // ACM SIGSOFT Software Engineering Notes, Volume 29 Issue 6, Nov. 2004, pp. 97—106.
[55] Daniel Quinlan, Thomas Panas. Source Code and Binary Analysis of Software Defects. // CSIIRW ’09, April 13-15, Oak Ridge, Tennessee, USA.
[56] Edison Design Group, http://www.edg.com/index.php
[57] clang: a C language family frontend for LLVM. http://clang.llvm.org/