Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)
Факультет управления и прикладной математики
Кафедра информатики

Коновалов Андрей Дмитриевич

Автоматический поиск ошибок работы с динамической памятью в ядре ОС Linux

010900 — Прикладные математика и физика

Выпускная квалификационная работа бакалавра

Научный руководитель:
к.т.н. Исходжанов Тимур Равилевич

Долгопрудный
2014 г.

Содержание

1 Введение

При написании программ разработчики часто допускают различные ошибки. Такие ошибки могут приводить к неправильному или непредсказуемому поведению программ: замедлению скорости работы, аварийному завершению исполнения, порче данных и т.п.. Результатом подобного поведения могут быть большие экономические потери и даже гибель людей [1].

Даже при отсутствии ошибок в написанной разработчиком программе во время ее выполнении все равно могут произойти ошибки. Виной этому могут быть ошибки в каком-либо из компонентов операционной системы, в том числе в ее ядре. Ядро операционной системы Linux является одним из самых больших и сложных проектов с открытым исходным кодом на текущий момент. Поиск и исправление ошибок в ядре является актуальной задачей, поскольку в настоящее время система Linux лидирует на рынках смартфонов, интернет-серверов и суперкомпьютеров [2].

Ошибки могут быть разных типов: ошибки в логике работы программы, ошибки синхронизации потоков, ошибки работы с памятью и т.п.. Поиск ошибок вручную не всегда бывает эффективен. Например, ошибки могут наблюдаться очень редко или не проявляться какое-то время после того, как они произошли. Для решения этой проблемы процесс поиска ошибок можно автоматизировать. В данной работе рассматривается автоматический поиск ошибок работы с памятью в ядре Linux.

Для автоматического поиска ошибок в существующих программах, используются специальные приложения, которые называются *детекторами* (англ. *detectors*). Детекторы ошибок в основном разделяются на статические и динамические [3]. Статические детекторы анализируют исходный код программы, не осуществляя ее запуск. Иногда простые статические детекторы встроены прямо в компилятор, который выдает предупреждения во время компиляции программы. Напротив, динамические детекторы анализируют поведение программы по мере ее исполнения и обнаруживают ошибку, только если она в действительности произошла.

Недостатками статического анализа является большая вычислительная сложность и низкая точность. Часто поиск нетривиальных ошибок статическим анализом с достаточной точностью возможен только для отдельных изолированных модулей программы и требует особой организации ее исходного кода [4]. К недостаткам динамического анализа относится необходимость наличия тестов, возникающая вслед-

ствие того, что обнаружение ошибки происходит только во время исполнения кода, который ее содержит.

В рамках данной работы был выбран динамический анализ, поскольку существенное редактирование и реорганизация исходного кода ядра Linux (более 15 млн. строк кода на языке C) за разумное время представлялись невозможными. Однако поскольку при работе ядра исполняется лишь ограниченный набор драйверов (исполняются только драйвера, отвечающие за работу присоединенных к тестирующему компьютеру устройств), обнаружение ошибок с помощью динамических детекторов в драйверах является затруднительным.

В большинстве современных операционных систем ядро операционной системы и программы пользователя имеют разные привилегии, т.е. обладают разными правами на исполнение инструкций, чтение памяти и т.п.. В операционной системе Linux различают два режима выполнения пользовательского процесса: *режим ядра* (англ. *kernel mode*) и *режим пользователя* (англ. *user mode*). Процесс начинает выполнение в режиме пользователя. Когда процесс производит обращение к операционной системе, режим выполнения процесса переключается с режима пользователя на режим ядра: операционная система пытается обслужить запрос пользователя, возвращая код ошибки в случае неудачного завершения операции.

В операционной системе Linux виртуальная память разделяется на две области: *пространство ядра* (англ. *kernel space*) и *пространство пользователя* (англ. *user space*). Пространство ядра резервируется для работы ядра и его компонентов: расширений ядра и большей части драйверов устройств, и может быть использовано только процессом, находящимся в режиме ядра. Пространство пользователя является областью памяти, в которой функционируют все приложения пользователя и может быть использовано процессом как находящимся в режиме пользователя, так и в режиме ядра.

В дальнейшем детекторы, которые ищут ошибки в приложениях пользователя, будем называть *детекторами ошибок для приложений пользователя* или *детекторами ошибок в пространстве пользователя*, а детекторы, которые ищут ошибки в ядре и его компонентах, – *детекторами ошибок для ядра* или *детекторами ошибок в пространстве ядра*.

Сейчас существует несколько довольно эффективных и точных детекторов оши-

бок работы с памятью в пространстве пользователя [5,6], чего нельзя сказать о существующих детекторах ошибок в пространстве ядра [7]. Цель данной работы: разработка новых методов динамического тестирования ядра Linux, позволяющих достичь большей эффективности и точности при поиске ошибок работы с памятью.

2 Ошибки работы с памятью

2.1 Некоторые типы ошибок

Существуют множество типов ошибок работы с памятью, некоторые из них:

- *выход за границы массива* или *переполнение буфера* (англ. *buffer-overflow*);
- *использование памяти после освобождения* (англ. *use-after-free*);
- *использование неинициализированных данных* (англ. *use-of-uninitialized-value*);
- *утечка памяти* (англ. *memory-leak*);
- *повторное освобождение* (англ. *double-free*);
- *неправильное освобождение* (англ. *invalid-free*);
- *перекрытие аргументов* функций `memcpy()` и `strcpy()`.

Ниже для иллюстрации приведены искусственные примеры каждого из этих типов ошибок.

Ошибка типа «переполнение буфера» заключается в записи или чтении программой данных за пределами выделенной для этого области памяти. Пример переполнения буфера приведен в листинге 1. Как видно, в нем неправильно написано условие выхода из цикла, в результате чего происходит запись за пределы массива. В общем случае массив может быть выделен в динамической памяти, на стеке или быть глобальной переменной.

```
void foo() {
    int* a = new int[42];
    for (int i = 0; i <= 42; i++) {
        a[i] = i;
    }
    delete[] a;
}
```

Листинг 1: Пример ошибки типа «переполнение буфера»

Ошибка типа «использование памяти после освобождения» состоит в записи или чтении данных из освобожденного участка памяти. Пример использования памяти после освобождения приведен в листинге 2. В этом примере происходит обращение к элементу массива после его освобождения.

```
void foo() {
    int* a = new int[42];
    delete[] a;
    a[4] = 2;
}
```

Листинг 2: Пример ошибки типа «использование памяти после освобождения»

Ошибка типа «использование неинициализированных данных» заключается в чтении и дальнейшем использовании области памяти с неприсвоенным значением. Пример использования неинициализированных данных приведен в листинге 3. В этом примере используется значение одного из элементов массива до того, как ему было присвоено какое-либо значение.

```
void foo() {
    int* a = new int[42];
    printf("%d\n", a[4]);
    delete[] a;
}
```

Листинг 3: Пример ошибки типа «использование неинициализированных данных»

Ошибка типа «утечка памяти» состоит в выделении области памяти без последующего ее освобождения. Пример утечки памяти приведен в листинге 4. В этом примере в динамической памяти выделяется массив и происходит возврат из функции. Поскольку указатель на массив не сохраняется, то его освобождение за пределами приведенной функции произойти не может.

```
1 void foo() {
2     int* a = new int[42];
3 }
```

Листинг 4: Пример ошибки типа «утечка памяти»

Ошибка типа «повторное освобождение» заключается в освобождении ранее освобожденной области памяти. Пример повторного освобождения приведен в листинге 5.

```
1 void foo() {
2     int* a = new int[42];
3     delete[] a;
4     delete[] a;
5 }
```

Листинг 5: Пример ошибки типа «повторное освобождение»

Ошибка типа «неправильное освобождение» заключается в несоответствующем использовании функций `malloc()` и `free()` и операторов `new`, `delete`, `new[]` и `delete[]`. Пример неправильного освобождения приведен в листинге 6. В этом примере массив, выделенный с помощью оператора `new[]`, освобождается с помощью оператора `delete` вместо `delete[]`.

```
1 void foo() {
2     int* a = new int[42];
3     delete a;
4 }
```

Листинг 6: Пример ошибки типа «неправильное освобождение»

Ошибка типа «перекрытие аргументов» функций `memcpy()` и `strcpy()` состоит в пересечении областей памяти, соответствующих назначению и источнику копирования. Согласно стандарту C/C++ эти функции могут использоваться только для неперекрывающихся областей памяти. Для перекрывающихся нужно использовать функцию `memmove()`. Пример этой ошибки приведены в листинге 7.

```
void foo() {
    char str[] = "aa11bb22";
    memcpy(str + 2, str, 6);
    // str != "aaaa11bb", str == "aaaa1111"
}
```

Листинг 7: Пример ошибки типа «перекрытие аргументов» функции `memcpy()`

Все описанные выше ошибки могут происходить как в приложениях пользователя, так и в ядре и его компонентах. Кроме них существуют ошибки специфичные только для ядра. Одной из таких ошибок является «использование памяти пользователя», которая заключается в прямом обращении ядра к адресному пространству пользователя в обход специально предназначенных для этого механизмов.

2.2 Неопределенное поведение

Многие ошибки работы с памятью приводят к *неопределенному поведению* (англ. *undefined behaviour*) программы. Это состояние возникает при нарушении программистом стандарта языка или спецификаций используемых программных функций или библиотек [4]. В этом случае компилятор или библиотека вправе выполнять некорректные и неожиданные действия. Особенностью таких ошибок является то, что обычно они не имеют наблюдаемых последствий, но иногда могут приводить к серьезным сбоям. При этом условия возникновения этих ошибок могут быть трудновоспроизводимыми – например, зависеть от версии компилятора, библиотеки или операционной системы; частоты процессора, количества ядер или даже его температуры. Более того, многие такие ошибки приводят не к мгновенным последствиям, а к порче данных, эффект от которой сможет наблюдаться лишь через некоторое время, затрудняя понимание и обнаружение ошибки.

Для демонстрации последствий неопределенного поведения рассмотрим пример программы [4], приведенный на листинге 8. В этой программе на последней, 64-й, итерации цикла происходит переполнение целочисленной знаковой переменной `value`. По стандарту C/C++ результат переполнения знаковых целочисленных переменных неопределен. Это очень распространенный тип ошибок, который нередко приводит к серьезным уязвимостям программного обеспечения [8].

```
#include <stdio.h>

void foo(int* array) {
  int value = 0x03020100;
  for (int i = 0; i < 64; i++) {
    printf("%d ", i);
    array[i] = value;
    value += 0x04040404;
  }
  printf("\n");
}

int main() {
  int values[64];
  foo(values);
  return 0;
}
```

Листинг 8: Пример программы с переполнением целочисленной переменной

Ожидаемый результат работы функции `foo` при запуске такого кода – печать значений от 0 до 63, а также заполнение аргумента-массива значениями. Этот результат действительно наблюдается при использовании компилятора gcc [9] версии 4.8.1 с настройками по умолчанию, как показано на листинге 9.

```
$ g++ test.cpp && ./a.out
0 1 2 ... 61 62 63
```

Листинг 9: Запуск программы с переполнением целочисленной переменной

Однако при включении компиляторных оптимизаций на 64-битной архитектуре результат получается другой. В данном случае печать значений продолжается и после 63 до тех пор, пока не произойдет ошибка сегментации, как показано на листинге 10.

```
$ g++ -O2 test.cpp && ./a.out
0 1 2 ... 61 62 63 64 65 66 ... 2062 2063 2064
Segmentation fault
```

Листинг 10: Запуск программы с переполнением целочисленной переменной, скомпилированной со включением компиляторных оптимизаций

Такое поведение программы связано с наличием в ней неопределенного поведения, поскольку в этом случае компилятор вправе породить произвольный код после

возникновения ошибки. При компиляции примера из листинга 8 получается ассемблерный код, приведенный в листинге 11. Видно, что в полученном коде нет условных переходов, которые могли бы прекратить исполнение цикла; также нет и инструкций возврата. Когда о таком поведении было сообщено разработчикам gcc, они отказались что-либо менять, сославшись на неопределенное поведение [10].

```
1  .LC0:
2    .string "%d␣"
3  foo:
4    push   %r12
5    mov    %rdi, %r12
6    push   %rbp
7    mov    $0x3020100, %ebp
8    push   %rbx
9    xor    %ebx, %ebx
10   xchg   %ax, %ax
11 .L2:
12   mov    %ebx, %edx
13   mov    .LC0, %esi
14   mov    $0x1, %edi
15   xor    %eax, %eax
16   callq  __printf_chks
17   mov    %ebp, (%r12,%rbx,4)
18   add    $0x4040404, %ebp
19   add    $0x1, %rbx
20   jmp    .L2
```

Листинг 11: Ассемблерный код, получаемый при компиляции функции с переполнением целочисленной переменной

Таким образом, следует отметить, что неопределенное поведение может приводить не только к некорректному результату ошибочной операции, но и к произвольным результатам любых дальнейших операций. Понятно, что если одна и та же ошибка по-разному проявляется при использовании одного и того же компилятора с разными настройками, то подобные ошибки могут становиться наблюдаемыми при переходе на новую версию компилятора, процессора и вообще программного и аппаратного обеспечения. Другими словами, при наличии в программе ошибки, приводящей к неопределенному поведению, даже если сегодня программа работает корректно, то никто не может гарантировать, что программа продолжит работать корректно завтра.

3 Обзор динамических детекторов

3.1 Применение детекторов

Детектирование ошибки с помощью динамического анализатора возможно только во время исполнения участка кода, в котором эта ошибка допущена. Вследствие этого для эффективного применения динамического тестирования необходимо иметь возможность исполнять все возможные участки кода, относящиеся к программе.

Одним из возможных способов исполнения различных участков кода является использование *автоматического модульного тестирования* (англ. *unit testing*). Модульные тесты – это небольшие программы или специальные функции программы-теста, которые обычно выполняют небольшое количество простых действий с программной компонентой, после чего сравнивают полученный результат с ожидаемым [4]. Модульные тесты хороши тем, что их можно многократно перезапустить в случае возникновения недетерминированных ложных срабатываний или пропусков ошибок. К сожалению, такой подход наследует от модульного тестирования основной недостаток – невозможность нахождения ошибок в коде, не исполняемом тестами.

Другим способом является использование *рандомизированного тестирования* (англ. *fuzz testing*). Рандомизированное тестирование заключается в автоматической генерации случайных данных, передаче их на вход программе и дальнейшее отслеживание появления исключительных ситуаций, например аварийного завершения [11]. Особенностью такого подхода является возможность находить ошибки, происходящие при определенных сложных условиях, которые не удается придумать программистам и тестировщикам.

При тестировании программ сами детекторы могут совершать ошибки. В соответствии с классификацией ошибок [4], *ошибкой первого рода* или *ложным срабатыванием* (англ. *false positive*) называется такая ситуация, когда в результате работы детектора пользователь получает отчет об ошибке, которой на самом деле в программе нет. *Ошибкой второго рода* или *пропуском ошибки* (англ. *false negative*) называется такая ситуация, когда в результате работы детектора пользователь не получает отчета об ошибке, которая на самом деле в программе есть. Следует отметить, что под ошибками первого и второго рода подразумеваются особенности поведения конкретного алгоритма детектора, а не ошибки в исследуемых программах.

Ложные срабатывания затрудняют применение детекторов, поскольку на анализ каждого отчета о найденной ошибке человек может тратить значительное количество времени. Пропуски ошибок, в свою очередь, снижают пользу от использования детекторов, так как означают, что некоторые ошибки так и не будут найдены.

3.2 Внутреннее устройство детекторов

В основном, динамические детекторы ошибок работы с памятью работают следующим образом. Для каждой области памяти детектор хранит информацию о том, доступна эта область для записи и чтения или нет, и проверяет ее доступность при каждом обращении. *Доступной* областью памяти называется область, выделенная приложением в динамической памяти, на стеке или являющаяся глобальной переменной. В случае недоступности этой области детектор выводит отчет, сообщающий об обнаруженной ошибке и содержащий информацию, полезную для локализации и исправления этой ошибки.

3.2.1 Теневая память

Многие детекторы для каждой ячейки памяти приложения хранят связанные с ней дополнительные данные. Такие данные описывают состояние каждой ячейки памяти, например, доступна ли она в текущий момент времени. Память, хранящая эти дополнительные данные, называется *теневой памятью* (англ. *shadow memory*) или *тенью* [4], а доступность ячейки памяти для чтения и записи называется *адресуемостью* (англ. *addressability*) этой ячейки.

Для проверки адресуемости при каждом обращении к ячейке памяти приложения детектор должен вычислять адрес соответствующей ему ячейки теневой памяти. Чем проще процедура вычисления этого адреса, тем выше производительность детектора. Другими словами для эффективной работы детектора требуется, чтобы можно было несложной процедурой получать адрес теневой памяти, зная адрес анализируемой ячейки памяти.

Одним из самых простых вариантов хранения теневой памяти является использование непрерывного участка адресного пространства, на который все адресное пространство отображается с помощью сжатия и сдвига. Такой вариант устройства теневой памяти используется в детекторах AddressSanitizer [5] и MemorySanitizer [12].

Другие детекторы, например Memcheck [13], используют более сложную многоуровневую систему организации теневой памяти.

3.2.2 Инструментация

При каждом обращении приложения к памяти детекторы производят проверку адресуемости этой области памяти. Обычно, для этого перед каждым таким обращением добавляется дополнительный проверочный код. Этот код проверяет доступность или недоступность памяти по данному адресу и сообщает об ошибке в соответствующем случае. Процесс внедрения дополнительного кода в программу без изменения ее основной функциональности называется *инструментированием* или *инструментацией* (англ. *instrumentation*) [4].

Инструментация выполняется либо во время компиляции, либо перед исполнением программы, либо во время ее исполнения, и называется *динамической*, *статической* и *компиляторной* инструментацией, соответственно [4]. Для приложений пользователя известными примерами систем, использующих динамическую инструментацию, являются Valgrind [14], Pin [15] и DynamoRIO [16], популярными примерами использования компиляторной инструментации являются утилита исследования покрытия кода тестами gcov [17] и детектор ошибок mudflap [18], а в качестве примера использования статической инструментации можно привести PEBIL [19]. Для ядра Linux примерами использования динамической инструментации являются PinOS [20], Kprobes [21] и KernInst [22], а в качестве примера использования компиляторной инструментации можно привести ftrace [23].

3.2.3 Библиотека времени исполнения

Для того чтобы хранить актуальное состояние памяти приложения в теневой памяти, необходимо отслеживать вызовы функций, выделяющих и освобождающих память приложения. Кроме того, при запуске приложения теневую память надо выделить и подготовить к использованию. Зачастую для этих целей одной только инструментации недостаточно. В программу необходимо также встроить *библиотеку времени исполнения* (англ. *runtime library*). Она может содержать новые функции и заменять реализации существующих.

Новые функции встраиваются в программу для того, чтобы их можно было вы-

зывать из внедренного инструментацией кода. Примером такой функции является функция печати отчета о найденной ошибке. Замена существующих функций бывает полезна, когда проще предоставить новую реализацию, чем заниматься инструментацией. Например, детекторам ошибок часто требуется заменять стандартные реализации функций работы с динамической памятью, чтобы модифицировать их поведение.

Библиотека времени исполнения может внедряться в программу на стадии компиляции (при использовании компиляторной инструментации), на стадии компоновки (например при использовании статической инструментации) и при запуске (при использовании динамической инструментации) [4].

3.3 Существующие детекторы в пространстве пользователя

3.3.1 Memcheck

Один из самых распространенных детекторов ошибок работы с памятью для приложений пользователя – Memcheck [13], основанный на системе инструментирования Valgrind [14]. Memcheck умеет находить следующие типы ошибок (см. раздел 2.1):

- переполнение буфера в динамической памяти;
- использование памяти после освобождения;
- использование неинициализированных данных;
- повторное освобождение;
- неправильное освобождение;
- утечка памяти;
- перекрытие аргументов `memcpy()` и `strcpy()`.

К недостаткам Memcheck можно отнести высокие накладные расходы: увеличение времени работы программ в 20-30 раз [6] и неумение находить ошибки типа «переполнение буфера» в стеке и при использовании глобальных переменных. Кроме того, Memcheck исполняет потоки в многопоточных программах поочередно (т.е. не одновременно), что оказывает существенное влияние на производительность при использовании современных многоядерных систем [4].

3.3.2 AddressSanitizer

AddressSanitizer [5] – новый детектор ошибок работы с памятью для приложений пользователя, основанный на компиляторной инструментации, реализованной в виде проходов Clang [24] и GCC [9]. AddressSanitizer умеет находить следующие типы ошибок (см. раздел 2.1):

- переполнение буфера в динамической памяти, на стеке и при использовании глобальной переменной;
- использование памяти после освобождения;
- повторное освобождение;
- неправильное освобождение;
- утечка памяти;
- перекрытие аргументов `memcpy()` и `strcpy()`.

Главным преимуществом AddressSanitizer по сравнению с Memcheck является высокая производительность: увеличение времени работы программ в среднем в 2 раза [5]. Кроме того, благодаря использованию компиляторной инструментации, AddressSanitizer умеет находить ошибки типа «переполнение буфера» на стеке и при использовании глобальных переменных, однако искать ошибки типа «использование неинициализированных данных» он не может. При этом при использовании AddressSanitizer потоки в многопоточных программах исполняются параллельно, что иногда позволяет на современном аппаратном обеспечении выиграть еще один порядок скорости у Memcheck [4]. Устройство AddressSanitizer подробно рассмотрено в разделе 4.1.

Сравнение детекторов Memcheck и AddressSanitizer приведено на таблице 1.

3.4 Существующие детекторы в пространстве ядра

3.4.1 Проверки в распределителе памяти

В распределитель памяти ядра Linux встроены специальные проверки корректности обращения с выделенной памятью. Эти проверки могут быть активированы с

Таблица 1: Сравнение детекторов Memcheck и AddressSanitizer (ASan)

	Memcheck	ASan
Переполнение буфера в динамической памяти	Да	Да
Переполнение буфера на стеке	Нет	Да
Переполнение глобального буфера	Нет	Да
Использование памяти после освобождения	Да	Да
Использование неинициализированных данных	Да	Нет
Утечка памяти	Да	Да
Повторное освобождение	Да	Да
Неправильное освобождение	Да	Да
Перекрытие аргументов `memcpy()` и `strcpy()`	Да	Да
Замедление	20x	2x

помощью включения специальных опций конфигурирования во время компиляции ядра. Несмотря на небольшие возможности, которые они предоставляют, их тоже можно использовать для поиска ошибок работы с памятью.

Опция CONFIG_DEBUG_SLAB позволяет обнаруживать некоторые ошибки типов «переполнение буфера» и «использование неинициализированных данных». Однако для обнаружения переполнения буфера некорректное обращение к памяти должно быть записью. Кроме того обнаружение ошибки происходит не сразу после того, как она произошла, а лишь спустя некоторое время.

Другая опция CONFIG_DEBUG_PAGEALLOC позволяет обнаружить некоторые ошибки типа «использования памяти после освобождения». Однако при использовании этой опции такие ошибки могут быть найдены довольно редко: при условии, что соответствующая страница памяти была выгружена из адресного пространства. Кроме того при включении этой опции наблюдается сильное уменьшение производительности.

3.4.2 kmemcheck

Детектор kmemcheck [7] является более качественным детектором ошибок работы с памятью в ядре Linux. По набору типов обнаруживаемых ошибок kmemcheck довольно похож на Memcheck (см. раздел 3.3.1). Однако реализация его совершен-

но другая, поэтому kmemcheck не так точен как Memcheck, но все равно довольно неплохо показывает себя на практике.

Детектор kmemcheck умеет обнаруживать ошибки типов «использование памяти после освобождения» и «использование неинициализированной памяти». Основным недостатком этого детектора является очень сильное уменьшение производительности ядра: примерно в 10 раз [25]. Кроме того при его использовании происходит увеличение использования динамической памяти ядром примерно в 2 раза.

4 Детектор KernelAddressSanitizer

Существующие детекторы ошибок работы с памятью в ядре обладают низкой точностью, производительностью и неудобны для использования. Поскольку детектор AddressSanitizer очень хорошо показал себя при тестировании приложений пользователя, идеи, лежащие в его основе, были использованы для разработки детектора для ядра и его компонентов KernelAddressSanitizer.

4.1 Устройство детектора AddressSanitizer

Рассмотрим внутреннее устройство AddressSanitizer более подробно.

4.1.1 Теневая память

AddressSanitizer использует теневую память для хранения информации о памяти приложения следующим образом. Память приложения разбивается на выровненные участки по 8 байт, каждый из которых может находиться в одном из 9 состояний. В каждом из этих состояний первые k ($0 \leq k \leq 8$) байт этого участка адресуемы, а оставшиеся $8 - k$ байт – нет. Таким образом, состояние каждых 8 байт памяти приложения можно закодировать одним байтом теневой памяти.

Кодирование байта теневой памяти осуществляется следующим образом. Если все 8 байт памяти приложения адресуемы, то значение соответствующего байта теневой памяти нулевое. Если адресуемы только первые k ($1 \leq k \leq 7$) байт, то эти 8 байт кодируются теневым байтом со значением k. Любое отрицательное значение байта теневой памяти означает, что все 8 байт памяти приложения неадресуемы. С помощью различных отрицательных значений можно различать разные типы неадресуе-

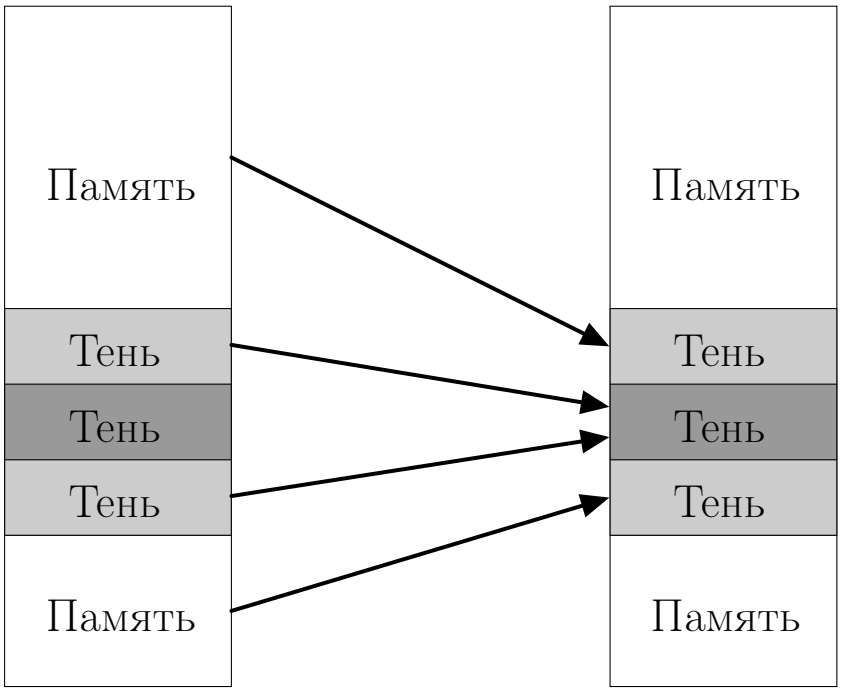

Рис. 1: Отображение адресного пространства в тень в детекторе AddressSanitizer. Та часть тени, которая имеет своим прообразом тень, отмечена более темным цветом.

мой памяти (освобожденная память; зоны безопасности (см. ниже) в динамической памяти, на стеке или вокруг глобальных переменных).

AddressSanitizer использует одну восьмую часть адресного пространства для хранения теневой памяти. Вычисление соответствующего адреса байта теневой памяти `ShadowAddr` по адресу памяти приложения `Addr` происходит как

```
ShadowAddr = (Addr >> 3) + Offset,
```

где `Offset` – это адрес начала участка теневой памяти в адресном пространстве приложения. Значение `Offset` выбирается фиксированным. Отображение адресного пространства в теневую память изображено на рис. 1.

4.1.2 Инструментация

Перед каждым обращением к участку памяти приложения детектор проверяет его адресуемость, обращаясь к соответствующей области теневой памяти. Проверка теневой памяти добавляется с помощью компиляторной инструментации перед непосредственным обращением к памяти приложения. Код, добавляемый при инструмен-

тация обращения размером `AccessSize` (1, 2, 4 или 8 байтов) по адресу `Addr`, показан на листинге 12. При обработке 16-байтных обращений нужно проверять сразу два байта теневой памяти, и добавляемый при инструментации код немного отличается от приведенного.

```
1 ShadowAddr = (Addr >> 3) + Offset;
2 k = *ShadowAddr;
3 if (k != 0 && ((Addr & 7) + AccessSize > k))
4     ReportAndAbort(Addr);
```

Листинг 12: Инструментация обращения к памяти приложения

Такой способ проверки адресуемости не имеет ошибок первого рода (т.е. ложных срабатываний). Однако могут быть пропущены редкие ошибки, когда происходит невыровненный доступ, частично переходящий через границу выровненного участка в 8 байт [5].

4.1.3 Библиотека времени исполнения

Главная цель библиотеки времени исполнения – это управление теневой памятью. Во время запуска приложения, область, соответствующая теневой памяти, резервируется, чтобы никакая другая часть приложения не могла ей воспользоваться. Кроме того, библиотека времени исполнения заменяет реализацию функций `malloc()` и `free()`.

Модифицированная функция `malloc()` выделяет дополнительную память вокруг возвращаемого участка памяти. Эти дополнительные области памяти называются *зонами безопасности* (англ. *redzones*). Зоны безопасности помечаются как неадресуемые. Чем больше размер зоны безопасности, тем большего размера выход за границы доступной области памяти детектор может обнаружить.

Модифицированная функция `free()` вместо того, чтобы сразу освобождать блоки памяти, переводит их в особое состояние – *карантина* (англ. *quarantine*). Память, помещенная в карантин, помечается недоступной для дальнейших доступов. Если бы блоки памяти освобождались сразу, то они могли бы быть выделены снова в другом модуле программы и помечены как доступные. В этом случае ошибки типа «использования неинициализированной памяти» были бы не обнаружены. Ясно, что удерживать блоки памяти в карантине вечно нельзя, поскольку иначе доступная

память просто закончится. Соответственно, после накопления определенного количества блоков в карантине, они начинают оттуда освобождаться.

Также функции `malloc()` и `free()` запоминают стек вызовов и сохраняют его в память, соответствующую зонам безопасности. Это позволяет показывать более информативные отчеты об ошибках.

4.2 Устройство детектора KernelAddressSanitizer

Устройство детектора KernelAddressSanitizer основано на устройстве детектора AddressSanitizer. KernelAddressSanitizer также использует теневую память для хранения информации о состоянии памяти ядра. Компиляторная инструментация используется для добавления проверок теневой памяти перед каждым обращением ядра к его памяти. Модуль времени исполнения резервирует теневую память и изменяет реализации некоторых функций распределителя памяти.

Одним из существенных отличий детектора KernelAddressSanitizer от детектора AddressSanitizer является то, что он ищет ошибки работы с физической памятью, а не с виртуальной (см. ниже).

4.2.1 Управление памятью в ядре Linux

Прежде чем говорить об устройстве детектора KernelAddressSanitizer, необходимо рассказать о внутреннем устройстве управления памятью в ядре операционной системы Linux. Далее будет рассмотрено устройство распределителя памяти SLAB для архитектуры x86-64.

Как говорилось ранее, в операционной системе Linux виртуальное адресное пространство разбито на две области: пространство ядра и пространство пользователя. В архитектуре x86-64 пространство ядра устроено таким образом, что его область с адресами из отрезка [`0xffff880000000000`, `0xffffc7fffffffff`] отведена для отображения на динамическую (оперативную) память [26]. В зависимости от размера доступной динамической памяти, начальный участок этой области памяти будет на нее однозначно отображен. Это означает, что чтение или запись значений в ячейки памяти, принадлежащие этому участку, будет чтением или записью в соответствующую ячейку динамической памяти. В дальнейшем этот участок виртуального адресного пространства ядра будем называть физической памятью (см. рис. 2).

Для выделения памяти в ядре Linux существует функция `kmalloc()` аналогичная функции `malloc()` в приложениях пользователя. Единственное существенное отличие состоит в том, что функция `kmalloc()` выделяет участок физической памяти, а не виртуальной. Функция `kmalloc()` для выделения памяти пользуется более низкоуровневыми механизмами распределителя памяти. В ядре Linux существует несколько различных вариантов реализации распределителя памяти. Самые популярные из них – это SLAB, SLUB и SLOB. В качестве распределителя памяти по умолчанию в ядре Linux используется SLAB.

Устройство распределителя памяти SLAB базируется на трех понятиях: *объект* (англ. *object*), *блок* (англ. *slab*) и *кэш* (англ. *cache*). Объект – это выделенный функцией `kmalloc()` участок физической памяти. Блок – это участок физической памяти размером в несколько страниц памяти, в котором один за другим хранятся несколько объектов одинакового размера. Блоки выделяются с помощью более низкоуровневых механизмов для работы с памятью. При выделении нового блока все объекты в нем помечаются как *незанятые*. Кэш – это список из нескольких блоков, в каждом из которых хранятся объекты одного и того же размера. *Размером кэша* называется размер объектов, которые хранятся в его блоках.

Распределитель памяти SLAB заранее создает несколько кэшей различного размера. При вызове функции `kmalloc()` сначала выбирается кэш соответствующего значению аргумента этой функции размера. Затем ищется блок, принадлежащий выбранному кэшу, в котором есть еще не занятые объекты. Если такой блок находится, то соответствующий объект помечается как *занятый*, и в качестве результата функции возвращается его адрес. Если все объекты во всех блоках оказываются занятыми, то выделяется новый блок и используется один из его объектов.

Заметим, что при такой реализации функции `kmalloc()` необходимо иметь кэши для каждого из возможных значений ее аргумента. Такая реализация не практична, поэтому в распределителе памяти кэши имеют размеры степеней двойки. Во время выполнения функция `kmalloc()` просто округляет значение аргумента до ближайшей степени двойки вверх.

Рис. 2: Теневая память в детекторе KernelAddressSanitizer

4.2.2 Теневая память

Во время загрузки ядра, часть его физической памяти резервируется в качестве теневой. Размер области теневой памяти `ShadowSize` зависит от размера доступной физической памяти `PhysSize` и вычисляется как

```
ShadowSize = PhysSize >> 3.
```

Отступ `ShadowOffset`, на который начало блока теневой памяти сдвинуто относительно начала участка физической памяти является фиксированным. Ясно, что поскольку область теневой памяти должна полностью лежать внутри физической, то это порождает требование `ShadowOffset + ShadowSize <= PhysSize`.

Поскольку физическая память во время запуска ядра может содержать случайные ненулевые значения, то после резервирования нужно ее обнулить. Кроме того, для резервирования теневой памяти необходимо, чтобы ядро уже инициализировало систему управления памятью, что происходит не совсем сразу после начала загрузки. До этого момента проверки адресуемости памяти включать нельзя. Это порождает дополнительную предварительную проверку того, была ли инициализирована теневая память или нет.

Вычисление адреса ячейки теневой памяти по адресу ячейки физической памяти происходит как

```
ShadowAddr = ((Addr - PhysOffset) >> 3) + PhysOffset + ShadowOffset,
```

где `PhysOffset` – адрес начала участка физической памяти. Несмотря на кажущуюся сложность, ясно, что это аналогично процедуре вида

```
ShadowAddr = (Addr >> 3) + Offset,
```

где

```
Offset = PhysOffset + ShadowOffset - (PhysOffset >> 3),
```

а значит метод вычисления адреса ячейки теневой памяти остается таким же, как и для детектора AddressSanitizer, только используется другое значение сдвига. Устройство виртуального адресного пространства для архитектуры x86-64 и схема хранения теневой памяти детектором KernelAddressSanitizer изображены на рис. 2.

4.2.3 Инструментация

KernelAddressSanitizer использует компиляторную инструментацию, реализованную как проход компилятора GCC [9]. В результате инструментации добавляются проверочные инструкции перед каждым обращением к физической памяти.

Как говорилось выше, процедура вычисления адреса ячейки теневой памяти по адресу ячейки физической памяти совпадает с соответствующей процедурой, используемой в детекторе AddressSanitizer. Однако в код, добавляемый инструментацией, необходимо включить несколько дополнительных проверок. Первая из них связана с необходимостью удостовериться, что теневая память уже была инициализирована. Затем для нахождения ошибок типа «использование памяти пользователя» выполняется проверка того, не лежит ли адрес обращения в адресном пространстве пользователя. Наконец, необходимо проверить, что адрес обращения лежит внутри физической памяти. Код, добавляемый при инструментации обращения к физической памяти размером `AccessSize` по адресу `Addr`, показан на листинге 13. Используемая в нем величина `KernelOffset` – это адрес начала области виртуальной памяти, соответствующей пространству ядра.

```
if (ShadowMemoryInitialized) {
    if (Addr < KernelOffset)
        ReportAndAbort(Addr);
    if (PhysOffset <= Addr && Addr < PhysOffset + PhysSize) {
        Offset = PhysOffset + ShadowOffset - (PhysOffset >> 3);
        ShadowAddr = (Addr >> 3) + Offset;
        k = *ShadowAddr;
        if (k != 0 && ((Addr & 7) + AccessSize > k))
            ReportAndAbort(Addr);
    }
}
```

Листинг 13: Инструментация обращения к памяти ядра

4.2.4 Библиотека времени исполнения

Библиотека времени исполнения реализована как модуль ядра. Он управляет теневой памятью и изменяет реализации некоторых функций, отвечающих за работу с физической памятью.

Во время загрузки ядра модуль резервирует и подготавливает к использованию теневую память. С помощью различных значений байтов теневой памяти различаются разные типы неадресуемой памяти (освобожденная память, зоны безопасности).

Модуль увеличивает размер всех создаваемых кэшей на фиксированное значение. Дополнительная память, занимаемая при создании объекта, используется в качестве зоны безопасности. При выделении объекта из этого кэша, последние несколько байт, соответствующие размерам зоны безопасности, помечаются как неадресуемые. Это означает, что у каждого объекта соответствующая ему зона безопасности находится справа. Поскольку объекты в блоке расположены один за другим, то слева от каждого из них кроме первого будет зона безопасности, соответствующая его левому соседу. В результате иногда выход влево за границы первого объекта может быть не обнаружен.

При создании нового блока он помечается неадресуемым и остается таким до тех пор, пока из него не будут выделены объекты. При выделении нового объекта соответствующая ему область памяти помечается адресуемой. Несмотря на то, что `kmalloc()` округляет размер объекта вверх до ближайшей степени двойки, адресуемым считается только участок памяти запрошенного размера. При освобождении

объекта соответствующая ему область памяти помечается неадресуемой.

Для повышения вероятности обнаружения ошибок типа «использование памяти после освобождения» KernelAddressSanitizer, как и AddressSanitizer, использует карантин. Кроме того, при создании и удалении объектов стек вызовов запоминается и сохраняется в память, соответствующую зонам безопасности.

4.3 Пример отчета об ошибке

Одним из ключевых преимуществ KernelAddressSanitizer по сравнению с существующими детекторами является очень подробный отчет о произошедшей ошибке. Ниже приведен отчет об одной из найденных ошибок.

Начинается отчет об ошибке с заголовка и стека вызовов для некорректного обращения к памяти как показано на листинге 14. В заголовке можно видеть, что произошла ошибка типа «использование неинициализированной памяти» (англ. use-after-free) в функции `ipv4_dst_check`, которая совершила некорректное чтение двух байт освобожденной памяти. Далее приводится стек вызовов, в котором показаны имена вызываемых функций, адреса вызывающих инструкций и даже указаны номера строк в соответствующих файлах с исходным кодом.

```
1  AddressSanitizer: heap-use-after-free in ipv4_dst_check
2  Read of size 2 by thread T15453:
3   [<ffffffff817daa3a>] ipv4_dst_check+0x1a/0x90
       ./net/ipv4/route.c:1116
4   [<ffffffff8175b789>] __sk_dst_check+0x89/0xe0
       ./net/core/sock.c:531
5   [<ffffffff81830a36>] ip4_datagram_release_cb+0x46/0x390 ??:0
6   [<ffffffff8175eaea>] release_sock+0x17a/0x230
       ./net/core/sock.c:2413
7   [<ffffffff81830882>] ip4_datagram_connect+0x462/0x5d0 ??:0
8   [<ffffffff81846d06>] inet_dgram_connect+0x76/0xd0
       ./net/ipv4/af_inet.c:534
9   [<ffffffff817580ac>] SYSC_connect+0x15c/0x1c0 ./net/socket.c:1701
10  [<ffffffff817596ce>] SyS_connect+0xe/0x10 ./net/socket.c:1682
11  [<ffffffff818b0a29>] system_call_fastpath+0x16/0x1b
       ./arch/x86/kernel/entry_64.S:629
```

Листинг 14: Отчет детектора KernelAddressSanitizer: заголовок сообщения об ошибке и стек вызовов для некорректного обращения к памяти

Далее в отчете приведены стеки вызовов, сохраненные во время освобождения

и предшествующего ему выделения соответствующего блока памяти как видно на листинге 15.

```
Freed by thread T15455:
 [<ffffffff8178d9b8>] dst_destroy+0xa8/0x160 ./net/core/dst.c:251
 [<ffffffff8178de25>] dst_release+0x45/0x80 ./net/core/dst.c:280
 [<ffffffff818304c1>] ip4_datagram_connect+0xa1/0x5d0 ??:0
 [<ffffffff81846d06>] inet_dgram_connect+0x76/0xd0
    ./net/ipv4/af_inet.c:534
 [<ffffffff817580ac>] SYSC_connect+0x15c/0x1c0 ./net/socket.c:1701
 [<ffffffff817596ce>] SyS_connect+0xe/0x10 ./net/socket.c:1682
 [<ffffffff818b0a29>] system_call_fastpath+0x16/0x1b
    ./arch/x86/kernel/entry_64.S:629

Allocated by thread T15453:
 [<ffffffff8178d291>] dst_alloc+0x81/0x2b0 ./net/core/dst.c:171
 [<ffffffff817db3b7>] rt_dst_alloc+0x47/0x50
    ./net/ipv4/route.c:1406
 [<     inlined    >] __ip_route_output_key+0x3e8/0xf70
    __mkroute_output ./net/ipv4/route.c:1939
 [<ffffffff817dde08>] __ip_route_output_key+0x3e8/0xf70
    ./net/ipv4/route.c:2161
 [<ffffffff817deb34>] ip_route_output_flow+0x14/0x30
    ./net/ipv4/route.c:2249
 [<ffffffff81830737>] ip4_datagram_connect+0x317/0x5d0 ??:0
 [<ffffffff81846d06>] inet_dgram_connect+0x76/0xd0
    ./net/ipv4/af_inet.c:534
 [<ffffffff817580ac>] SYSC_connect+0x15c/0x1c0 ./net/socket.c:1701
 [<ffffffff817596ce>] SyS_connect+0xe/0x10 ./net/socket.c:1682
 [<ffffffff818b0a29>] system_call_fastpath+0x16/0x1b
    ./arch/x86/kernel/entry_64.S:629
```

Листинг 15: Отчет детектора KernelAddressSanitizer: стек вызовов для освобождения блока памяти, используемого при некорректном обращении к памяти

Затем в отчете приводится информация об используемом блоке памяти: адрес его начала и конца, его размер, а также адрес некорректного обращения, как показано на листинге 16.

```
The buggy address ffff880024ff2266 is located 102 bytes inside
 of 192-byte region [ffff880024ff2200, ffff880024ff22c0)
```

Листинг 16: Отчет детектора KernelAddressSanitizer: адрес некорректного обращения к памяти и описание используемого блока памяти

Последняя часть отчета представляет собой описание состояния памяти вокруг адреса некорректного обращения как видно на листинге 17. Каждый из символов

в описании соответствует выровненному 8-байтному участку памяти ядра. Разными символами кодируются различные значения теневого байта, соответствующего этому участку. Символ «.» означает, что все 8 байт адресуемы, «f» – освобождены, а «r» – принадлежат зоне безопасности. Стрелочками указано местоположение байта теневой памяти, который соответствует адресу обращения. В данном случае, этот теневой байт кодируется символом «f». Это означает, что произошло обращение к освобожденному участку памяти.

```
1  Memory state around the buggy address:
2   ffff880024ff1d00: ffffffff fffrrrrr rrrrrrrr rrrrrrrr
3   ffff880024ff1e00: ffffffff ffffffff ffffffff fffrrrrr
4   ffff880024ff1f00: rrrrrrrr rrrrrrrr rrrrrrrr rrrrrrrr
5   ffff880024ff2000: rrrrrrrr rrrrrrrr rrrrrrrr rrrrrrrr
6   ffff880024ff2100: rrrrrrrr rrrrrrrr rrrrrrrr rrrrrrrr
7  >ffff880024ff2200: ffffffff ffffffff ffffffff rrrrrrrr
8                                ^
9   ffff880024ff2300: rrrrrrrr rrrrrrrr ........ ........
10  ffff880024ff2400: ........ rrrrrrrr rrrrrrrr rrrrrrrr
11  ffff880024ff2500: ffffffff ffffffff ffffffff rrrrrrrr
12  ffff880024ff2600: rrrrrrrr rrrrrrrr ffffffff ffffffff
13  ffff880024ff2700: ffffffff rrrrrrrr rrrrrrrr rrrrrrrr
14 Legend:
15  f - 8 freed bytes
16  r - 8 redzone bytes
17  . - 8 allocated bytes
18  x=1..7 - x allocated bytes + (8-x) redzone bytes
```

Листинг 17: Отчет детектора KernelAddressSanitizer: состояние памяти вокруг адреса некорректного обращения к памяти

После отправки этого отчета разработчикам ядра [27], благодаря его подробности[1], ошибка была найдена и устранена [28].

4.4 Производительность

Измерение производительности детектора KernelAddressSanitizer производилось в виртуальной машине VirtualBox 4.3.12 с установленной операционной системой Ubuntu 14.04 LTS. Машине было выделено 2 ГБ памяти и 2 ядра процессора Intel

[1] «Yeah, we had many reports in the past that something was wrong... Your nice report made me take a look, finally.», Eric Dumazet, Netdev Mailing List, 2014-06-06.

Таблица 2: Производительность детектора KernelAddressSanitizer

	Чистое ядро	Ядро с KASan	Увеличение
Используемая память	36.93 МБ	83.66 МБ	2.27x
Время загрузки	22.95 сек	32.98 сек	1.44x

Таблица 3: Сравнение детекторов kmemcheck и KernelAddressSanitizer (KASan)

	kmemcheck	KASan
Переполнение буфера	Да	Да
Использование памяти после освобождения	Иногда	Да
Использование неинициализированных данных	Да	Нет
Использование памяти пользователя	Нет	Да
Замедление	10x	1.5x
Увеличение использования памяти	2x	2x

Core i5-2500, 3.7 ГГц. Опции конфигурирования для компиляции ядра были взяты из предустановленного ядра в Ubuntu и отличались только добавлением опции, включающей детектор KernelAddressSanitizer.

Производилось измерение двух величин: увеличения потребления памяти ядром и уменьшения скорости его работы. Для оценки средних значений этих величин использовались соответственно количество используемой ядром физической памяти после его загрузки и время загрузки ядра. Результаты приведены в таблице 2. Дополнительное увеличение использования памяти на фиксированную величину, возникающее из-за использования карантина, не учитывалось в этом эксперименте.

Сравнение детекторов kmemcheck и KernelAddressSanitizer приведено в таблице 3. Необходимо пояснить, что kmemcheck редко находит ошибки типа «использование памяти после освобождения», поскольку не использует карантин (см. раздел 4.2). Несмотря на то, что производительность KernelAddressSanitizer выше, чем производительность kmemcheck, говорить, что первый лучше, некорректно, поскольку они находят разные типы ошибок.

4.5 Найденные ошибки

Тестирование ядра с помощью динамических детекторов осложняется отсутствием набора модульных тестов с достаточно высоким покрытием кода[2]. В результате, приходится использовать рандомизированное тестирование.

Trinity [29] – это одна из утилит, применяемая для рандомизированного тестирования ядра Linux. Принцип ее работы заключается в вызове системных вызовов со случайными аргументами. Однако, передавая в системные вызовы абсолютно случайные аргументы, их выполнение часто не проходит дальше, чем проверка этих аргументов на корректность. Trinity поступает умнее и передает не совсем случайные аргументы. Например, при запуске она генерирует список всех доступных файловых дескрипторов и передает один из них в качестве аргумента тем системным вызовам, соответствующим аргументом которых должен быть именно файловый дескриптор.

В процессе тестирования ядра с применением детектора KernelAddressSanitizer и утилиты Trinity было найдено 15 ошибок, из них 6 ошибок типа «переполнение буфера» и 9 ошибок типа «использование памяти после освобождения». Часть ошибок были подтверждены (11 ошибок) и исправлены (7 ошибок) разработчиками ядра.

Одна из найденных ошибок оказалась серьезной уязвимостью подсистемы ядра ipv6 [30]. С помощью этой уязвимости, можно вызвать аварийное завершение удаленного компьютера посредством отправки на него определенных UDP пакетов. Уязвимы оказались несколько современных операционных систем на базе Linux: Ubuntu 12.04 LTS, Red Hat Enterprise Linux Server 6 и другие [31].

5 Заключение

Для автоматического поиска ошибок в ядре Linux был разработан детектор KernelAddressSanitizer. Основными преимуществами нового детектора над существующими является высокая производительность, эффективность и качество отчетов о найденных ошибках. В результате тестирования ядра Linux с применением разработанного детектора, были найдены более 10 ранее неизвестных дефектов в ядре Linux, часть которых была впоследствии исправлена разработчиками ядра.

[2] «"Regression testing"? What's that? If it compiles, it is good; if it boots up, it is perfect.», Linus Torvalds, Linux Kernel Mailing List, 1998-04-08.

Основные результаты работы:

- Реализована инструментация кода ядра ОС Linux, основанная на компиляторе GCC.
- Разработан прототип детектора ошибок работы с памятью в ядре ОС Linux KernelAddressSanitizer, основанный на алгоритме, используемом в детекторе AddressSanitizer.
- Исходный код детектора KernelAddressSaitizer открыт и доступен для использования.
- Детектор KernelAddressSanitizer внедрен в регулярный процесс тестирования ядра ОС Linux в компании Google.
- Найдено более 10 настоящих ошибок работы с памятью в ядре ОС Linux, часть которых подтверждена и исправлена разработчиками ядра.
- Результаты представлены на 56-й научной конференции МФТИ [32].

Список литературы

[1] Leveson Nancy G, Turner Clark S. An investigation of the therac-25 accidents // Computer. — 1993. — V. 26, no. 7. — P. 18–41.

[2] Linux. — http://ru.wikipedia.org/wiki/Linux.

[3] Ernst Michael D. Static and dynamic analysis: Synergy and duality // WODA 2003: ICSE Workshop on Dynamic Analysis / Citeseer. — 2003. — P. 24–27.

[4] Исходжанов Т.Р. Автоматический поиск ошибок в компьютерных программах с применением динамического анализа: дис. канд. физ.-мат. наук / Т.Р. Исходжанов. — МФТИ. — 2013.

[5] AddressSanitizer: a fast address sanity checker / Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov // Proceedings of the 2012 USENIX conference on Annual Technical Conference. — USENIX ATC'12. — Berkeley, CA, USA, 2012. — P. 28–28.

[6] Seward Julian, Nethercote Nicholas. Using Valgrind to detect undefined value errors with bit-precision // USENIX Annual Technical Conference. — 2005. — P. 17–30.

[7] GETTING STARTED WITH KMEMCHECK. — https://www.kernel.org/doc/Documentation/kmemcheck.txt.

[8] 2011 CWE/SANS Top 25 Most Dangerous Software Errors. — http://cwe.mitre.org/top25/.

[9] GCC, the GNU Compiler Collection. — http://gcc.gnu.org/.

[10] GCC Bugzilla — Bug 33498 — Optimizer (-O2) may convert a normal loop to infinite. — http://gcc.gnu.org/bugzilla/show_bug.cgi?id=33498.

[11] Fuzz testing. — http://ru.wikipedia.org/wiki/Fuzz_testing.

[12] MemorySanitizer. — https://code.google.com/p/memory-sanitizer.

[13] Memcheck: a memory error detector. — http://valgrind.org/docs/manual/mc-manual.html.

[14] Nethercote Nicholas, Seward Julian. Valgrind: A framework for heavyweight dynamic binary instrumentation // Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. — PLDI '07. — New York, NY, USA: ACM, 2007. — P. 89–100.

[15] Pin: building customized program analysis tools with dynamic instrumentation / Chi-Keung Luk, Robert Cohn, Robert Muth et al. // Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. — PLDI '05. — New York, NY, USA: ACM, 2005. — P. 190–200.

[16] Bruening Derek. Efficient, Transparent, and Comprehensive Runtime Code Manipulation: Ph. D. thesis / Derek Bruening. — M.I.T. — 2004.

[17] gcov — a Test Coverage Program. — `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[18] Eigler Frank Ch. Mudflap: pointer use checking for C/C++ // GCC Developers Summit / Red Hat Inc. — 2003.

[19] Pebil: Efficient static binary instrumentation for linux / Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, Allan Snavely // Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on / IEEE. — 2010. — P. 175–183.

[20] Bungale Prashanth P, Luk Chi-Keung. PinOS: a programmable framework for whole-system dynamic instrumentation // Proceedings of the 3rd international conference on Virtual execution environments / ACM. — 2007. — P. 137–147.

[21] Kprobes. — `https://sourceware.org/systemtap/kprobes/`.

[22] Tamches Ariel, Miller Barton P. Fine-grained dynamic instrumentation of commodity operating system kernels: Ph. D. thesis / Ariel Tamches, Barton P Miller. — University of Wisconsin–Madison. — 2001.

[23] ftrace - Function Tracer. — `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`.

[24] clang: a C language family frontend for LLVM. — `http://clang.llvm.org/`.

[25] Finding errors with kmemcheck. — http://www.diku.dk/hjemmesider/ansatte/julia/cocciwk/nossum.pdf.

[26] Linux Kernel: Virtual memory map. — https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt.

[27] Potential race in ip4_datagram_release_cb. — http://www.spinics.net/lists/netdev/msg285419.html.

[28] ipv4: fix a race in ip4_datagram_release_cb(). — http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=9709674e68646cee5a24e3000b3558d25412203a.

[29] Trinity: A Linux System call fuzz tester. — http://codemonkey.org.uk/projects/trinity/.

[30] CVE-2013-4387. — http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4387/.

[31] Linux Kernel CVE-2013-4387 Memory Corruption Vulnerability. — http://www.securityfocus.com/bid/62696.

[32] Коновалов А.Д., Вьюков Д.С. Автоматический поиск ошибок в ядре операционной системы Linux с применением динамического анализа // Труды 56-й научной конференции МФТИ. Управление и прикладная математика. Т. 2 / МФТИ. — 2013. — С. 162–163.