ПРОГРАММИРОВАНИЕ, 2021, № 6, с. 62–80

АНАЛИЗ ДАННЫХ

УДК 004.9

ПОИСК УЯЗВИМОСТЕЙ НЕБЕЗОПАСНОГО ИСПОЛЬЗОВАНИЯ ПОМЕЧЕННЫХ ДАННЫХ В СТАТИЧЕСКОМ АНАЛИЗАТОРЕ SVACE

© 2021 г. А. Е. Бородин[a,*], А. В. Горемыкин[a, b,], С. П. Вартанов[a,***], А. А. Белеванцев[a, b,****]**

[a] *Институт системного программирования им. В.П. Иванникова РАН, 109004 Москва, ул. А. Солженицына, д. 25, Россия*

[b] *Московский государственный университет имени М.В. Ломоносова, 119991 Москва, Ленинские горы, д. 1, Россия*

**E-mail: alexey.borodin@ispras.ru*

***E-mail: alexey.goremykin@ispras.ru*

****E-mail: svartanov@ispras.ru*

*****E-mail: abel@ispras.ru*

Поступила в редакцию 05.07.2021 г.
После доработки 16.07.2021 г.
Принята к публикации 22.07.2021 г.

В статье рассматривается поиск ошибок помеченных данных в исходном коде программ, т.е. ошибок, вызванных небезопасным использованием данных, полученных из внешних источников, которые потенциально могут быть изменены злоумышленником. В качестве основы использовался межпроцедурный статический анализатор Svace. Анализатор осуществляет как поиск дефектов в программе, так и поиск подозрительных мест, в которых логика программы может быть нарушена. Целью является найти как можно больше ошибок при приемлемой скорости и низком уровне ложных срабатываний (<20–35%). Для поиска ошибок Svace с помощью компилятора строит низкоуровневое типизированное промежуточное представление, которое подается на вход основному анализатору SvEng. Анализатор строит граф вызовов, после чего выполняет анализ на основе резюме. При таком анализе функции обходятся в соответствии с графом вызовов, начиная с листьев. После анализа функции создается ее резюме, которое затем будет использовано для анализа инструкций вызова. Анализ имеет как высокую скорость, так и хорошую масштабируемость. Внутрипроцедурный анализ основан на символьном выполнении с объединением состояний в точках слияния путей. Для отсеивания несуществующих путей для некоторых детекторов может использоваться SMT-решатель. При этом SMT-решатель вызывается, только если есть подозрение на ошибку. Анализатор был расширен возможностью поиска дефектов, связанных с помеченными данными. Детекторы реализованы в виде плагинов по схеме источник-приемник. В качестве источников используются вызовы библиотечных функций, получающих данные извне программы, а также аргументы функции main. Приемниками являются обращение к массивам, использование переменных как шага или границы цикла, вызов функций, требующих проверенных аргументов. Реализованы детекторы, покрывающие большинство возможных типов уязвимостей, для непроверенных целых чисел и строк. Для оценки покрытия использовался проект Juliet. Уровень пропусков составил от 46.31% до 81.17% при незначительном количестве ложных срабатываний.

DOI: 10.31857/S0132347421060030

1. ВВЕДЕНИЕ

В статье описывается реализация поиска ошибок помеченных данных с помощью статического анализатора Svace [1–4].

К наиболее важным особенностям анализатора Svace можно отнести следующее.

• Межпроцедурный анализ на основе резюме, при котором функции обходятся по графу вызовов, начиная с листьев. Каждая функция анализируется только один раз.

• Неконсервативный анализ. За счет отказа от консервативности анализ получает более высокую точность и производительность.

• Анализ внутри одной функции основан на анализе значений. Анализ отслеживает значения переменных и ячеек памяти и ассоциирует большинство свойств со значениями переменных.

В разд. 2 описывается, какие виды ошибок мы будем искать, в разд. 3 и 4 описывается устройство статического анализатора Svace и модуля

SvEng соответственно. Разд. 5 посвящен реализации анализа помеченных данных, и разд. 6 содержит оценку результатов для проектов Juliet и Tizen 6.

2. ПОМЕЧЕННЫЕ ДАННЫЕ

В статье рассматриваются ошибки небезопасного использования данных, не контролируемых программой, т.е. полученных из внешних источников. Такие данные могут быть изменены злоумышленником. Если эти данные используются без соответствующей проверки, то в программе присутствует уязвимость.

Неконтролируемыми данными являются данные из файлов, пользовательский ввод, данные, переданные по сети. Мы рассматриваем два вида помеченных данных: помеченные целые числа и помеченные строки.

Ниже перечислены виды уязвимостей, связанных с помеченными данными.

• При использовании для доступа к массиву происходит переполнение буфера, что может позволить злоумышленнику захватить контроль над устройством [5]. По данным национальной базы уязвимостей США (NVD) ошибки подобного рода являются причиной 9.49% всех уязвимостей, занесенных в базу CWE в 2018 году [6].

• Использование в циклах. Если помеченные данные используются в качестве ограничения цикла, то цикл может выполниться большее количество раз, чем предусмотрено. Это может приводить как к излишнему расходованию процессорного времени, так и к другим ошибкам, например, переполнению массива. Если помеченные данные используются в качестве шага итератора цикла, то можно так подобрать данные, чтобы цикл стал бесконечным.

• Использование для некоторых операций, например, для выделения памяти. Злоумышленник может заставить программу выделить излишне много памяти.

Листинг 1 иллюстрирует уязвимости, связанные с целыми числами. Для исправления уязвимости переполнения буфера, необходимо проверить диапазон переменной n, чтобы он принадлежал интервалу [0; 99]. Безопасный диапазон для других видов уязвимостей зависит от логики программы.

Помеченные строки могут как содержать произвольные символы, так и иметь произвольную длину. При копировании такой строки в массив фиксированного размера, может происходить его переполнение.

Фрагмент кода на листинге 2 иллюстрирует уязвимости с помеченными строками.

3. СТАТИЧЕСКИЙ АНАЛИЗАТОР SVACE

Анализатор Svace осуществляет поиск дефектов в исходном коде; при этом поиск происходит как для ошибок, возможных при выполнении программы, так и подозрительных мест, где, возможно, логика программы нарушена. Целью является найти как можно больше дефектов при приемлемом времени работы и высоком качестве результатов.

Инструмент может как пропускать реальные дефекты, так и выдавать ложные срабатывания, не соответствующие реальным дефектам. Для анализируемой программы не требуется никакой подготовки, достаточно, чтобы исходный код компилировался. Время анализа сравнимо со временем компиляции программы. Для больших программ целесообразно использовать Svace во время ночной сборки.

Задача поиска дефектов является алгоритмически неразрешимой [7], т.е. нельзя найти все ошибки определенного типа в произвольной программе и не выдать ложных срабатываний. Для решения этой задачи используются всевозможные компромиссы. Одним из подходов является поиск приближенного решения. Возможны два вида приближений.

• Приближать в сторону отсутствия ложных срабатываний. В этом случае выдаются только истинные сообщения, но возможен пропуск множества дефектов. Типичным примером является выдача ошибок компилятора, для которого недопустима возможность отказаться компилировать корректную программу.

• Поиск всех ошибок определенного типа. Процент ложных срабатываний при этом может быть высоким. Более качественный анализ может достигаться за счет существенного увеличения времени работы.

Такие приближения называются консервативными, так как они всегда округляют решение в одну сторону. В Svace не используется консервативное приближение. Это позволяет как ускорить анализ программ, так и существенно повысить уровень истинных срабатываний.

В статье [8] утверждается, что консервативный анализ алиасов необходим для оптимизации программ, но является опциональным для анализаторов. Когда авторы потоково- и контекстно-чувствительного анализа [9, 10] удалили один шаг, необходимый для обеспечения консервативности, общее время анализа значительно сократилось. В одном случае время уменьшилось от нескольких дней до нескольких минут.

На рис. 1 иллюстрируются возможные подходы. По оси ординат показан процент найденных ошибок, по оси абсцисс процент истинных. Подход, используемый в Svace, позволяет максими-

```
char buf[100];


int n;
scanf("%d", &n); //n - tainted


//переполнение буфера, если n меньше нуля
либо больше 99
buf[n] = 0;


//выделение памяти
char*p = (char*)malloc(n * sizeof(struct
Fmt));
int i = 99;


while(i > 0) {
  buf[i] = '0' + (i % 10);
  //бесконечный цикл, если n равно 0
  i -= n;
}
```

Листинг 1. Помеченные целые числа
Listing 1. Tainted integers

зировать площадь прямоугольника, показанного пунктирными линиями.

Подобный подход достаточно популярный и не является ноу-хау, используемым в Svace. В статье [11] предлагается термин *soundy*, означающий в целом консервативный анализ, который допускает отказ от консервативности для некоторых конструкций.

2.1. Архитектура Svace

Для анализа программы анализатору требуется исходный код и скрипт сборки. Svace осуществляет поиск дефектов в программах на языках C, C++, Java, Kotlin, Go[1].

Типичная схема анализа показана на рис. 2. Svace перехватывает команды запуска компилятора и компоновки [12]. Затем запускается модифицированный компилятор, который строит абстрактное синтаксическое дерево (АСД) и запускаются детекторы для поиска ошибок на АСД, а также генерируется промежуточное представление программы для последующего анализа. Промежуточное представление подается на вход анализатору SvEng[2], выполняющему межпроцедурный анализ.

Анализаторы, построенные на базе АСД, осуществляют проход по узлам АСД и делают относительно простые проверки анализируемых правил. С помощью АСД-анализаторов можно найти подозрительные паттерны на деревьях разборов, различные опечатки, но при этом

[1] Первый релиз Svace, поддерживающий язык Go, ожидается в январе 2021 г.

[2] Сокращение от **Sv**ace **Eng**ine – движок анализатора Svace.

```
char*p = getenv("aaa");

//потенциальное
переполнение буфера
//размер p может быть
меньше 10
char x = p[10];

char buf[10];
int n = *((int*)p);
//переполнение буфера
buf[n] = 0;
```

Листинг 2. Помеченные строки
Listing 2. Tainted strings

• сложно отследить зависимости между переменными,

• сложно выполнить анализ указателей,

• сложно сделать межпроцедурный и межмодульный анализ.

Поскольку поиск ошибок помеченных данных, как правило, требует анализа вышеперечисленных свойств, мы не будем реализовывать детекторы помеченных данных на этом уровне.

4. АНАЛИЗАТОР SvEng

4.1. Общая схема

Анализатор SvEng осуществляет глубокий межпроцедурный потоково- и контекстно-чувствительный анализ. Наиболее важными являются следующие особенности анализатора SvEng.

• Межпроцедурный анализ на основе резюме, при котором функции обходятся по графу вызовов начиная с листьев. Каждая функция анализируется только один раз. По завершению анализа функции создается ее резюме, которое затем будет использовано при анализе вызова функции. Резюме содержит описание свойств функции. Анализ сохраняет баланс между компактностью анализа и точностью описания семантики функции.

• Неконсервативный анализ. За счет отказа от консервативности анализ получает лучшую точность и более высокую производительность.

• Анализ внутри одной функции основан на анализе значений. Анализ отслеживает значение переменных и ячеек памяти и ассоциирует большинство свойств со значениями переменных.

• Запуск всех анализов одновременно. Невысокая стоимость одного детектора.

Дизайн SvEng позволяет находить множество типов дефектов: разыменование нулевых указателей, недостижимый код, переполнение массива, утечки памяти и ресурсов, неправильное использование библиотечных функций, двойные блокировки мьютексов.

Мы реализовали поиск уязвимостей помеченных данных таким образом, чтобы по максимуму использовать возможности анализатора. Детекторы реализованы в виде анализа источник-приемник, где источником являются функции, возвращающие помеченные данные, а приемником – операции, в которых эти данные должны быть проверены перед использованием. Анализатор хорошо обнаруживает ошибки, где источник и приемник не сильно удалены друг от друга на графе вызовов.

Анализ производится по следующем схеме:

1 – на вход анализатору подаются файлы с промежуточным представлением;

2 – строится граф вызовов;

3 – запускается предварительная фаза анализа, на которой доступны легковесные анализы; на этой фазе, в том числе, собирается информация об указателях на функцию и виртуальных вызовах;

4 – достраивается граф вызовов для виртуальных функций и вызовов по указателю;

5 – запускается основная фаза анализа.

4.2. Промежуточное представление

Для анализа используется собственное промежуточное представление, которое строится уже после запуска анализатора (Svace IR). На вход анализатору подаются файлы в промежуточном представлении, зависящие от анализируемого языка:

• LLVM-файлы для C/C++,

• Java-байт код для Java/Kotlin,

• собственный JSON-формат для Go.

После конвертации исходного кода в представление Svace IR, анализ производится одинаково для всех языков программирования.

Svace IR является низкоуровневым типизированным языком в SSA-форме. Язык имеет процедуры, в том числе возможность их вызова по указателю. Для моделирования виртуальных вызовов в C++ явно строятся виртуальные таблицы. Для моделирования исключений используются конструкции goto.

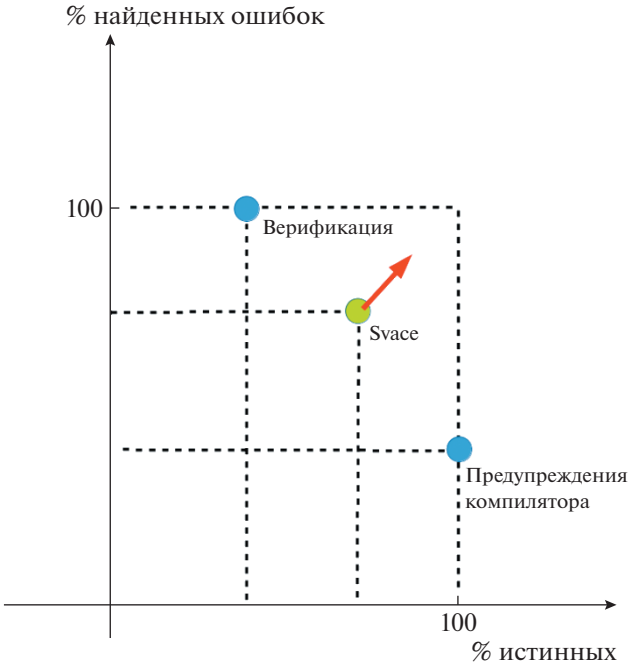

Рис. 1. Подходы к обнаружению ошибок.

Листинг 3 содержит пример кода на C. Соответствующий фрагмент на Svace IR приведен на листинге 4.

Использование низкоуровневого промежуточного представления имеет как свои плюсы, так и минусы. Недостатки такого представления:

• затруднен анализ высокоуровневых конструкций;

• нет информации об оригинальных синтаксических конструкциях;

• трансформация представления в эквивалентное компилятором.

Основным преимуществом является простота анализа. Семантика точно моделируется компилятором, язык имеет небольшое количество инструкций, которые редко меняются. Значительное количество языковых конструкций являются синтаксическим сахаром (исключения, конструкторы и деструкторы, циклы и др.) и не требуют добавления новых инструкций в IR.

Низкоуровневое IR для задачи поиска помеченных данных представляется хорошим выбором, т.к. для этого вида дефектов важна принципиальная возможность эксплуатации уязвимости, которая не меняется при переходе на низкоуровневый язык. Анализ синтаксических конструкций при этом не очень важен.

4.3. Межпроцедурный анализ

Используется межпроцедурный анализ на основе резюме.

При таком анализе для каждой функции строится резюме – краткое описание поведения функции. Резюме используется для анализа инструкции вызова функции и позволяет избежать повторного анализа тела функции. Резюме создается после анализа функции.

На рис. 3 показан граф вызовов для сборки двух программ. Наверху графа вызовов находятся функции main, которые вызывают другие функции. Листьями графа являются функции h и j, анализ начинается с листьев графа. После того как они будут проанализированы, станут доступны их резюме и возможность начать анализ функции foo.

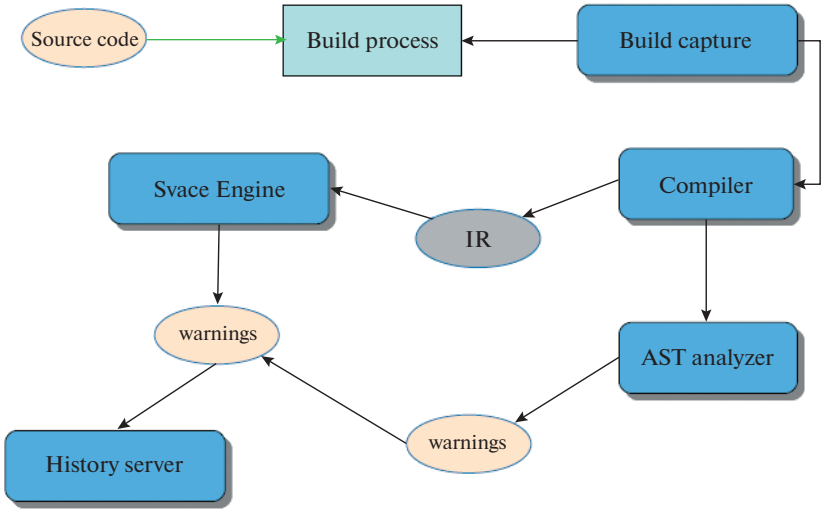

Рис. 2. Схема анализа.

```
int calc(int f)
{
    int a, b;
    int*p;
    a = 1;
    b = 2;
      p = f ?
&a : &b;

    int x = *p;
    *p = 3;

    int y = *p;

     return x +
y;
}
```

Листинг 3. Код на C
Listing 3. C code

Анализ имеет как высокую скорость, так и хорошую масштабируемость. Последняя достигается за счет того, что резюме создается достаточно компактным и не включает все детали поведения функции. Возможно ограничивать размер резюме. В этом случае резюме не будет описывать все интересующие эффекты вызова функции, но анализ вызова функции будет иметь константное время.

Благодаря своим преимуществам, этот анализ является довольно популярным и используется во многих инструментах статического анализа: Prefix [13], Saturn [14], Calysto [15], CSharpChecker [16].

4.4. Предварительная фаза

При использовании только анализа на основе резюме каждая функция посещается ограниченное количество раз. При этом для ряда анализов необходима некоторая информация обо всей программе, либо о функциях, находящихся выше по графу вызовов.

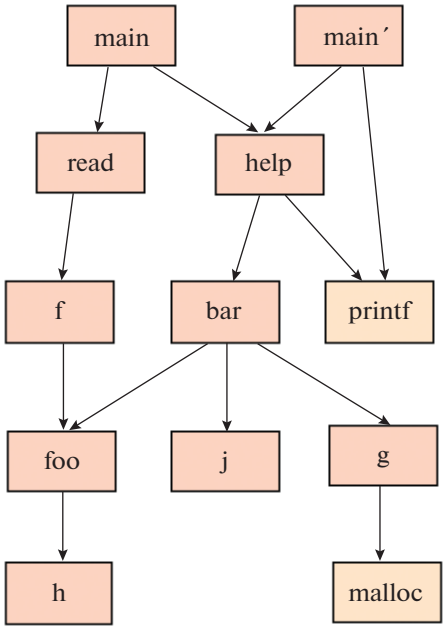

Рис. 3. Пример графа вызовов.

Для получения такой информации была создана предварительная фаза, которая включает следующие виды анализов:

- анализ присваиваний значений указателям на функции,
- анализ заполнения виртуальных таблиц C++,
- анализ иерархии классов для Java,
- анализ значений глобальных переменных.

Эти анализы на вход получают информацию о глобальных переменных и инструкции для каждой функции. Анализ производится параллельно для разных модулей.

Анализ не учитывает порядок инструкций, но доминирующая инструкция всегда анализируется первой. Потоково-нечувствительный анализ был выбран, чтобы не слишком замедлять время общего анализа, т.к. основные свойства будут проанализированы во время основного анализа.

Цель данных анализов – получить необходимую информацию, не затрачивая существенное количество времени.

4.5. Девиртуализация

Цель девиртуализации разрешить вызов процедур для указателей на функции и виртуальных таблиц. Таблица виртуальных функций представляет собой глобальную переменную типа структуры с полем – массивом, содержащим указатели на виртуальные функции, доступ к которым осуществляется через константные значения.

```
def calc(int f) int
{
    a = alloca();
    b = alloca();
     p = alloca ref
int();
    store 1, a;
    store 2, b;

    if(f != 0)
          goto true-
label;
    else
         goto false-
label;

true-label:
    store a, p;
    goto label2;
```

```
false-label:
    store b, p;
    goto label2;

label2:
    t1 = load p;
    x = load t1;
    store 3, t1;
    t2 = load p;
    y = load t2;
    t3 = x + y;
    ret t3;
}
```

Листинг 4. Код на Svace IR
Listing 4. Svace IR code

Анализ отслеживает значения переменных, структур и константные индексы массивов.

Для каждого указателя собираются все возможные записи. Результатом анализа для каждой пары “функция – указатель” является множество функций, которые могут быть вызваны, либо пустое множество, если анализ не смог определить всех кандидатов.

Результаты анализа используются двумя способами:

• при достраивании графа вызовов для добавления ребер,

• при обработке инструкции вызова по указателю.

Анализу на основе резюме нужна информация о графе вызовов. Поэтому после получения информации о виртуальных функциях достраивается граф вызовов. Эта процедура не является сложной, достаточно просто добавить ребро, где **может** быть вызвана функция в результате виртуального вызова. При анализе функции на основном анализе эту информацию в некоторых случаях можно будет уточнить. Более точный анализ на предварительной фазе добавит меньшее количество лишних ребер. До тех пор, пока основной анализ в состоянии убрать лишнее ребро, они не влияют на точность анализа. Но время анализа может увеличиваться, а потребление памяти возрастать, т.к. каждое ребро в графе вызовов накладывает ограничения на возможный обход функций и требует наличия резюме.

В основном анализаторе реализован плагин, отслеживающий возможных кандидатов для каждого указателя. Этот плагин получает данные о

кандидатах для каждого указателя, а затем используя потоково-чувствительный анализ распространяет эти данные внутри процедуры независимо. В некоторых случаях, используя информацию о значениях переменных и недостижимых инструкциях, анализ может получить более точную информацию, чем та, которая доступна после предварительной фазы.

Для случая, если указателю соответствуют несколько кандидатов, моделируется их условный вызов. Для каждого кандидата применяется резюме, а затем происходит объединение для каждого контекста использования. Таким образом каждое резюме применяется отдельно для каждого контекста.

4.6. Внутрипроцедурный анализ

Для анализа отдельной функции используется символьное выполнение с объединением состояний анализа в точках слияния путей. Для функции строится граф потока управления, и анализ начинает обход графа. С ребрами графа ассоциируются абстрактные состояния. На каждом шаге для инструкции по абстрактному состоянию на входных ребрах формируется абстрактное состояние на выходном ребре.

Для сильно связанных компонент (ССК) выполняются несколько итераций. Абстрактные состояния для выходных ребер ССК сохраняются после каждой итерации. После завершения анализа ССК, эти состояния объединяются. Во время анализа используются несколько эвристик для моделирования всевозможных путей выполнения ССК. В некоторых случаях моделирование происходит некорректно, если эвристики не сработали.

Все дополнительные анализы и детекторы запускаются одновременно, что позволяет уменьшить одновременно и время анализа, и потребляемую память. Время анализа сокращается за счет того, что общие для всех детекторов свойства анализируются только один раз. А потребление памяти уменьшается, т.к. в большинстве случаев после анализа инструкции, состояние анализа на входном ребре может быть удалено.

Для моделирования значений переменных и ячеек памяти Svace использует абстракцию, названную идентификатором значений. Двум переменным сопоставляется один идентификатор значений, если при выполнении эти переменные будут иметь одинаковые значения (решается задача нумерации значений).

Большинство свойств ассоциируются с идентификаторами значений. Сами свойства также можно описывать с помощью идентификаторов значений. Для описания свойств используются атрибуты. Атрибут обозначает некоторое анализируемое свойство.

Примеры атрибутов:

- Null – значение указателя имеет нулевое значение,
- ValueInterval – значения целочисленной переменной принадлежат отрезку [a;b],
- PointsTo – указатель указывает на ячейки памяти из множества. Сами ячейки памяти обозначаются идентификатором значений, который моделирует адреса ячеек памяти,
- Ness – необходимые условия достижения ребра в графе потока управления. Представляет собой формулу булевой логики, где в качестве переменных используются идентификаторы значений.

4.7. Использование SMT-решателя

Для реализации чувствительности к путям в Svace используются условные атрибуты, которые выражают свои свойства в виде формул булевой логики над идентификаторами значений. Условные атрибуты, описывающие свойства значений переменных, ассоциируются с идентификаторами значений.

Формулы имеют вид, показанный на рис. 4, где *Val* – идентификаторы значений, *Const* – константы. Для каждой литеральной формулы *Atom* определена инструкция отрицания, возвращающая другую литеральную формулу. В формулах могут использоваться конъюнкции, дизъюнкции от других формул, а также отрицание для литеральных формул.

SMT-решатель запускается только перед выдачей предупреждения для того, чтобы отсеять несуществующие пути. Таким образом SMT-решатель запускается не больше, чем количество неотфильтрованных предупреждений. В общем случае используется следующая формула:

$$Ness, \wedge Error_i,(v),$$

где *Ness* необходимое условие достижимости для ребра i, $Error_i,(v)$ – условие ошибки для значения, моделируемого идентификатором v.

4.8. Анализ потока данных

Для анализа недостижимого кода был реализован консервативный анализ потока данных (АПД[3]) [17]. Этот анализ запускается перед началом анализа каждой функции в основном анализе. Анализ помечает недостижимые ребра на графе потока управления. Основная причина, по которой он был реализован – недостаточная точность основного анализа. Недостижимое ребро сильно влияет на все остальные анализы, поэтому неконсервативность

[3] Под АПД в статье мы будем подразумевать движок анализа, основанный на анализе потока данных.

$\bowtie ::= > | < | = | \neq | \leq | \geq$
$\oplus ::= + | - | * | /$
Op ::= Val | Const
Atom ::= True | False | Op $\bowtie$ Op | Op = Op $\oplus$ Op
Conj = Atom | Conj $\wedge$ Conj
CFormula ::= Conj
SFormula ::= Conj $\wedge$ $\overline{\text{Conj}}$
FFormula ::= Atom | FFormula $\wedge$ FFormula | FFormula $\vee$ FFormula

Рис. 4. Используемые формулы.

в данном случае может приводить к значительно худшим результатам.

Позже были добавлены другие анализы:

- анализ интервалов,
- анализ исключений,
- анализ живых переменных.

Помимо получения консервативных результатов, этот анализ позволяет получить данные о функции перед запуском основного анализа. Поскольку в основном анализе все детекторы запускаются одновременно, возникает проблема предварительного получения свойств, которая решается АПД.

4.9. Анализ сверху вниз на предварительной фазе

Рассмотрим пример, показанный на листинге 5. Функция f получает помеченные данные и передает их в функцию g. Если функция g небезопасно использует свой аргумент, то надо выдать предупреждение. Для этого в анализе на основе резюме, надо протащить информацию о небезопасном использовании аргумента y через резюме.

В общем случае, сделать это не тривиально. При анализе функции g ничего не известно о контекстах, где она будет использована. Поэтому не известно надо ли сохранять информацию о небезопасном использовании. Резюме является компактным, сохранение всей возможной информации лишит анализ основных преимуществ.

Для решения описанной проблемы мы добавили анализ на предварительной фазе на основе АПД. На этой фазе процедуры обходятся в произвольном порядке; в рамках контекста процедуры отслеживаются помеченные данные и для каждого вызова процедуры в этом контексте запоминаются аргументы, которые являются помеченными. Таким образом, сохраненная информация относится только к рассмотренным контекстам.

```
void f() {
    int x = input();
    g(x);
}

void g(int y) {
    // Известен контекст, в котором
переменной y
      // было передано помеченное
значение
}
```

Листинг 5. Мотивация для предварительной фазы
Listing 5. Motivation for preliminary phase

Идея этого подхода в некотором смысле схожа с динамическим анализом, при котором происходит исследование свойств конкретного пути программы и все выводы которого остаются справедливыми только в рамках этого пути. При таком анализе производится исследование свойств, которые не обладают всеобщностью и справедливы для некоторого контекста вызова или цепочки вызовов функций.

Таким образом, предварительная фаза собирает информацию о контекстах использования функции. Данные о помеченных аргументах функции используются во время основного анализа функции. Соответствующие атрибуты, описывающие помеченные данные, устанавливаются на основе результатов предварительной фазы. Для конкретных детекторов ситуация выглядит так, как будто аргумент является результатом вызова функции, возвращающей помеченные данные. При этом возникает проблема, что информация о помеченных данных может попасть в резюме, что не является корректным. Резюме описывает результат вызова функции для произвольного контекста вызова, а предварительный анализ собирает данные для некоторых контекстов. Для решения этой проблемы был выбран следующий способ. Функция, у которой некоторые аргументы являются помеченными согласно предварительному проходу, анализируется два раза:

1 – используются данные от предварительного прохода, резюме не создается,

2 – обычный анализ без использования данных от предварительного прохода с созданием резюме.

Например, если для функции `input` известно, что она всегда является источником помеченных данных, то при анализе функции `f`, можно сделать вывод о существовании контекста вызова функции `g`, в котором ее аргумент имеет помеченное значение. На основании этого, верным будет сообщить об ошибке. Основная фаза принимает решение на основе только анализа функции `g`.

4.10. Спецификации в Svace

Для анализа библиотечных функций, поведение которых известно, используются спецификации. Спецификация в Svace это еще одно определение функции, написанное на анализируемом языке. Спецификация описывает поведение функции в компактной форме. Помимо конструкций языка, спецификации могут содержать вызовы специальных предопределенных функций, называемых спец-функциями.

Дистирбутив Svace содержит спецификации для широкоиспользуемых библиотек. Пользователь может добавлять свои собственные спецификации.

Для примера разберем спецификацию для функции `strcat` из стандартной библиотеки C. Исходный код спецификации:

```
char *strcat(char *s, const char *append) {
   char d1 = *s;
   char d2 = *append;
   sf_set_trusted_sink_ptr(s);
   sf_set_trusted_sink_ptr(append);
   sf_append_string(s, append);
     sf_vulnerable_fun("This function is unsafe, use strncat instead.");
   sf_null_terminated(s);
   return s;
}
```

В данном коде содержатся следующие спецфункции:

- `void sf_set_trusted_sink_ptr(const void* str)` – данная спецфункция показывает, что ее аргумент str должен быть из надежного источника, иначе данный указатель может вызвать уязвимость,
- `void sf_vulnerable_fun(const char*const reason)` – данная спецфункция показывает, что текущая функция небезопасна и имеет safe-аналоги,
- `void sf_append_string(char* dst, const char* src)` – данная спецфункция показывает, что выполнено добавление строки src к dst,
- `void sf_null_terminated(char *p)` – данная спецфункция показывает, что строка `p` заканчивается нулевым символом.

При анализе данной спецификации детекторы могут извлечь следующие свойства:

строки `s`, `append` были разыменованы, это может быть полезно для детекторов, которые ищут разыменование нулевых указателей,

строки `s`, `append` должны быть из надежного источника, информация используется детекторами помеченных данных,

функция `strcat` опасна и нужно использовать ее безопасную замену `strncat`,

строка s заканчивается нулевым символом.

5. ДЕТЕКТОРЫ ПОМЕЧЕННЫХ ДАННЫХ В SvEng

5.1. Плагины и детекторы

Анализ в SvEng разделен на ядро и плагины. Ядро анализа отслеживает граф указателей, выполняет сильные и слабые обновления ячеек памяти и вызывает обработчики для соответствующих ситуаций. Все дополнительные анализы реализуются внутри плагинов в виде обработчиков инструкций, оперирующих не переменными, а

соответствующими идентификаторами значений. Дополнительные анализы получают на вход абстрактное состояние на входном ребре инструкции, и формируют абстрактное состояние на выходном ребре. Детекторы также реализуются в плагинах и выдают предупреждения на основе входного абстрактного состояния и передаточной функции обрабатываемой инструкции.

Всем дополнительным анализам доступна информация на входных ребрах и недоступна на выходных. Различные анализы и детекторы не должны влиять друг на друга во время анализа инструкции. В текущей версии анализаторы и детекторы запускаются по очереди[4], но результаты должны быть такими же, как если бы они запускались параллельно.

В абстрактном состоянии для описания свойств используются атрибуты. Атрибуты могут ассоциироваться с идентификаторами значений и с ребрами графа потока управления для некоторого абстрактного состояния. Атрибут обозначает некоторое анализируемое свойство. Атрибуты должны иметь функцию объединения двух атрибутов $\sqcup$. Эта функция используется при объединении состояний в точках слияния путей. Атрибуты позволяют разделять абстрактное состояние между дополнительными анализами.

Атрибут может иметь произвольную структуру, но некоторые виды атрибутов используются достаточно часто. Можно выделить следующие виды атрибутов:

- двоичные атрибуты,
- тернарные атрибуты,
- интервальные атрибуты,
- условные атрибуты,
- множество идентификаторов значений.

Каждый вид атрибутов может также иметь трассу – односвязный список из пар “точка в программе – краткое текстовое описание события”. Трассы меняются при распространении атрибутов и используются в момент выдачи предупреждений, чтобы показать дополнительные точки в программе, позволяющие лучше понять, в чем ошибка.

Двоичные атрибуты. Имеют два значения: *true* и *false*. Значение *true* означает, что некоторая переменная точно имеет анализируемое свойство, значение *false* все остальное, т.е. либо переменная не имеет это свойство, либо недостаточно информации. В зависимости от функции объединения эти атрибуты могут быть двух видов:

- or-атрибуты – результат будет истинным, если хотя бы один аргумент является истинным,
- and-атрибуты – результат будет истинным, если оба аргумента является истинными.

Тернарные атрибуты могут принимать следующие значения:

true – для переменной выполняется свойство в данной точке для всех путей[5], проходящих через нее,

maybe – существует путь, возможно, недостижимый, для которого свойство выполняется,

false – остальные случаи: данное свойство либо не выполняется, либо недостаточно информации.

Функция объединения атрибута:

- $true \sqcup false = maybe$,
- $maybe \sqcup false = maybe$,
- $true \sqcup maybe = maybe$.

Фактически, тернарный атрибут является объединением or и and двоичных атрибутов. Значение *true* будет, если оба двоичных атрибута имеют истинное значение. Значение *maybe*, если or-двоичный атрибут имеет истинное значение, а and-атрибут не имеет, и *false* в остальных случаях.

Интервальные атрибуты связывают переменную с некоторым целочисленным интервалом, который описывает произвольное свойство. Например, возможный размер выделенной памяти для указателя или же значение, которое может принимать целочисленная переменная. Интервал может принимать следующие значения: $[a,b] - a \le b$; $a, b \in [MIN\_INT + 1, MAX\_INT - 1]$, данная запись означает, что свойство у переменной имеет значение в пределах интервала от a до b. Значения MIN_INT и MAX_INT зарезервированы для обозначения бесконечностей $-\infty$ и $+\infty$.

Цепочка интервалов представляет собой несколько интервалов. Цепочка интервалов позволяет моделировать интервалы с выколотыми точками.

Условные атрибуты хранят в себе формулу, описывающую выполнение некоторого свойства (см. 4.7). Выполнимость формулы проверяется SMT-решателем перед выдачей предупреждения. Данная формула состоит из следующих конъюнкций:

- условие достижимости, данная формула содержит условия при которых точка, где выдается срабатывание, будет достижима; данную формулу хранит атрибут *Ness*;
- условия помеченных данных, данная формула содержит условие, при котором указатель или зна-

[4] Распараллеливание запуска детекторов потенциально позволяет немного ускорить анализ, но при этом значительно усложняет внутрипроцедурный анализ из-за необходимости синхронизации. Поэтому выбор для распараллеливания был сделан на уровне графа вызовов: отдельные функции могут анализироваться параллельно, но анализ внутри функции последовательно.

[5] Для случая статического анализа учитываются все пути, которые анализ посчитал достижимыми. Чем точнее анализ, тем больше недостижимых путей он сможет отсеять.

чение целочисленной переменной будет получено из ненадежного источника; данную формулу отслеживают атрибуты *TaintedPtrIf* (для ненадежного указателя) и *TaintedIntIf* (для ненадежного значения целочисленной переменной), которые будут описаны позже;

• некоторое дополнительное свойство, зависящее от детектора, например, для обращения к массиву, проверяется что значение индекса меньше нуля, либо больше размера массива.

5.2. Межпроцедурное распространение атрибутов

Ядро анализа при создании резюме определяет какие идентификаторы значений туда попадут и для каждого вызывает обработчик *annotate*. В момент применения резюме ядро анализа сопоставляет формальные аргументы фактическим и вызывает обработчик *apply*. Для того, чтобы сделать атрибут межпроцедурным, достаточно подписаться на эти два обработчика, и реализовать соответствующую логику в зависимости от семантики атрибутов.

Для тернарных, двоичных, интервальных атрибутов резюме формируется путем передачи свойств без изменений (обработчик *annotate*).

Условные атрибуты – прежде, чем передать формулу в резюме, она упрощается:

1 – добавляем к условию конъюнкцию с условием достижимости,

2 – все атомарные условия, которые содержат в себе идентификаторы, добавленные ядром в резюме, сохраняются в специальное множество,

3 – берем не посещенное простое условие из формулы; если оно не содержится в списке, то преобразуем его в *False*, иначе помечаем условие как посещенное,

4 – для получившейся формулы применяем правила поглощения ($False \wedge Cond = False$, где Cond – некоторое условие),

5 – долучившаяся формула сохраняется в резюме.

Далее будем считать, что все стандартные атрибуты являются межпроцерурными и используют описанный выше механизм создания резюме, если не сказано иного.

5.3. Используемые атрибуты

Опишем вспомогательные атрибуты, предоставляющие нужную информацию для множества детекторов, которые используют помеченные данные.

Атрибут *MustTaintedInterval* хранит интервал возможных значений помеченной переменной. Значения из этого интервала должны быть проверены перед использованием. Функция объединения атрибута: пересечение интервалов ($[10, 20] \cap \cap \varnothing = \varnothing$, $[10,20] \cap [10,11] = [10, 11]$).

Атрибут *MightTaintedInterval* хранит максимально возможный помеченный интервал. Функция объединения атрибута: объединение с пустым интервалом) и пересечение с непустым ($[10,20] \sqcup [10,11] = [10,11]$).

Со значением помеченной переменной также связаны атрибуты, которые показывают осуществлялась ли проверка значения данной переменной:

MinIsTainted – двоичный or-атрибут, указывает на наличие проверки нижней границы. По умолчанию значение *true* (проверка отсутствовала);

MaxIsTainted – двоичный or-атрибут, указывает на наличие проверки верхней границы. По умолчанию значение *true* (проверка отсутствовала);

Эти атрибуты нужны, чтобы не выдавать бесполезное срабатывание для случаев, когда переменную сравнили с некоторым параметром функции:

```
char* allocate(int max) {
    unsigned int n;
    scanf("%d", n);

    if (n > max) {
        printf ("parameter too big,
use %d", max);
        return 0;
    }

return malloc(n);
}
```

Атрибут подавляет выдачу предупреждений для случаев, когда неизвестна точная безопасная граница. Например, для выделения памяти из кучи нет возможности определить такое ограничение статически. Для доступа к массиву с известным размером, такая граница известна, и атрибут не влияет на выдачу предупреждения.

5.4. Целочисленные помеченные значения

Значения целочисленных переменных могут контролироваться злоумышленником. Все реализованные детекторы являются подвидом источник-приемник, где в качестве источников используются функции получения данных из внешних источников, а приемников – операции, где данные необходимо проверить.

Источником являются все данные, которые получены извне программы (файл, сеть, пользовательский ввод). В большинстве случаев эти данные в Svace получаются из спецификаций.

Исключением являются параметры функции `main – argc` и `argv`. Svace с помощью атрибута *ArgvVarAttr* связывает `argv` с `argc`, данный атрибут позволяет детекторам избегать ложных срабатываний связанных с использованием параметров `main`. Например, когда указатель `argv` смещают, используя `argc`.

Приемником являются:

• использование как индекса массивов; перед использованием надо проверить диапазон,

• библиотечные функции,

• использование как шага цикла либо как ограничение цикла,

• использование как индекса для указателя; хотя неизвестен его точный размер, это не значит, что можно использовать любой.

Особенностью помеченных целочисленных переменных является то, что проверка их диапазона может производиться довольно сложным образом с использованием битовой арифметики.

Кроме этого, некоторые переменные могут быть связаны друг с другом какой-либо операцией, в этом случае проверяться может только одна переменная. При реализации анализа важно учитывать такие взаимосвязи. Для анализа взаимосвязей между переменными используется SMT-решатель: строятся формулы, описывающие эти взаимосвязи, а затем вызывается SMT-решатель, который определит может ли формула иметь решение.

В Svace реализованы следующие типы детекторов для поиска помеченных целых чисел:

• TAINTED_ARRAY_INDEX – доступ к массиву по непроверенному индексу,

• TAINTED.INT_OVERFLOW – приемником является потенциальное целочисленное переполнение,

• TAINTED_INT.PTR – доступ к указателю с помощью смещения,

• TAINTED_INT – доступ к функции, где входные параметры должны быть проверены,

• TAINTED_INT.LOOP – приемником помеченных данных является использование переменной как ограничения цикла, либо как шага цикла.

Для перечисленных детекторов могут использоваться суффиксы, имеющие следующее значение:

• .MIGHT – не на всех путях к опасному использованию данных эти данные помеченные,

• .COND – приемник находится в вызываемой функции и не достижим на всех путях внутри этой функции.

Например, TAINTED_ARRAY_INDEX.MIGHT – доступ к массиву в качестве индекса, где не все пути содержат помеченные данные.

5.4.1. Предупреждение TAINTED_INT. В коде может встретиться ситуация, когда программист использует переменную, значение которой получено из внешнего источника, в таких функциях как `strncpy`, `malloc` или в качестве условия выхода из цикла. Так как злоумышленник может передать любое значение, то использование таких переменных может повлечь за собой уязвимости: бесконечный цикл, переполнение массива.

```
void test(int fd) {
    int sizeBuf;
    //sizeBuf получен из ненадежного источника
    int ret = recv(fd, &sizeBuf, sizeof(sizeBuf), 0);
    if(ret<0)
        return;
    if (sizeBuf < 0) {
       return;
    }
    //использование помеченной переменной в calloc
    char*x = calloc(1, sizeBuf);//TAINTED_INT
}
```

В данном примере размер выделенной памяти зависит от помеченной переменной, *sizeBuf* может быть очень большим, что повлечет за собой выделение чрезмерно больших объемов памяти, которая не будет использоваться или же *sizebuf* может быть равен нулю, тогда обращение к *x* может вызвать ошибку переполнения массива.

Детектор ищет ситуации, когда помеченные целочисленные переменные передаются в функции, где они могут вызвать уязвимость.

Для идентификации помеченных целочисленных переменных используется атрибут *TaintedIntIf*, который хранит в себе формулу, при выполнении которой переменная будет содержать помеченное значение. Функция объединения атрибута: конъюнкция формул с веток.

Для межпроцедурного распространения свойств об использовании данных, которые должны быть из надежного источника, используется атрибут *TrustedIntSinkFlag*. Фактически, этот атрибут при применении резюме создает еще один приемник, для которого может быть выдано предупреждение. Заметим, что анализ спецификаций является частным случаем межпроцедурного анализа. При обработке спецификаций для функций типа *malloc* используется этот атрибут.

Условия выдачи срабатывания:

• после вызова функции ее аргумент имеет следующие свойство: *TrustedIntSinkFlag* – *true* или же переменная используется в условиях цикла,

• переменная имеет не пустой атрибут *MustTaintedInterval* или *MightTaintedInterval*,

• у переменной не проверяли нижнюю и верхнюю границу; атрибуты *MinIsTainted* и *MaxIsTainted* для переменной имеют значение *false*,

• конъюнкция формулы из *TaintedIntIf* с условием, что переменная лежит в интервале из *MustTaintedInterval* или *MightTaintedInterval* выполнима.

Использование атрибутов *MustTaintedInterval* и *MightTaintedInterval* является лишним для логики программы, но позволяют оптимизировать запросы к SMT-решателю.

5.4.2. Предупреждение TAINTED_INT.LOOP. Является подвидом TAINTED_INT для уязвимостей, связанных с циклами.

Выдается для двух случаев.

• Помеченное значение используется для ограничения количества итераций цикла. Ошибка заключается в том, что цикл может выполняться излишне много итераций. Для определения, что переменная ограничивает количество итераций цикла используется информация о графе потока управления. Обработчик условных инструкций проверяет, что инструкция принадлежит сильно связанной компоненте и входное ребро является ребром входа в эту компоненту.

• Помеченное значение используется как шаг цикла. В этом случае, если злоумышленнику удастся установить такое значение, чтобы переменная не менялась на разных итерациях, в программе будет бесконечный цикл. Определение шага цикла реализовано более сложно. На первом этапе на анализе потока данных вычисляются инварианты цикла, т.е. такие переменные, которые имеют одно и то же значение на всех итерациях. Для всех остальных переменных проверяется, что они используются в арифметических инструкциях, их диапазон допускает нежелательное значение[6] и условные инструкции, проверяющие значения этих переменных.

Возможны ситуации, когда переменная используется в цикле в вызываемой функции. Для анализа используется атрибут *LoopBoundFlag*, который устанавливается в истину в тех случаях, когда анализ посчитал переменную ограничителем количества итераций цикла. Далее этот атрибут распространяется межпроцедурно, и если в момент применения резюме формальный аргумент имеет этот атрибут, а фактический – атрибут *MustTaintedInterval*, то будет выдано предупреждение.

5.4.3. Предупреждение TAINTED_INT.PTR. Если использовать помеченную целочисленную переменную без каких-либо проверок в качестве смещения указателя, то можно выйти за пределы выделенной памяти, так как значение помеченной переменной может быть произвольным.

[6] Чаще всего это 0. В некоторых ситуациях к ошибке может дополнительно приводить целочисленное переполнение.

```
void test(int fd, int *ptr) {
    int index;
    //значение index помечено
    int ret = recv(fd, &index, sizeof(index), 0);
    //использование помеченной переменной index как смещение
    ptr[index] = 3;//TAINTED_INT.PTR
}
```

В данном примере значение переменной *index* может быть любым, поэтому возможна ошибка переполнения буфера.

Детектор ищет ситуации, когда помеченные целочисленные переменные не проверяются и используются в качестве смещения указателя.

Условия выдачи срабатывания:

инструкция получения доступа к указателю,

• смещение имеет непустой *MustTaintedInterval* или *MightTaintedInterval*,

• у смещения не проверяли нижнюю и верхнюю границу; атрибутов *MinIsTainted* и *MaxIsTainted* у смещения имеют значение *false*,

• размер выделенной памяти для указателя неизвестен, или же *MustTaintedInterval* или *MightTaintedInterval* интервал может превышать его размер.

5.5.4. Предупреждение TAINTED_ARRAY_INDEX. Предупреждение во многом похоже на TAINTED_INT.PTR с той разницей, что оно выдается при работе с массивами для которых статический анализатор знает размер.

```
void test(int fd) {
    int ptr [6];
    int index;
    //значение index получено из ненадежного источника
    int ret = recv(fd, &index, sizeof(index), 0);
    //использование помеченного значения в качестве индекса
    ptr[index] = 3;//TAINTED_ARRAY_INDEX
}
```

В данном примере нам известен размер массива *ptr*, и значение *index* может быть больше 5.

Детектор ищет ситуации, когда происходит доступ к массиву по индексу, где индекс получен из ненадежного источника и может иметь значение больше, чем размер массива. При этом размер массива известен анализатору.

Для определения размера массива используется атрибут *BufferSizeAttrVal*, который хранит возможный размер массива в виде интервала.

Условия выдачи срабатывания:

• доступ к массиву по индексу,

• нам известен размер массива; интервал из *BufferSizeAttrVal* не пустой и не [–∞, +∞],

• индекс имеет не пустой *MustTaintedInterval* или *MightTaintedInterval* атрибут,

• для индекса составляется формула с условием, что его значение может превышать интервал из *BufferSizeAttrVal*, эта формула выполнима.

5.5.5. Ошибки с целочисленным переполнением. Так как значение переменной, пришедшей из внешнего источника, может быть любым, то если осуществлять арифметические действия с данной переменной без предварительной проверки, то может произойти целочисленное переполнение и ее значение будет некорректным. Это может привести как к уязвимостям, так и к нарушению логики выполнения программы. Для таких ситуаций выдается предупреждение TAINTED.INT_OVERFLOW.

```
void test(int fd) {
    int ptr [6];
    int index;
    //значение index получено из ненадежного источника
    int ret = recv(fd, &index, sizeof(index), 0);
    //целочисленное переполнение
    index += 1; //TAINTED.INT_OVERFLOW
    if(index > 4) {
       return;
    }
    ptr[index] = 3;
}
```

В данном примере *index* может иметь максимальное значение для переменной типа *int*, поэтому при добавлении единицы может произойти целочисленное переполнение и ее значение станет некорректным.

Детектор проверяет ситуации, когда переменная из ненадежного источника участвует в арифметических операциях: сложение, умножение, вычитание. При помощи интервала из атрибута *MustTaintedInterval* проверяется, возможно ли переполнение.

Условия выдачи срабатывания:

• инструкция сложения, вычитания, умножения,

• атрибут *MustTaintedInterval* у одного из аргументов инструкции не пустой,

• интервал из *MustTaintedInterval* = $[-\infty; +\infty]$ или он может переполнить тип второго аргумента.

5.5. Помеченные строки

В некоторых ситуациях необходимо, чтобы указатель содержал в себе проверенные данные. Например, при открытии файла с помощью open или же когда осуществляется объединение или копирование строк. В случае `strcpy` и `strcat`, если строка `src` помечена, то ее размер может быть больше чем у строки `dst`, что вызовет переполнение буфера.

```
void test(int fd, int *ptr, int size) {
    //env получен из ненадежного источника
    char* env = getenv("PATH");
    char *buf = malloc(size);
    //строка в env может быть любого размера
    //поэтому возможно переполнение buf при копировании
    strcpy(buf, env);//TAINTED_PTR
}
```

В данном примере размер строки в `env` может быть любым, поэтому возможна ошибка переполнения буфера при ее копировании в `buf`.

Детектор ищет ситуации, когда ненаджедный указатель используется в функциях, где он может вызвать уязвимость.

Для идентификации испорченного указателя используются следующие атрибуты:

TaintedPtrIf – атрибут хранит условия, при которых указатель будет содержать помеченные данные. Представляет собой формулу логики высказываний. Функция объединения атрибута: конъюнкция формул с веток.

TaintedPtr – тернарный атрибут, показывает, содержит ли указатель помеченные данные.

С помощью тернарного атрибута *TrustedPtrSinkFlag* детектор находит указатели, которые использовались в опасных функциях, где помеченное происхождение указателя может вызвать уязвимость (`open`, `strcpy`, `strcat` и т.д.).

Условия выдачи срабатывания:

• атрибут *TaintedPtr* имеет значение *true* или *maybe*; указатель точно или же возможно получен из ненадежного источника,

• атрибут *TrustedPtrSinkFlag* имеет значение *true*, указатель использовался в функции, где он может вызвать уязвимость,

• формула в атрибуте *TaintedPtrIf* выполнима.

Также существуют ситуации, когда помеченный указатель не может вызвать уязвимость. Например, если при использовании `strcpy` известно, что для `dst`-строки выделено памяти достаточно, чтобы скопировать испорченную `src`-строку:

```
void test(int fd, int *ptr, int size) {
    //env содержит помеченные данные
    char* env = getenv("PATH");
    //размер buf зависит от длины строки в env
    char *buf = malloc(strlen(env) + 1);
    //переполнение невозможно
    strcpy(buf, env);//TAINTED_PTR
}
```

Таблица 1. Процент ложных срабатываний для проекта Tizen 6

Предупреждение	Количество предупреждений	Процент истинных срабатываний
TAINTED_ARRAY_INDEX	102	62.5
TAINTED_INT	137	65.5
TAINTED_INT.LOOP	137	76
TAINTED_INT.PTR	82	58
TAINTED_PTR	242	70.5
TAINTED.INT_OVERFLOW	796	85

В данном примере для массива *buf* выделено достаточно памяти для копирования туда строки *env*.

Для отсеивания таких ситуаций при копировании строки детектор узнает идентификатор, который является размером выделенной памяти для строки, после чего проверяет длины каких строк содержит данный идентификатор, если среди этих строк присутствует `dst`-строка, то ошибка не выдается. В примере, для переменной `buf` выделена память размера (*strlen*(*env*) + 1), данный размер выделенной памяти включает в себя длину строки `env`, поэтому срабатывание не выдается.

Похожая ситуация есть и со `strcat`. Разница в том, что при присоединении новой строки проверяется включает ли в себя размер выделенной памяти `src`-строку со множеством строк из которых состоит `dst`-строка.

В случае, если помеченную строку сравнивали с другой строкой при помощи функций сравнения (`strcmp`), то срабатывания так же не выдается. Для идентификации таких помеченных строк используется тернарный атрибут *SanitizationInvoked*. Он помечает переменные, которые использовались в функциях сравнения строк.

6. РЕЗУЛЬТАТЫ

6.1. Анализ проектов с открытым исходным кодом

Для оценки процента истинных срабатываний использовался исходный код проекта Tizen 6 [18]. Проект Tizen – это открытая операционная система на базе ядра Linux. Общий размер проанализированного кода с помощью Svace составил более 32 миллионов строк. Результаты анализа приведены в табл. 1. Для оценки было размечено как минимум по 40 предупреждений для каждого вида детекторов. При этом не оценивалось эксплуатируемость ошибки. Предупреждение считалось истинным, если код содержит передачу небезопасных данных в критические операции; проверка достижимости пути из точек входа в программу не производилась.

Детектор TAINTED.INT_OVERFLOW оказался довольно шумным. Возможно, для него требуется доработка, чтобы исключить малополезные предупреждения. Большинство найденных срабатываний выглядят следующим образом:

```
int count;
count = strtol(arg, NULL, base);
```

Функция *strtol* возвращает тип *long*, поэтому потенциально возможна потеря значимой части возвращаемого значения.

6.2. Оценка Juliet 1.3

Проект Juliet [19] является тестовым набором для проверки возможностей статических анализаторов. Он включает как корректные тесты, где анализатор должен выдавать данные, так и ошибочные тесты, где анализатор не должен выдавать предупреждение.

Таблица 2. Процент пропусков для проекта Juliet

CWE	Количество тестов	Покрытие	FN(%)
CWE680	384	206	46.35
CWE194	816	444	45.59
CWE195	816	444	45.59
CWE789	384	92	76.04
CWE127	240	104	56.67
CWE124	240	104	56.67
CWE126	390	104	73.33
CWE400	624	164	73.72
CWE134	1200	226	81.17
Всего	5094	1888	62.93

Общее покрытие тестов составило 38.32%.

Из набора типов ошибок в Juliet были взяты те, которые могут быть связаны с помеченными данными: CWE680, CWE194, CWE195, CWE789, CWE127, CWE124, CWE126, CWE134, CWE400. Данные тесты компилировались с опцией omitgood, которая скрывает все тесты, где ошибка отсутствует. Тесты используют специальную систему именования: имя теста можно разделить на части, каждая часть несет некоторую информацию, например, тип ошибки или же источник и приемник [20]. Основную информацию об используемых функциях в тесте можно получить из части под названием Functional Variant Name. Из получившейся выборки были отсеяны следующие тесты:

• тесты, которые компилируются только для windows; у таких тестов в Functional Variant Name содержатся w32, wchar;

• тесты, которые не содержат помеченные данные; у таких тестов в Functional Variant Name содержится не испорченный источник, а именно, функции: rand, new и другие.

На получившейся выборке было измерено покрытие тестов. Если один из помеченных детекторов выдавал ошибку в тесте, он считался покрытым. Все непокрытые тесты относятся к FN (False Negative). Табл. 2 содержит данные о проценте непокрытых текстов.

Для данной выборки также было измерено количество ложных срабатываний. Для этого компиляция тестов происходила с опцией omitbad, которая скрывает все тесты с ошибкой. Соответственно любое срабатывание будет ложным. Для тестовой выборки количество ложных срабатываний незначительно и составило 0.47%.

7. ЗАКЛЮЧЕНИЕ

Был описан межпроцедурный контекстно- и потоково-чувствительный анализ помеченных данных для программ на C, C++, Java, Kotlin и Go для поиска уязвимостей. В анализе используются хорошо известные и проверенные решения, которые можно встретить в других анализаторах.

Общая уникальная схема анализа была разработана за более, чем 10-летний опыт написания статических анализаторов. Получившееся решение не позволяет найти все уязвимости в программе, но процент найденных ошибок выше 38.32% для тестов Juliet.

8. БЛАГОДАРНОСТИ

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект № 20-01-00581 А.

СПИСОК ЛИТЕРАТУРЫ

1. *Belevantsev A., Borodin A., Dudina I. et al.* Design and development of Svace static analyzers. In Proc. of the 2018 Ivannikov Memorial Workshop (IVMEM), 2018. P. 3–9.
2. *Бородин А.Е. и Белеванцев А.А.* Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН. 2015. Т. 27. Вып. 6. С. 111–134. https://doi.org/10.15514/ISPRAS-2015-27(6)-8 / A. Borodin, A. Belevancev. A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels. Trudy ISP RAN/Proc. ISP RAS. 2015. V. 27. Iss. 6. P. 111–134 (in Russian).
3. *Borodin A., Belevantsev A., Zhurikhin D., and Izbyshev A.* Deterministic static analysis. In Proc. of the 2018 Ivannikov Memorial Workshop (IVMEM), 2018. P. 10–14.
4. *Иванников В.П., Белеванцев А.А., Бородин А.Е. и др.* Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН. 2014. Т. 26. Вып. 1. С. 231–250. https://doi.org/10.15514/ISPRAS-2014-26(1)-7 / V. Ivannikov, A. Belevantsev, A. Borodin et al. Svace: static analyzer for detecting of defects in program source code. Trudy ISP RAN/Proc. ISP RAS. V. 26. Iss. 1, 2014. P. 231–250 (in Russian).
5. Aleph One. Smashing the stack for fun and profit. Phrack magazine. 1996. V. 7. Iss. 49. P. 14–16.
6. National Vulnerability Database – CWE Over Time. 2020. URL: https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time. Accessed 15.01.2021.
7. *Landi W.* Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS). 1992. V. 1. № 4. P. 323–337.
8. *Hind M.* Pointer analysis: haven't we solved this problem yet? In Proc. of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2001. P. 54–61.
9. *Landi W.* Interprocedural aliasing in the presence of pointers. PhD Thesis, The State University of New Jersey, 1992, 268 p.
10. *Landi W., Ryder B.G.* A safe approximate algorithm for interprocedural aliasing. ACM SIGPLAN Notices. 1992. V. 27. № 7. P. 235–248.
11. *Livshits B., Sridharan M., Smaragdakis Y. et al.* In defense of soundiness: a manifesto. Communications of the ACM. 2015. V. 58. № 2. P. 44–46.
12. *Белеванцев А., Избышев А., Журихин Д.*. Организация контролируемой сборки в статическом анализаторе Svace. Системный администратор. 2017. Вып. 7–8. С. 135–139 / A. Belevantsev, A. Izbyshev, D. Zhurikhin. Monitoring program builds for Svace static analyzer. System Administrator. 2017. Iss. 7–8. P. 135–139 (in Russian).
13. *Bush W.R., Pincus J.D., and Sielaff D.J.* A static analyzer for finding dynamic programming errors. Software-Practice and Experience. 2000. V. 30. Iss. 7. P. 775–802.

14. *Aiken A., Bugrara S., Dillig I. et al.* An overview of the saturn project. In Proc. of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. 2007. P. 43–48.

15. *Babic D. and Hu A.J.* Calysto: scalable and precise extended static checking. In Proc. of the 30th international conference on Software engineering. 2008. P. 211–220.

16. *Кошелев В.К., Игнатьев В.Н., Борзилов А.И.* Инфраструктура статического анализа программ на языке C#. Труды ИСП РАН. 2016. Т. 28. Вып. 1. С. 21–40. DOI: / V. Koshelev, V. Ignatiev, A. Borzilov, and A. Belevantsev. SharpChecker: static analysis tool for C# programs. Programming and Computer Software. 2017. V. 43. № 4. P. 268–276. https://doi.org/10.15514/ISPRAS-2016-28(1)-2

17. *Мулюков Р. Р., Бородин А.Е.* Использование анализа недостижимого кода в статическом анализаторе для поиска ошибок в исходном коде программ. Труды ИСП РАН, 2016г., том 28, вып. 5, стр. 145–158 / R.R. Mulyukov, A.E. Borodin. Using unreachable code analysis in static analysis tool for finding defects in source code. Trudy ISP RAN/Proc. ISP RAS. 2016. V. 28. Iss. 5. P. 145–158 (in Russian). https://doi.org/10.15514/ISPRAS-2016-28(5)-9.

18. Tizen 6.0 Public M2 Release. URL: https://www.tizen.org/blogs/bighoya/2020/tizen-6.0-public-m2-release-0, accessed 15.01.2021.

19. *Black P.E.* Juliet 1.3 Test Suite: Changes from 1.2. US Department of Commerce, National Institute of Standards и Technology, 2018, 37 p.

20. Juliet Test Suite v1.2 for C/C++ User Guide. Center for Assured Software, National Security Agency, 2012, 41 p.