Поиск уязвимостей в программном коде при помощи методов машинного обучения на примере парадокса Лебланка.

Легерова Валерия Евгеньевна

Московский государственный университет имени М.В.Ломоносова, 2024 г.

1) ***Представление целых чисел в памяти компьютера***

Существует несколько способов представления целых чисел в памяти компьютера. Прямой код, обратный, а в настоящее время используется так называетмый дополнительный код.

~ Краткая справка:

- Прямой код (Sign and magnitude)

Это представление числа в двоичной системе счисления, при котором первый (старший разряд) отводится под знак числа (знак числа хранится
в знаковом бите). Если 0 - число положительно, 1 - отрицательно.

Особенности представления чисел с помощью прямого кода:

-Получить прямой код числа достаточно просто.
-Коды положительных чисел относительно беззнакового кодирования остаются неизменными.
-Количество положительных чисел равно количеству отрицательных.
-Существование двух нулей.

- **Обратный код** (Ones complement)

Отрицательные числа получаются путем инвертирования всех его битов. Отрицательный нуль никуда не делся, диапазон остался тот же, но теперь бит знака не требует дополнительной обработки и вычитание можно заменить сложением.

- Дополнительный код (Twos complement)

Используется в современных компьютерах. Чтобы получить отрицательное число для начала производится инвертирование, а затем к полученному значению прибавляется единица.

Каждый тип данных в языке С занимает определенное количество байтов. Конкретное число зависит от типа системы. Благодаря этому можно легко определить min/max значения, которые может принимать данный тип.

В таблице ниже приведены минимальные и максимальные значения для 32-битной платформы:

Typical Sizes and Ranges for Integer Types on 32-Bit Platforms

Type	Width (in Bits)	Minimum Value	Maximum Value
signed char	8	-128	127
unsigned char	8	0	255
short	16	-32,768	32,767
unsigned short	16	0	65,535
Int	32	-2,147,483,648	2,147,483,647
unsigned int	32	0	4,294,967,295
long	32	-2,147,483,648	2,147,483,647
unsigned long	32	0	4,294,967,295
long long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long	64	0	18,446,744,073,709,551,615

В представленной ниже таблице указаны основные 64-битные системы, использующиеся на сегодняшний день. Указано количество бит, выделенных конкретно под каждый тип данных.

64-Bit Integer Type Systems

Type	ILP32	ILP32LL	LP64	ILP64	LLP64
char	8	8	8	8	8
short	16	16	16	16	16
int	32	32	32	64	32
long	32	32	64	64	32
long long	N/A	64	64	64	64
pointer	32	32	64	64	64

Много проблем вытекает из того факта, что арифметические операции над целыми числами могут приводить к значению, не попадающему в диапазон представленного типа (numeric underflow).

Простой пример1

(-) unsigned int a;
(-) a=0;
(-) a=a-1;

Пример2

```
        1110 0000 0000 0000 0000 0000 0010 0000
+       0010 0000 0000 0000 0000 0000 0010 0000
=     1 0000 0000 0000 0000 0000 0000 0100 0000
```

Он показывает, что арифметические операции могут давать неожиданный неверный результат, связанный с усечением дополнительных битов, появившихся из -за переполнения.

2) ***Парадокс Лебланка***

Все эти нюансы с представлением чисел в компьютере привели к открытию малоизвестного среди программистов так называемого парадокса Лебланка.

~ Краткая справка:

Дэвид Леблан – ведущий исследователь проблем целочисленного представления.
Именно он обнаружил интересный парадокс по время работы с одним своим коллегой. Итак, приведем содержание парадокса на примере типа int.
Рассмотрим число (0x80000000)16 (или (-2147483648)10). Это число является нижней границей допустимых значений типа int. Попробуем получить из
него отрицательное, используя правила представления отрицательных чисел в дополнительном коде:

1000 0000 0000 0000 0000 0000 0000 0000 ---> 0111 1111 1111 1111 1111 1111 1111 1111 ---> 1000 0000 0000 0000 0000 0000 0000 0000

Число не изменилось!

Эта особенность может проявиться, когда разработчики, пользуясь отрицательными числами, пытаются получить их абсолютное значение.

Пример3

```
#include <stdio.h>
int hashbank(int index_e, int value_e);
int hashbank(int index, int value) {
        int bank[4096];
        if (index < 0) {
                index = -index;
        }
        if (index >= 4096) return -1;

        bank[index] = value;
        return 0;
}
int main() {
        int a,b;
        a = 0x80000000;
        b = 0x414243;
        printf("%d", hashbank(a,b));
}
```

Итог:

```
macpro:~ lera$ gcc -o leblanc_example leblanc_example.c -fsanitize=address
macpro:~ lera$ ./leblanc_example
80000000-2147483648,0
AddressSanitizer:DEADLYSIGNAL
=================================================================
==2661==ERROR: AddressSanitizer: SEGV on unknown address 0x7ffcea2e0880 (pc
 0x00010591bc34 bp 0x7ffeea2e4a40 sp 0x7ffeea2e0860 T0)
==2661==The signal is caused by a WRITE memory access.
    #0 0x10591bc33 in hashbank (leblanc_example:x86_64+0x100000c33)
    #1 0x10591bd14 in main (leblanc_example:x86_64+0x100000d14)
    #2 0x7fff598c43d4 in start (libdyld.dylib:x86_64+0x163d4)
```

Ошибка состоит в том, что происходит попытка записи значения по индексу большого по модулю отрицательного числа. Указатель как бы 'проскакивает' массив и пытается записать какое-то значение в невыделенную для этого область памяти.

Данного рода ошибки являются обычно весьма неожиданными для разработчиков, так как, умножив на -1, они думают, что получили таким образом модуль числа.

В представленной работе как раз и предлагается методика распознавания такого рода уязвимостей при помощи машинного обучения.

3) ***SEMGREP***

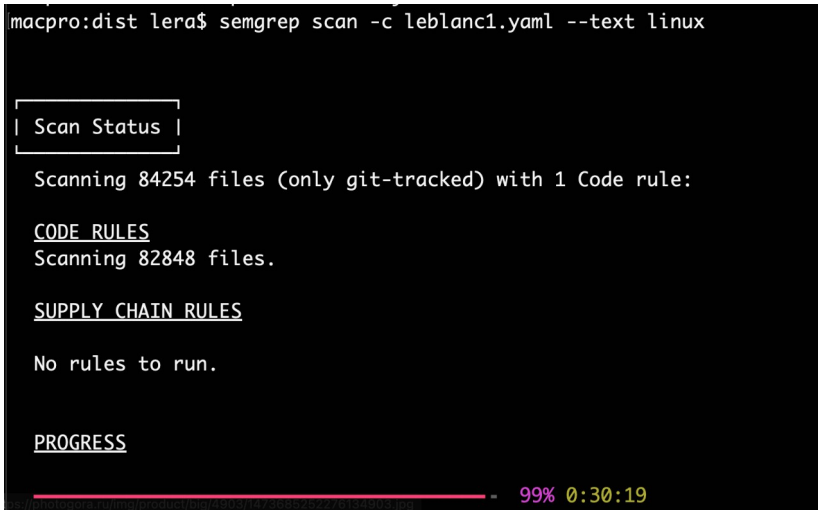
Semgrep — это инструмент статического анализа с открытым исходным кодом, который помогает сканировать исходный код и находить различные уязвимости с помощью предопределенных правил.
Semgrep сканирует заданные файлы и находит части кода заданного шаблона по написанным разработчиком правилам.

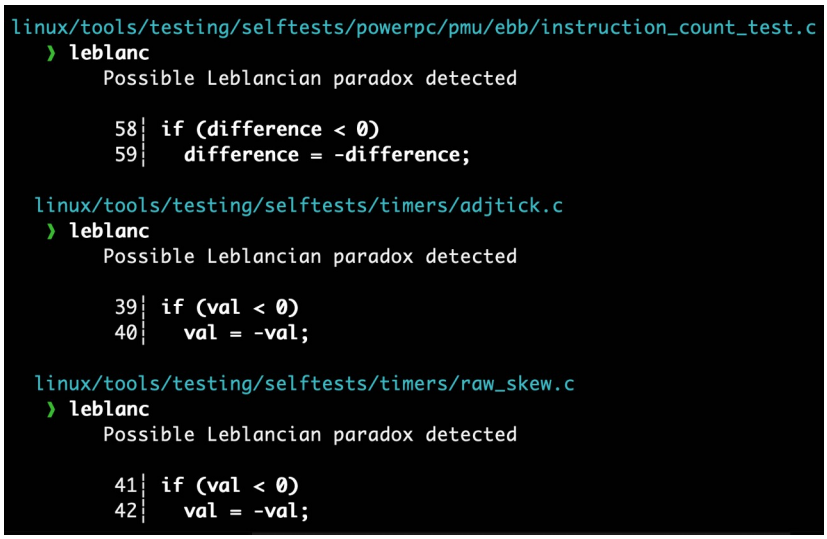
Схема написания правил достаточно проста. Вот как выглядело использованное мною:

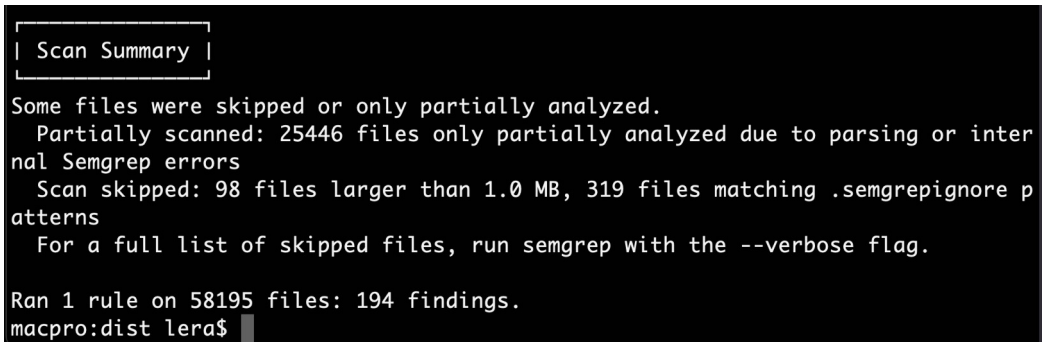
```
rules:
  - id: leblanc
    metadata:
      author: project Fitzel
      references:
        - https://github.com
      confidence: LOW
    message: >-
        Possible Leblancian paradox detected
    severity: INFO
    languages:
      - c
      - cpp
    pattern-either:
      - pattern: |
                 if ($X < 0) {
                    $X = -$X;
                 }
      - pattern: |
                 if (<... $X < 0 ...>) {
                     ...
                     $X = -$X;
                     ...
                 }
      - pattern: |
                 if (<... $X <= 0 ...>) {
                     ...
                     $X = -$X;
                     ...
                 }
      - pattern: |
                 if ($Z < 0) $Z = -$Z;
      - pattern: |
                 if ($Z <= 0) $Z = -$Z;
      - pattern: |
                 if (<... $Z < 0 ...>) $Z = -$Z;
      - pattern: |
                 $Y = $X < 0 ? -$X : $X;
      - pattern: |
                 $Y = $X <= 0 ? -$X : $X;

```

Запуск сканирования и сам процесс происводятся следующим образом:
(в данном примере производилось сканирование ядра Linux)

Пример4







Так выглядит вывод Semgrep-а.

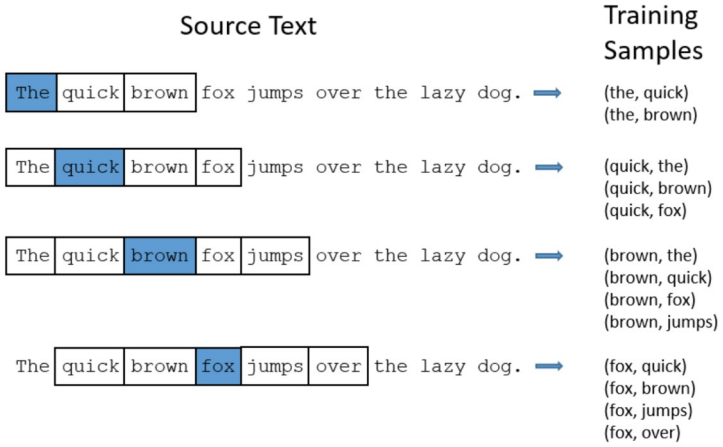
Стоит отметить, что из-за внутренних ошибок Semgrep сканирует не все поданные ему файлы (в основном примерно 50%). Так, здесь он отсканировал 25446/58195.

*4) **Word2Wec***

Для того, чтобы нейронные сети могли работать со словами, людям необходимо было научиться представлять слова в виде векторов. Одной из самых лучших разработок оказалась модель на основе нейронных сетей под названием word2wec. Программное обеспечение *под названием «word2vec» было разработано группой исследователей* Google *в 2013 году.*

Если говорить вкратце, то модель берет на вход текстовый корпус, а на выходе сопоставляет каждому отдельному слову вектор, основываясь на контекстной близости слов друг к другу.

Координатами в векторе на выходе будут вероятности, с которыми можно встретить каждное слово из нашего словаря в контексте с этим словом (чей вектор подавался на вход).



В word2wec реализованы 2 основных типа архитектур – *CboW* и *Skip-gram*.

CBoW — архитектура, которая предсказывает текущее слово, исходя из окружающего его контекста. Архитектура типа Skip-gram использует текущее слово, чтобы предугадывать окружающие его слова.

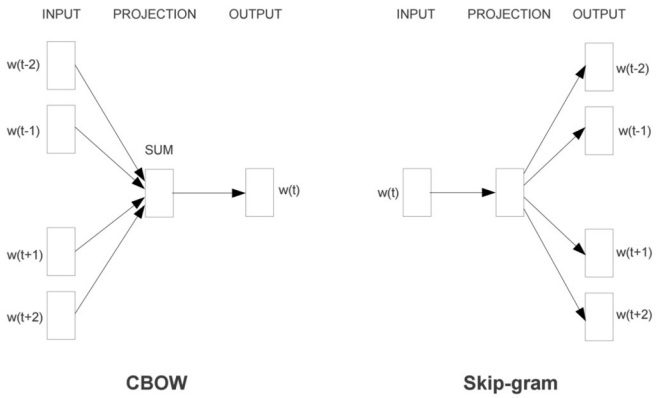
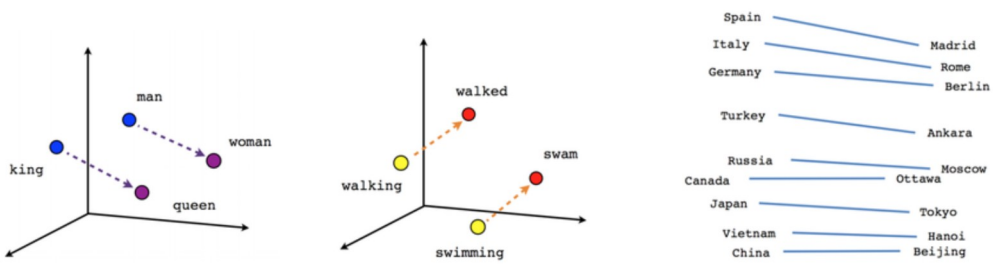


Иллюстрация к основным типам архитектур word2wec.

Так как вектора, построенные таким образом, частично отражают контекстный смысл слов, то с ними можно строить интересные линейные соотношения.

Популярный *пример5*

(http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/ источник)



3: Visualization of semantic relationships, e.g. male-female, verb tense and even country-capital relationships between words (Mikolov et al., 2013b).

Существует библиотека Gensim (для обработки естественного языка) , в которой алгоритмы word2wec-а реализованы на C. Она и была использована внастоящей работе.

https://radimrehurek.com/gensim/models/word2vec.html – ссылка на документацию

Также существуют улучшенные версии классического word2wec (см. Doc2Vec *и* FastText), однако в настоящей работе они не используются.

6) ***Сбор датасата***

Процесс постояения датасета выглядел следующим образом:

1) Запущен semgrep и отсканированы искодный код inux, android, firefox, chrome и 100 репозиториев из github на предмет наличия потенциальных уязвимостей.

2) Получен файл с результатами сканирования, для каждого срабатывания в которам указаны имя файла с уязвимостью и номер строки, начиная с которой находится интересующая нас конструкция (см. Правило для парадокса Лебланка).

3) Производится разборка файла, и для каждого файла, на который среагировал semgrep, находится номер строки N, начиная с которой можно найти парадокс:

Для создания подвыборок для обучения модели (train и validation) производилась обработка дата сета, которая включала в себя:

3.1) Копирование строк с N -5 до N + 5 в отдельный файл. Это будут данные для тренировочной выборки, содержащей уязвимость.

3.2) Копирование строк с N + 10 до N + 20 в файл. Это будут данные для тренировочной выборки, содержащей не уязвимость.

3.4) Копирование строк с N – 3 до N + 7 в файл. Это данные для валидационной выборки, содержащей уязвимость.

3.3) Копирование строк с N + 12 до N + 22 в файл. Это данные для валидационной выборки, не содержащей уязвимость.

Train dataset, vulnerable example:

```
WORD32 id = in_data[band];
if (id != 0) {
   id_sign = 0;
   if (id < 0) {
      id = -id;
      id_sign = 1;
   }
}
```

Validation dataset, vulnerable example:

```
if (id != 0) {
   id_sign = 0;
   if (id < 0) {
      id = -id;
      id_sign = 1;
   }
}
huff_bits += ixheaace_write_bits(pstr_bit_buf, p_huff_tab[id].value, p_huff_tab[id].length);
if (id != 0) {
   huff_bits += ixheaace_write_bits(pstr_bit_buf, id_sign, 1);
}
```

Validation dataset, not vulnerable example:

```
double sin_half_angle = std::sqrt(1.0 - cos_half_angle * cos_half_angle);
if (sin_half_angle < kEpsilon) {
    return *this;
}
double half_angle = std::acos(cos_half_angle);
double scaleA = std::sin((1 - t) * half_angle) / sin_half_angle;
```

TRain dataset, not vulnerable sample:

```
if (cos_half_angle > 1)
    cos_half_angle = 1;
double sin_half_angle = std::sqrt(1.0 - cos_half_angle * cos_half_angle);
if (sin_half_angle < kEpsilon) {
    return *this;
}
double half_angle = std::acos(cos_half_angle);
```

Такая обработка частично схожа с обработкой изображений в компютерной зрении (аугментация). Такая методика по сути своей является сдвигом окна, которое анализирует модель на наличие уязвимости. Это позволяет, во первых, значительно увеличить размер дата сета, а также приучить модель работать на чуть более вариабельных данных.

7) ***Реализация нейронной сети***

Моя модель содержала 2 линейных слоя, между которыми нелинейная функция активации LeakyReLU.

```
class MLP_Model(nn.Module):
    def __init__(self):
        super().__init__()

        self.fc1 = nn.Linear(30*100, 100)
        self.fc2 = nn.Linear(100, 2)
        self.relu = nn.LeakyReLU()
```

Линейный слой:

Выполняет линейное преобразование входных данных вида y=xAT+b.

dim A = out_features * in_features

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$.

LeakyReLU:

Модификация популярной функции активации ReLU. (Благодаря функциям активации нейронные сети способны преобразовывать данные *нелинейным образом*, что позволяет порождать более информативные признаковые описания).

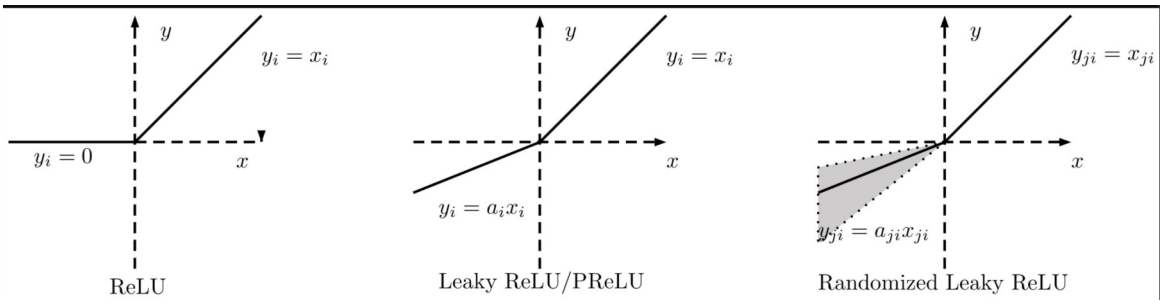
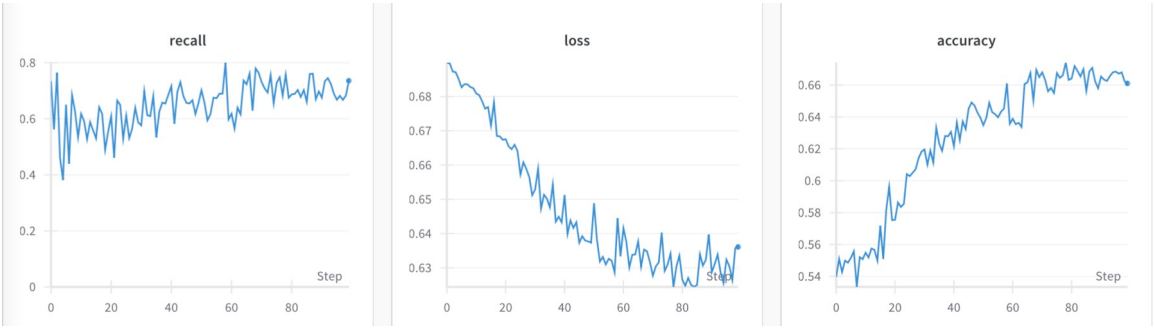


Иллюстрация отражает основное различие между классической ReLU и LeakyReLU.

После обучение модели на тестовой выборке ей был предложена для классификации валидационная выборка, постоенная по уже изложенному выше принципу.

По результатам ее работы на валидационной выборке были построены графики loss, recall и accuracy:



$$Accuracy = \frac{True_{positive} + True_{negative}}{True_{positive} + True_{negative} + False_{positive} + False_{negative}}$$

$$Recall = \frac{True_{positive}}{True_{positive} + False_{negative}}$$

Данные графики отражают результаты работы модели на тестовой выборке:
max recall: 0.78
max accuracy: 0.674

Как видно, соблюдается тенденция типичного поведения нармально обученной модели.

8) ***Выводы***

Данный парадокс мало известен среди разработчиков, поэтому может привести к достаточно трудно отлавливаемым уязвимостям в реальных программах.

Ниже приведены примеры потенциально отпасных мест в реальных, обработанных мною файлах на пример парадокса Лебланка.

Проект android/external/lua/

```
static int num2straux (char *buff, int sz, lua_Number x) {
   ...
    int e;
    lua_Number m = l_mathop(frexp)(x, &e);  /* 'x' fraction and exponent */
    int n = 0;  /* character count */
    if (m < 0) {  /* is number negative? */
      buff[n++] = '-';  /* add sign */
      m = -m;  /* make it positive */
    }
    ...
  ...
}
```

Видна попытка сделать переменную 'm' неотрицательной.

Проект - android/external/libaom/third_party/libyuv

```
// Copy a plane of data
LIBYUV_API
void CopyPlane(const uint8_t* src_y,
               int src_stride_y,
               uint8_t* dst_y,
               int dst_stride_y,
               int width,
               int height) {
  int y;
  void (*CopyRow)(const uint8_t* src, uint8_t* dst, int width) = CopyRow_C;
  // Negative height means invert the image.
  if (height < 0) {
    height = -height;
    dst_y = dst_y + (height - 1) * dst_stride_y;
    dst_stride_y = -dst_stride_y;
  }
  // Coalesce rows.
  if (src_stride_y == width && dst_stride_y == width) {
    width *= height;
    height = 1;
    src_stride_y = dst_stride_y = 0;
  }
  // Nothing to do.
  if (src_y == dst_y && src_stride_y == dst_stride_y) {
    return;
  }
  // Copy plane
  for (y = 0; y < height; ++y) {
    CopyRow(src_y, dst_y, width);
    src_y += src_stride_y;
    dst_y += dst_stride_y;
  }
}
```

Здесь если переменная 'height' будет отрицательной, то указатель 'dst_y' после операции:
"dst_y = dst_y + (height -1) * dst_stride_y" будет указывать в область слева от dst_y,
так как смещение отрицательное.
Далее в цикле 'for' произойдет запись в границы буфера dst_y.
Это беглый анализ, для более подробного анализа нужно время.

Ссылки на использованные источники:

[1] https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BF%D0%BE%D0%BB%D0%BD%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4

[2] https://www.amazon.com/Art-Software-Security-Assessment-Vulnerabilities/dp/0321444426

[3] https://semgrep.dev/

[4] https://source.android.com/docs/setup/reference/build-numbers#source-code-tags-and-builds

[5] https://github.com/trending/c++?since=monthly

[6] https://en.wikipedia.org/wiki/Word2vec

[7] http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

[8] *https://radimrehurek.com/gensim/models/word2vec.html*

[9] https://pytorch.org/docs/stable/nn.html