Министерство образования и науки Российской Федерации

МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ

(государственный университет)

ФАКУЛЬТЕТ УПРАВЛЕНИЯ И ПРИКЛАДНОЙ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАММИРОВАНИЯ

(Специализация «Прикладная математика и физика»)

Разработка и реализация метода поиска дефектов связанных с динамической памятью в бинарном коде

Выпускная квалификационная работа бакалавра

студента 375 группы

Петтика Никиты Юрьевича

Научный руководитель

Гайсарян С.С., к. ф.-м. н., доцент

г. Долгопрудный

2017

Содержание

Введение

Данная работа является исследованием, относящимся к области статического анализа кода. В настоящее время статический анализ, как исходного, так и бинарного кода является ключевой составляющей разработки и написания программного обеспечения. Так как ошибки неизбежны, то следует своевременно осуществлять их поиск и исправления, поскольку они приводят к неправильному поведению программы, а иногда и являются потенциальными источниками уязвимостей. Стоит отметить, что поиск ошибок без специальных инструментов отнимает значительное количество сил и времени. Сам статический анализ кода подразделяется на два основных уровня: анализ исходного кода и бинарного. Каждый из подходов имеет свои недостатки и преимущества, однако даже после статического анализа исходного кода остаются ошибки, просачивающиеся в бинарный код. Так же не стоит забывать про принцип WYSINWYX[1] — современные оптимизирующие компиляторы могут проводить множества неочевидных и сложных преобразований кода, вследствие чего бинарный код существенно отличается от того, что ожидал получить программист. Например, порядок вычисления операндов у некоторых операций в языках C/C++ не регламентирован стандартом, следовательно разные компиляторы могу выдать последовательности инструкций, производящих вычисления в разном порядке. Кроме того, компилятор может удалить часть кода, которая кажется ему бессмысленным. Так, в случае с GCC можно применить опцию компиляции -fstrict-aliasing, которая говорит компилятору, что указатели разных типов не могут ссылаться на одну и ту же ячейку памяти, давая тем самым право на удаление любых обращений к памяти, если тип указателя и тип хранящихся данных не соответствуют друг другу.

Важность своевременного анализа и поиска дефектов так же косвенно подтверждается неумолимо растущей с каждым годом базой CWE и нахождением уязвимостей “нулевого дня” как в новых, так и казалось бы, проверенных временем проектах.

Стоит отметить, что статический анализ бинарного кода становится особенно актуальным и в связи с появлением так называемого интернета-вещей, где исключительно важным является защищенность от уязвимостей используемых прошивок. Иногда, код для них пишется на самом низком уровне — напрямую в машинных инструкциях.

[1] What You See Is Not What You eXecute

В коммерческих целях нередко приходится использовать сторонние, уже скомпилированные библиотеки. Поэтому, чтобы обеспечить надежность разрабатываемого программного обеспечения, следует быть уверенным в безопасности используемых библиотек и отсутствии в оных наличия уязвимостей, дефектов и закладок.

Работы по поиску ошибок в программном коде начали появляться с 90-х годов прошлого века, однако долгое время они не могли похвастаться удовлетворительным количеством истинных срабатываний и демонстрировали продолжительное время работы.

Постановка задачи

При статическом анализе код исследуется без запуска самой программы. Этот факт накладывает некоторые ограничения на возможности статического анализа. К тому же, в бинарном коде отсутствует информация о типах и переменных как таковых, так как инструкции оперируют непосредственно регистрами и ячейками памяти. Более того, не всегда удается восстановить информацию о прототипах функций. Однако, основным преимуществом анализа бинарного кода является тот факт, что анализируются именно те инструкции, которые в конечном итоге и выполняются при запуске программы, при условии отсутствия самомодифицирующегося кода.

Вообще говоря, статический анализ бинарного кода несколько сложнее исходного[1], поэтому большинство инструментов для поиск ошибок, связанных с динамической памятью используют динамический подход. Это связано с тем, что динамический анализ позволяет относительно просто отслеживать алиасы — копии данного указателя. Более того, исчезают неоднозначности выполнения предиката ветвления, ведь в статическом анализе не всегда можно проверить истинность захождения выполнения программы в ту или иную ветвь графа потока управления, в связи с чем возникают неоднозначности в определении статусов указателей. Однако, динамический анализ не всегда может сгенерировать входные данные для получения произвольного пути графа потока управления или на это может потребоваться большое количество времени. По этой причине, статический анализ является более подходящим методом анализа, если требуется сфокусироваться на покрытии кода. Основными трудностями статического анализа принято считать точный потоко-нечувствительный анализ указателей и растущее количество ограничений, вызываемое предикатами ветвления[6].

Важным классом критических ошибок являются ошибки, связанные с динамической памятью. В то время, как такие высокоуровневые языки программирования как Java, Python,

C# обладают автоматической системой выделения динамической памяти и сборки мусора, их более быстрые аналоги — C и C++ предоставляют только возможности выделения и освобождения памяти вручную, с использованием функций стандартной библиотеки — malloc, calloc, new, free, delete и их производных. Такое управление памятью с одной стороны дает большой простор для оптимизаций. Например, можно сразу отдавать системе освобожденную память, что в особенности актуально для встроенных систем, где не такой уж и большой объем памяти и её следует экономно расходовать. Другое применение — запросить сразу большой блок динамической памяти для последующего размещения в ней множества мелких объектов, без повторного вызова функции выделения памяти (этот прием реализуется в C++ за счет оператора placement new), что увеличивает скорость работы. С другой стороны такой подход возлагает полную ответственность за происходящее на программиста. Последствия некорректного использования динамической памяти могут варьироваться от неопределенного поведения программы и аварийного завершения, до банальных утечек памяти. Ключевыми критическими ошибками, связанными с динамической памятью являются использование памяти после её освобождения и двойное освобождение памяти.

Таким образом, целью работы является создание метода поиска ошибок связанных связанных с динамической памятью. Для этого необходимо:

- Реализовать алгоритм в среде BinSide
- Разработать план внутрипроцедурного анализа
- Построить формальную модель динамической памяти
- Улучшить алгоритм символьного выполнения для корректной обработки указателей
- Произвести сравнительную оценку методов межпроцедурного анализа
- Протестировать разработанный и реализованный алгоритм поиска ошибок как на наборе синтетических тестов, так и на реальных проектах

1. Обзор работ

1. “WYSINWYX”[1] — в данной работе описываются основополагающие принципы статического анализа бинарного кода. Авторы подчеркивают особенности анализа бинарного кода и приводят описание различных подходов для его реализации. Кроме того, в данной работе описывается абстрактная модель памяти, которая подходит для описания как статической так и динамической памяти. Так же дается описание VSA — потоко-чувстви-

тельного, контексто-зависиммого, межпроцедурного алгоритма, основанного на абстрактной интерпретации. Данный подход используется в большинстве статических инструментов для определения множеств значений переменных. Основная идея заключается в задании множества значений для первоначальных значений регистров и ячеек памяти и последовательного применений передаточных функций для каждой инструкции. В итоге, получается набор формул, описывающих произвольное состояние программы, включающий в себя абстрактные домены. Таким образом, данная техника позволяет узнать поведение программы на всех возможных входах и для всех точек выполнения, которые могут быть достигнуты. Как результат работы, авторы представили инструмент CodeSurfer, способный проводить анализ бинарного кода для архитектуры x86, а так же набор программ для поиска различных дефектов в коде. CodeSurfer предоставляет возможность строить системный граф зависимостей, а так же содержит графический интерфейс для навигации по графу, позволяя получать статические сечения — набор переменных и предикатов, описывающих текущее состояние программы.

2. “Statically detecting use after free on binary code”[2] — в данной статье представлен инструмент для статического поиска Use-After-Free уязвимостей в бинарном коде. Авторы использовали IDA PRO для получения дизассемблерного кода и графа потока управления. Далее, в среде BinNavi код транслируется в REIL представление, используемое для анализа во фрейморке MonoREIL. Так же применяется абстрактная интерпретация для отслеживания значений указателей. Однако для межпроцедурного анализа используется инлайнинг заранее выбранных функций, что существенно замедляет процесс анализа. Еще одной особенностью является возможность извлечения путей графа потока управления, которые привели к появлению уязвимости: от места выделения памяти, до использования уже освобожденной памяти. Сам проект является прототипом и написан на языке Jython, который сам по себе является достаточно медленным для анализа кода.

3. “LoongChecker: Practical summary-based semi-simulation to detect vulnerability in binary code”[3] — в данной статье описывается созданный алгоритм для поиска различных дефектов в бинарном коде. Аналогично предыдущей работе, авторы используют фрейморк BinNavi и дизассемблер IDA PRO. Однако, в этом случае используется расширение языка eREIL, которое лучше подходит для статического анализа. Например, при трансляции ин-

струкции сравнения в оригинальное REIL представление, так же транслируются и значения, выставленные в регистре $eflags$, которые могут быть использованы только при динамическом анализе, а при статическом не несут никакой пользы. После трансляции, запускается внутрипроцедурный анализ для построения резюме выполнения функции, которое далее используется в межпроцедурном анализе. Подход используемый в данной работе называется “полу-симуляция” — комбинация точного VSA алгоритма, описанного в статье[1], и анализа зависимости данных. Кроме того, данный детектор соединили с фаззингом, для подтверждения найденных уязвимостей. Авторы продемонстрировали работоспособность данного инструмента обнаружив две уязвимости нулевого дня в проектах Serenity Player и Kingsoft Office Writer.

2. Идеи и методы решения задачи

2.1. Общая структура анализа

Как правило, предметом анализ кода является граф потока управления и/или граф потока данных. Дизассемблирование бинарного кода, а так же построение графа вызовов и графа потока управления являются нетривиальными задачами. Поэтому для этих целей был использован дизассемблер IDA PRO. В некоторых случаях, IDA так же позволяет восстановить прототип функции, что положительно сказывается на точности анализа. Далее, все части экспортируется через специальный плагин во фреймворк BinSide, который содержит бинарный транслятор во внутреннее REIL представление. Этот код и используется для анализа. Задаются начальные аннотации для функций выделения и освобождения памяти. Сам межпроцедурный анализ запускается по слоям — от вызванных функций к вызывающим. Таким образом, анализ стартует из листьев и заканчивается в функции $main$. Внутри каждой функции запускается внутрипроцедурный анализ, основой которого является символьное выполнение. Осуществляя проход по графу потока управления, для каждой инструкции вычисляется набор актуальных регистров и ячеек памяти и множество символьных выражений, хранящихся в них. Для каждой инструкции загрузки и сохранения по указателю на динамическую память создается ограничение, содержащее используемый указатель. В конце анализа функции строится набор аннотаций, который используется при вызове данной функции. Так же для всех найденных ограничений проверяется условие появления критической ошибки: если указатель в

данном состоянии имел статус “освобожден”, то в результат анализа функции добавляется информация о возможном дефекте.

2.2. Ограничения и предположения подхода

Так как нет никаких сведений о реальном размещении блоков динамической памяти и работы системы выделения памяти в целом, то используется стандартное предположение о том, что каждый раз выделяется новый блок памяти, что, естественно не так. Пример такого поведения рассматривается в разделе 2.1. В связи с использованием символьного вычисления и межпроцедруного анализа, на выходе некоторых функций получаются длинные символьные выражения. Поэтому, иногда нет смысла передавать дальше такие формулы, так как они плохо поддаются какой-либо обработке, что снижает точность анализа. Кроме того, несмотря на то, что используется внутреннее представление ассемблерного кода, по прежнему многое зависит от исходной архитектуры и соглашения о вызовах. Например, параметры функции могут передаваться через стек, а могут через регистры, возвращаемое значение так же может передаваться по-разному. В данный момент инструмент показывает стабильную работу только для x64 архитектуры, так как аргументы в x86 передаются через стек, а распознание стековых аргументов не является простой задачей. Некоторые ограничения так же связаны и со статической природой анализа: не всегда можно предсказать, по какой ветке пойдет исполнение программы, так как невозможно вычислить значение предиката условного перехода. В связи с этим, приходится использовать некоторые эвристики, которые не могут гарантировать точность анализа, но тем не менее показывают хороший результат приближения.

2.3. Программные средства

В качестве фреймворка для анализа бинарного кода был выбран разрабатываемый в ИСП РАН проект BinSide, который предоставляет широкий спектр возможностей для анализа и редактирования ассемблерного кода. Стоит отметить, что кроме работы с бинарным кодом исходной машины, BinSide позволяет для анализа использовать свое внутреннее представление, называемое REIL[2], которое основывается на RISC архитектуре виртуального процессора[4]. Такое представление получается за счет бинарной трансляции и поддерживается для большинства современных архитектур: x86, x86_64, ARM, PowerPC и тд. Данная

[2] Reverse Engineering Intermediate Language

трансляция дает возможность несколько абстрагироваться от архитектуры конкретной машины и упростить анализ бинарного кода. Немаловажным является тот факт, что REIL представление содержит всего 17 инструкций, в то время как только в x64 насчитывается более 400 инструкций. Следует подчеркнуть, что необязательно реализовывать транслятор для каждой инструкции исходной архитектуры, так как редко встречающиеся команды можно транслировать в инструкцию *UNKNOWN*, которая никак не влияет на анализ кода. Каждая из REIL инструкций содержит ровно три операнда, при этом в некоторых инструкциях операнды могут опускаться, и выполняет только одно действие без каких-либо сторонних эффектов, таким образом избавляя нас от необходимости следить за флагами переноса, сравнения и т.д., что существенно облегчает анализ кода. Каждая из машинных инструкций транслируется в последовательность REIL команд, при этом используется неограниченное количество временных регистров. Временные регистры не имеют физического отображения и используются только локально, т.е. для каждой транслируемой инструкции будет создан, если требуется, новый временный регистр, значение которого будет использовано только для трансляции исходной инструкции. Для поддержания соответствия между ассемблеровскими и REIL инструкциями, последние имеют дополнительное смещение, умноженное на 0x100. Кроме того, BinSide содержит интерпретатор REIL кода, который позволяет запускать и отлаживать код в данном представлении.

Для первоначального дизассемблирования, построение графа потока управление и графа вызовов для исходного бинарного кода, используется дизассемблер IDA PRO, с последующим импортом в базу данных BinSide с помощью соответствующего плагина.

3. Поиск дефекта

3.1. Описание ошибки

Одной из самых частых и легко эксплуатируемых ошибок, связанной с динамической памятью является Use-After-Free(CWE-416), иными словами — использование указателя на динамически выделенную память после её освобождения. Наиболее простой, и в то же время одной из наиболее распространенных, иллюстрацией данного дефекта является следующий код:

```
char *ptr = malloc(SIZE);
…
if (error){
   free(ptr);
}
…
printf("%s", ptr);
```

Пример 1

Нетрудно заметить, что при выполнении некоторых условий, исполнение программы зайдет в ветку, в которой вызывается функция $free$. В этом случае, вызов функции $printf$ будет содержать ошибку. Таким образом, данная ошибка имеет место, если выполняются следующие 3 условия:

1) Выделяется динамическая память
2) Память освобождается, но указатель все еще доступен
3) Указатель разыменовывается, что приводит к доступу к освобожденной памяти

Стоит отметить, что в зависимости от платформы и от реализации системы распределения динамической памяти, данный дефект может привести к разным последствиям и разным способам эксплуатации. В большинстве случаев это приводит к утечке информации. Например, функции из библиотеки glibc используют first-fit алгоритм для выбора свободного блока памяти[5]:

```
char* a = malloc(512);
char* b = malloc(256);
char* c;
strcpy(a, "this is A!");
free(a);
c = malloc(500);
strcpy(c, "this is C!”);
```

Пример 2

После выполнения данного кода, переменная *c* будет содержать указатель на ту же память, что и переменная *a*, т.е. обратившись к указателю *a* получим строку "this is C!", несмот-

ря на то, что мы освободили память, на которую указывает *a*. Поэтому, часто хорошей практикой чистого кода является обнуление указателей после их освобождения.

Но что более интересно, в некоторых случаях данную ошибку можно эксплуатировать для запуска вредоносного кода. Такой прием осуществляется посредством следующих шагов:

1) Программа выделяет, а затем освобождает блок динамической памяти A

2) Атакующая сторона выделяет блок динамической памяти B, используя память выделенную под A

3) Атакующая сторона записывает исполняемый код в блок B

4) Программа использует освобожденный блок A, запуская тем самым вредоносный код

В C++ такой способ часто используется в комбинации с подменой указателя на таблицу виртуальных функций для объектов, содержащих виртуальные методы.

Другой, но так же распространенной ошибкой является Double-Free(CWE-415), то есть двойное освобождение одного и того же указателя на динамическую память. Рассмотрим следующий пример:

```
int *a = malloc(8);
int *b = malloc(8);
free(a);
//free(a);  Повторное освобождение приведет к ошибке, так как в данный момент в
//начале списка освобожденной памяти будет указатель a
free(b);
free(a); //Теперь в списке освобожденной памяти будут указатели a, b, a
```

Пример 3

Следует подчеркнуть две особенности. Во-первых, если освободить два раза подряд один и тот же указатель, то скорее всего программа просто аварийно завершится. Во-вторых, если после освобождения указателя *a*, освободить какой-нибудь другой указатель, а потом снова *a*, то *a* дважды попадет в список доступной для выделения памяти. Таким образом, *malloc* дважды вернет указатель на одну и ту же память.

Кроме того, механизм открытия/закрытия файлов в C/C++ организован так же через указатели, поэтому описаный выше подход можно так же использовать и для поиска обращений к закрытым файлам, ведь согласно стандарту, любое обращение к указателю на закрытый

файл влечет за собой неопределенное поведение. Это можно сделать, заменив функции выделения динамической памяти на функции открытия файлов, а функции освобождения — функциями закрытия файлов.

3.2. Особенности REIL представления

В REIL представлении непосредственно с памятью взаимодействуют только две инструкции:

1. $STM\ reg, , addr$ — сохраняет содержимое регистра reg по адресу $addr$
2. $LDM\ addr, , reg$ — загружает содержимое по адресу $addr$ в регистр reg

Так же есть инструкция $STR\ reg1, , reg2$, которая записывает содержимое регистра или числа $reg1$ в регистр $reg2$.

Вызов функции происходит за счет инструкции условного перехода: $JCC\ , , funcAddr$

Перед вызовом функции аргументы кладутся в регистры $rdi/rsi/rdx$ и т.д.

После вызова функции выделения памяти, значение указателя сохраняется из возвращаемого регистра в ячейку памяти. Разыменовавание указателя можно определить по загрузке из памяти по адресу $@(ptr + *base)$, где $*base$ - смещение сегмента. В символьном выражении это может быть $ssbase$ для стека, $dsbase$ и $csbase$ для сегмента глобальной памяти.

4. Внутрипроцедурный анализ

Существует два подхода к анализу графа потока управления:

1. Внутрипроцедурный анализ, который ограничен пределами анализируемой функции
2. Межпроцедурный анализ, который предоставляется возможность ходить по графу вызовов и собирать любую информацию

Очевидно, что внутрипроцедурный анализ гораздо проще реализовать и для поиска некоторых ошибок этого вполне хватает. Однако, если целью является нахождение ошибок связанных с динамической памятью, то следует отслеживать полный путь жизни блока памяти — от выделения до освобождения. Как показывает практика, зачастую это происходит в разных функциях. Поэтому имеет смысл сначала проводить внутрипроцедурный анализ каждой функции, суммировав каким-либо образом информацию о том, что происходит внутри функции и проводить анализ остальных функций, используя эту информацию.

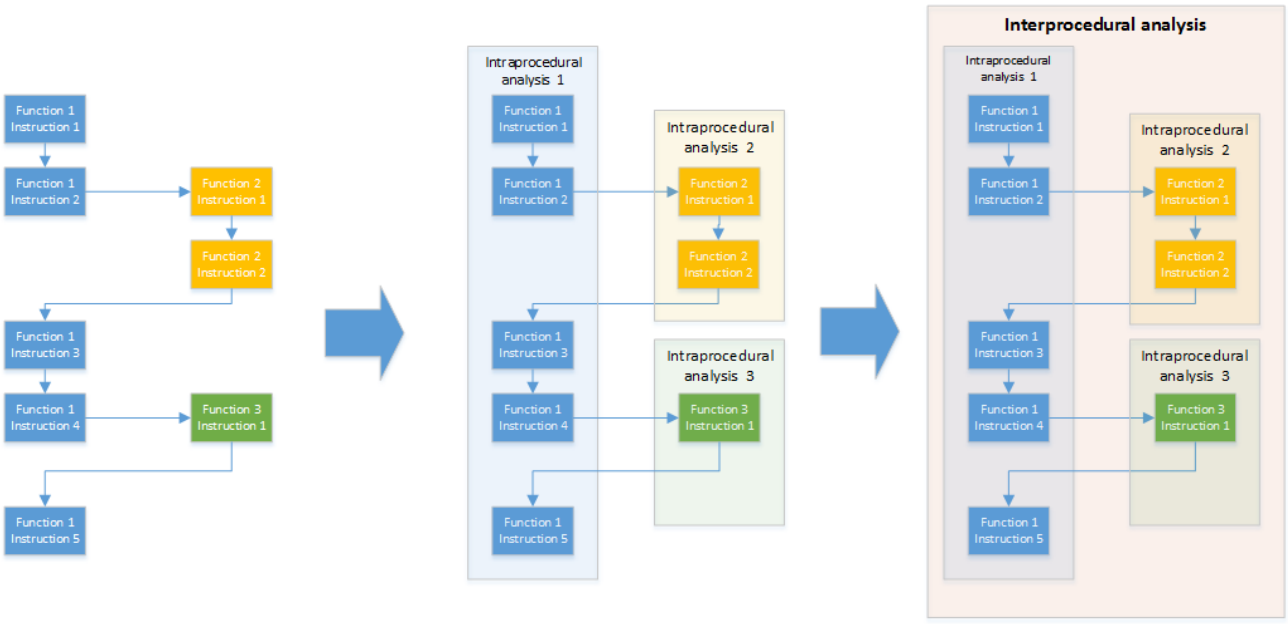

Схема 1: Межпроцедурный анализ как объединение результатов внутрипроцедурного

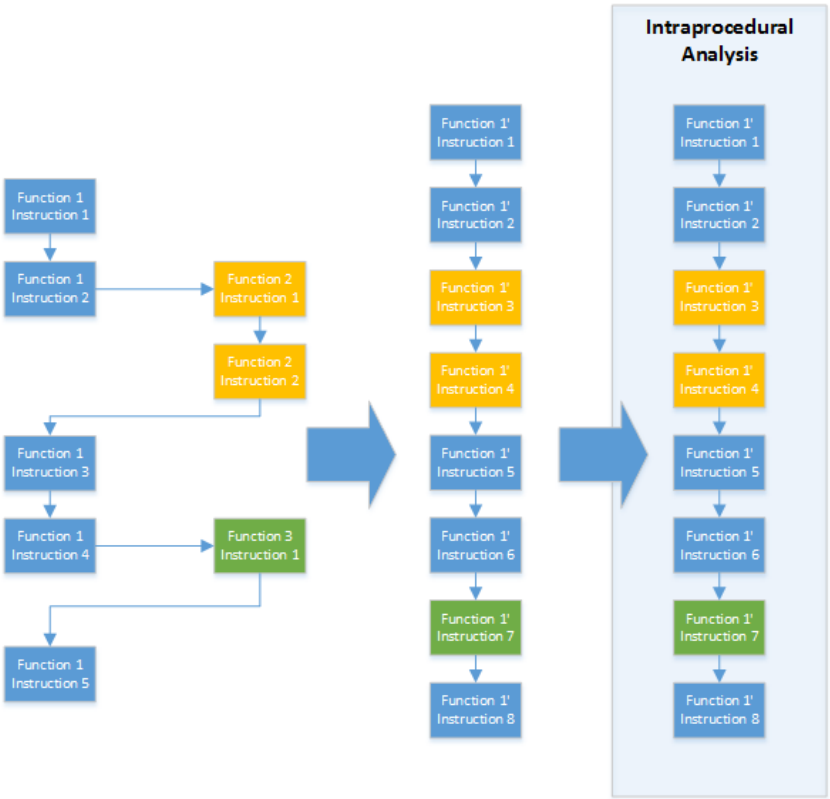

Схема 2: Межпроцедурный анализ с использованием подстановки тела функции

Существуют и другие подходы. Например, в работе [2] используется полное включение тела вызываемой функции. Такой подход достаточно прост в реализации, но накладывает некоторые ограничения. Так, функция, вызванная два раза будет и проанализированная дважды, даже если имеет одинаковый результат работы. Таким образом, данный метод потребляет в разы больше памяти и времени.

В данной работе реализована первая модель межпроцедурного анализа, называемая анализом с помощью аннотаций. Для начала рассмотрим схему поиска ошибок для внутрипроцедурного анализа.

4.1. Формальная модель динамической памяти

Статический поиск ошибок связанных с динамической памятью в бинарном коде сопряжен с некоторыми трудностями по сравнению с поиском других дефектов из-за отсутствия какого-либо шаблона, по которому можно искать появление ошибки. Данная задача так же осложняется из-за особенностей реализации функций, выделяющих и освобождающих память, а так же модели памяти в целом, отслеживанием значений и статусов указателей. Вследствие чего, приходится использовать свою модель памяти, отслеживать указатели и их перемещения по всем путям графа потока управления.

Построим упрощенную версию модели памяти, описанной в работе [1]. В такой модели, обозначим множество HA как множество всех выделенных и не освобожденных элементов динамической памяти, HF — множество всех освобожденных элементов динамической памяти, HE — множество всех возможных элементов динамической памяти. Тогда $HA \in HE$, $HF \in HE$. Элемент множества HE задается уникальным идентификатором и размером, которые образуют блок памяти. Пусть PC — множество всех точек выполнения программы. Так же для каждой точки выполнения программы $p \in PC$ определим множество $HeapAccess$, которое для каждого элемента $h \in HE$ содержит множество возможных регистров и ячеек памяти, содержащих указатель на h. После каждого вызова функции выделения или освобождения памяти будет применяться соответствующая передаточная функция, которые имеют следующий вид:

$f_{alloc}(p, HA, HF, HeapAccess, size, id) = (HA', HF, retVal, id')$, где $retVal = (id, size)$ — возвращаемый функцией новый блок памяти, $HA' = HA(p) \cup \{retVal\}$, $id' = id + 1$.

$f_{free}(p, HA, HF, HeapAccess, arg) = (HA', HF')$, где arg — аргумент функции, $HA' = HA(p) \backslash HeapAccess(arg), HF' = HF(p) \cup HeapAccess(arg)$.

Для отслеживания перемещений указателей и алиасов, следует так же записать передаточные функции для инструкций загрузки и сохранения в том случае, когда их аргументами являются элементы из множества $HeapAccess$:

$f_{STM}(reg, addr, HeapAccess) = HeapAccess'$, где $\exists h : reg \in HeapAccess(h)$, $HeapAccess' = HeapAccess \cup \{h, addr\}$ — добавляем в множество алиасов для элемента h ячейку с адресом $addr$. Если $\nexists h : reg \in HeapAccess(h)$ и $\exists h : @addr \in HeapAccess(h)$, то $HeapAccess' = HeapAccess \backslash \{h, @addr\}$

$f_{STR}(reg1,reg2,HeapAccess) = HeapAccess'$, где $\exists h : reg1 \in HeapAccess(h)$, $HeapAccess' = HeapAccess \cup \{h, reg2\}$. Если $\nexists h : reg1 \in HeapAccess(h)$, но $\exists h : reg2 \in HeapAccess(h)$, то $HeapAccess' = HeapAccess \backslash \{h, reg2\}$.

$f_{LDM}(addr, reg, HeapAccess) = HeapAccess'$, где $\exists h : @addr \in HeapAccess(h)$, т.е. по в ячейке памяти с адресом $addr$ лежит указатель на блок динамической памяти, $HeapAccess' = HeapAccess \cup \{h, reg\}$. Если $\nexists h : @addr \in HeapAccess(h)$ и $\exists h : reg \in HeapAccess(h)$, то $HeapAccess' = HeapAccess \backslash \{h, reg\}$

То есть, если перезаписать указатель значением не из списка указателей на динамическую память, например обнулив указатель, то надо убрать его из списка синонимов.

Тогда, обозначим $danglingSet$ как множество указателей на освобожденную память: $danglingSet = \{(p, (id, size)) | (id, size) \in (HeapAccess(p) \cap HF(p))\}$. Таким образом, если в какой-либо точке выполнения программы происходит разыменовывание “висячего” указателя, то регистрируется Use-After-Free ошибка. В REIL представлении это будет команда $LDM\ addr, , reg$, где $\exists h : @addr \in HeapAccess(h)$ и $h \in danglingSet$. В ходе выполнения анализа так же возможно обнаружение следующих дефектов:

- Double-Free ошибок: вызов передаточной функции $f_{free}(p, HA, HF, HeapAccess, arg)$, где $HeapAccess(arg) \in HF$.

- Утечек памяти: в конце выполнения функции $HA \neq \varnothing$ и $heapAccess(HA(p)) = \varnothing$, т.е. указателей на память не осталось, но сама выделенная память присутствует.

- Глобальных "висячих" указателей: в конце выполнения функции $\exists HeapAccess(global) \in HF$.

- Вызова функции освобождения памяти от неинициализированных указателей: $f_{free}(p, HA, HF, HeapAccess, arg)$,где $HeapAccess(arg) \notin HF(p), HeapAccess(arg) \notin HA$.

Несмотря на кажущуюся простоту такой модели, она позволяет полностью отслеживать все перемещения и синонимы указателей.

4.2. Символьное выполнение

Для реализации алгоритма приведенного выше, необходимо отслеживать значения, передаваемые в функцию освобождения памяти. Есть несколько способов анализа значений переменных, один из которых, символьное выполнение, и реализован в BinSide.

В BinSide существует фреймворк MonoREIL для анализа потока данных на основе работы решеток. Суть этого подхода заключается в определении решетки L — упорядоченного множества элементов с определенной функцией сравнения, передаточных функций ϕ для инструкций, которые переводят один элемент решетки в другой, и определение функции F для задания порядка зависимости инструкций друг от друга. Далее запускается итеративный процесс преобразования решетки. Условием завершения служит нахождение точки сходимости — такой точки, которая преобразуется сама в себя, либо ограничение по количеству итераций, но в таком случае получается не полное решение. В нашем случае, используется именно последний вариант, причем количество итераций устанавливается равным двум. Это соответствует одному проходу по циклу и корректному объединению состояний решетки после ветвления.

На основе этого инструмента реализовано символьная интерпретация значений выражений в функциях. Для этого были написаны передаточные функции ϕ для всех REIL инструкций, которые отображают входные операнды в конечный выходной, строя символьное выражение. При этом элементами решетки L являются наборы значений регистров и ячеек памяти в каждой инструкции, а функция F соответствует графу потока управления.

Поскольку входные параметры функции, значения глобальных переменных и адреса динамически выделенных блоков памяти неизвестны, вводятся абстрактные символьные элементы для описания значений выражений в функции: Addition — сложение двух элементов; Multiplication — умножение двух элементов; Subtraction — вычитание двух элементов; Division — деление двух элементов; BitwiseAnd — битовое И двух элементов; BitwiseOr — битовое ИЛИ двух аргументов; BitwiseXor — битовое исключающее ИЛИ для двух элемен-

тов; BitwiseShift — бинарный сдвиг; Dereference — получение значения по адресу; Either — одно из двух значений; Literal — конкретное число; NullCheck — проверка значения на 0; Range — интервал значений; Symbol — начальное значение регистра или смещение сегмента; Subtraction — вычитание двух элементов; MemoryCell — элемент, представляющий ячейку стека, или глобальной памяти; Undefined — неопределенное значение; DynMemory — динамически выделенная память; Pointer — элемент, содержащий DynMemory и смещение, относительно начала памяти. Все элементы содержат внутри себя ссылки на поддеревья операндов. В итоге, для каждого выражения получается дерево разбора выражения, где листьями являются элементы Pointer, Literal, и Symbol, а все бинарные и унарные операции являются узлами дерева. Таким образом, каждый элемент решетки, т.е. состояние для каждой инструкции, содержит ассоциативный массив <регистр/ячейка памяти, символьное значение> для всех определенных регистров и ячеек памяти. Однако, так как временные регистры имеют локальное время жизни, то для экономии памяти их значения после использования не отслеживаются. Кроме того, дополнительно хранится ассоциативный массив <указатель, статус> для анализа указателей.

Поскольку существуют ветвление и циклы, необходимо реализовать объединение значений. Так два различных числа могут объединится в интервал. При этом для объединения независимых выражений генерируется элемент типа Either, содержащий объединяемые значения в качестве операндов.

Оперировать с большими символьными формулами не очень удобно, поэтому было реализовано большое количество эвристик, позволяющих упрощать полученные выражения, как простейшие типа $X\ xor\ X = 0$ и $X\ +\ 0 = X$ так и более сложные, основанные на булевой алгебре. Все формулы с константными операндами сворачиваются.

При анализе циклов не всегда можно определить количество итераций, поэтому инструмент пытается свести возможные значения к какому-либо диапазону, либо построить формулу на основе переменной цикла. В случае невозможности построения такой формулы анализ завершается на определенной итерации и используется неполное значение.

На вход данному инструменту подается граф потока управления в REIL представлении. Анализ начинается с первой инструкции функции и проходит по графу потока управления,

последовательно вычисляя состояния для каждой инструкции, пока не обработает все инструкции.

4.3. Описание работы алгоритма

Сама схема работы устроена следующим образом: вначале программист задает множество начальных функций, которые выделяют динамическую память, таких как malloc, calloc, new и их производные. Последовательно перебирая функции, находятся соответствующие обертки из системной библиотеки и их адреса. Как было сказано ранее, шаблон REIL кода для выделения памяти на куче: $jmp\ 1,\ ,addrOfMalloc$, где $addrOfMalloc$ — адрес одной из функций, выделяющих память. В соотвествии с соглашением о вызове функций $cdecl$ и $fastcall$ для x86 и x86_64, результат вызова функции будет помещен в регистр eax или rax. Следовательно, после вызова функции выделения памяти, следует отследить, куда сохраняется значение из данного регистра: это может быть ячейка стека, глобальной памяти или динамической памяти. Но значения адресов динамической памяти доступны только во время выполнения программы, поэтому для идентификации различных блоков памяти был использован новый символьный элемент — Pointer. Это сделано из-за особенности символьного вычисления: если просто отслеживать значение регистра eax, то оно будет всегда одним и тем же, т.е. никак не учитывается разница между первым вызовом функции выделения памяти и всеми остальными. Поэтому Pointer хранит уникальный идентификатор, и после каждого вызова функции выделения памяти это значение инкрементируется. Далее, вызывается передаточная функцию для функции выделения памяти и передаточная функция для инструкции STM. Аналогично, находятся адреса функций освобождения памяти. Однако в данном случае так же необходимо знать значение аргумента функции — номер блока выделенной динамической памяти. Для этого используется символьное выполнение, которое позволяет найти значение, записанное в стек или регистр. После того, как мы узнали значение аргумента, вызывается передаточную функцию для функции освобождения памяти. Так же необходимо отслеживать перемещения указателей, поэтому каждый раз, когда встречается инструкция STM или LDM с аргументами из множества HeapAccess, то вызывается соответствующую передаточную функция. В итоге, когда встречается шаблон, соответствующий разыменовыванию указателя — инструкцию LDM с аргументами из множества HeapAccess, проверяется их наличие во множестве HF: если они там присутствует, то это и есть Use-After-Free ошибка.

Аналогично, если при вызове функции освобождения памяти указатель уже содержится в HF, то это Double-Free ошибка. Если в конце анализа функции остаются элементы в HeapAccess, являющиеся локальными переменными, то это ни что иное как утечка памяти. Если функция освобождения памяти вызывается с неинициализированным аргументом, или с указателем равным $NULL$, будет выдана соответствующая ошибка. Так же осуществляется учет глобальных указателей, которые на момент завершения выполнения функции указывают на освобожденную память — так называемые “висячие указатели”.

5. Модель межпроцедурного анализа

Описанная выше модель работает, но у нее есть ряд недостатков, вызванных ограничением видимости анализа. Например, если в программе используются обертки функций выделения памяти для структур, то приходится перед запуском добавлять их вручную в специальный список. То же самое относится и к функциям освобождения памяти. Так же, никак не учитывается состояние указателей находящихся в статическом сегменте памяти. Более того, даже в самом банальном случае, когда указатель передается в функцию, внутри которой он освобождается или разыменовывается, ошибка не поддается обнаружению в случае внутрипроцедурного анализа. Поэтому ниже рассмотрена схема межпроцедурного анализа.

Для реализации межпроцедурного анализа была выбрана модель на основе аннотаций к функциям. Основная схема анализа следующая:

1. Граф вызовов разбивается на сильносвязные компоненты

2. В каждой компоненте удаляются обратные ребра

3. Функции подвергаются топологической сортировке и разбиваются на слои в обратном порядке вызова: каждый слой содержит функции, которые вызывают только функции из предыдущих слоев

4. Идет обход графа вызовов, начиная с листовых функций. После анализа каждой функции, строятся аннотации и ограничения, которые необходимы для последующего анализа каждого типа ошибок

5. В конце анализа функции значения ограничений отфильтровываются, и остаются только те, которые потенциально могут привести к тому или иному дефекту

Очевидное преимущество такой схемы перед инлайнингом, который применяется в некоторых других работах, состоит в том, что анализ каждой функции происходит один раз, в то

время как заинлайненные функции анализируются каждый раз заново, к тому же экспоненциально увеличивая объем кода. Более того, после составления аннотаций, их можно сохранять в базу данных. Таким образом, проанализировав один раз какую-либо библиотеку, мы можем использовать результаты анализа во всех программах, использующих функции из этой библиотеки, без необходимости повторного анализа, что, конечно же, в разы ускоряет время работы анализатора. Кроме того, после внесения изменений в проект, анализу можно подвергать только измененные функции, используя уже готовые аннотации с прошлой итерации проверки проекта. Стоит отметить, что такую схему межпроцедурного анализа можно использовать не только для символьного вычисления и анализа указателей, но и для анализа помеченных данных и поиска ошибок форматной строки.

5.1. Типы аннотаций

Для реализации обозначенной схемы анализа, были созданы следующие аннотации:

1. Аннотация регистров исходной архитектуры — аннотация к функции, которая содержит все изменения нативных (т.е. не временных) регистров в данной функции. Исключение составляют регистры флагов и стековые указатели rsp и rbp - они специально отсеиваются, так как не несут никакой полезной для анализа информации. Данные хранятся в ассоциативном массиве <регистр, новое значение>. Основной смысл использования данной аннотации заключается в том, чтобы отслеживать возвращаемое функцией значение — согласно соглашению о вызове, оно хранится в регистре rax для архитектуры x64.

2. Аннотация глобальной памяти — аннотация к функции, которая содержит все значения изменений глобальной памяти, т.е. элементы $@(addr + csbase)$ или $@(addr + dsbase)$, в зависимости от того, были ли инициализированы данные, их их символьные значения. Информация хранится в ассоциативном массиве <ячейка памяти, новое значение>.

3.Аннотация на выделение динамической памяти — аннотация к функции, которая служит индикатором того, что функция возвращает новый элемент динамической памяти. Ей по умолчанию помечаются функции $malloc$, $calloc$, new, $new[]$. Разработчик так же имеет возможность добавлять в список базовых аллокаторов собственные реализации функций, которые внутри себя имеет иную структуру распределения динамической памяти. Аннотация так

же содержит регистр или ячейку памяти, в котором после вызова функции содержится новаый блок памяти. Это сделано для поддержки разных архитектур: по умолчанию это rax для $x64$ и eax для $x86$. В аннотацию так же входит регистр, в котором хранится размер запрашиваемой памяти. По умолчанию rdi для $malloc$ и $rdi * rsi$ для $calloc$ в случае архитектуры $x64$, и ячейка в стеке по адресу $esp + 4$, и $esp + 8$ для второго аргумента $calloc$ для архитектуры $x86$. Так же хранится тип функции: $malloc$, $calloc$, new, $new[]$ или $custom$.

4. Аннотация на освобождение динамической памяти — аннотация к функции, которая служит индикатором того, что функция освобождает элемент динамической памяти. Ей по умолчанию помечаются функции $free$, $delete$, $delete[]$. Аннотация содержит регистр или ячейку памяти, в которой хранится освобождаемый указатель (по умолчанию rdi для $x64$ и ячейка памяти в стеке по адресу $esp + 4$) и тип функции: $free$, $delete$, $delete[]$ или $custom$.

5. Аннотация на разыменовывание указателя — аннотация к функции, которая служит индикатором того, что функция получает доступ по адресу, содержащимся в аргументе, или элементе глобальной памяти, хотя бы по одному из путей выполнения функции. Содержит список регистров и ячеек памяти, соответствующих разыменованным указателям.

5.2. Работа символьного вычисления с аннотациями

В добавок к обычной работе символьного вычисления, следует учесть вызовы функций, у которых есть какие-либо аннотации. Таким образом, при встрече вызова функции, у которой есть набор аннотаций, следует обновить текущее состояние элементов, в соответствие с применяемой аннотацией. В случае с регистрами исходной архитектуры (аннотации с глобальными переменными работают аналогично), следует взять символьное значение регистра из аннотации и обновить его в текущем состоянии. Например, если до вызова функции с аннотацией $rax = 10$, значение регистра rax было равно $rsp + 20$, то после вызова оно станет равным 10. При этом так же происходит подстановка значений аргументов функции, если таковые присутствуют в конечном результате аннотации. Например, если до вызова с аннотацией $rax = rdi + 10$, значение регистра rdi было $rbx + 10$, то после вызова значение регистра rax станет равным $(rbx + 10) + 10 = rbx + 20$. Подстановка осуществляется рекурсивным разбором дерева выражения и заменой листьев, соответствующих аргументным реги-

страм, поддеревьями значений из текущего состояния решетки. В случае с аннотацией к выделению динамической памяти, генерируется символьный элемент — новый указатель на динамическую память, и добавляется в список указателей со статусом “не освобожден”. Если функция имеет аннотацию на освобождение одного из аргументов, то его статус в списке указателей меняется на “освобожден”. Если символьное выражение не соответствует элементу “указатель”, т.е. не содержится в списке текущих указателей, а такая ситуация может возникнуть в случае освобождения одного из аргументов функции, ведь анализ идет от вызываемых функций к вызывающим, следовательно сведения о аргументах функции недоступны, то вместо указателя добавляется в список символьное выражение. Такая ситуация так же может случится из-за неполного упрощения символьной формулы. Для решения таких проблем возможно использование решателя, на вход которому подается предикат равенства двух формул. Если решение существует при любых значениях переменных из формулы, то такие формулы тождественны и представляют один и тот же элемент.

Для отслеживания динамической памяти, в инструмент для символьного вычисления были добавлены специальные элементы DynMemory и Pointer. DynMemory содержит уникальный номер блока динамической памяти, который генерируется статическим полем аннотации, во время перехода на функцию, которая помечена аннотацией на выделение памяти; статус памяти — флаг “освобождена”, “дважды освобождена” и “возможно освобождена”; размер самой памяти, который передается в аргументе функции выделения памяти и так же является символьным выражением. Pointer содержит элемент DynMemory и смещение относительно начала выделено памяти.

Кроме того, были добавлена эвристика упрощения формулы к элементу Addition в случае, если левая/правая часть сложения Pointer, а правая/левая - Literal. В этом случае создается новый элемент Pointer с тем же значением DynMemory, но с новым значением смещения. Так же, если одним из членов элемента Either является Pointer, а другим — Literal или Range, что соответствует ситуации, когда в одной ветке графа потока управления выделяется память, а в другой — нет, то остается один элемент Pointer со статусом “возможно освобожден”.

Для отслеживания статусов указателей в элемент решетки был добавлен ассоциативный массив, состоящий из пар <указатель, статус указателя>, где в статусе хранится информация о указателе — набор флагов и адреса инструкций выделения и освобождения. Была добавлена в

функцию комбинирования состояний возможность склейки двух списков указателей, с соответствующим изменением их статусов: если в одном состоянии статус указателя “не освобожден”, а в другом - “освобожден” или “возможно освобожден”, то после объединения указатели приписывается статус “возможно освобожден”. Если в обоих состояниях один и тот же статус, то он и остается в результирующем состоянии. Аналогично происходит слияние для указателей со статусами “дважды освобожден”.

5.3. Анализ аннотаций

После основного анализа функции на предмет нахождения в ней каких-либо ошибок, так же следует определить, содержит ли она сама какие-либо аннотации.

Для начала следует найти все точки возврата из функции — такие инструкции JCC помечаются специальным флагом при трансляции исходного ассемблерного кода (обычно это инструкции ret). В случае с регистрами исходной архитектуры, для данных инструкций берутся состояния всех регистров после символьного выполнения и и отфильтровываются незначимые регистры: регистры флагов, стековые указатели, временные регистры. В случае, если у функции больше одной точки возврата, эти значения объединяются, аналогично тому, как происходит слияние двух путей выполнения после ветвления, с помощью функции комбинирования из инструмента для символьного вычисления. С ячейками глобальной памяти проводится аналогичная процедура.

Для определения аннотации на выделение динамической памяти проверяется значение регистра, в котором возвращается результат работы функции. Для архитектуры x64 этим регистром является rax: если в нем содержится элемент Pointer со статусом “не освобожден”, то такой функции приписывается аннотация на выделение памяти. Кроме того, в такой функции не осуществляется поиск аннотаций на регистры исходной архитектуры. Если вдруг в регистр rax попадет указатель извне тела функции, то он не будет элементом Pointer, а будет задаваться символьным выражением.

Для поиска аннотаций на освобождение памяти, осуществляется просмотр массива указателей для инструкций возврата: если в нем присутствует символьное выражение, включающее в себя регистр, через который передаются аргументы функции (rdi, rsi, rdx, rcx и т.д.) и соответствующий статус указателя “освобожден”, то данной функции следует приписать

аннотацию на освобождение памяти для того регистра, который присутствовал в формуле. Данной аннотацией так же помечаются функции, возвращающие освобожденный указатель. В таком случае, освобождаемый регистр будет регистром rax.

Поиск аннотаций на разыменовывание аргументов функции осуществляется путем проверки аргументов каждой инструкции загрузки и сохранения: если адресом служит регистр для хранения аргумента, то приписываем функции данную аннотацию.

Стоит отметить, что так как нет возможности проанализировать функции стандартной библиотеки, то аннотации на них приходится задавать вручную перед запуском анализа.

6. Результаты

Прежде всего, инструмент был протестирован на Juliet — наборе тестов, созданным Центром Верификации Программного Обеспечения Агентства Национальной Безопастности. Данный тестовый набор предназначается для проверки способностей статических анализаторов кода обнаруживать ошибки. Все примеры были составлены на основе реально существующих дефектов из различных проектов.

Характеристики машины, на которой запускались тесты: процессор Intel Core i7-3770@3.4GHz, 32 гигабайта оперативной памяти. Во время запуска тестов максимальное потребление памяти для Java-машины было ограничено 8 гигабайтами.

Набор тестов для Use-After-Free содержит 960 тестовых функций, из которых 480 содержат ошибку. Разработанный инструмент показал следующие результаты:

Всего функций	Содержащих дефект	Истинных срабатываний	Ложных срабатываний
960	480	190	0

Таким образом, общий процент срабатываний около 41%, что является неплохим результатом. Стоит отметить нулевое количество ложных срабатываний. Время анализа составило ~500 секунд для 10 проходов и ~150 секунд для двух проходов вычислений символьных значений, что соответствует одному проходу по циклу. Это подтверждает гипотезу о том, что сходимость и анализ циклов мало влияет на обнаружение Use-After-Free ошибок.

Кроме искусственных тестов, инструмент так же запускался и на реальных проектах, трестируемые версии которых содержали известные Use-After-Free ошибки. Результаты анализа следующие:

	Количество дефектов	Истинных Срабатываний	Ложных Срабатываний
gnome-nettool	1	1	1
openjpeg	1	1	3
jasper	1	1	9
accelppp	2	2	2

Существование ложных срабатываний в некоторых случаях объясняется сложной структурой функций и большим количеством ветвлений.

7. Выводы

Несмотря на то, что уровень срабатываний ниже, чем у статических анализаторов исходного кода, а стандартом для них считается уровень срабатываний не ниже 60%, инструмент показал способность находить ошибки в реальном программном обеспечении.

Стоит отметить, что некоторые ложные срабатывания появляются из-за того, что пока детектор не может проанализировать выполнения условия ветвления. Рассмотрим следующую функцию:

```
void* customFree(void* ptr) {
  if (isPointerValid(ptr)) {
    free(ptr);
  }
  return ptr;
}
```

Пример 4

В некоторых случаях, а именно когда выполняется условие, задаваемое функцией *isPointerValid*, указатель будет освобожден, в то время как в остальных случаях — нет. Таким образом, у указателя *ptr* в конце выполнения функции будет статус “может быть осво-

божден”. Тогда открытым остается вопрос, стоит ли в данном случае приписывать функции аннотацию на освобождение памяти, или нет? Однако, даже если предикатом *if* является константа и компилятор не провел оптимизацию данного ветвления, то пока мы не сможем сказать, что это условие выполняется всегда. Аналогично и с аннотациями на разыменование аргументов — оно может происходить только лишь по одному из путей выполнения и заранее нельзя сказать, когда будет выполняться это условие. Для анализа выполнения предикатов ветвления можно использовать решатель — инструмент для проверки существования решения системы символьных уравнений.

С другой стороны, можно приписывать аннотации на освобождение памяти и разыменовывание аргументов только тогда, когда оно происходит по всем путям выполнения. Тогда процент срабатываний будет очень маленьким, но все они, скорее всего, будут истинными. Точно так же, как и только по одному пути выполнения. Тогда общий процент срабатываний будет высоким, но высоким так же будет и уровень ложных срабатываний. Таким образом, можно задавать различную “чувствительность” анализа.

В некоторых других работах, межроцедурность достигается за счет инлайнинга функций, однако такой подход демонстрирует долгое время анализа. Например, в работе[2] используется аналогичная схема анализа бинарного кода, но время проверки проекта gnome-nettool составляет 30 минут на практически идентичной машине, в то время как разработанный инструмент завершает полный анализ за ~5 минут.

Заключение

В результате работы был разработан метод статического поиска ошибок динамической памяти в бинарном коде:

- Алгоритм реализован в среде BinSide
- Реализован метод анализа указателей для внутрипроцедурного анализа
- Построена формальная модель динамической памяти
- Улучшен алгоритм символьного выполнения для корректной обработки указателей
- Была выбрана и реализована схема межпроцедурного анализа
- Составлены эвристики для поиска разных типов аннотаций для функций
- Приведены результаты работы инструмента для поиска критических ошибок как на наборе искусственных тестов, так и на реальных проектах.

Во время разработки были так же исправлены некоторые ошибки как в самом инструменте для символьного выполнения, так и в бинарном трансляторе REIL представления. Написаны специальные юнит тесты для проверки работоспособности каждой части инструмента.

Список использованных источников

1. *G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum2* WYSINWYX: What You See Is Not What You eXecute // http://research.cs.wisc.edu/wpis/papers/wysinwyx05.pdf
2. *Josselin Feist, Laurent Mounier, Marie-Laure Potet* Statically detecting use after free on binary code // http://binsec.gforge.inria.fr/pdf/JCVHT-verimag.pdf
3. *Shaoyin Cheng, Jun Yang, Jiajie Wang, Jinding Wang and Fan Jiang* LoongChecker: Practical summary-based semi-simulation to detect vulnerability in binary code //
4. *Thomas Dullien, Sebastian Porst* REIL: A platform-independent intermediate representation of disassembled code for static code analysis// https://static.googleusercontent.com/media/www.zynamics.com/ru//downloads/csw09.pdf
5. *François Goichon* Glibc Adventures: The Forgotten Chunks // https://www.contextis.com/documents/120/Glibc_Adventures-The_Forgotten_Chunks.pdf
6. *Susan Horowitz* Precise flow-insensitive may-alias analysis is NP-hard // http://research.cs.wisc.edu/wpis/papers/toplas97a.pdf