Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра системного программирования

Михаил Андреевич Седлярский

Двунаправленное направляемое конфликтами резюмирование кода в символьной виртуальной машине V#

Магистерская диссертация

Научный руководитель:
доцент кафедры СП, к. ф.-м. н. Мордвинов Д. А.

Санкт-Петербург
2021

1 Введение

С каждым днём программное обеспечение становится всё сложнее и сложнее. А тестов, написанных людьми, уже недостаточно, чтобы проверять программы на корректность. Пропущенные программные ошибки в различных сферах человеческой деятельности могут приводить к большим потерям: начиная от задержек во времени и заканчивая человеческими жизнями. В 2002 году Грегори Тассей оценивал годовой ущерб экономике США от ошибок программного обеспечения в 59.5 миллиардов долларов США [9].

Например, в момент написания этих строк была найдена критическая уязвимость в одной из самых распространённых библиотек логирования – Log4J [6]. Данная брешь позволяет выполнять произвольный код в любом JVM приложении, использующем эту библиотеку. Символично, что патч, исправляющий уязвимость, содержал ещё одну угрозу [10].

В силу своей простоты самым часто используемым методом верификации ПО является тестирование. Входные данные для тестов могут генерироваться случайно, их могут подбирать люди, данные могут имитировать некоторые пользовательские сценарии. Также входы тестов можно подбирать таким образом, чтобы увеличивать покрытие кода тестами. Но, как было сказано выше, обычных тестов недостаточно, чтобы гарантировать корректность программного обеспечения. Например, имея тест с одним 32 битным входном, нам потребуется проверить 2^{32} различных тестовых входа. Очевидно, что за разумное время перебрать все варианты невозможно.

В связи с этим, вполне закономерно использовать дополнительные способы верификации ПО. Например, различные виды статического и динамического анализа. Статический анализ можно также использовать для формальной проверки программ т.е. для доказательства некоторых свойств корректности программного обеспечения. Среди формальных методов можно выделить символьное исполнение. Это метод моделирования работы программы с использованием символьных значений вместо конкретных. Во время символьного исполнения, встретив оператор ветвления, анализатор добавит условие (ограничение) к себе в список и продолжит анализ в обеих ветвях. Затем используются решатели ограничений, которые по заданным условиям находят входные данные, приводящие программу в интересующие нас ветви.

К сожалению, символьное исполнение тоже имеет ограничения. Количество возможных путей выполнения программы может быть очень большим. Чтобы получить комбинаторный взрыв достаточно использовать цикл с неизвестным заранее количеством итераций. Эта особенность ограничивает применение прямого символьного исполнения в промышленной разработке.

Но что если мы хотим обойти не все пути, а сойтись к какому-нибудь конкретному месту в коде? Здесь поможет *обратное символьное исполнение*. Оно начинает работу не с точки входа в программу, а с некоторого заданного места, и далее движется в обратном направлении. Данный подход можно применять в том числе и для целевой отладки приложения. Но, к сожалению, он тоже подвержен проблеме комбинаторного взрыва: одну и ту же функцию можно вызывать из множества различных мест. В том числе, и из недостижимых при нормальном выполнении.

Перспективным направлением является двунаправленное символьное исполнение. Подход заключается в том, что мы поочерёдно выполняем пря-

мые и обратные шаги. Тем самым, мы сходимся к целевой инструкции с двух сторон.

Ещё одним способом сокращения пространства поиска является резюмирование кода. Мы можем аппроксимировать функции сверху [3, 8], а затем, комбинируя полученные аппроксимации, можно отсекать целые поддеревья возможных ветвей символьного исполнения.

После обнаружения невыполнимого ограничения пути будет находится минимальное unsat-ядро формулы и на основе атомов полученного ядра строится аппроксимация исследуемого участка кода.

Данная работа посвящена разработке и реализации алгоритма резюмирования кода в двунаправленном символьном исполнении в виртуальной символьной машине V# для платформы .net core.

2 Постановка задачи

Целью данной работы является разработка и реализация алгоритма алгоритм двунаправленного направляемого конфликтами резюмирования кода в символьной виртуальной машине для платформы .net core. Для достижения этой цели были выделены следующие задачи:

1. Провести детальный обзор существующих подходов резюмирования кода и стратегий ветвлений в символьном исполнении
2. Разработать алгоритм двунаправленного направляемого конфликтами резюмирования кода
3. Реализовать алгоритм в символьной виртуальной машине V#.
4. Провести эксперименты, доказывающие эффективность предложенного алгоритма и его реализации

3 Обзор

3.1 Символьное исполнение

Символьное исполнение — это метод анализа программного обеспечения, позволяющий определить какие входные данные приводят к выполнению каждой из его частей. Символьное исполнение предлагает отличный (от обычного) метод исполнения программы, в рамках которого производится абстракция от конкретных значений.

Данный метод анализа позволяет автоматически находить возможные ошибки и уязвимости в программном обеспечении. Также перспективной областью применения может быть целевая отладка. Т.е.такая отладка, при которой программист указывает точку останова, а входные данные, приводящие в неё, подбирает символьная машина.

Механизм символьного исполнения обрабатывает каждую входную переменную как некоторое символьное значение. Символьный анализ может обрабатывать поведение программы при каждой комбинации входных значений. Благодаря символьным значениям можно надежно искать возможные пути выполнения, а также определять условия, при которых они до-

стигаются. Если у программы дерево всех путей исполнения конечно, то символьное выполнение также будет конечным.

Рассмотрим небольшой пример на рисунке 1. Попробуем определить входные данные, при которых условие в assert будет ложным. Заменим все конкретные переменные, чьи значения нельзя выяснить заранее, на символы a_i. В каждый момент времени состояние символьной машины можно представить тройкой $stmt, \sigma, \pi$, где:

1. $stmt$ – операция, которая будет выполнена на следующем шаге
2. σ – символьное хранилище, связывающее переменные программы с конкретными, либо символьными значениями
3. π – ограничения пути. Это формула выражает условия относительно символов a_i, необходимые для достижения $stmt$. Т.е. ограничения пути это обязательные условия для достижения текущей ветви. Изначально $\pi = true$

```
void foobar(int a, int b) {
        int x = 1, y = 0;
        if (a != 0) {
                y = 3 + x;
                if (b == 0)
                        x = 2 * (a + b);
        }
        assert (x - y != 0);
}
```

Рис. 1: Пример простой программы. При каких входных значениях условие assert не выполнится?

Выражение вида $x = e$ означает обновление символьного хранилища путём связывания переменной x с новым символьным значением e.

Встретив выражение вида *if e then* s_{true} *else* s_{false}, символьная машина разветвляет своё исполнение, создавая два состояния выполнения. Созданные состояния отличаются ограничениями пути: $\pi_{true} = \pi \wedge e$ и $\pi_{false} = \pi \wedge \neg e$.

На рисунке 2 в виде дерева изображено символьное исполнение функции *foobar*.

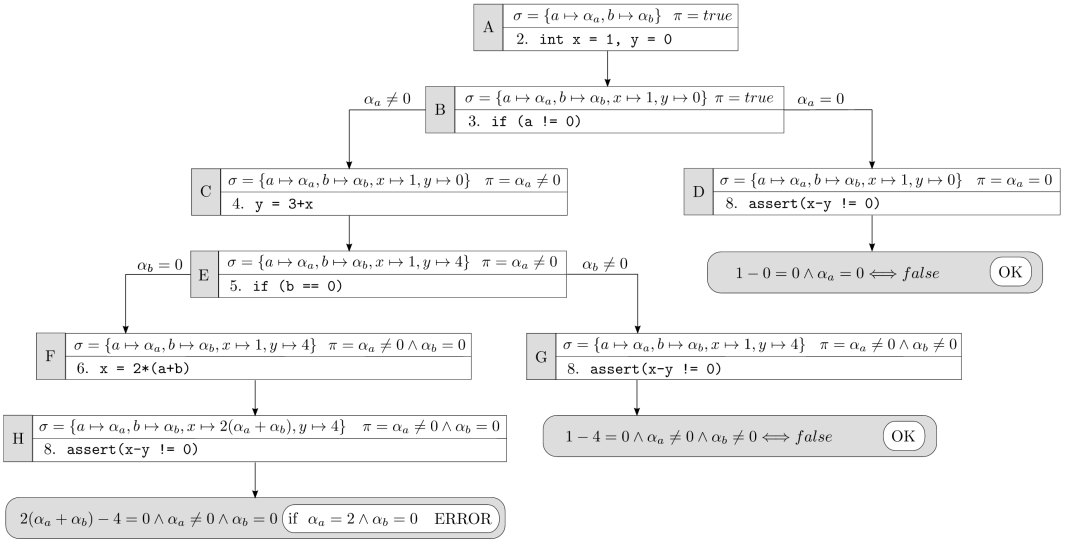

Рис. 2: Символьное дерево исполнения (взято из [2])

Изначально (состояние A) ограничения пути истинны, а входные аргументы a и b связаны с символьными значениями. После инициализации переменных x и y, в символьное хранилище добавляются их отображения на конкретные значения (состояние B). Затем символьное исполнение встречает условный оператор $if-else$ и делает два предположения относительно истинности предиката. Получаем состояния C и D. Затем выполнение продолжается в каждой из ветвей до тех пор, пока не достигается операция *assert*. Затем ограничения пути текущей ветви объединяются при помощи связки "И"с отрицанием предиката из функции *assert*. Полученная формула передаётся в SMT решатель, который пробует найти значения символьных переменных так, чтобы переданная формула оказалась истинной т.е. чтобы выполнялись ограничения пути и не выполнялось условие *assert*. Из примера видно, что рассматриваемая программа будет "сломана"при $a = 2$ и $b = 0$. Пример взят из [2].

В данном примере мы обошли все ветви символьного исполнения, но в настоящих программах полный анализ, к сожалению, невозможен. Несмотря на множество стратегий ветвления, символьное исполнение упирается в проблему комбинаторного взрыва. Поскольку количество ветвей в дереве исполнения велико и растёт кратно с каждым встречным условием. Даже на относительно небольших программах становится невозможным проанализировать дерево полностью. Следовательно, нужен способ отсечения менее перспективных ветвей исполнения т.е. необходимо выбрать *стратегию ветвления*.

3.2 Обратное и двунаправленное символьное исполнение

Как уже было сказано, обратное символьное исполнение начинает работать не с начала программы, а с некоторой заданной локации и движется в обратном направлении [4]. Данный подход можно использовать для генерации тестовых данных, увеличивающих покрытие кода тестами. Также полученные данные можно использовать для целевой отладки: разработчик помечает нужную строчку кода и нажимает кнопку "debug а отладчик сам подбирает данные, приводящие программу в обозначенное место. С помощью обратного исполнения можно искать специальные виды ошибок. Например, null pointer exception. Но, к сожалению, обратное символьное исполнение также подвержено проблеме комбинаторного взрыва путей. Возможным выходом из сложившейся ситуации и одним из перспективных направлений является двунаправленное символьное исполнение. Его суть заключается в поочерёдном исполнении прямого и обратного подходов и определении состояний, когда эти подходы пересекаются [5, 7].

3.3 Композициональное символьное исполнение и резюмирование кода

Очевидно, что если мы уже проанализировали тело цикла или функции, то переиспользование полученных результатов сильно бы ускорило весь анализ. *Композициональное символьное исполнение* – это подход, позволяющий получать новые символьные состояния путём композиции старых без символьного выполнения кода [1].

Как построить аппроксимацию исследуемой функции? В области верификации программного обеспечения существует подход аппроксимации сверху. Мы описываем все возможные состояния программы, но, возможно, включим и те, что на самом деле никогда не будут достигнуты. Очевидно, что для описания всех состояний нужна достаточно лаконичная и ёмкая форма. Предлагается для этого использовать формулы логики первого порядка. Рассмотрим пример на рисунке 3:

```
int abs(int x) {
        return x < 0 ? -x : x;
}
```

Рис. 3: Функция нахождения модуля целого числа

Мы можем аппроксимировать сверху эту функцию с помощью выражения:

$$x \neq INT\_MIN \Rightarrow abs(x) \geq 0$$

Можно заметить, что данное приближение включает и достаточно бессмысленные состояния. Например, $x = -6$ и $abs(x) = 20$. Но, тем не менее, мы можем доказать недостижимость таких состояний, как: $abs(x) = -24$. Подобное приближение позволяет не анализировать условия вида *if abs(x) < 0 then p else m*, что сильно сокращает пространство поиска.

Список литературы

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. pages 367–381, 03 2008.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[3] Franck Cassez, C. Müller, and K. Burnett. Summary-based interprocedural analysis via modular trace refinement. *Leibniz International Proceedings in Informatics, LIPIcs*, 29:545–556, 12 2014.

[4] Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. *ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 31–36, 09 2014.

[5] Kin-Keung Ma, Yit Khoo, Jeffrey Foster, and Michael Hicks. Directed symbolic execution. volume 6887, pages 95–111, 09 2011.

[6] The Hacker News. Extremely critical log4j vulnerability leaves much of the internet at risk. https://thehackernews.com/2021/12/extremely-critical-log4j-vulnerability.html.

[7] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach. 11 2017.

[8] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Interpolation-based function summaries in bounded model checking. pages 160–175, 12 2011.

[9] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing, 2002.

[10] Ars Technica. Patch fixing critical log4j 0-day has its own vulnerability that's under exploit. https://arstechnica.com/information-technology/2021/12/patch-fixing-critical-log4j-0-day-has-its-own-vulnerability-thats-under-exploit/.