ПРОГРАММИРОВАНИЕ, 2020, № 4, с. 46–64

ЯЗЫКИ, КОМПИЛЯТОРЫ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

УДК 004.415.533

СРЕДСТВА ДИНАМИЧЕСКОГО АНАЛИЗА ПРОГРАММ В КОМПИЛЯТОРАХ GCC И CLANG

© 2020 г. Н. И. Вьюкова[a],*, В. А. Галатенко[a],, С. В. Самборский[a],*****

[a]Федеральное государственное учреждение "Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук"
117218 Москва, Нахимовский проспект, д. 36, к. 1, Россия

**E-mail: niva@niisi.ras.ru*

***E-mail: galat@niisi.ras.ru*

****E-mail: sambor@niisi.ras.ru*

Поступила в редакцию 28.07.2019 г.
После доработки 13.09.2019 г.
Принята к публикации 23.09.2019 г.

Усиливающиеся требования безопасности, предъявляемые к программному обеспечению, рост объемов программных проектов и необходимость постоянно сокращать сроки разработки и выпуска новых версий вызвали настоятельную потребность в средствах динамического анализа, которые бы позволяли эффективно выявлять уязвимости программ на языках C и C++ и предотвращать их эксплуатацию. В статье рассмотрены два вида средств динамического анализа, реализованных в компиляторах gcc и clang и получивших название "санитайзеры". Первый вид санитайзеров используется на стадии тестирования и предназначен для обнаружения ошибок работы с памятью, ошибок несоответствия типов и других уязвимостей, часто остающихся не выявленными при традиционном тестировании. Более эффективному выявлению уязвимостей способствует применение санитайзеров в сочетании с тестированием на автоматически генерируемых случайных наборах данных. Другой вид санитайзеров предназначен для противодействия угрозам безопасности программ, выполняющихся в производственном режиме. Эти средства имеют низкие накладные расходы и ориентированы на поддержание целостности потока управления программы. Применение санитайзеров в значительной мере компенсирует недостатки языков C и C++, такие как небезопасность операций с памятью, возможность небезопасной работы с типами и другие. В то же время в этой области остается ряд нерешенных задач, краткий обзор которых приведен в заключении.

DOI: 10.31857/S0132347420010082

1. ВВЕДЕНИЕ

Языки C и C++ проектировались и развивались как инструменты для создания высокоэффективных программ. Платой за эффективность стало отсутствие требований по осуществлению проверок корректности вычислений во время выполнения. Например, от реализаций этих языков не требуются проверки безопасности доступа к объектам в памяти или отсутствия переполнения при операциях над целочисленными данными. Ответственность за обеспечение разнообразных условий корректности возлагается на программиста. В стандартах языков C и C++ это свойство отражено наличием многочисленных ситуаций неопределенного поведения: "при несоблюдении условия A результат операции B не определен". Неопределенное поведение программы, согласно комментариям к стандартам C и C++, может варьироваться от полного игнорирования ситуации с непредсказуемыми последствиями, до некоторого детерминированного поведения, отраженного в документации, включая, например, прекращение трансляции или выполнения с выдачей диагностики.

Наличие неопределенного поведения в программе далеко не всегда приводит к видимым дефектам ее поведения – программа может годами работать без проблем. Тем не менее оно считается ошибкой по следующим причинам.

• Программа может повести себя непредсказуемо при необычных редко встречающихся входных данных.

• Многие виды неопределенного поведения представляют угрозу безопасности, так как являются уязвимостями, которые могут эксплуатироваться вредоносным программным обеспечением (ПО).

• Наличие неопределенного поведения ухудшает портируемость программы и может стать "бомбой замедленного действия" – программа

может перестать работать корректно при переходе на другую инструментальную платформу или даже к другой версии компилятора либо операционной системы.

Компиляторы старого поколения для некоторых видов неопределенного поведения зачастую обеспечивали интуитивно ожидаемое поведение программ. Современные же компиляторы имеют тенденцию максимально использовать предоставляемую стандартами языков C и C++ свободу для проведения агрессивной оптимизации кода, что может приводить к неожиданному поведению программ. Подробное рассмотрение различных аспектов неопределенного поведения программ представлено в публикации [1].

Для выявления неопределенного поведения в программах традиционно применялись средства статического анализа. Методы статического анализа программ постоянно развиваются (см., например, [2]), тем не менее они остаются принципиально ограниченными в силу невозможности промоделировать все возможные варианты выполнения программы. К числу недостатков средств статического анализа относится возможность ложных срабатываний.

В дополнение к статическим анализаторам применяются инструменты динамического анализа программ для выявления ошибок, имеющие давнюю историю [3]. В настоящее время широко применяются свободно доступные инструменты Memcheck на основе инфраструктуры Valgrind, Dr. Memory, а также различные коммерческие продукты, такие как BoundsChecker, IBM Rational PurifyPlus, Intel® Inspector, продукты компании Parasoft.

Инструменты динамического анализа могут быть классифицированы по ряду признаков.

• Языки программирования, для которых предназначен инструмент анализа.

• Набор выявляемых ошибок. Например, ошибки обращения к памяти, ошибки синхронизации многопоточных программ или другие виды ошибок.

• Набор поддерживаемых аппаратно-программных платформ.

• Представление программы, с которым работает инструмент анализа. Это может быть выполняемый модуль или исходный код программы.

• Поведение при обнаружении ошибки. Возможные варианты – аварийное завершение программы с предварительной выдачей диагностики (или без нее), выдача диагностики и продолжение работы, продолжение выполнения с корректировкой ошибочного поведения программы. Пример последнего из перечисленных подходов представлен в [4], где описан метод контекстно-зависимого обхода ошибок переполнения буфера в некоторых стандартных функциях языка C.

• Основное назначение. Инструменты динамического анализа могут использоваться для тестирования и отладки программ либо для предотвращения эксплуатации уязвимостей в программах, выполняющихся в производственном режиме. В последнем случае ключевым требованием к инструменту анализа являются низкие накладные расходы, связанные с его применением.

Эта статья посвящена средствам динамического анализа, появившимся в последнее десятилетние в компиляторах clang и gcc и получившим название “санитайзеры”. Соответствующий англоязычный термин, sanitizer, переводится на русский как “дезинфицирующее средство”, что неплохо отражает назначение этих инструментов. Санитайзеры предназначены для выявления ситуаций неопределенного поведения во время работы программы, таких как ошибки работы с памятью, ошибки синхронизации в многопоточных программах и другие. В сравнении с инструментами на основе Valgrind и Dr. Memory, санитайзеры clang и gcc поддерживают распознавание ряда дополнительных классов ошибок работы с памятью, в частности, переполнение буфера в статически выделяемой памяти и в локальной памяти (стеке) функций, доступ к локальным объектам функции после выхода из нее.

Поддерживается также распознавание широкого класса ошибок, не связанных с доступом к памяти. Коэффициент замедления типичной программы при использовании санитайзера значительно ниже, чем при использовании упомянутых выше инструментов, что позволяет шире применять динамический анализ для тестировании кода в цикле разработки ПО. В частности, высокую результативность показало применение санитайзеров в сочетании с массированным фаззинг-тестированием, когда используется генерация случайных наборов входных данных для тестируемых приложений или библиотечных функций.

В clang и (в меньшей степени) в gcc поддерживается также ряд средств, предотвращающих эксплуатацию уязвимостей программ. Эти средства можно классифицировать как санитайзеры целостности потока управления (Control Flow Integrity – CFI). Они нейтрализуют последствия различных ошибок, таких как переполнение буфера, не позволяя злоумышленнику перенаправить поток управления программы и выполнить нужный ему код.

Дальнейшее содержание статьи построено по следующему плану. В главе 2 рассмотрены санитайзеры, применяемые для тестирования ПО. Они позволяют обнаруживать ошибки доступа к объектам в памяти, ошибки синхронизации многопоточных программ и многие другие виды не-

определенного поведения программ, которые обычно остаются не выявленными при традиционном тестировании. В главе 3 дается представление о фаззинг-тестировании и преимуществах его применения совместно с санитайзерами. Четвертая глава посвящена санитайзерам целостности потока управления, применяемым для отражения угроз безопасности при промышленной эксплуатации программ. В заключение анализируются некоторые из нерешенных пока задач в этой области и рассматриваются направления дальнейших исследований и разработок.

2. САНИТАЙЗЕРЫ ТЕСТИРОВАНИЯ И ОТЛАДКИ

В этом разделе представлены санитайзеры, предназначенные для применения на стадии тестирования ПО: санитайзер ошибок адресации (AddressSanitizer), санитайзер утечек памяти (LeakSanitizer), санитайзер неинициализированной памяти (MemorySanitizer), санитайзер многопоточных программ (ThreadSanitizer) и санитайзер неопределенного поведения (UndefinedBehaviorSanitizer).

2.1. AddressSanitizer и LeakSanitizer

Инструмент AddressSanitizer [5], поддерживаемый компиляторами clang и gcc, предназначен для выявления ошибок работы с памятью. В GCC, начиная с версии 4.9, он заменил имевшийся там ранее инструмент аналогичного назначения Mudflap.

AddressSanitizer выявляет как пространственные, так и темпоральные ошибки работы с памятью. К первому классу относятся ошибки обращения к памяти по адресу, находящемуся за пределами объекта, такие как выход за границу массива. К темпоральным ошибкам относится доступ к объекту, время жизни которого еще не началось или уже закончилось. Ошибки, выявляемые при помощи AddressSanitizer:

• выход за границы массива (буфера) в динамической памяти, стеке и в статически выделяемой памяти;

• использование динамической памяти после ее освобождения;

• использование локального объекта после выхода из функции, где объект был определен;

• использование объекта после выхода из области его определения;

• повторное или некорректное освобождение памяти;

• некорректные операции над указателями, такие как сравнение указателей, не указывающих на один и тот же объект;

• некорректные аргументы стандартных функций `strcat`, `strcpy`, `memcpy` и других;

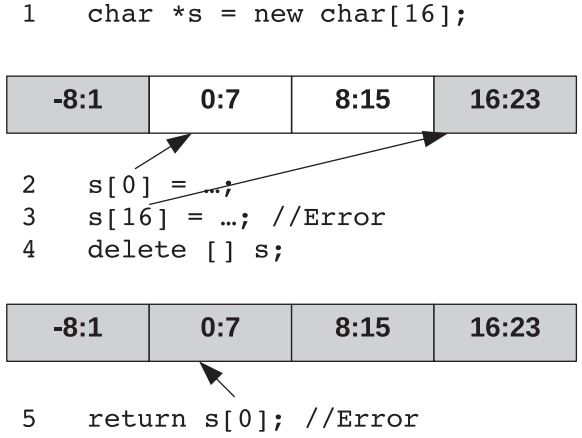

Рис. 1. Принципы работы AddressSanitizer.

• некорректный порядок инициализации глобальных переменных в программах на C++;

• утечки памяти.

AddressSanitizer распознает все основные классы ошибок, выявляемые другими аналогичными средствами, за исключением ошибок использования неинициализированной памяти, для выявления которых предназначен MemorySanitizer. Реализация AddressSanitizer включает модуль компилятора, выполняющий инструментирование программы, и библиотеку времени выполнения. Библиотека времени выполнения включает санитарные версии ряда стандартных функций, таких как `malloc`, `free`, `strcat`, `strcpy`, `memcpy` и других.

Выявление ошибок основано на механизме теневой памяти и выделении “санитарных зон” между объектами в памяти. Рис. 1 иллюстрирует принципы работы AddressSanitizer. Санитарные зоны, выделяемые при создании объекта (строка 1) выделены на рисунке серым фоном. Разметка теневой памяти позволяет определить для каждого байта основной памяти, принадлежит ли он объекту или санитарной зоне. Перед каждым обращением к памяти добавляется проверка ее статуса, и попытка обратиться к памяти санитарной зоны (строка 3) приводит к ошибке. Нетрудно видеть, что, увеличив значение индекса, можно получить доступ к памяти другого объекта, находящегося за санитарной зоной. Такие ошибки AddressSanitizer не обнаруживает.

При уничтожении объекта (строка 4) освобождаемая память помечается как санитарная зона и помещается в карантин, так что ее повторное выделение максимально откладывается; это позволяет идентифицировать ошибки вида “использование после освобождения” (строка 5).

Для активации AddressSanitizer необходимо задать ключ `-fsanitize=address`. Анализатор утечек памяти LeakSanitizer является частью AddressSanitizer и активируется по ключу `-fsani-`

tize=address. Но он также может быть использован самостоятельно при помощи ключа -fsanitize=leak.

Типичное замедление программы, использующей AddressSanitizer, составляет 2x, что существенно меньше, чем замедление при использовании других инструментов. Более высокая эффективность достигается в какой-то мере за счет того, что AddressSanitizer инструментирует код во время компиляции, что позволяет избежать больших задержек на старте программы. Объем требуемой памяти увеличивается в 2–4 раза. Размер стека может увеличиваться примерно в 3 раза. Подробное сравнение AddressSanitizer с аналогичными средствами приведено в [5] и [6].

2.2. MemorySanitizer

Инструмент MemorySanitizer [7] предназначен для отслеживания ситуаций, когда в программе на C или C++ используются неинициализированные данные, то есть когда чтение из стека или из динамически выделенной памяти происходит до записи в эту память. В настоящее время он реализован только в компиляторе clang и активируется ключом -fsanitize=memory.

Отслеживание инициализации данных ведется на уровне отдельных бит памяти, то есть MemorySanitizer способен диагностировать ошибку вплоть до битовых полей в структурах. Копирование неинициализированных данных и простейшие логические или арифметические операции над ними не приводят к ошибке, поскольку это не запрещено стандартами. MemorySanitizer молча отслеживает распространение неинициализированных данных и выдает сообщение об ошибке лишь тогда, когда выполнение программы достигает условного перехода, системного вызова или разыменования указателя, зависящего от неинициализированного значения. MemorySanitizer также включает экспериментальную реализацию проверок использования объектов после выполнения деструкторов.

Стандартная диагностика показывает место, где произошло некорректное использование неопределенного значения, но этого может быть недостаточно для локализации ошибки. Источник ошибки использования неопределенного значения может находиться далеко от места ее проявления, как текстуально, так и по потоку выполнения. Опция -fsanitize-memory-track-origins, аналогичная опции –track-origins=yes инструмента Memcheck, позволяет отслеживать источник ошибки; при этом в диагностическую выдачу включается место создания объекта и все события записи в память неинициализированного значения. В листинге 1 приведен пример программы и диагностики с отслеживанием источника.

```
$ cat -n msan.C
 1  #include <stdio.h>
 2  int x[1] = {9};
 3  int main(int argc, char** argv) {
 4    typedef int* intp;
 5    intp *a = new intp [10];
 6    a[argc] = x;
 7    int *b = a [2];
 8
 9    printf ("*b = %d\n", *b);
10    return 0;
11  }
$ clang++ msan.C -g -Wall -fsanitize=memory \
-fsanitize-memory-track-origins
$ ./a.out
==27687==WARNING: MemorySanitizer:
use-of-uninitialized-value
  #0 0x4a205d in main /home/user/memsan/msan.C:9:24
  #1 0x7fb87418a430 in __libc_start_main
(/lib64/libc.so.6+0x20430)
  #2 0x41b929 in _start (/home/user/memsan/a.out+0x41b929)
Uninitialized value was stored to memory at
  #0 0x4a1fe2 in main /home/user/memsan/msan.C:7:8
```

```
    #1 0x7fb87418a430 in __libc_start_main
(/lib64/libc.so.6+0x20430)
Uninitialized value was created by a heap allocation
    #0 0x49f104 in operator new[](unsigned long)
/home/user/llvm-project-8.0.0/compiler-rt/lib/
msan/msan_new_delete.cc:48
    #1 0x4a1dec in main /home/user/memsan/msan.C:5:13
    #2 0x7fb87418a430 in __libc_start_main
(/lib64/libc.so.6+0x20430)
SUMMARY: MemorySanitizer: use-of-uninitialized-value
/home/user/memsan/msan.C:9:24 in main
```

Листинг 1: Пример программы и диагностики MemorySanitizer с отслеживанием источника.

Условием корректной работы MemorySanitizer является компиляция всего приложения, включая стандартные библиотеки, с ключом `-fsanitize=memory`. Несоблюдение этого условия может приводить к ложным срабатываниям. Поэтому вам придется самостоятельно собрать требуемую версию стандартной библиотеки C++ согласно процедуре, приведенной в документации. Необходимо будет также заменить в программе ассемблерные модули и ассемблерные вставки на код на языке C. Для того чтобы упростить использование MemorySanitizer, в его библиотеку времени выполнения включены “санитарные” версии около 300 наиболее употребительных функций библиотеки языка C, что позволяет применять его с неинструментированной библиотекой libc.

Использование памяти при работе с MemorySanitizer увеличивается в два раза, а при отслеживании источника – в 3 раза. Выполнение типичной программы, использующей MemorySanitizer, замедляется примерно в 3 раза, что значительно меньше, чем при использовании Memcheck или Dr. Memory. Более высокая эффективность MemorySanitizer по сравнению с другими средствами объясняется, во-первых, тем, что инструментирование выполняется на стадии компиляции, и тем самым исключается длительная задержка при старте программы; вторая причина заключается в том, что MemorySanitizer выполняет только проверку использования неинициализированной памяти, в то время как Memcheck и Dr. Memory совмещают функциональность MemorySanitizer и AddressSanitizer. Хотя совместное использование MemorySanitizer и AddressSanitizer не поддерживается, но их последовательное применение занимает, как правило, значительно меньше времени, чем тестирование с помощью Memcheck или Dr. Memory. Сравнение MemorySanitizer с аналогичными инструментами, а также вопросы реализации представлены в [7].

Низкие накладные расходы позволяют применять AddressSanitizer и MemorySanitizer на регулярной основе в цикле разработки ПО как для модульного и регрессивного тестирования, так и в сочетании с фаззинг-тестированием. MemorySanitizer был опробован для тестирования ряда больших проектов, включая сам компилятор clang, компилятор gcc, а также различные серверные приложения Google, где с его помощью было найдено более 500 ошибок.

2.3. ThreadSanitizer

ThreadSanitizer [8] представляет инструмент для выявления ситуаций гонки данных (data races), возникающих в результате ошибок синхронизации многопоточных программ. Реализация включает модуль инструментирования программы в компиляторе и библиотеку времени выполнения.

```
$ cat -n race1.cc
 1 #include <pthread.h>
 2 #include <stdio.h>
 3
 4 int Global;
 5
 6 void *Thread1(void *x) {
 7   Global++;
 8   return NULL;
```

```
 9 }
10
11 void *Thread2(void *x) {
12   Global--;
13   return NULL;
14 }
15
16 int main() {
17   pthread_t t[2];
18   pthread_create(&t[0], NULL, Thread1, NULL);
19   pthread_create(&t[1], NULL, Thread2, NULL);
20   pthread_join(t[0], NULL);
21   pthread_join(t[1], NULL);
22 }
$ clang++ race1.cc -fsanitize=thread -g
$ ./a.out
==================
WARNING: ThreadSanitizer: data race (pid=10915)
 Write of size 4 at 0x0000011a78c8 by thread T2:
  #0 Thread2(void*) /home/user/tsan/race1.cc:12:9
(a.out+0x4c711e)
Previous write of size 4 at 0x0000011a78c8 by thread T1:
  #0 Thread1(void*) /home/user/tsan/race1.cc:7:9
(a.out+0x4c70be)
Location is global 'Global' of size 4 at 0x0000011a78c8
(a.out+0x0000011a78c8)
Thread T2 (tid=10918, running) created by main thread at:
  #0 pthread_create /home/user/llvm-project/compiler-rt
/lib/tsan/rtl/tsan_interceptors.cc:975 (a.out+0x429596)
  #1 main /home/user/tsan/race1.cc:19:3 (a.out+0x4c718a)
Thread T1 (tid=10917, finished) created by main thread at:
  #0 pthread_create /home/user/llvm-project/compiler-rt/
lib/tsan/rtl/tsan_interceptors.cc:975 (a.out+0x429596)
  #1 main /home/user/tsan/race1.cc:18:3 (a.out+0x4c7171)
SUMMARY: ThreadSanitizer: data race /home/user/tsan/
race1.cc:12:9 in Thread2(void*)
```

Листинг 2: Пример программы и диагностики ThreadSanitizer.

Во время работы программы регистрируются события доступа к памяти и события синхронизации. Под управлением наблюдаемых событий и некоторого конечного автомата изменяется текущее состояние программы, которое включает глобальное состояние и состояния отдельных потоков. Состояния, соответствующие некорректным последовательностям событий, трактуются как ошибки. В листинге 2 показан пример программы и диагностика, выданная ThreadSanitizer. Другие примеры типичных ошибок представлены в [9].

Замедление программы в результате использования ThreadSanitizer составляет от 5 до 15 раз. Расход памяти может возрастать в 5 – 10 раз. ThreadSanitizer проверялся преимущественно на программах, использующих библиотеку pthread; работа с библиотекой потоков C++11 проверялась пока недостаточно.

Весь код должен быть скомпилирован с опцией `-fsanitize=thread`, включая стандартные библиотеки C и C++, в противном случае возможны как ложноположительные, так и ложноотрицательные срабатывания, а также могут быть показаны не все кадры стека. Не поддерживается статическая компоновка с libc, libstdc++. Диагностируются только ошибки, фактически произошедшие при данном выполнении, поэтому

при тестировании программы желательно обеспечить реалистичную нагрузку.

2.4. UndefinedBehaviorSanitizer

Санитайзер UndefinedBehaviorSanitizer позволяет динамически выявлять в программе разнообразные виды неопределенного поведения, помимо рассмотренных в предыдущих подразделах. Вся совокупность проверок активируется ключом `-fsanitize=undefined`. Ниже перечислены отдельные виды проверок, которые можно задавать ключами вида `-fsanitize=`*проверка*.

`signed-integer-overflow` – переполнение в операциях знаковой целочисленной арифметики. Ошибки этого типа трудно поддаются обнаружению и представляют собой уязвимости, создающие угрозу безопасности ПО [10]. В списке известных уязвимостей от 2011 г. они фигурируют в числе 25 наиболее опасных [11].

`float-cast-overflow` – переполнение при преобразовании из вещественного типа в целочисленный или обратно, а также при преобразовании между двумя вещественными типами.

`bounds` – выход за границы массива при индексной адресации в случаях, когда границы массива могут быть вычислены статически. В листинге 3 приведен пример программы и диагностики UndefinedBehaviorSanitizer.

```
$ cat -n bounds.c
 1 int ops [13] = {11, 12, 46, 3, 2, 2, 3, 2, 1, 3, 2, 1, 2};
 2 int num = 13;
 3
 4 int main()
 5 {
 6   int i;
 7   for (i = 0; i < num; i++)
 8     {
 9        int j;
10        for (j = num - 1; j >= i; j--)
11          {
12             if (ops[j-1] < ops[j])
13              {
14                 int op = ops[j];
15                 ops[j] = ops[j-1];
16                 ops[j-1] = op;
17              }
18          }
19     }
20   return 0;
21 }
clang -g -O1 bounds.c -fsanitize=bounds -g -Wall
$ ./a.out
bounds.c:12:15: runtime error: index -1 out of bounds
 for type 'int [13]'
```

Листинг 3: Пример программы и диагностики UndefinedBehaviorSanitizer.

`shift` – некорректные операнды операторов сдвига, например, когда величина сдвига отрицательна или превышает разрядность сдвигаемого значения.

`alignment` – использование невыровненного значения указателя или создание невыровненной ссылки. Проверяется также выравнивание значений в соответствии с атрибутами `assume_aligned` и `align_value`, а также выравнивание в соответствии с параметром aligned в директивах OpenMP.

`bool` – проверка считываемых из памяти значений типа `bool`. Ошибка диагностируется, если значение не является ни `true`, ни `false`.

`enum` – проверка считываемых из памяти значений типа `enum`. Полезность этой проверки ограничена, поскольку проверяется лишь то, что значение находится в диапазоне разрядности, со-

ответствующей данному типу. Например, если тип содержит значения 1, 5, 6, 7, 99, то ошибкой будет значение вне диапазона 0..127.

`float-divide-by-zero`, `integer-divide-by-zero` – вещественное и целочисленное деление на ноль.

`null` – разыменование нулевого указателя, создание нулевой ссылки. Проверка указателей, отмеченных атрибутами `nonnul` (аргументы функций), `returns_nonnul` (возвращаемое значение функции).

`object-size` – попытка использовать байты, не являющиеся частью объекта, к которому осуществляется доступ. Выявление различных видов ошибок доступа к объектам по указателям. Например, ошибочное приведение к типу-наследнику или вызов методов по некорректному указателю.

`return` – в программах на C++ достижение конца функции, возвращающей значение, без возврата значения.

`unreachable` – достижение вызова встроенной функции `__builtin_unreachable()`, что является неопределенным поведением. Вызов указанной функции заменяется на вызов диагностического сообщения.

`vla-bound` – создание массива переменного размера, где размер не является положительным значением.

`vptr` – проверка указателей на таблицу виртуальных функций. Проверяются ситуации использования объектов с некорректным динамическим типом, а также объектов, время жизни которых уже закончилось или еще не началось.

Только в clang (но не в gcc) поддерживаются следующие проверки.

`function` – косвенный вызов функции не соответствующего типа по указателю (только для C++ на платформах x86/x86_64 под Darwin/Linux).

`builtin` – передача некорректных аргументов встроенным функциям компилятора.

`pointer-overflow` – арифметические действия над указателями, приводящие к переполнению.

Clang поддерживает, в дополнение к перечисленным выше, проверки ряда ситуаций, которые не относятся к категории неопределенного поведения, но зачастую не соответствуют ожиданиям программиста:

`unsigned-integer-overflow` – переполнение в операциях беззнаковой целочисленной арифметики.

`implicit-unsigned-integer-truncation`, `implicit-signed-integer-truncation` – неявное преобразование целочисленного значения к целочисленному типу с меньшей разрядностью с потерей данных.

`implicit-integer-sign-change` – неявное преобразование между целочисленными типами, при котором происходит смена знака.

`nullability-assign`, `nullability-arg`, `nullability-return` – проверка указателей, отмеченных спецификатором `_Nonnull`.

В отличие от других рассмотренных выше санитайзеров, UndefinedBehaviorSanitizer может быть применен на любой платформе, поддерживаемой компилятором, если задан ограниченный режим, не требующий использования библиотеки времени выполнения `libubsan`. Ограниченный режим активируется ключом `-fsanitize-undefined-trap-on-error` и подразумевает, что при первой же ошибке выполнение программы будет аварийно завершено вызовом встроенной функции `__builtin_trap` без выдачи диагностического сообщения от санитайзера.

2.5. Настройки санитайзеров

Эксперименты показывают, что наиболее точную информацию о локализации ошибки можно получить при компиляции без оптимизации (хотя в документации рекомендуется использовать `-O1 -fno-optimize-sibling-calls -fno-omit-frame-pointer`). Опция отладки `-g` нужна для отображения места возникновения ошибки с указанием имен файлов и номеров строк исходного кода.

Дополнительное управление работой санитайзеров осуществляется при помощи переменных окружения `ASAN_OPTIONS`, `MSAN_OPTIONS`, `TSAN_OPTIONS`, `LSAN_OPTIONS`, `UBSAN_OPTIONS`. В частности, с их помощью можно подавлять известные ошибки. Например: `UBSAN_OPTIONS=suppressions=`*файл*, где *файл* содержит список директив, специфицирующих список игнорируемых ошибок:

```
signed-integer-overflow:module.cpp
alignment:function
vptr:shared_object.so
```

Поддерживается также атрибут функций, позволяющий отменять для них заданные проверки, например, `__attribute__((no_sanitize("null")))`. Это полезно для игнорирования известных ошибок, для функций, выполняющих низкоуровневые манипуляции с системными данными или повышения производительности заведомо корректно реализованных функций.

В clang (но не в gcc) отмена проверок на стадии компиляции поддерживается при помощи ключа `-fsanitize-blacklist=`*файл*, где *файл* – имя

файла с директивами отмены проверок в заданных файлах или функциях. Еще одна полезная возможность в clang – условная компиляция в зависимости от использования санитайзеров. Она управляется директивами препроцессора вида #if `__has_feature`(*санитайзер*).

3. САНИТАЙЗЕРЫ И ФАЗЗИНГ-ТЕСТИРОВАНИЕ

Фаззингом (англ. fuzzing) называется метод тестирования программ, как правило, автоматизированного, с использованием случайных входных данных, возможно неправильных и не ожидаемых тестируемой программой. Фаззинг применяется для тестирования программ или библиотек, работающих с достаточно сложными по структуре входными данными, такими как программы сериализации/десериализации, программы сжатия/разжатия данных, медиа кодеки, криптографические программы, текстовые процессоры, компиляторы, ассемблеры, интерпретаторы и другие.

Растущий интерес к фаззинг-тестированию вызван в значительной степени повышением требований к безопасности ПО. В частности, это относится к применению фаззинга в сочетании с санитайзерами или другими средствами динамического анализа программ. В ряде компаний фаззинг-тестирование является обязательным звеном в цикле разработки ПО, имеющего требования по безопасности. Известно, что хакеры также применяют фаззинг для выявления еще неизвестных им уязвимостей программ. Соответственно, производители ПО, подвергнув свои разработки фаззингу, должны избавиться от ошибок безопасности до того, как у хакеров появится шанс воспользоваться ими [12].

Обстоятельный обзор методов и систем фаззинг-тестирования с обширной библиографией представлен в [13]. Введением в фаззинг-тестирование, хотя и несколько устаревшим, может послужить вышедшая в 2009 году книга на русском языке [14]. Не претендуя на полноту, мы лишь кратко остановимся здесь на основных признаках, по которым обычно классифицируют системы фаззинг-тестирования.

Степень использования информации о тестируемой программе. По этому признаку выделяются 3 категории систем. К первой категории относятся системы, тестирующие целевую программу по принципу “белого ящика” и требующие полной информации о ней, включая исходные тексты и спецификации различных аспектов ее функционирования. Ко второй категории относятся системы, не имеющие никакой информации о тестируемой программе (кроме способа ее запуска). Промежуточное положение занимают системы, использующие частичную информацию о тестируемой программе, которую они могут собирать динамически в ходе выполнения программы (например, покрытие кода) или получать путем статического анализа ее исходного кода.

Объекты, варьируемые при фаззинг-тестировании. Это могут быть аргументы командной строки, значения переменных окружения, входные файлы, содержимое области оперативной памяти, пакеты сетевых протоколов, данные, вводимые интерактивно, и другие.

Степень автоматизации. Система фаззинг-тестирования как минимум должна автоматически создавать входные данные (Csmith [16]). Далее, она может автоматически в цикле запускать тестируемую программу, анализировать результат запуска и сохранять входные данные, которые приводят к ошибкам (AFL [17], libFuzzer [19]). Дополнительный сервис может включать минимизацию набора входных данных, на котором воспроизводится ошибка, автоматическую генерацию и отправку отчета об ошибке с проверкой того, что эта ошибка не была зафиксирована ранее. Пример среды фаззинг-тестирования с полным циклом автоматизации – общедоступный сервис OSS-fuzz [18] для тестирования проектов с открытыми исходными текстами на серверах Google.

Способ порождения входных данных. Фаззеры можно разделить на две группы: генерирующие и мутационные. Первые генерируют каждый очередной тест “с нуля”. Они могут использовать набор правил или грамматику, описывающую синтаксис входных данных. Тесты могут порождаться как строго по правилам, так и с отклонениями от них. Такой подход может быть хорош для тестирования компиляторов, интерпретаторов, ассемблеров и других подобных программ (Csmith). Мутационные фаззеры порождают новые тестовые данные путем мутации тестов из заданного набора. К этому классу относятся фаззеры AFL, libFuzzer. Пример фаззера, поддерживающего оба способа – PEACH [20].

Стратегия порождения данных. Хорошую результативность показали появившиеся в последние годы фаззеры, использующие генетические алгоритмы, управляемые покрытием кода тестируемой программы (coverage-guided fuzzing). Принцип их действия в общем виде описывается следующим алгоритмом.

```
Скомпилировать тестируемую программу с
  инструментацией для измерения покрытия кода.
Сформировать начальный набор тестов ("корпус").
Цикл: {
  Создать новый тест путем случайной мутации
    теста из корпуса
  Выполнить новый тест с измерением покрытия.
  Если новый тест увеличивает суммарное покрытие,
    то добавить его к корпусу.
  }
```

На этом принципе основаны системы AFL, libFuzzer, gofuzz [21] и другие. Покрытие кода здесь детализируется с учетом порядка и количества прохождения линейных участков. Например, в AFL различаются покрытия $A \to B \to C \to D \to E$ и $A \to B \to D \to C \to E$; различаются также покрытия с разным числом счетчиков выполнения блоков с точностью до нескольких диапазонов: 1, 2, 3, 4–7, 8–15, 16–31, 32–127, 128+. В libFuzzer помимо этого поддерживается фаззинг, управляемый потоком данных (data-flow-guided fuzzing): инструментирование операций сравнения во время компиляции и целенаправленные мутации входных данных для изменения результата сравнения. Это позволяет быстрее достигать новых покрытий кода за счет изменения направления переходов, зависящих от результатов сравнений.

Рассмотрим подробнее преимущества фаззинга в сравнении с традиционными видами тестирования. Например, при модульном тестировании обычно также стремятся достичь максимального полного покрытия кода тестами. Однако фаззинг-тестирование способно улучшить покрытие кода, понимаемое в более широком смысле, чем просто число строк кода в процентах. Помимо собственно исходного кода есть данные, обрабатываемые этим кодом, например, значения индексов в обращениях к массивам. Возможны также многочисленные вариации потока управления: например, для того чтобы воспроизвести ошибку, может быть необходимо зайти в `then`-часть одного оператора `if`, но не зайти в `then`-часть другого, затем выполнить такой-то цикл определенное число раз и т.д. Еще один важный пример, иллюстрирующий расширенное понимание термина “покрытие кода” – накопленное состояние программы, от которого может зависеть ход ее выполнения. Преимуществом фаззинг-тестирования является его способность обеспечить множество различных вариаций покрытия кода.

Другая сильная сторона фаззинга – беспристрастность. Программист-тестировщик может быть подвержен различным иллюзиям, например, иллюзии “простоты” и “очевидности” тестируемых модулей. Или он может положиться на общепризнанную высокую квалификацию программиста-разработчика ПО, и так далее. Фаззинг-тестирование исключает влияние подобных факторов.

Немаловажным преимуществом фаззинг-тестирования является дешевизна его использования в силу высокой степени автоматизации.

Наконец, отметим преимущества совместного применения фаззинга и санитайзеров. Санитайзеры способны выявлять лишь те ошибки, которые произошли при данном пуске программы. Фаззинг позволяет получить максимальный эффект от применения санитайзеров, предоставляя обширную тестовую базу, включающую как корректные, так и некорректные данные, не ожидаемые тестируемой программой. Применение санитайзеров, в свою очередь, позволяет повысить результативность фаззинг-тестирования за счет расширения класса фиксируемых ошибок.

К слабым сторонам фаззинг-тестирования можно отнести то, что оно позволяет, как правило, фиксировать лишь ошибки, приводящие к аварийному завершению (ошибки сегментирования, вызовы assert(), неперехватываемые исключения C++ и другие) либо к зависанию программы. Но, поскольку используются случайные входные данные, обычно невозможно бывает проверить корректность вычисленных результатов. Тем не менее, выявление логических ошибок в тестируемых программах при фаззинг-тестировании вполне возможно. Способы зависят от специфики тестируемых программ, и требуют творческого подхода. Простой пример – выполнение прямой и обратной функции (сжатие-расжатие, шифрование-дешифрование и т. п.) с проверкой того, что результат совпадает с исходными данными: $f^{-1}(f(x)) = x$. Другие методы обсуждаются, например, в [15].

В заключение рассмотрим пример использования системы AFL для тестирования утилиты gawk-5.0.0, скомпилированной при помощи компилятора clang с применением MemorySanitizer. Процедура фаззинг-тестирования при помощи AFL достаточно проста. После сборки са-

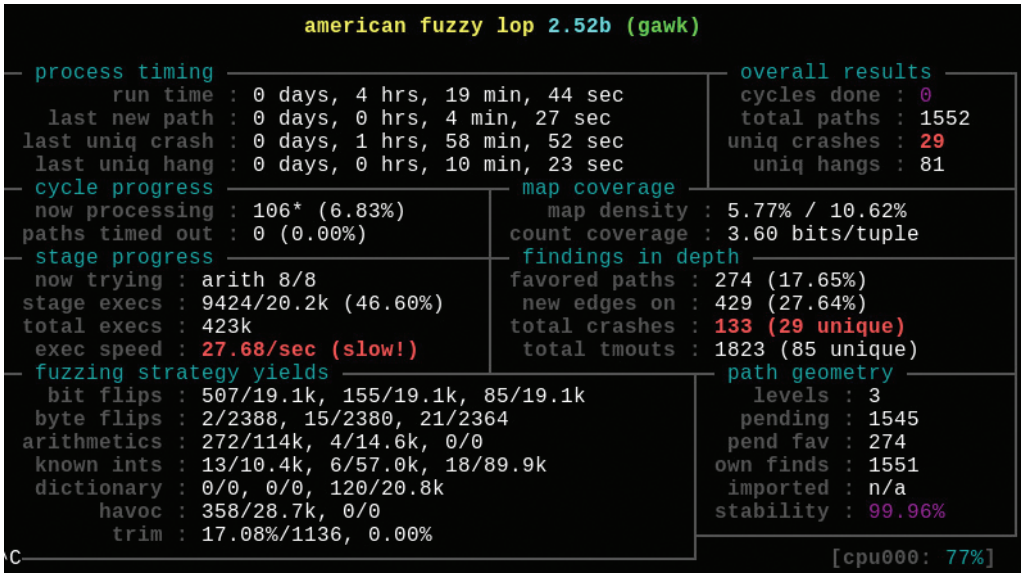

Рис. 2. Экран состояния фаззинг-тестирования при помощи AFL.

мого фаззера согласно описанию нужно определенным образом собрать тестируемое приложение. В данном случае сборка gawk проводилась с использованием переменных окружения `CFLAGS="-g -O2 -fsanitize=memory"`, `CC=~/AFL_PROJECTS/afl-2.52b/afl-clang-fast`. Здесь `afl-clang-fast` – обертка для вызова компилятора clang с определенными ключами, в частности, с ключами сбора тестового покрытия.

Затем необходимо создать подкаталог с корпусом входных данных. Мы поместили в этот каталог одну awk-программу из набора тестов gawk-5.0.0. Запуск фаззера AFL:

```
AFL_USE_MSAN=1 ~/AFL_PROJECTS/afl-2.52b/afl-fuzz -m none -i ./inputs \-o ./out ~/AFL_PROJECTS/local/bin/gawk -f @@ ./fpat1.in
```

Здесь `-m none` означает отсутствие ограничений по использованию памяти, `-i ./inputs` задает каталог с начальным корпусом входных данных, `-o ./out` задает выходной каталог, где AFL будет накапливать корпус сгенерированных входных данных и сохранять файлы данных, вызвавшие аварийное завершение или зависание тестируемой программы. Далее следует имя тестируемой программы и ее аргументы. Опция `-f` задает файл с awk-программой, а символы `@@` обозначают место подстановки имени очередного сгенерированного фаззером файла. Последний аргумент `./fpat1.in` – файл с данными, обрабатываемыми утилитой gawk. Этот файл также взят из набора тестов gawk-5.0.0 и он в ходе тестирования изменяться не будет.

AFL поддерживает фаззинг только одного входного файла, то есть в данном случае можно применить фаззинг к файлу с awk-программой либо к файлу с обрабатываемыми данными, но не к тому и другому сразу.

Фаззинг-тестирование будет продолжаться, пока вы не нажмете `Ctrl-C`. На экране будет отображаться таблица с текущим состоянием тестирования (рис. 2).

В правой верхней части таблицы показано число случаев аварийного завершения (`uniq crashes: 29`) и зависания (`uniq hangs: 81`). Наборы данных для воспроизведения этих ситуаций сохраняются в подкаталогах `out/crashes` и `out/hangs`. Запуск `gawk` с awk-программи, сохраненными в `out/crashes` показал, что все они вызывают одну и ту же ошибку использования неинициализированных данных в лексическом анализаторе. Эта ошибка была исправлена разработчиками программы `gawk`.

4. САНИТАЙЗЕРЫ ЦЕЛОСТНОСТИ ПОТОКА УПРАВЛЕНИЯ

В разделе 2 были рассмотрены инструменты динамического анализа, предназначенные для тестирования ПО. Настоящий раздел посвящен реализованным в компиляторах clang и gcc средствам динамического анализа, предназначенным для обеспечения безопасности выполнения приложений в производственном режиме. Эти средства позволяют исключить или существенно затруднить эксплуатацию уязвимостей, возможно присутствующих в программе. Санитайзеры этого вида могут применяться и на стадии тестирования ПО, но важно понимать, что их срабатывание не всегда соответствует месту возникновения ошибки. Их основное назначение – предотвра-

тить опасные последствия ошибки, а не указать ее местоположение и характер.

Наличие в программе косвенных переходов создает потенциальную возможность выполнить переход на произвольный адрес. Если злоумышленник сумеет модифицировать значение адреса, по которому осуществляется переход, то он сможет использовать существующий код для своих целей. Термин "целостность потока управления" (Control Flow Integrity – CFI) обозначает совокупность методов безопасности, направленных на то, чтобы ограничить возможные пути исполнения программы в рамках графа потока управления, определяемого семантикой программы [22].

Для обеспечения широкого применения CFI важно, чтобы соответствующие механизмы защиты были интегрированы непосредственно в промышленные компиляторы и были совместимы с другими технологиями разработки, такими как разделяемые библиотеки и инкрементальная компиляция. Поскольку средства CFI должны встраиваться в промышленно поставляемое ПО, накладные расходы по памяти и производительности должны быть приемлемыми.

Принято выделять прямые и обратные косвенные переходы. Прямые переходы происходят, например, при вызовах функций по указателю или при вызовах виртуальных методов в языке C++. На графе потока управления они обозначаются прямыми дугами (forward edges). Обратные переходы соответствуют возвратам из функций; они обозначаются обратными дугами (backward edges). Далее будут рассмотрены средства защиты прямого и обратного потоков управления, поддерживаемые компиляторами clang и gcc.

4.1. Защита обратного потока управления

Для защиты обратного потока управления, то есть адресов возврата из функций, в gcc поддерживается ключ `-fstack-protector`, затрудняющий эксплуатацию уязвимостей в стеке (stack smashing). В стек записывается дополнительная переменная-маркер, отделяющая локальные данные функции от сохраненных значений регистров. Перед возвратом из функции значение маркера сравнивается с эталонным и при несовпадении происходит аварийное завершение программы с выдачей диагностики.

Опция `-fstack-protector` применяется только к уязвимым функциям, а именно, к функциям, содержащим вызовы alloca или буферы размером более 8 байт (этот размер буферов регулируется параметром компилятора `ssp-buffer-size`). Поддерживается также тотальная защита всех функций (`-fstack-protector-all`), но она может приводить к ощутимой деградации производительности. Начиная с версии 4.9 в gcc реализован усиленный режим защиты `-fstack-protector-strong`), который защищает функции, содержащие любые локальные массивы, даже внутри структур или объединений, или использующие адреса локальных переменных как аргументы функций либо в правой части присваиваний. Это обеспечивает более сильную защиту в сравнении с `-fstack-protector` без чрезмерной потери производительности.

Компилятор gcc поддерживает также режим выборочной защиты (`-fstack-protector-explicit`), применяемой только к функциям с атрибутом `stack_protect`. Этот атрибут действует и при наличии любой из перечисленных выше опций защиты стека. Таким образом, компилятор предоставляет гибкие возможности для настройки защиты адресов возврата в стеке.

Заметим, что компилятор gcc для ОС Ubuntu использовал `-fstack-protector` по умолчанию с момента реализации данной опции; в компиляторе, поставляемом с последними версиями ОС Ubuntu, по умолчанию действует `-fstack-protector-strong` и `ssp-buffer-size=4`.

Описанный способ, хотя и охватывает большинство уязвимостей переполнения буферов в стеке, встречающихся на практике, не обеспечивает абсолютной защиты. Он защищает при переполнениях в результате циклической записи в непрерывный диапазон адресов, но в других ситуациях может не сработать.

В clang, в дополнение к описанному выше методу, поддерживается санитайзер безопасного стека (ключ `-fsanitize=safe-stack`), являющийся частью проекта CPI (Code Pointer Integrity) [23]. Идея метода заключается в том, что приложение использует два стека вместо одного: безопасный и небезопасный. В безопасном стеке хранятся данные, доступ к которым не может привести к перезаписи других значений в стеке: адрес возврата, регистры, вытолкнутые в память, скалярные локальные переменные. В небезопасном стеке размещается все остальное, в частности, локальные массивы и переменные, от которых берутся указатели. Наборы переменных, сохраняемых в каждом из стеков, определяются путем статического анализа кода. Для защиты безопасного стека применяются различные методы изоляции памяти, которые могут быть специфическими для разных архитектур, см. [23].

Накладные расходы, связанные с использованием санитайзера `safe-stack`, значительно ниже, чем для метода `-fstack-protector`, и составляют не более 0.1%. Это связано с тем, что дополнительный, небезопасный, стек требуется в среднем лишь примерно для 25% функций. Иногда использование второго стека приводит даже к ускорению программы за счет более эффективного кеширования данных: поскольку массивы пе-

реносятся в небезопасный стек, то часто используемые небольшие локальные переменные располагаются более компактно. Еще одно преимущество санитайзера `safe-stack` в сравнении с опцией `-fstack-protector` заключается в том, что он не приводит к аварийным завершениям приложений. Правда, `safe-stack` не всегда применим, поскольку он не поддерживается для разделяемых библиотек и реализован не для всех операционных систем.

Упомянем также санитайзер теневого стека (`-fsanitize=shadow-call-stack`), поддерживаемый компилятором clang. Метод защиты, реализуемый этим санитайзером, заключается в том, что адреса возврата из функций сохраняются в отдельном, теневом стеке и, следовательно, не могут быть переписаны в ситуации переполнения буфера в стеке. Для совместимости с существующими ABI, адрес возврата размещается и в обычном стеке, но его значение там не используется. Санитайзер теневого стека задуман как более сильный вариант защиты адреса возврата в сравнении с методом `-fstack-protector`, поскольку обеспечивает защиту от произвольных, а не только циклических, записей в буфер. Однако в настоящее время он реализован только для архитектуры aarch64. Недостатком этого метода в сравнении с санитайзером `safe-stack` является то, что он защищает только адреса возврата, в то время как `safe-stack` защищает все данные, хранимые в безопасном стеке.

4.2. Защита прямого (восходящего) потока управления

Способы защиты адресов возврата из функций и другой критической информации, размещаемой в стеке, разработаны и поддерживаются в компиляторах довольно давно. В связи с этим злоумышленники переключились на создание альтернативных подходов к эксплуатации уязвимостей с задействованием восходящего потока управления. Например, они могут попытаться перезаписать хранящиеся в динамической памяти указатели на функции или таблицы виртуальных функций. Это может стать возможным при наличии в программе таких уязвимостей, как переполнение буфера либо использование объекта после освобождения. Для противодействия угрозам такого рода в gcc и clang были реализованы механизмы динамического анализа, предотвращающие нарушения целостности восходящего потока управления [24].

В gcc при компиляции программ на C++ поддерживается опция `-fvtable-verify=`, обеспечивающая для каждого виртуального вызова проверку того, что используемая таблица виртуальных методов (virtual method table, VMT) соответствует типу объекта, для которого делается вызов, и что эта таблица не была испорчена или перезаписана. Если в результате такой проверки выявлен некорректный указатель VMT, то выдается диагностическое сообщение и выполнение программы аварийно завершается. Срабатывание может происходить также в результате некорректного приведения типов в программе.

При использовании этой опции перед каждым вызовом виртуального метода вставляется вызов функции, которая проверяет корректность указателя на VMT. Эти проверочные функции используют служебные vtable-map-переменные, указывающие на наборы допустимых VMT для каждого полиморфного класса. Наборы допустимых указателей формируются всегда до входа в функцию main. Опция `-fvtable-verify=` имеет аргумент, который уточняет, когда именно происходит формирование этих наборов: до загрузки и инициализации разделяемых библиотек (аргумент `preinit`) или после (аргумент `std`).

Для корректной верификации необходимо, чтобы весь проект был скомпилирован с ключом `-fvtable-verify=`, иначе наборы допустимых указателей на VMT могут оказаться неполными, что приведет к ложным срабатываниям. В [24] описывается подход, позволяющий обойти эту проблему, если, например, в проекте используются сторонние библиотеки, поставляемые без исходных текстов.

Другое затруднение, с которым вы можете столкнуться, попытавшись воспользоваться этой функциональностью, заключается в том, что предустановленный в системе компилятор, скорее всего, не поддерживает ее. Для этого при конфигурировании gcc должна быть указана опция `--enable-vtable-verify`, которая по умолчанию не активна в силу следующих причин. Обозначим для краткости через vtv-gcc компилятор, сконфигурированный с опцией `--enable-vtable-verify`. Стандартная библиотека C++ в vtv-gcc собирается с ключом `-fvtable-verify=`, чтобы обеспечить корректность верификации. Программа на C++, компилируемая при помощи vtv-gcc даже без `-fvtable-verify=`, все равно будет работать медленнее, чем при сборке стандартным gcc, так как функции библиотеки C++ содержат вызовы верификации. Хотя в этом случае вызываются лишь заглушки, эти дополнительные вызовы замедляют выполнение программ. Для тестов на C++ из SPEC CPU2006, согласно [24], замедление составляет до 4,7%. Но, как показывают эксперименты, замедление может быть и более значительным. Поэтому компилятор vtv-gcc целесообразно применять только для сборки ПО, использующего динамическую верификацию виртуальных вызовов, а в остальных случаях пользоваться стандартным gcc.

В компиляторе clang санитайзер целостности потока управления доступен начиная с версии 3.7. Поддерживается несколько схем динамического анализа, которые могут быть включены по отдельности либо все вместе при помощи ключа `-fsanitize=cfi`.

В clang реализация этой функциональности основывается на доступности графа потока управления для всей программы. Поскольку обычно программа собирается из множества модулей, полный граф становится доступен только на стадии компоновки. Поэтому вместе с `-fsanitize-cfi` необходимо использовать ключ `-flto` для включения оптимизаций времени компоновки (Link Time Optimization, LTO). В рамках прохода LTO выполняется анализ программы и ее трансформация с генерацией заданных видов динамических проверок.

Рассмотрим теперь возможности различных динамических проверок CFI. Опция `-fsanitize=cfi-icall` включает верификацию косвенных вызовов функций. Для каждого косвенного вызова функции по указателю добавляется проверка двух условий: (1) адрес вызова соответствует началу некоторой функции в программе и (2) сигнатура вызываемой функции соответствует сигнатуре указуемой функции, определенной во время компиляции. Данная проверка реализована только для платформ x86 и x86_64.

Упомянутые условия могут нарушаться при эксплуатации уязвимостей, связанных с перезаписью содержимого памяти. Злоумышленник таким образом может попытаться передать управление на фрагменты существующего в программе кода, который реализует нужные ему действия. Эти фрагменты кода (называемые гаджетами) часто не соответствуют началу какой-либо функции; такая подмена указателя не пройдет проверку `cfi-icall`. Не сработает также попытка передать управление на функцию с несоответствующей сигнатурой. Однако эта проверка не спасет от подмены корректного указателя на указатель функции с такой же сигнатурой (например, `delete_user(const char *user)` на `make_admin(const char *user)`).

Опция `-fsanitize=cfi-mfcall` активирует верификацию косвенных вызовов по указателю на метод класса. Проверяется, что метод применяется к объекту подходящего динамического типа и что указуемая функция имеет соответствующий тип.

Опция `-fsanitize=cfi-vcall` включает верификацию вызовов виртуальных методов. Виртуальные методы класса могут быть специализированы в его производных классах, и для них применяется динамическое связывание, то есть конкретный метод определяется во время выполнения в зависимости от типа объекта. Поэтому виртуальные вызовы реализуются как косвенные. При задании `cfi-vcall` проверяется, что вызываемый метод относится к классу из иерархии базовых для объекта, к которому он применяется. Эта проверка выявляет, в частности, ошибки несоответствия типов (type confusion), являющиеся уязвимостями, типичными для программ со сложными иерархиями классов.

Опция `-fsanitize=cfi-nvcall` защищает от вызовов невиртуальных методов для объектов, не относящихся к классам, для которых данные методы были определены. Эта опция подобна `cfi-vcall`, но применяется к невиртуальным вызовам. Поскольку адреса невиртуальных методов известны во время компиляции, то, строго говоря, данный механизм защиты не имеет отношения к CFI. Для каждого невиртуального вызова осуществляется динамическая проверка типа объекта, к которому применяется метод. Динамический тип объекта должен быть производным от типа, известного во время компиляции, или совпадать с ним.

Срабатывания этой проверки могут возникать в результате перезаписи содержимого памяти, ошибок несоответствия типов или ошибок десериализации. Перенаправления потока управления при этом не происходит; опасность заключается в том, что метод может быть применен к данным, для которых он не предназначен.

Опции `-fsanitize=cfi-unrelated-cast`, `-fsanitize=cfi-derived-cast` позволяют динамически проверять и отвергать некорректные приведения типов объектов. Эти проверки также не связаны с целостностью потока управления, а направлены на предотвращение эксплуатации ошибок несоответствия типов. Причинами их срабатывания могут быть также порча содержимого памяти, ошибки десериализации.

Опция `cfi-unrelated-cast` отвергает приведение типов между объектами, типы которых не связаны друг с другом. Такие ошибки часто возникают из-за того, что адреса объектов передаются между разными частями программы как указатели типа `void *`. При приведении от типа `void *` к типу класса будет проверяться, что объект действительно имеет указанный тип. При приведении от одного типа класса к другому проверяется, что эти типы связаны отношением наследования. Опция `cfi-derived-cast` запрещает приведение от базового типа к производному, если объект в действительности не имеет указанный производный тип.

Проверка `cfi-derived-cast` не отвергает приведение от базового типа к производному типу, если производный класс имеет единственный базовый, не вводит своих виртуальных методов и не переопределяет никаких виртуальных методов, за исключением виртуального деструктора.

В этом случае раскладка памяти объектов совпадает, и проблем безопасности не возникает. С точки зрения стандарта языка C++, такое приведение является неопределенным поведением, но этот прием используется во многих проектах. Для того чтобы подобные приведения типов отвергались, необходимо дополнительно использовать опцию `-fsanitize=cfi-cast-strict`.

В заключение рассмотрим пример программы, нарушающей условия проверки `-fsanitize=cfi-nvcall`, который показан на листинге 4. Это сокращенный вариант примера из публикации [25], где можно найти примеры срабатывания и других проверок CFI.

```
01 #include <iostream>
02 #include <string>
03
04 struct Account {
05   Account(const std::string &s) : name(s) {}
06   virtual ~Account() {}
07   void showName() {
08     std::cout << "Account name is: "
09               << name << std::endl;
10   }
11   void adminStuff() { std::cout
12         << "Not Implemented" << std::endl; }
13   std::string name;
14 };
15 struct UserAccount : Account {
16   UserAccount(const std::string &s) : Account(s) {}
17   virtual ~UserAccount() {}
18   void adminStuff() {
19     std::cout
20     << "Admin Work not permitted for a user account!"
21     << std::endl;
22   }
23 };
24 struct AdminAccount : Account {
25   AdminAccount(const std::string &s) : Account(s) {}
26   virtual ~AdminAccount() {}
27   void adminStuff() {
28     std::cout << "Would do admin work in context of: "
29      << this->name << std::endl;
30   }
31 };
32 int main(int argc, const char *argv[]) {
33   UserAccount* user = new UserAccount("user");
34   AdminAccount* admin = new AdminAccount("admin");
35   admin->showName();
36   admin->adminStuff();
37   user->showName();
38   user->adminStuff();
39
```

```
40   Account *account = static_cast<Account*>(user);
41   AdminAccount *admin_it =
42        static_cast<AdminAccount*>(account);
43   admin_it->showName();
44   std::cout << "CFI Should prevent the actions below:"
45             << std::endl;
46   admin_it->adminStuff();
47   return 0;
48 }
```

Листинг 4: Пример программы нарушающей условия проверки `-fsanitise=cfi-nvcall`.

На листинге 5 показаны выдачи этой программы, скомпилированной без `-fsanitise=cfi-nvcall`, и той же программы, скопилированной с `-fsanitise=cfi-nvcall`. Строки 40–42 программы эмулируют ситуацию подмены данных. В результате по указателю `AdminAccount *admin_it` оказывается объект типа `UserAccount`. Ошибка обнаруживается при вызове метода `adminStuff()` в строке 46.

```
$ ./no-cfi-nvcall
Account name is: admin
Would do admin work in context of: admin
Account name is: user
Admin Work not permitted for a user account!
Account name is: user
CFI Should prevent the actions below:
Would do admin work in context of: user
$ ./cfi-nvcall
Account name is: admin
Would do admin work in context of: admin
Account name is: user
Admin Work not permitted for a user account!
Account name is: user
CFI Should prevent the actions below:
nvcall.cpp:46:3: runtime error: control flow integrity
check for type 'AdminAccount' failed during non-virtual call
(vtable address 0x000000437c00)
0x000000437c00: note: vtable is of type 'UserAccount'
  00 00 00 00 90 f0 42 00 00 00 00 00 10 f1 42 00
  00 00 00 00 00 00 00 00 00 00 00 00 88 7b 43 00
```

Листинг 5: Выдача программы из листинга 4

5. ЗАКЛЮЧЕНИЕ

В условиях усиливающихся требований безопасности, предъявляемых к ПО, роста объемов разрабатываемых проектов и необходимости постоянно сокращать сроки разработки и выпуска новых версий возникла настоятельная потребность в инструментах динамического анализа, которые бы позволяли эффективно выявлять уязвимости программ на языках C и C++ и могли применяться на регулярной основе в цикле разработки. Санитайзеры AddressSanitizer, MemorySanitizer, ThreadSanitizer, UndefinedBehaviorSanitizer, реализованные в компиляторе clang, а в дальнейшем вошедшие и в gcc, в значительной мере восполнили этот пробел. Эти средства позволяют обнаруживать многие критические с точки зрения безопасности классы ошибок. Инструменты фаззинг-тестирования, рассмотренные в разделе 3, автоматизируют процесс создания тестов и позволяют многократно повысить эффект применения санитайзеров за счет более полного

покрытия множества возможных путей выполнения программы. Сервис OSS-fuzz (доступный, правда, только для проектов с открытыми исходными текстами) добавляет еще один уровень автоматизации, обеспечивая генерацию отчетов об ошибках и минимизацию тестовых данных для воспроизведения ошибки. Можно ожидать, что следующим шагом на пути автоматизации устранения уязвимостей в программах станет автоматическое исправление типичных ошибок ([26]).

Еще одну линию защиты предоставляют санитайзеры целостности потока управления. Их применение в сочетании с системными средствами защиты, такими как рандомизация размещения адресного пространства (ASLR), предотвращение выполнения кода, находящегося в сегментах данных (DEP), существенно затрудняет эксплуатацию уязвимостей программ. Дополнительную защиту могут обеспечить методы диверсификации программного кода [27].

Рассмотренные в работе санитайзеры позволяют значительно повысить безопасность ПО. В то же время, остается ряд нерешенных задач, относящихся к доступности, простоте использования и полноте существующих средств динамического анализа.

Под *доступностью* понимается набор платформ, для которых поддерживаются рассмотренные средства динамического анализа. В компиляторах gcc и clang они реализованы для ряда наиболее употребительных операционных систем общего назначения, таких как ОС Linux, Android, MacOS, MS Windows. Несомненно, подобные средства были бы полезны и при разработке ПО для встроенных и бортовых систем, к которому предъявляются повышенные требования надежности и безопасности. Портирование средств динамического анализа на платформы, используемые во встроенных системах, затрудняется ограничениями по ресурсам, особенностями функционирования ОС реального времени и, возможно, спецификой процесса разработки ПО для этих систем. В связи с этим интерес представляют работы [29], где рассмотрены вопросы портирования AddressSanitizer на платформу Myriad, и [28], где представлен опыт портирования санитайзеров AddressSanitizer, MemorySanitizer, UndefinedBehaviorSanitizer на платформы под управлением ОС реального времени JetOS. В [28] отмечается также, что подобные инструменты могут быть дополнены средствами проверки требований контрактов для кода при сертификации ПО.

С точки зрения *простоты использования*, положительными характеристиками рассмотренных санитайзеров является их доступность непосредственно в компиляторах gcc и clang и, в большинстве случаев, приемлемые накладные расходы. Наличие различных ограничений может усложнять применение санитайзеров. Например, не все они в полной мере поддерживают разделяемые библиотеки. Неудобства создает также невозможность одновременного применения нескольких санитайзеров, в частности, AddressSanitizer, MemorySanitizer, ThreadSanitizer. Из-за этого тестирование с каждым из них должно проводиться раздельно. Согласно [31], невозможность совместного применения – общая беда многих санитайзеров, связанная с тем, что для анализа им требуются различные несовместимые метаданные.

Для использования MemorySanitizer, ThreadSanitizer необходимо, чтобы все приложение, включая стандартные библиотеки, было скомпилировано с соответствующей опцией. Желательно, чтобы компилятор включал версии библиотек C++, подходящие для применения с этими санитайзерами и обеспечивал их автоматическое подключение при компоновке.

Важный фактор, снижающий популярность санитайзеров среди разработчиков, согласно исследованию [31], – наличие ложноположительных срабатываний. Например, при использовании MemorySanitizer причиной ложноположительных срабатываний может стать невозможность инструментировать весь проект, если он содержит внешние библиотеки. Опыт применения UndefinedBehaviorSanitizer, согласно [31], также показывает значительное число ложных срабатываний. Возможность ложноотрицательных срабатываний, согласно тому же источнику, в меньшей степени влияет на популярность инструмента.

Полнота функциональности. Не менее важная проблема заключается в том, что существующие санитайзеры охватывают далеко не все случаи неопределенного поведения программ на языках C и C++. Например, AddressSanitizer обнаруживает не все ошибки адресации. В некоторых случаях обнаружение ошибки санитайзером зависит от заданного уровня оптимизации. Примеры подобных ситуаций, а также полезные рекомендации по использованию санитайзеров и других способов повышения безопасности программ обсуждаются в публикации [30].

Далее, существующие санитайзеры, обеспечивающие безопасное выполнения программ в производственном режиме, ориентированы в основном на поддержание целостности потока управления. Но, во-первых, они не гарантируют абсолютной целостности потока управления, а,во-вторых, существуют атаки, реализуюемые путем подмены данных (data-only attacks). По этой причине некоторые производители ПО пытаются использовать AddressSanitizer в промышленных релизах, что вряд ли можно считать приемлемым решением из-за высоких накладных расходов. К тому же злоумышленник, зная принципы работы AddressSanitizer, может обмануть за-

щиту. В связи с этим сейчас активно ведутся исследования по созданию санитайзера, основанного на аппаратной технологии теггирования памяти (memory tagging, MT) и указателей, [33], [34]. В сравнении с AddressSanitizer этот подход требует существенно меньших накладных расходов, а предоставляемую им защиту труднее обойти. Поэтому MT-санитайзер может применяться не только при обычном и фаззинг-тестировании, но и для противодействия эксплуатации ошибок. Правда, аппаратная поддержка технологии MT имеется пока лишь на двух платформах: SPARC M7/M8 (Application Data Integrity, ADI) и ARM v8.5 (Memory Tagging Extension, MTE).

Наконец, для некоторых видов неопределенного поведения не существует пока надежных методов обнаружения. Например, это относится к ошибкам перекрытия объектов в памяти (strict aliasing rules, см. [35]) и к нарушениям правил модификации объектов между точками следования (sequence points). Современные компиляторы выдают диагностику в относительно простых случаях, но в целом отсутствие диагностики не гарантирует отсутствия ошибок. Помимо неопределенного поведения, стандарты языков C и C++ описывают ситуации неспецифицированного поведения, которые также могут быть источником ошибок. В частности, не определен порядок вызовов функций при вычислении аргументов функций. В [31] также рассматриваются некоторые легальные языковые средства, являющиеся источником распространенных уязвимостей.

Тем не менее, рассмотренные в работе санитайзеры в значительной мере компенсируют негативные свойства языков C и C++, такие как небезопасность работы с памятью, возможность небезопасной работы с типами и другие. Несомненно, дальнейшее развитие и внедрение подобных средств будет способствовать повышению надежности и безопасности ПО. Отметим, что попытки свести к минимуму эффекты неопределенного поведения на уровне стандарта языка C не привели пока к желаемому результату. В C11 были введены необязательные требования, относящиеся к функциям манипуляций со строками с безопасными (bounds checking) интерфейсами (Annex K) и к свойству анализируемости (Analyzability, Annex L). Хотя компиляторы поддерживают опции, позволяющие снимать некоторые виды неопределенного поведения (strict aliasing, переполнение целых), требование анализируемости в целом, по-видимому, не было реализовано ни в одном из них. Что касается функций с безопасными интерфейсами, то в документе [32], где анализируются недостатки этого пункта стандарта и предлагается в конечном счете отказаться от него, в качестве альтернативных решений рассматриваются прежде всего средства динамического анализа.

6. БЛАГОДАРНОСТИ

Исследование выполнено в рамках государственного задания ФГУ ФНЦ НИИСИ РАН (проведение фундаментальных научных исследований 47 ГП) по теме № 0065-2019-0002 “Исследование и реализация программной платформы для перспективных многоядерных процессоров” (рег. № АААА-А19-119012290074-2).

СПИСОК ЛИТЕРАТУРЫ

1. *Латтнер К.* Что каждый программист на C должен знать об Undefined Behavior. https://habr.com/ru/post/341144/.
2. *Дудина И.А., Белеванцев А.А.* Применение статического символьного выполнения для поиска ошибок доступа к буферу // Программирование. 2017. № 5. С. 3–17.
3. *Glenn R. Luecke, Coyle J., Hoekstra J., Kraeva M., Li Y., Taborskaia O., and Yanmei Wang.* A Survey of Systems for Detecting Serial Run-Time Errors // Concurrency and Computation: Practice and Experience, 2006. P. 1885–1907.
4. *Rigger M., Pekarek D., Mossenbock H.* Context-aware Failure-oblivious Computing as a Means of Preventing Buffer Overflows // Proceedings of the 12th International Conference, NSS 2018. P. 376–390.
5. *Serebryany K., Bruening D., Potapenko A., Vyukov D.* AddressSanitizer: a fast address sanity checker. // Proceedings of the 2012 USENIX conference on Annual Technical Conference. Berkeley, CA, USA, 2012, p. 309–318.
6. AddressSanitizerComparisonOfMemoryTools. https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools.
7. *Stepanov E., Serebryany K.* MemorySanitizer: fast detector of uninitialized memory use in C++ // Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2015. P. 46–55.
8. ThreadSanitizerCppManual. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual
9. ThreadSanitizerPopularDataRaces. https://github.com/google/sanitizers/wiki/ThreadSanitizerPopularDataRaces
10. *Dietz W., Li P., Regehr J.* Understanding Integer Overflow in C/C++. http://www.cs.utah.edu/regehr/papers/tosem15.pdf
11. *Christey S., Martin R.A., Brown M., Paller A., Kirby D.* 2011 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/.
12. *Ализар. А.* Разработан универсальный фаззер, объединивший 15 разных фаззинг-приложений. 2011. https://xakep.ru/2011/04/25/55501/
13. *Man'es V.J.M., Han H., Han C., Cha S.K., Egele M., Schwartz E.J., Woo M.* The Art, Science, and Engineering of Fuzzing:*A Survey*. https://arxiv.org/pdf/1812.00140.pdf
14. *Саттон М., Грин А., Амини П.* Fuzzing: исследование уязвимостей методом грубой силы. Пер. с англ. СПб.: Символ Плюс, 2009. 560 с.

15. *Вьюков Д.* C++ Russia 2017: Fuzzing: The New Unit Testing. https://www.youtube.com/watch?v=FD30Qzd6ylk
16. Csmith. https://embed.cs.utah.edu/csmith/
17. American fuzzy lop. http://lcamtuf.coredump.cx/afl
18. OSS-Fuzz – continuous fuzzing of open source software. https://github.com/google/oss-fuzz
19. libFuzzer – a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html.
20. PEACH Fuzzer. https://www.peach.tech/products/peach-fuzzer/
21. Gofuzz. https://github.com/google/gofuzz.
22. *Abadi M., Budiu M., Erlingsson U., Ligatti J.* Control-Flow Integrity Principles, Implementations, and Applications // ACM Conference on Computer and Communication Security (CCS). November, 2005. P. 340–353.
23. *Kuznetsov V., Szekeres L., Payer M., Candea G., Sekar R., Song D.* Code-Pointer Integrity // Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October, 2014. P. 147–163.
24. *Tice C., Roeder T., Collingbourne P., Checkoway S., Erlingsson U., Lozano L., Pike G.* Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM // SEC'14 Proceedings of the 23rd USENIX conference on Security Symposium. August, 2014. P. 941–955.
25. Let's talk about CFI: clang edition. https://blog.trailofbits.com/2016/10/17/lets-talk-about-cfi-clang-edition/
26. Getafix: How Facebook tools learn to fix bugs automatically. https://engineering.fb.com/developer-tools/getafix-how-facebook-tools-learn-to-fix-bugs-automatically/
27. *Нурмухаметов А.Р., Курмангалеев Ш.Ф., Каушан В.В., Гайсарян С.С.* Применение компиляторных преобразований для противодействия эксплуатации уязвимостей программного обеспечения // Труды ИСП РАН. 2014. Т. 26. Вып. 3. С. 113–126.
28. *Cheptsov V., Khoroshilov A.* Dynamic Analysis of ARINC 653 RTOS with LLVM // Ivannikov Isp Ras Open Conference, Moscow, 22–23 November 2018. P. 9–15.
29. *Lee W.* Address Sanitizer on Myriad. https://docs.google.com/document/d/1oxmk0xUojyb-DaQDAuTEVpHVMi5xQX74cJPyMJbaSaRM
30. UB-2017. Часть 1. https://habr.com/ru/post/341694/
31. *Song D., Lettner J., Rajasekaran P., Na Y., Volckaert S., Larsen P., Franz M.* SoK: Sanitizing for Security. 2019, https://oaklandsok.github.io/papers/song2019.pdf.
32. Updated Field Experience With Annex K – Bounds Checking Interfaces. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1969.htm.
33. *Serebryany K., Stepanov E., Shlyapnikov A., Tsyrklevich V., Vyukov D.* Memory Tagging and how it improves C/C++ memory safety. Google, February 2018. https://arxiv.org/pdf/1802.09517.pdf.
34. Hardware-assisted AddressSanitizer Design Documentation. http://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.
35. *Horgan P.* Understanding C/C++ Strict Aliasing. http://dbp-consulting.com/tutorials/StrictAliasing.html.