Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок

Арутюн Аветисян <arut@ispras.ru>, Алексей Бородин <alexey.borodin@ispras.ru>

Аннотация. В ИСП РАН разрабатывается инструмент статического анализа Svace для поиска ошибок в исходном коде программ на языках Си и Си++. Цель Svace - найти как можно больше ошибок при низком количестве ложных срабатываний и разумном использовании имеющихся ресурсов. Важными требованиями, предъявляемыми к системам статического анализа являются масштабируемость и расширяемость. В статье описывается встроенный механизм, поддерживающий включение в систему Svace детекторов новых видов ошибок, сохраняющий ее масштабируемость. Использование механизма иллюстрируется на примере четырёх разработанных детекторов ошибок.

Ключевые слова: статический анализ; анализ потока данных; расширяемость; ошибки разыменования нулевого указателя; ошибки работы с динамической памятью.

1. Введение

В ИСП РАН разрабатывается инструмент статического анализа Svace для поиска ошибок в исходном коде программ. Текущая версия инструмента позволяет анализировать программы, написанные на языках Си и Си++. Цель Svace - найти как можно больше ошибок при низком количестве ложных срабатываний и разумном использовании имеющихся ресурсов.

Важными требованиями, предъявляемыми к Svace являются масштабируемость и расширяемость. Масштабируемость необходима для возможности анализа больших по размеру программ с пропорциональным увеличением потребляемых ресурсов по сравнению с небольшими программами. Расширяемость требуется для возможности обработки новых видов ошибок. Расширяемость достигается с помощью системы расширяемых модулей и спецификаций. Основное внимание данной статьи будет посвящено проблеме расширяемости.

2. Краткое описание Svace

Входная программа на языке Си (или Си++) преобразуется с помощью компилятора gcc в набор модулей с промежуточным представлением на языке LLVM, этот набор подаётся на вход анализатору Svace.

На первом этапе Svace просматривает все файлы и строит граф вызовов функций.

На втором этапе этот граф обходится в обратном топологическом порядке, таким образом, что каждая функция посещается после того, как были посещены вызываемые из неё функции. При этом рекурсивные вызовы игнорируются, что может вносить некоторые неточности в анализ. На практике существенных проблем, связанных с рекурсивными функциями, обнаружено не было.

Для каждой посещаемой функции производится её анализ. Результатом анализа функции будет выдача найденных предупреждений, а также создание аннотации функции. Аннотация описывает эффект от вызова данной функции в произвольном месте программы. После этого функция уже не будет переанализироваться, а при её вызове для вычисления побочных эффектов используется аннотация.

Более подробно узнать о Svace можно в статьях [1-9].

3. Внутри процедурный анализ

При анализе функции строится ее граф потока управления, после чего проводится потоково-чувствительный анализ, аналогичный анализу потока данных. С каждой дугой графа потока управления ассоциируется контекст – информация о потоке данных, установленная для путей выполнения, проходящих через данную дугу. Большинство инструкций имеют один входной контекст, который затем преобразуется в выходной контекст в соответствии с семантикой инструкции. Инструкции слияния потока управления принимают несколько входных контекстов и создают на их основе один выходной. А инструкции ветвления могут иметь несколько выходных контекстов.

Контекст описывает взаимосвязь между следующими элементами: абстрактными ячейками памяти, идентификаторами значений и атрибутами. Абстрактные ячейки памяти моделируют ячейки памяти, к которым происходит обращение в программе на различных путях исполнения.

Идентификаторы значений обозначают группу возможных значений, которые может иметь ячейка памяти. Несколько ячеек памяти могут разделять одни и теже идентификаторы значений. Например, подобное происходит после

обработки инструкции присваивания – обе ячейки памяти, участвующие в инструкции присваивания будут иметь одно и тоже значение после её выполнения.

```
//a → val1, b →val2
a = b;
//a →val2, b →val2.
```

Рис. 1. Присваивание.

На рис. 1 показано, что 2 ячейки памяти a и b после выполнения инструкции будут иметь одни и те же значения val2. При этом val2 – это не одно какое-либо конкретное значение, а целый класс значений, которые ячейка памяти может иметь в результате работы программы на различных путях выполнения.

Атрибуты позволяют описывать интересующие свойства. Атрибутами можно помечать идентификаторы значений переменных, области памяти и точки графа потока управления. Большинство видов анализов в Svace оперируют с атрибутами идентификаторов значений, т.к. интересующие свойства являются свойствами значений. Например, при поиске ошибок разыменования нулевого указателя необходимо проверить, что значение переменной, которую разыменовывают, не равно нулю. Для этого можно создать специальный атрибут, который будет обозначать свойство, что значение равно нулю (противоположное свойство - “значение не обязательно равно нулю”), и помечать им идентификаторы значений переменных, которые точно равны нулю. После этого в момент разыменования можно проверить значение указателя и значение атрибута.

Иногда для анализа свойств только значений не достаточно. Примерами могут служить функции завершения программы, подобные exit. В этом случае необходимо пометить путь выполнения программы после вызова функции exit как недостижимый, для чего граф потока управления помечается специальным атрибутом. Контекст на этой части графа не участвует в операциях слияния, а код, после вызова функции exit, может быть помечен как недостижимый.

Для примера атрибутов, которые ассоциируются с абстрактными ячейками, можно рассмотреть анализ утечек памяти. Утечки ищутся только для таких ячеек, которые выделены в динамической памяти. Чтобы правильно производить анализ необходимо специальным атрибутом пометить сами абстрактные ячейки памяти.

```
int* p = new[10];

p[2] = 7;
```

Рис 2. Атрибуты ячеек памяти.

На рис. 2 область памяти, на которую указывает указатель p, выделена в динамической памяти. Вторая инструкция присваивает константу 7 ячейке

памяти p[2]. При этом меняется значение ячейки, но сама ячейка по прежнему остаётся в динамической памяти. Поэтому атрибут, обозначающий динамическую память, некорректно ассоциировать со значением ячейки (т.к. после операции присваивания ячейке будет соответствовать новое значение без нужного атрибута).

4. Описание спецификаций функций

Для многих библиотечных функций Svace имеет спецификации, описывающие побочные эффекты от вызова функции. Наличие таких спецификаций продиктовано 2-мя обстоятельствами:

1. Исходный код библиотечных функций как правило отсутствует.

2. Часто сложно вывести семантику работы функции, анализируя её код.

Спецификация представляет собой функцию на языке Си, включающую вызов специальных предопределённых функций Svace, которые обрабатываются анализатором особым образом. Результатом анализа спецификации также является аннотация, хранящая все указанные эффекты от вызова функции.

Сокращённая спецификация для функции malloc приведена на рис. 3.

```
void *malloc(size_t size) {
      void* ptr;
      sf_overwrite(&ptr);
      sf_overwrite(ptr);
      sf_set_alloc_possible_null(ptr);
      sf_new(ptr, MALLOC_CATEGORY);
      return ptr;
}
```

Рис 3. Спецификация для malloc.

sf_overwrite сообщает, что изменяется значение указываемой ячейки памяти. Таким образом, с помощью двух вызовов sf_overwrite сообщается, что malloc инициализирует значение указателя ptr, а также указываемой памяти. Далее с помощью функции sf_set_alloc_possible_null сообщается, что указатель может иметь нулевое значение (но не обязательно имеет). И затем вызывается функция sf_new, чтобы показать, что ptr указывает на вновь выделенную в динамической области память. Константа MALLOC_CATEGORY указывает способ выделения новой памяти. Это требуется для возможности в дальнейшем проверить, что память была корректно освобождена с помощью нужной функции освобождения памяти.

Спецификация для free приведена на рис. 4.

```
void free(void *ptr) {
      sf_overwrite(ptr);
```

```
        sf_delete(ptr, MALLOC_CATEGORY);
}
```

Рис. 4. Спецификация для free.

В спецификации для free показывается, что изменяется состояние памяти, на которую указывает входной аргумент. А также то, что память освобождается. При этом освобождение происходит для памяти, выделенной с помощью функции malloc.

Пользователь может создавать свои собственные спецификации. Например, можно легко создать детектор, который будет проверять, что вся память, созданная пользовательской функцией xmalloc, освобождена с помощью xfree. Для этого достаточно создать аналогичные спецификации для этих функций и изменить константу MALLOC_CATEGORY на некую другую, например XMALLOC_CATEGORY.

Не смотря на то, что спецификации обрабатываются несколько иначе, чем обычные функции, момент вызова функции обрабатывается одинаково, при этом используется только аннотация функции, которая имеет единый формат.

В настоящее время созданы спецификации для большинства функций стандартной библиотеки языка Си. Для поддержки другой библиотеки необходимо создать спецификации для её основных функций, корректно описывающие побочные эффекты вызова.

5. Описание расширяемых модулей

Детектор - это компонент инструмента анализа, ответственный за поиск предупреждений определенного типа (или схожих типов). Детектор считается межпроцедурным, если он распространяет атрибуты за пределы одной функции и при выдаче предупреждений учитывает побочные эффекты от вызова функций.

Большинство детекторов реализовано в виде расширяемых модулей, которые регистрируются в диспетчере расширений и затем получают уведомления об интересующих событиях. Расширяемые модули расширения представляют из себя классы языка Java, реализующие интерфейс расширения Svace. Некоторые модули являются вспомогательными и не участвуют в выдаче предупреждений, а только распространяют атрибуты.

Примеры событий, которые доступны модулям расширения: add (сложение), sub (вычитание), deref (разыменование), bufferAccess (доступ к элементу массива), apply (трансляция атрибутов из контекста вызываемой функции в контекст вызывающей), annotate (создание аннотации). Все события

позволяют работать с атрибутами: проверять другие атрибуты или устанавливать свои, а также выдавать предупреждения.

Для межпроцедурных детекторов необходимо реализовать обработчики annotate и apply. Annotate позволяет внутренние атрибуты перевести в аннотацию функции, а apply применить аннотацию к контексту вызывающей функции в момент вызова.

Система расширяемых модулей позволяет не только улучшить расширяемость программы, но и в некоторых случаях уменьшить время анализа. Все общие действия выполняются 1 раз, при необходимости вызываются обработчики, на которые подписаны расширяемые модули, при этом работа всех модулей осуществляется одновременно. Модули не видят результаты друг друга до завершения операции, но при этом используют общие структуры данных. Благодаря чему после реализации нового детектора общее время работы программы увеличивается незначительно. Верно и обратное – если отключить половину всех детекторов, то скорость анализа не увеличится вдвое, т.к. Svace по прежнему должен тратить время на вычисление общих данных.

Каждый расширяемый модуль работает со своим набором атрибутов. Изменения этих атрибутов не должны влиять на другие модули. В некоторых случаях атрибут необходимо сделать видимым для других модулей, чтобы они могли переиспользовать её. Для этого необходимо зарегистрировать атрибут как видимый.

6. Описание типов атрибутов

Атрибуты группируются по типам атрибутов, которые позволяют разделять разные свойства и задавать возможное поведение самих атрибутов. Разные типы атрибутов могут иметь разный формат значений. Svace имеет набор предопределённых типов атрибутов, которые могут использоваться в качестве основы для пользовательских атрибутов. Наиболее важные будут перечислены в этой главе.

Часто во время анализа надо отследить какое-то свойство. Например, что значение указателя равно нулю. В данном случае не важно какое конкретное значение имеет указатель. При этом не интересуют случаи, когда указатель может быть равен нулю. Необходимо определить ситуации, когда указатель равен нулю на всех путях выполнения программы, и только в этом случае выдать предупреждение в случае его разыменования. В противном случае получится слишком много ложных срабатываний. Для того, чтобы отследить подобные свойства имеется двоичный атрибут BooleanFlag, который имеет 2 возможных значения – истина или ложь. Во время слияния графа потока управления есть 2 основные стратегии объединения двоичных атрибутов – для

описания отношений “может” и “должен”. Отношение “может” означает некоторое свойство, которое выполняется хотя бы на одном пути выполнения программы, отношения “должен” обозначает свойство, которое истинно на всех путях выполнения программы.

Соответственно имеются 2 подтипа OrBooleanFlag и AndBooleanFlag, которые ведут себя по разному в момент слияния. В примере с разыменованием описывается отношение “должен”, поэтому лучше использовать атрибут AndBooleanFlag, который будет сопоставлен результату слияния только в том случае, если описываемое свойство выполнялось на всех ветвях слияния.

В некоторых случаях интересны сразу оба отношения “может” и “должен”. Можно создать два двоичных атрибута для этого случая, но удобнее оперировать одним. Для этого используется троичный атрибут TernaryFlag с тремя значениями: “истина”, “ложь” и “возможно истина” (или “может быть”). Значение “истина” обозначает, что некоторое свойство всегда выполняется, значение “может быть” - истинно только на некоторых путях выполнения программы. Т.е. “истина” реализует отношение “должен”, а “может быть” отношение “может”.

Существуют и множество других предопределённых атрибутов. Интервальный атрибут позволяет оценить диапазон возможных значений для целых чисел. Параметризованный атрибут позволяет связать вместе две переменные. Подобное бывает необходимым при доступе к массиву. При этом в зависимости от детектора, либо массив имеет атрибут с ссылкой на индекс, либо наоборот.

7. Детекторы

7.1 FREE_OF_ARITHM

Наиболее простой из описываемых детекторов. Находит подозрительное использование функций освобождения памяти, заключающееся в удалении памяти по смещённому указателю. Позволяет найти ошибки, допущенные по невнимательности. В хорошо отлаженных проектах как правило подобное использование является намеренным.

```
static void
limited_free (gpointer mem)
{
      gpointer real = ((char*)mem) - HEADER_SPACE;
      //...
      free (real);
```



```
}
```

Рис. 5. Освобождение памяти по смещённому указателю (gtk+2.0).

В примере выше память, на которую ссылается указатель real, получена по смещению HEADER_SPACE. Такое использование является подозрительным, поэтому Svace сообщает об ошибке. В данном случае такое использование не является ошибкой, и пакет gtk+2.0 соответствующим образом переопределяет другие операции работы с динамической памятью.

Для реализации детектора, позволяющего находить подобные ситуации необходимо запомнить, что указатель был получен смещением другого указателя, а затем проверить, что значение такого указателя, не передаётся в функцию освобождения памяти.

Всю информацию о переменных и их значениях в Svace необходимо указывать с помощью атрибутов. Для того, чтобы пометить указатель как смещённый, был создан атрибут resultOfArithmeticOpFlag (результат арифметической операции). Так как предупреждение необходимо выдавать когда указатель сдвинут на всех путях, то в качестве типа атрибута был выбран AndBooleanFlag. Детектор подписывается на события add и sub, и проверяет что первый операнд имеет тип указателя, а второй является целым ненулевым числом. Операции add (сложение) и sub (вычитания), таким образом, применённые к указателям обозначают операцию смещения. После чего можно установить атрибут resultOfArithmeticOpFlag на значение результата операции. Больше не требуется никаких действий с этим атрибутом. В момент слияния путей флаг результата будет установлен автоматически, если этот флаг был установлен для всех сливаемых значений. В случае операции присваивания переменная получит новое значение, поэтому не требуется никаких дополнительных действий, чтобы снять данный флаг.

```
static void
limited_free (gpointer mem, int mode)
{
      gpointer real;
      if(mode==1) real = ((char*)mem) - HEADER_SPACE;
      else real = ((char*)mem) - HEADER_SPACE_EX;
      //...
      free (real);
}
```

Рис. 6. Слияние.

Если немного модифицировать экземпляр кода, как на рис. 6., то предупреждение также будет выдано, т.к. в момент слияния путей на разных ветках условного оператора, Svace сопоставит что переменная real имеет абстрактное значение val1 (равное ((char*)mem) – HEADER_SPACE) на одном

пути, и абстрактное значение val2 (равное ((char*)mem) – HEADER_SPACE_EX) на другом пути. После слияния переменной будет присвоено новое объединённое абстрактное значение val3, которое будет включать общие свойства значений на всех путях. После чего будут вызваны обработчики слияния для всех атрибутов этих значений. Эти действия производятся инфраструктурой Svace и нет необходимости заботиться о них при реализации детекторов.

На этом обработка смещения завершена, но требуется также правильно обработать вызов функции free для освобождения памяти. Для этого достаточно создать обработчик функции free, который будет вызван в нужный момент. Для очень многих детекторов такого решения достаточно, но в данном случае оно имеет некоторые недостатки. Во-первых, появляется жёсткая привязка к функции free. Если будут вызваны другие функции освобождения памяти, то придётся менять исходный код анализатора. Во-вторых, не будут обработаны межпроцедурные ситуации, когда вызывается некоторая функция, которая в свою очередь вызывает функцию free.

Чтобы решить первую проблему необходимо описание функции сделать спецификацией. Для обработки межпроцедурных ситуаций, надо создать атрибут, который будет реагировать на операции apply и annotate (см. Описание расширяемых модулей). Но в данном случае этих действий тоже не требуется, т.к. Svace уже имеет спецификацию для функции free (рис. 4), и модуль, который отслеживает, что память будет освобождена; при этом эта область памяти будет помечена специальным атрибутом deleteAttr. При такой реализации наш детектор будет находить также ситуации, в которых память освобождается оператором delete языка Си++, либо нестандартной функцией, для которой была создана аналогичная спецификация.

Осталось только среагировать на подозрительную ситуацию и в случае необходимости выдать предупреждение. Для этого надо подписаться на событие apply, которое будет вызвано в момент трансляции атрибутов из контекста вызываемой функции в контекст вызывающей. Это событие имеет два параметра: dst и src – оба параметра обозначают одну и ту же область памяти, но разные атрибуты (в случае, если область именованная, то они обозначают переменные в вызывающей функции и вызываемой функции). Параметр dst обозначает ячейку памяти на стороне вызывающей функции до инструкции вызова, параметр src обозначает эту же ячейку на стороне вызываемой функции после инструкции вызова. Все межпроцедурные атрибуты должны быть перенесены с параметра src на параметр dst, при этом каждый модуль переносит только те атрибуты, за которые он отвечает. В момент распространения атрибутов необходимо решить какие атрибуты из вызываемой функции перейдут в вызывающую. Для данного детектора достаточно только проверить, что dst имеет значение, помеченное флагом resultOfArithmeticOpFlag,а область памяти src помечена атрибутом deleteAttr.

Фактически это будет означать, что смещённый указатель был передан в функцию, которая освободит память, на которую он указывает.

7.2 FREE_INCOMPATIBLE и DELETE_INCOMPATIBLE

Предупреждение FREE_INCOMPATIBLE выдаётся в случае нахождения ситуаций несоответствия функций выделения памяти и функций освобождения. Например, при смешивании Си++ операторов new/delete с функциями Си библиотеки malloc/free. На практике встречается не часто.

Интересен подвид предупреждения (DELETE_INCOMPATIBLE), описывающий часто-встречающуюся ошибку, связанную с непониманием правильной работы с операторами выделения/освобождения памяти языка Си++ операторов new[]/delete. Если память была выделена с помощью оператора new[], то необходимо освободить её оператором delete[] для массивов. Частой ошибкой является освобождение памяти с помощью оператора delete для одиночных элементов. Нарушение этого правила ведёт к неопределённому поведению. Даже если в текущей версии программы не было выявлено ошибок, они могут возникнуть при смене компилятора, или изменении опций компилирования. Поэтому такие ошибки сложно находить в ходе ручного тестирования, но статические анализаторы легко справляются с такой задачей.

```
size_t SndFile::write(int srcChannels, float** src,
size_t n) {
      // ...
      float *buffer = new float[n * dstChannels];
      if (srcChannels == dstChannels) {
            // ...
      else {
            printf("SndFile:write channel mismatch %d ->
%d\n",
                  srcChannels, dstChannels);
            delete buffer; //необходимо использовать
delete[] buffer.
            return 0;
       }
      int nbr = sf_writef_float(sf, buffer, n) ;
      delete buffer; //необходимо использовать delete[]
buffer.
      return nbr;
}
```

Рис. 7. Неправильное освобождение памяти (проект muse).

Далее будет приведено описание работы детектора DELETE_INCOMPATIBLE, т.к. он немного проще, но при этом принципиальных различий в реализации с FREE_INCOMPATIBLE нет.

Для реализации необходимо создать один атрибут createPtrAttr для того, чтобы пометить результат new[], и один атрибут deletePtrAttr, чтобы пометить, что оператор delete освобождает свой аргумент несовместимым образом. Эти атрибуты устанавливаются на значения указателей, чтобы понять почему именно на значения, а не на сами указатели, стоит посмотреть следующий пример:

```
char* array = new char[MAX_PATH];

char* ptr = array;

delete ptr;
```

Рис. 8. Вновь присваивание.

В момент присваивания ptr = array свойства области памяти, в которой находится указатель ptr, не меняются, поэтому область ptr не будет помечена атрибутом createPtrAttr, и ошибка не будет найдена. Если же пометить значение указателя этим атрибутом, то в момент присваивания указатель ptr получит значение указателя array, которое уже помечено этим атрибутом, и ошибка будет найдена.

Т.к. в Си++ часто встречаются маленькие функции, то анализ надо делать межпроцедурным. Это в том числе позволит найти ошибки, когда выделение массива происходит в конструкторе некоторого класса, а некорректное освобождение в деструкторе. Чтобы сделать детектор межпроцедурным, достаточно реализовать обработчики событий apply и annotate. В данном случае реализация тривиальная – надо просто скопировать атрибут из источника в приёмник.

Детектирование ошибки так же не представляет сложности и делается аналогично как для детектора FREE_OF_ARITHM. В обработчике apply необходимо проверить, что переменная в контексте вызова имеет значение, помеченное атрибутом createPtrAttr, а значение переменной в контексте вызываемой функции помечено атрибутом deletePtrAttr.

При описании этого детектора заметно, что очень многие детали похожи, либо идентичны другим детекторам. Аналогично при реализации детектора не возникает множества новых проблем, практически всё необходимое уже встречалось в других детекторах и вынесено в общую для всех модулей часть, работа с который осуществляется анализатором Svace.

7.3 FREE_INCONSISTENT

Как правило, функция освобождения памяти должна обрабатывать входной аргумент в любом случае, независимо ни от каких условий. Если же освобождение происходит только в некоторых случаях, то это может привести к трудноуловимым утечкам памяти. Детектор FREE_INCONSISTENT проверяет, что входной параметр некоторой функции освобождается на некоторых путях, и не освобождается на других.

Как и FREE_INCOMPATIBLE анализ также выполняется над значениями указателей. Но в данном случае двоичного атрибута уже будет недостаточно. Если функция освобождения вызывается на всех путях, то ошибки нет, если же она не вызывается вообще, то ошибке тоже нет. Фактически необходимо найти ситуации, когда функция освобождения вызывается на некоторых путях, для этой цели прекрасно подходит троичный атрибут TernaryFlag.

При реализации нет необходимости создавать отдельную спецификацию для функции освобождения памяти, т.к. она уже создана. Необходимо только добавить действия на обработчик sf_delete, который будет устанавливать атрибут deletePtrTernaryAttr.

Троичный атрибут уже имеет обработчик, вызываемый в момент слияния путей. Если на обоих путях значение было равно “истина”, то и результат будет “истина”, если же на обоих атрибут имел значение “ложь”, то и результат будет “ложь”. В остальных случаях будет установлено значение “может быть”.

Для детектора FREE_INCONSISTENT значение “ложь” будет означать, что функция освобождения ни разу не была вызвана для данного указателя. Значение “истина” означает, что функция была вызвана на всех путях. А значение “может быть”, что функция была вызвана на некоторых путях, фактически это и есть искомая ситуация. Когда завершится анализ функции, Svace начнёт создавать аннотацию, для этого он вызовет обработчик annotate для необходимых атрибутов. В этот момент и можно проверять, что флаг deletePtrTernaryAttr имеет значение либо “ложь”, либо “истина”.

Важно подчеркнуть, что анализ будет производиться для каждого отдельного значения.

```
void freeSomething(char* ptr1, char* ptr2, int index) {
	if(index==1)
		free(ptr1);
	else
		free(ptr2);
}
```

Рис. 9. Слияние путей.

В этом надуманном примере функция free вызывается на всех путях, но в первом случае она вызывается для ptr1, а во втором для ptr2. Таким образом

для входного параметра ptr1 функция free вызывается только на одном из путей выполнения, поэтому будет выдано предупреждение.

Если бы в начале функции существовало присваивание ptr1 = ptr2, тогда оба указателя имели бы одно и тоже значение и ошибки выдано не было бы. Т.к. присваивания нет, то Svace считает параметры разными, если по каким-либо причинам параметры являются псевдонимами друг друга, то будет выдано ложное предупреждение.

В ближайших планах является создание дополнительного анализа указателей, который во многих случаях позволит понять, что указатели являются псевдонимами.

7.4 DEREF_OF_NULL.RET

Часто, когда функция возвращает указатель, нулевое значение используется в случае возникновения какой-либо ошибки. В этом случае необходимо проверить, что указатель не нулевой, перед его разыменованием. Для случая отсутствия подобной проверки Svace имеет предупреждение DEREF_OF_NULL.RET.

Все такие функции можно разделить на 2 категории: библиотечные и пользовательские.

В документации к библиотечной функции как правило явно задано, что она может вернуть нуль, при этом тело функции может быть недоступно. Поэтому имеет смысл делать явные спецификации для подобных функций.

```
char *getenv(const char* key) {

      char *str;
      sf_overwrite(&str);
      sf_set_possible_null(str);
      return str;
}
```

Рис. 10. Спецификация для getenv.

Обработчик специальной функции sf_set_possible_null должен установить атрибут PossibleNull на значение аргумента. Этот атрибут означает, что значение некоторого указателя может быть равно нулю, поэтому указатель желательно проверить на нуль перед разыменованием. Атрибут имеет подтип OrBooleanFlag – в случае слияния путей, результат будет иметь этот атрибут, если его имеет хотя бы один из аргументов. Очевидно, что реализуется отношение “может”, то есть ищутся ситуации, когда переменная равна нулю хотя бы на одном пути выполнения программы.

Необходимо заметить, что создавать спецификации нужно только для таких функций, которые возвращают нуль в случае неконтролируемой ошибки, которая напрямую не зависит от входных аргументов, например в случае отсутствия файла. Иначе будет множество ложных срабатываний, т.к. результат функции может быть детерминирован и зависеть от потока управления. На рис. 11 на первой строке у массива buf последнему элементу присваивается символ новой строки. Таким образом массив будет гарантированно иметь этот символ хотя бы 1 раз. Далее с помощью функции strchr ищется первое вхождение символа новой строки и вместо него устанавливается символ окончания строки, таким образом обрубая строку в этом месте. Функция strchr возвращает нуль, если ничего не было найдено. Таким образом для предотвращения ошибки разыменования нулевого указателя необходимо проверить, что результат strchr не нулевой. Но в данном примере, очевидно, что функция всегда вернёт ненулевое значение.

```
buf[sizeof(buf) - 1] = '\n';
*strchr(buf, '\n') = '\0';
```

Рис. 11. Вызов детерминированной функции (проект busybox).

Для реализации в самом простейшем случае необходимо проверить, что во время разыменования значение указателя имеет атрибут PossibleNull, и в этом случае выдать предупреждение (подтипа DEREF_OF_NULL.RET.LIB). Также необходимо реагировать на сравнение указателя с нулём, чтобы понять, что указатель проверен на нуль. Для простой реализации достаточно снимать этот атрибут, если указатель сравнивается с нулевой константой.

Более полная реализация будет отслеживать, что другие указатели могут иметь нулевое значение, и соответственно сравнение с ними может быть эквивалентно сравнению с нулевой константой.

Довольно интересно использование функции malloc стандартной библиотеки. Эта функция выделяет память заданного размера и в случае неудачи возвращает нулевой указатель. На современных системах, где при нехватке оперативной памяти, её часть может сгружаться на диск, такая ситуация довольно редка. Кроме этого, часто в случае ошибки сделать уже ничего нельзя, поэтому многие программисты не проверяют результат malloc. По этой причине для функции malloc выделено специальное предупреждение DEREF_OF_NULL.RET.ALLOC. Если в проекте сознательно игнорируют проверку результата функции malloc, то это предупреждение можно легко отключить.

Поиск ошибок для пользовательских функций немного сложнее. Необходимо по коду определить, что функция иногда возвращает нулевое значение. Механизм определения не будет подробно описываться. Основная идея – использовать вспомогательный тернарный атрибут, у которого значение “истина” означает, что указатель равен нулю, а значение “ложь”, что не равен. В случае слияния путей, если один из атрибутов истинен, а другой ложен, значение результата будет “возможно истинен”, что фактически означает, что

указатель может иметь нулевое значение. А это почти эквивалентно семантике атрибута possibleNull, разница лишь в том, что атрибут possibleNull устанавливается на возвращаемое значение.

Детектирование ошибок с пользовательскими функциями абсолютно аналогично, с той лишь разницей, что выдаётся предупреждение DEREF_OF_NULL.USER. И также как и в случае с библиотечными функциями возможны ложные срабатывания, когда код, возвращающий нуль, каким-либо образом зависит от входных аргументов. В этом случае при использовании функции можно заранее знать, что она не вернёт нулевое значение, т.к. её аргументы не случайны. Поэтому можно не проверять возвращённый указатель. Отличить подобные ситуации с помощью статического анализа очень сложно и первое время предупреждение имело очень высокий уровень ложных срабатываний. Какие-либо попытки улучшить результат за счёт усложения модели анализа не давали существенных результатов. Svace выдавал огромное количество предупреждений, которые на первый взгляд казались истинными, но при более тщательной проверке выяснялось, что в данном конкретном случае функция не может вернуть нулевое значение.

Улучшить ситуацию удалось с помощью технологии Z-ranking. Подробно она описана в [10]. Эта технология позволяет откинуть часть предупреждений на основе статистической информации. Идея заключается в том, что если результат функции никогда не проверяется в коде, следовательно, существует негласное правило, что она не может вернуть нулевое значение. Если же результат, в большинстве случаев проверяется, а иногда нет, то в последнем случае с большой долей вероятности удалось найти истинную ошибку. Z-ranking позволяет дать количественную оценку вероятности, что предупреждение будет ложным. Таким образом, можно не только отсеить предупреждения, которые могут быть ложными, но и отсортировать все предупреждения в порядке уменьшения вероятности, что они будут истинными.

Список литературы

[1] С.С. Гайсарян, А.В. Чернов, А.А. Белеванцев, О.Р. Маликов, Д.М. Мельник, А.В. Меньшикова. О некоторых задачах анализа и трансформации программ. Труды ИСП РАН, No5, с.7-41, 2004.

[2] О.Р. Маликов, А.А. Белеванцев. Автоматическое обнаружение уязвимостей в программах. Материалы конференции «Технологии Майкрософт в теории и практике программирования», Москва, 2004.

[3] О.Р. Маликов. Автоматическое обнаружение уязвимостей в исходном коде программ. Известия ТРТУ, No4, с. 48-53, 2005.

[4] В.С. Несов, О.Р. Маликов. Использование информации о линейных зависимостях для обнаружения уязвимостей в исходном коде программ. Труды ИСП РАН, No9, с. 51-57, 2006.

[5] О.Р. Маликов, В.С. Несов. Автоматический поиск уязвимостей в больших программах. Известия ТРТУ, Тематический выпуск «Информационная безопасность», No7 (62), с.114-120, 2006.

[6] В.С. Несов. Использование побочных эффектов функций для ускорения автоматического поиска уязвимостей в программах. Известия ЮФУ. Технические науки. Тематический выпуск «Информационная безопасность». Таганрог: Изд-во ТТИ ЮФУ, 2007. No 1(76), с. 134-139.

[7] В.С. Несов, С.С. Гайсарян. Автоматическое обнаружение дефектов в исходном коде программ. Методы и технические средства обеспечения безопасности информации: Материалы XVII Общероссийской научно-технической конференции. СПб.: Изд-во Политехн. ун-та, 2008, с.107.

[8] В.С. Несов. Автоматическое обнаружение дефектов при помощи межпроцедурного статического анализа исходного кода. Материалы XI Международной конференции «РусКрипто'2009».

[9] Vladimir Nesov. Automatically Finding Bugs in Open Source Programs. Electronic Communications of the EASST 20. ISSN 1863-2122, 2009.

[10] T. Kremenek, D. Engler. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. Static Analysis, pp. 295-315, 2003 — Springer.