ИСП РАН

Гибридный фаззинг фреймворка машинного обучения TensorFlow

Кобрин И.А., Вишняков А.В., Федотов А.Н.

29 июня 2022 г.

ИСП РАН

Актуальность

Системы искусственного интеллекта повсеместно внедряются в обыденную жизнь, в связи с чем повышается актуальность вопроса безопасности фреймворков машинного обучения, применяемых для проектирования таких систем.

Программист самостоятельно может тестировать ПО только на ограниченном наборе тестов, поэтому необходимо использовать методы автоматического тестирования.

Одними из самых распространенных технологий для автоматического тестирования остаются **фаззинг** и **символьная интерпретация** программ.

Цели работы

- Добавить новые фаззинг-цели TensorFlow
- Провести аудит фреймворка TensorFlow с помощью разработанного метода
- Наладить CI фаззинга TensorFlow

Символьная интерпретация

- **Символьная интерпретация** — метод автоматического тестирования программ, при котором происходит интерпретация программы, где конкретным значениям переменных, зависящих от входных данных, сопоставляются символьные переменные, принимающие произвольные значения.
- **Предикат пути** — система из уравнений и неравенств над символьными переменными и константными значениями, решение которой обеспечивает прохождение потока управления по исследуемому пути.
- **Предикат безопасности** — дополнительные условия на предикат пути, которые позволяют обнаружить ошибку в программе.

Предикаты безопасности

Предикат безопасности составляется и проверяется каждый раз, когда во время символьной интерпретации встречается опасное место в программе.

Пример построения предиката безопасности для ошибки разыменования нулевого указателя:

Символьное состояние	Инструкция	Множество формул	Предикат пути
$rax = \phi_1, rbx = \phi_2$	—	$S = \varnothing$	$\prod = true$
$rax = \phi_1, rbx = \phi_3$	`add rbx, 2`	$S = \{\phi_3 = \phi_2 + 2\}$	$\prod = true$
$rax = \phi_1, rbx = \phi_3$	`cmp rbx, 2048` `jge .out`	$S = \{\phi_3 = \phi_2 + 2\}$	$\prod = (\phi_3 < 2048)$
$rax = \phi_1, rbx = \phi_3$	`cmp rbx, 0` `jl .out`	$S = \{\phi_3 = \phi_2 + 2\}$	$\prod = (\phi_3 < 2048) \wedge (\phi_3 \geq 0)$
$rax = \phi_1, rbx = \phi_3$	`mov [rax + rbx], 1`	$S = \{\phi_3 = \phi_2 + 2, \phi_4 = \phi_1 + \phi_3\}$	$\prod = (\phi_3 < 2048) \wedge (\phi_3 \geq 0)$

Предикат безопасности

$$\prod_{security} = (\phi_3 < 2048) \wedge (\phi_3 \geq 0) \wedge (\phi_4 = 0)$$

Предикаты безопасности

- Деление на нуль
  - Проверяем символьный делитель на равенство нулю
- Целочисленное переполнение
  - Проверяем флаги `CF`/`OF` на равенство 1 при выполнении символьных арифметических инструкций
  - Обнаруживаем стоки ошибок — места, использование переполненных значений в которых приводит к более серьезным последствиям. Например, условные переходы, разыменование адреса, аргументы функций.
- Выход за границы массива
  - Проверяем символьный адрес при его разыменовании, не выходит ли он за границы массива
  - Для массива на куче узнаем границы из теневой кучи
  - Для массива на стеке вычисляем нижнюю границу эвристически, верхнюю узнаем из теневого стека

Фаззинг

- **Фаззинг** — метод тестирования ПО, при котором программе на вход подаются входные данные, после чего анализируется реакция программы и генерируются новые входные данные.
- **Цель фаззинга** — поиск наборов входных данных, приводящих выполнение программы к аварийному завершению, переполнению по памяти или зависанию.
- Одним из наиболее распространенных видов фаззинга остается **фаззинг с обратной связью по покрытию**.
- Объединение работы фаззинга и символьной интерпретации называется **гибридным фаззингом**.

Метод автоматизированного поиска ошибок

Разработанный метод состоит из следующих этапов:

- **Гибридный фаззинг** программы для расширения корпуса входных данных с помощью libFuzzer и Sydr
- **Минимизация** корпуса для получения меньшего количества входных данных, дающих то же покрытие
- Проверка **предикатов безопасности** на каждом файле из минимизированного корпуса для поиска ошибок целочисленного переполнения, выхода за границы массива, деления на нуль и других
- Верификация санитайзерами результатов проверки предикатов безопасности
- Анализ аварийных завершений с помощью Casr
- Ручной анализ найденных ошибок

TensorFlow в OSS-Fuzz

OSS-Fuzz – репозиторий от Google для фаззинга проектов с открытым исходным кодом.

В OSS-Fuzz были добавлены очень примитивные цели для фаззинга TensorFlow, покрывающие маленькие единичные функции.

В репозитории TensorFlow были найдены цели, покрывающие более сложный и интересный код, однако сборочные файлы для этих целей оказались нерабочими.

Фаззинг-цели TensorFlow

Найденные в ядре TensorFlow цели представляют собой код, в котором фактически создается один нейрон сети и вызываются функции, отвечающие, например, за декодирование файлов формата WAV, PNG, парсинг тензоров и другой функционал ядра TensorFlow.

Эти цели покрывают огромное количество кода как самого TensorFlow, так и его зависимостей, поэтому их сборка позволяет провести качественный аудит фреймворка.

Сборка фаззинг-целей

Для сборки найденных целей пришлось в истории изменений OSS-Fuzz искать какие-то намеки на то, как это сделать.

Найденные в ранних изменениях OSS-Fuzz сборочные файлы также не приводили к успешной сборке целей для фаззинга.

В итоге пришлось написать сборочные скрипты, совмещавшие в себе сборку целей из текущей версии OSS-Fuzz и сборку, найденную в старом коммите OSS-Fuzz.

Сборка фаззинг-целей

В изначальной версии TensorFlow цели собираются как динамическая библиотека, однако это приводит к ошибкам линковки, поэтому мы собираем фаззинг-цели как статически слинкованные исполняемые файлы.

Была налажена система сборки фаззинг-целей, цели были собраны в трех вариантах:

- С санитайзерами для libFuzzer;
- Без инструментации для инструмента символьной интерпретации Sydr;
- Для просмотра покрытия llvm-cov.

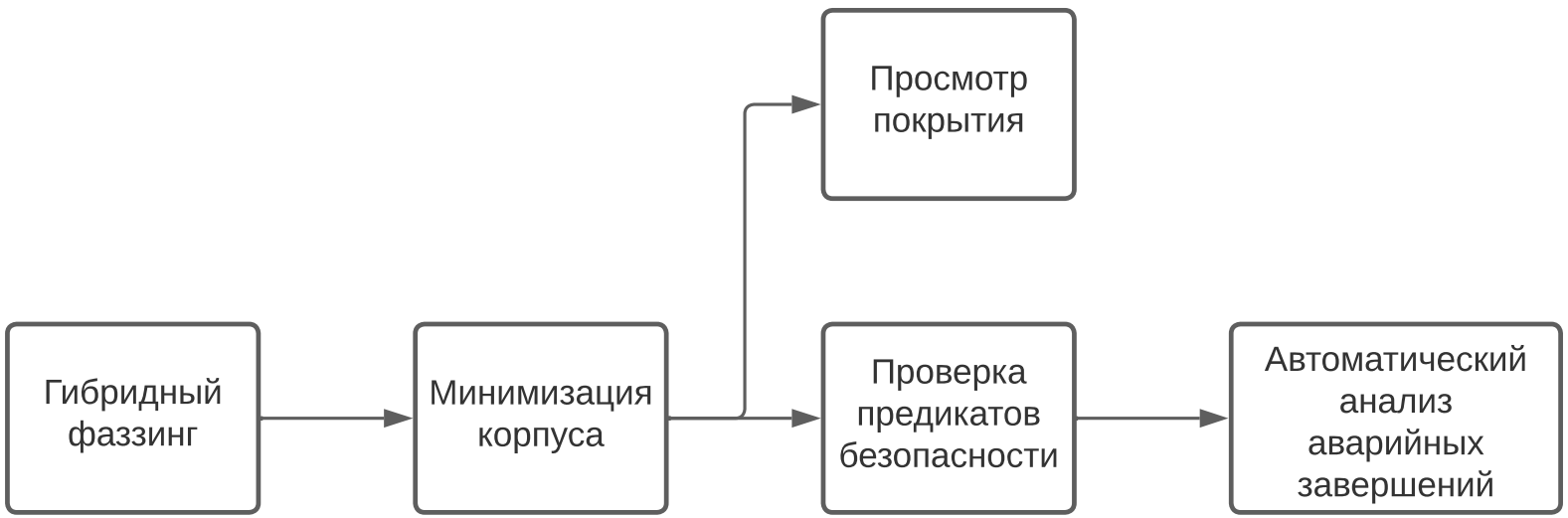
Окружение для фаззинга TensorFlow

Для фаззинга TensorFlow мы создали Docker-контейнер, в котором присутствуют:

- 3 версии каждой фаззинг-цели (для libFuzzer, для Sydr и для llvm-cov),
- Изначальные корпуса входных данных для некоторых фаззинг-целей,
- Словари для мутации входных данных для некоторых фаззинг-целей,
- Подготовленные конфигурационные файлы для гибридного фаззинга при помощи libFuzzer и Sydr для каждой фаззинг-цели.

CI фаззинга TensorFlow (OSS-Sydr-Fuzz)

В описанном окружении был налажен CI фаззинга TensorFlow. Все файлы для сборки фаззинг-целей и докер-контейнеров расположены здесь: github.com/ispras/oss-sydr-fuzz/tree/master/projects/tensorflow



Ошибка целочисленного переполнения в TensorFlow

При проверке предикатов безопасности на обертке `parse_tensor_op` была найдена ошибка целочисленного переполнения:

```
1  char* FastInt32ToBufferLeft(int32 i, char* buffer) {
2    uint32 u = i;
3    if (i < 0) {
4      *buffer++ = '-';
5      // i may be equal to -2^31,
6      // negatiation of which is equal to itself.
7      u = -i;
8    }
9    return FastUInt32ToBufferLeft(u, buffer);
10 }
```

Ручной анализ ошибки показал, что на практике она не может привести программу к некорректному поведению.

Бесконечный цикл в TensorFlow

На этапе гибридного фаззинга цели `decode_wav` был найден набор входных данных, приводящий к бесконечному циклу:

```
1   while (found_text != kFormatChunkId) {
2       if (found_text != "JUNK" && found_text != "bext" &&
3           found_text != "iXML" && found_text != "qlty" &&
4           found_text != "mext" && found_text != "levl" &&
5           found_text != "link" && found_text != "axml") {
6         return errors::InvalidArgument("Unexpected field ",
7                                        found_text);
8       }
9       // size_of_chunk may be equal to -8,
10      // which may lead to endless loop.
11      uint32 size_of_chunk;
12      TF_RETURN_IF_ERROR(ReadValue<uint32>(wav_string,
13                                           &size_of_chunk, &offset));
14      TF_RETURN_IF_ERROR(
15          IncrementOffset(offset, size_of_chunk,
16                          wav_string.size(), &offset));
17      TF_RETURN_IF_ERROR(ReadString(wav_string, 4,
18                                    &found_text, &offset));
19  }
```

github.com/tensorflow/tensorflow/pull/56455

Заключение

- В коде TensorFlow была найдена истинно положительная ошибка целочисленного переполнения, но она не приводит к серьезным последствиям;
- В коде TensorFlow была найдена ошибка бесконечного цикла;
- Вышеупомянутые ошибки были найдены автоматически в процессе CI фаззинга TensorFlow;
- Разработанная система сборки фаззинг-целей была принята в проект Google OSS-Fuzz: github.com/google/oss-fuzz/pull/7704