УДК 519.686

D. A. Maksimenkov

Test generation system for x86 with random instruction sequences

Abstract

The paper concerns an approach to debugging of complex software systems like binary compilers where a wide variety of test cases is required. Test generation system with random x86 instruction sequences is considered. Several automatic generation typical issues are investigated and methods for their resolving are advised.

Система генерации тестов со случайным набором команд в кодах x86

Д. А. Максименков

ЗАО “МЦСТ”, г. Москва

shark@mcst.ru

Аннотация

При создании сложных программных продуктов, таких как системы бинарной компиляции, требуется широкий набор различных отладочных тестов. В статье рассматривается опыт создания генератора тестов в кодах архитектуры x86 со случайной последовательностью команд. Приводится анализ ряда практических проблем автоматической генерации и предлагаются методы их эффективного решения.

Система генерации тестов со случайным набором команд в кодах x86

Д. А. Максименков

ЗАО “МЦСТ”, г. Москва

shark@mcst.ru

Введение

В статье рассматривается технология тестирования системы бинарной компиляции (СБК) [1], разрабатываемой для вычислительного комплекса “Эльбрус-3М1” (далее ВК) [1]. Основные компоненты СБК, – интерпретатор, шаблонный транслятор, оптимизирующие компиляторы 3-го и 4-го уровней, профилировщик и наборщик регионов. Главная цель СБК – обеспечение возможности запуска приложений в бинарных кодах одной платформы на другой архитектурной платформе. В зависимости от профиля исполняемого кода используются те или иные компоненты СБК, обеспечивающие наиболее эффективное исполнение кода.

Для тестирования этих компонент, самих средств отладки и для проверки аппаратуры ВК предлагается технология генерации тестов в кодах исходной архитектуры (в рассматриваемой СБК это архитектура IA-32 [2]). Генераторы тестов для тестирования бинарных компиляторов и аппаратуры можно встретить, например, в проектах корпораций Transmeta – TiGeR [3], IBM – Genesys-x86 [4] и др.

Статья является продолжением публикаций [5, 6], в которых излагаются способы организации тестирования оптимизирующего бинарного компилятора на примере создания генератора самопроверяющихся тестов со сложным графом управления для архитектуры IA-32. Такие тесты используют довольно большой, но все же ограниченный, набор наиболее часто используемых команд, учитывают особенности оптимизирующего бинарного компилятора и хорошо подходят для организации круглосуточных запусков при контроле надежности системы. В них не возникают какие-либо исключительные ситуации, все они являются детерминированными и хорошо подходят для тестирования статических оптимизирующих компиляторов [1], что зачастую позволяет упростить процедуру выявления ошибок.

Ниже предлагается еще одна технология отладки на примере создания генератора тестов со случайным набором инструкций. Целью тестирования с их помощью является проверка максимально большого количества всевозможных последовательностей инструкций IA-32 (в том числе и недопустимых), редко встречающихся в реальных задачах, и их различных комбинаций на СБК ВК. Также одной из главных целей рассматриваемого тестирования является формирование маленьких простых тестов, удобных для анализа и не требующих больших ресурсов при их генерации и запуске. Помимо этого ставится задача создания тестов, способных вырабатывать различные исключительные ситуации, что позволило бы проверить адекватное поведение системы при возникновении различных типов исключительных ситуаций во время исполнении кодов IA-32.

Описание технологии

В CISC-архитектурах, к которым, в частности, относится и семейство процессоров IA-32, длина команды не имеет фиксированного размера и может варьироваться от 1 до 15 байт [2]. Таким образом, чтобы перебрать все возможные операции, необходимо сформировать 256^{15} различных вариантов сочетаний бит и проверить их работоспособность на СБК, с одной стороны, а также использовать при тестировании

аппаратуры ВК, с другой. Количество различных комбинаций таких команд резко увеличивается с ростом длины цепочки операций. Нельзя забывать и о большом разнообразии возможных аргументов для таких команд. Итак, в нашем распоряжении имеется большое поле деятельности в отношении проверки надежности СБК. Особый интерес в этом переборе команд представляют операции, редко или вообще не встречающиеся в реальных задачах, и таким образом являющиеся либо плохо оттестированными, либо вообще не тестированными на рассматриваемой СБК. Также интересна реакция оптимизирующих компонент компилятора на случайные комбинации команд, хотя для большей части таких инструкций возможность попадания в регионы [1], подвергающиеся в дальнейшем оптимизациям, ограничена. В то же время срабатывания этих ограничений и корректную работу компилятора в таких ситуациях также необходимо проверить.

От случайных комбинаций команд со случайными аргументами стоит ожидать различных исключительных ситуаций: в цепочке инструкций может оказаться недопустимая операция, обращение к закрытой на чтение/запись странице памяти, переход на не принадлежащий программе адрес, некорректный вызов функции и т.д. Здесь возможен любой ситуационный сценарий, приводящий к возникновению различных исключений. Все эти ситуации также представляют интерес при проверке надежности системы. СБК должна быть отказоустойчивой при исполнении любой комбинации байт, интерпретируемой в качестве команд процессора x86.

Алгоритм генерации тестов со случайным набором команд может быть следующим. За основу берется шаблон простой программы на ассемблере, где вместо тела теста стоит макрос, определенный в отдельном внешнем файле. Это делается для того, чтобы отделить основной код исходного текста программы на ассемблере, который не меняется при каждой генерации нового теста, от изменяемой ее части – подменяемого макроса (рис. 1).

Данный макрос представляет собой строчку символов, заполненную произвольным набором байт. Например, это может быть строчка из потока случайных символов /dev/urandom в ОС Linux (рис. 2). Длина строчки L варьируется режимом тестирования и может составлять от нескольких символов до сотен байт, в зависимости от целей тестирования. Нужно отметить тот факт, что при увеличении длины генерируемой строки возрастает вероятность появления в тесте некорректных инструкций, приводящих, в свою очередь, к проявлению различных исключительных ситуаций. В случае успешной отработки тела теста (подставляемого макроса из случайных команд) приложение завершается системным вызовом exit с кодом возврата 0. Именно по нему можно будет судить об успешной работе созданной цепочки операций.

После того, как строчка случайных символов занесена в тело макроса, находящегося в отдельном файле, запускается процесс компиляции и линковки теста. Эта операция не потребует больших ресурсов для такого небольшого кода. При исполнении задачи описанным способом в ОС Linux результатом ее работы может быть одна из следующих ситуаций:

- успешное завершение с нулевым кодом возврата и передача управления в командный интерпретатор
- зацикливание теста (довольно редкий случай – при возникновении бесконечного цикла из случайных инструкций макроса)
- выработка тестом сигнала, являющегося следствием возникновения исключительной ситуации (Segmentation fault, Illegal instruction и т.д.) – наиболее вероятная ситуация, напрямую зависящая от количества созданных случайных команд.

В первом случае никаких дополнительных действий предпринимать не нужно.

В случае зацикливания теста – поможет запуск с ограничением по времени работы задачи. Например, для командного интерпретатора bash в ОС Linux строчка запуска может выглядеть так:

bash –icv “ulimit –t **$TIME**; ./**$TEST**”

где переменные: TIME – время, отводимое для работы теста в секундах, TEST – имя запускаемого теста. Как показал опыт, нескольких секунд вполне достаточно для того, чтобы задачи, не имеющие бесконечных циклов, успели завершить свою работу.

Поведение, описанное в последнем случае, было бы критичным для бинарных компиляторов приложений. Например, бинарный компилятор Execution Layer фирмы Intel для процессоров Itanium2 [7] не гарантирует корректного исполнения задачи при возникновении в результате ее вычислений не-чисел (Nan, Qnan, Inf), не говоря уже о корректной работе приложения со случайным набором команд. В СБК, реализованной на процессоре “Эльбрус” ВК “Эльбрус-3М1”, такое поведение тестов вполне допустимо, и система должна отрабатывать возникающие сигналы исключительных ситуаций аналогично архитектуре IA-32. Именно это и позволяет проверять тесты, создаваемые описанным алгоритмом.

Однако при генерации достаточно длинных цепочек случайных байт, интерпретируемых в качестве инструкций процессора, у команд, находящихся в конце программы, при описанном подходе вероятность быть исполненными оказывается меньше всего из-за возможных прерываний на более ранних операциях. На рис. 3 приведена гистограмма вероятности возникновения исключительной ситуации при исполнении таких команд, сформированных из случайной последовательности байт. Как видно из гистограммы, при интерпретации 10 байт кода в качестве инструкций x86 вероятность возникновения сигнала оказывается выше 90%.

Для работы шаблонного кода системы бинарной трансляции и оптимизирующих компиляторов необходимо многократное повторение одних и тех же команд приложения для определения системой “горячих” участков задачи с целью их дальнейшей оптимизации [1]. При описанной реализации теста такого эффекта можно достичь несколькими повторными запусками одного и того же приложения. Но этот способ оказывается неэффективным из-за сравнимости времени запуска/окончания работы задачи в ОС со временем исполнения инструкций самого теста. Гораздо эффективней было бы иметь внутренний цикл программы, повторяющий цепочку случайных операций заранее заданное число раз. Но как этого добиться, если вероятность возникновения прерывания (а значит и досрочного завершения работы задачи) при генерации случайной последовательности команд оказывается довольно высока?

Предлагается следующий механизм, устраняющий описанные выше проблемы. На наиболее часто возникающие исключительные ситуации в самом тесте производится регистрация обработчика сигналов – функции, вызываемой при появлении исключительных ситуаций в задаче. Рис. 3 показывает статистику возникновения различных сигналов для случайной цепочки байт. Оказывается, что при длине цепочки команд в 10 байт вероятности появления сигналов SEGV (Segmentation fault) и ILL (Illegal instruction) составляют 90% и 5% соответственно от общего числа возникающих сигналов. Такая тенденция сохраняется и при более длинных цепочках байт, исполняемых в качестве команд процессора x86. Остальные 5% составляют сигналы TRAP, BUS, FPE, INT и другие, встречающиеся достаточно редко. Такое неравномерное распределение вероятности возникновения различных типов прерываний создает дисбаланс в тестировании исключительных ситуаций. Для подавления “лидеров” – сигналов SEGV и ILL в тесте регистрируется обработчик на эти типы сигналов. Тем самым мы собираемся откорректировать работу теста в 9 из 10

случаев возникновения исключительных ситуаций при исполнении программы. Кроме того, коррекция этих ситуаций должна уменьшить количество случаев досрочного завершения работы теста из-за возникающих прерываний. Это позволит увеличить количество тестов, имеющих “горячие” участки кода, т.е. пригодных для набора регионов (тесты с кодом возврата 0). Одновременно возрастает вероятность проявления при работе теста других исключительных ситуаций, ранее маскируемых сигналами SEGV и ILL.

Операционная система в момент выполнения инструкции, приведшей к возникновению исключительной ситуации, посылает тесту сигнал. Работа программы прерывается с сохранением ее контекста и управление передается соответствующему обработчику сигнала. В случае отсутствия в пользовательском приложении такого обработчика запускается стандартный обработчик, который завершает работу программы с соответствующим данному сигналу ненулевым кодом возврата. Так, например, для сигнала SEGV код возврата равен 139. Если же программист хочет отработать данную исключительную ситуацию особым образом (как обстоит дело в нашем случае) и для этих целей зарегистрировал свой обработчик сигналов, то ОС передает управление обработчику (в нашем случае единому для нескольких сигналов) вместе с дополнительной информацией о контексте возникновения исключения, в том числе и адрес команды.

Внутри самого обработчика запускается алгоритм, корректирующий в теле теста код инструкции, на которой возникло исключение, путем замены ее другой случайной цепочкой значений, начиная с адреса некорректной команды. Чтобы исправить ситуацию, заменять всю цепочку не нужно – достаточно модифицировать несколько первых байт. Замена должна иметь случайный, но детерминированный характер. Новые значения будут зависеть от значений байт старой цепочки. Это делается для того, чтобы поведение теста не менялось от запуска к запуску. В описываемой реализации используется коррекция 32 бит цепочки (переменная value) по следующей формуле генератора псевдослучайных чисел:

```
long value;
value = 1664525L * value + 1013904223L;
```

После коррекции команды (фактически ее подмены) обработчиком, управление возвращается в тело теста на адрес, на котором было прервано его исполнение, и восстанавливается исходный контекст. Далее снова производится попытка выполнить инструкцию, находящуюся по этому адресу. В этот раз там окажутся новые значения байт кода, которые в соответствии с системой команд IA-32 будут интерпретированы как другая команда. Если при ее исполнении вновь возникнет исключительная ситуация, мы опять окажемся в обработчике сигналов и заново скорректируем код. Если же команда получилась корректной, то тест продолжит свою работу до следующей недопустимой команды. Этот процесс будет повторяться до тех пор, пока мы не достигнем конца цепочки операций. Иллюстрация описанного выше алгоритма приведена на рис. 4.

Здесь в качестве примера приведена цепочка из 8 команд макроса **rand** длиной в 13 байт (а). Цифрами показаны длины команд. Во время исполнения 2-ой инструкции размером в 3 байта возникла исключительная ситуация. Управление передается обработчику, который изменяет 4 байта цепочки, начиная с адреса команды, на которой возникло исключение. В результате на месте второй команды получилась новая цепочка операций, состоящей из двух-, одно- и начала трехбайтной операции (б). Все они отрабатывают успешно. Пятая команда цепочки (б) оказывается некорректной, и управление снова передается обработчику. Образованный им код (в) оказался также

недопустимым, и мы вновь запускаем функцию, корректирующую исполняемый код. Полученная цепочка (г) дорабатывает до конца макроса **rand** успешно.

Чтобы избежать затирания команд, которые стоят в тесте после вызова макроса, во время процесса коррекции недопустимого кода, обработчиком сигналов после макроса **rand** в тесте изначально добавляется несколько nop′ов – команд-пустышек. Таким образом, если исключение возникло на однобайтовой команде, являющейся последней в цепочке операций макроса, то функция-корректор исправит эти операции nop, не испортив идущий далее осмысленный код теста (например, системный вызов exit).

Теперь у нас есть цепочка команд, образованная самим тестом в результате нескольких итераций автокоррекции кода. Полученную последовательность инструкций процессора уже можно поместить в цикл, количество итераций которого будет достаточным для формирования в тесте “горячего” участка исполнения кода программы с заданным числом повторений. Запуск такого теста в системе бинарной компиляции позволит подключить к работе важные компоненты системы [1]. Например, начнет работу шаблонный транслятор, профилировщик, оптимизирующие компиляторы разных уровней и другие сложные компоненты системы [1]. Это дает возможность тестирования перечисленных компонентов СБК и аппаратуры ВК “Эльбрус-3М1” при работе на случайной цепочке команд.

Далее, при создании тестов необходимо обратить внимание на воспроизводимость выявляемых ими ошибочных ситуаций. Невоспроизводимых ошибок не существует. Бывают не до конца воспроизведены условия (окружение запуска), при котором возникла ошибка. Для того, чтобы повысить вероятность получить детерминированный тест, а значит, в случае обнаружения ошибки повысить шансы на ее воспроизведение, необходимо учитывать следующее. Поскольку создаваемый код теста является случайным и может при вычислениях использовать различные значения регистров, то разумно было бы перед его исполнением провести инициализацию всех регистров общего назначения, а также статусных и управляющих регистров [2]. Значения для инициализации также могут быть получены случайным образом, как и при получении случайных команд. Т.е. в сегмент данных вставляется макрос со случайной строчкой символов, интерпретируемой далее как значения для перечисленных выше регистров, инициализация которых стоит в начале теста. Теперь исходный код нашего теста будет выглядеть, как показано на рис. 5.

В начале теста регистрируется обработчик сигналов на ситуации, описанные выше (SEGV и ILL). Далее идет инициализация регистров (целочисленных, вещественных, векторных, статусных и управляющих), значения которых могут влиять на ход исполнения программы. Инициализация производится посредством случайных значений бит, доступных по метке **mem** в сегменте данных. В данном случае за этой меткой расположен макрос **rand2**, определяемый аналогично макросу **rand** в подключаемом файле **rand.mac**. Размер строки во втором макросе должен быть достаточным для независимой инициализации всех регистров (на рисунке приведен пример укороченной строки в макросе **rand2**).

После инициализации регистров в программе организован цикл с управляющей 32-битной переменной **counter** в памяти, обеспечивающей выполнение 32 тыс. итераций тела цикла. В качестве последнего выступает случайный набор команд, определенный в макросе **rand**. За ним следуют 3 операции **nop** для предотвращения описанного выше краевого эффекта при самомодификации кода. В сегменте данных помимо массива **mem** инициализируется счетчик цикла **counter**. Количество итераций цикла можно изменить, задав необходимое начальное значение этой переменной.

При инициализированных всех регистрах и коррекции обращений в недопустимую память вероятность недетерминированного поведения задачи и использование

неинициализированных данных, влияющих на поведение задачи, резко уменьшается. Также стоит производить запуск теста в ОС с обнулением окружения командного интерпретатора. Т.е. предложенная выше строчка запуска принимает вид:

bash –norc –icv “ulimit –t **$TIME**; exec –c ./**$TEST**”

Однако, несмотря на все наши усилия, остается вероятность некорректного поведения задачи: например, при генерации перехода на доступный адрес внутри теста или вызов библиотечной функции с некорректными параметрами. Несомненно, внутри такого кода произойдет исключительная ситуация и никакая модификация инструкций не позволит корректно закончить работу теста. Такие случаи следует отсеивать и не рассматривать в процессе тестирования как допустимые. Для этого внутри обработчика сигналов реализован механизм контроля модифицируемых адресов. Если адрес команды, на которой возникло исключение, принадлежит промежутку значений от метки loop до адреса loop+L, то код нужно скорректировать и продолжить его исполнение. Если же по каким-то причинам мы оказались вне макроса **rand**, то управление теста было передано в недопустимую область программы и продолжать исполнение кода бессмысленно. В этом случае управление передается системному вызову exit с кодом возврата -1. Именно по нему мы можем отсеивать такие “бракованные” тесты.

Результаты

После отсеивания “бракованных” тестов, составляющих почти 50% от общего количества получаемых при генерации задач, для оставшихся тестов распределение получаемых кодов возвратов приведено на круговой гистограмме рис. 6. 57% тестов заканчивают свою работу с кодом возврата 0, то есть являются пригодными для тестирования различных компонент системы бинарной компиляции (на таких тестах цепочка случайных команд повторяется заданное количество раз, достаточное для формирования региона). 19% тестов, несмотря на регистрацию обработчика сигналов SEGV, завершают работу с кодом 139. Такая ситуация возникает при переполнении стека в результате исполнения случайной цепочки команд или из-за перезаписи регистра %esp – стекового указателя. 8% тестов получаются с бесконечными или долго работающими циклами. Остальные 16% составляют другие сигналы, такие как TRAP, BUS, FPE, INT и т.д. Добиться уменьшения тех или иных исключений можно, зарегистрировав тот же самый обработчик на соответствующий тип сигналов, при этом процент тестов с кодом возврата 0 будет увеличиваться.

Тесты, получаемые генератором, реализованным по описанной выше технологии, обладают следующими свойствами:

– они невелики по размеру (несколько Кб) и имеют небольшое время исполнения (меньше 1 сек);

– их формирование не требует больших временных и машинных ресурсов, а также сложных скриптов генерации и запуска;

– благодаря своей простоте и небольшому размеру они удобны для анализа и разбора ошибок;

– вероятность воспроизведения ошибок, выявленных на таких тестах, довольно высока

– возможность легкого повторного воспроизведения всех обнаруженных на них ошибок;

–в получаемых тестах есть самомодификация кода, что позволяет проверить некоторые режимы работы наборщика регионов и базы кодов регионов системы [1];

– во время исполнения тестов возможно возникновение любых исключительных ситуаций, что также актуально при отладке системы;

– в создаваемых тестах возможны любые (что немаловажно) последовательности IA-32 инструкций (в том числе и недопустимые), позволяющие проверить работу СБК при различных комбинациях команд и их аргументов;
– такие тесты адаптированы для активизации работы всех уровней СБК, благодаря наличию в них “горячих” участков кода внутри циклов.

Для тестирования можно использовать два режима генерации тестов:
– более простой, с запуском непосредственно на отлаживаемой системе процесса генерации теста, его сборку и исполнение (генерация тестов без эталонной x86-машины);
– более сложный, с регистрацией обработчиков сигналов для автокоррекции кода, с отсеиванием “бракованных” тестов на эталонной машине и с последующим запуском итогового набора тестов на СБК.

Получаемые тесты не являются самопроверяющимися, поэтому тестирование и выявление ошибок при их запуске рассчитано на внешние средства верификации. В рассматриваемой СБК обнаружение ошибок происходило:
– посредством всевозможных внутренних проверок отладочной версии СБК: с помощью чекеров и ловушек самой системы;
– с помощью системы сравнения оптимизированного и неоптимизированного кода СБК [8].

При тестировании кодов, полученных с помощью описанного генератора, обнаружено более 25 ошибок на уже отлаженной версии СБК, успешно работающей с различными ОС и пользовательскими приложениями. Среди найденных ошибок – нереализованные, редко используемые и недопустимые инструкции СК IA-32, ошибки в различных компонентах СБК, в частности, в наборщике и компиляторе регионов, ошибки в шаблонном коде и в интерпретаторе. С помощью таких тестов удалось обнаружить несколько ошибок и в средствах отладки; была также выявлена аппаратная проблема, приводящая к остановке ВК “Эльбрус-3М1”.

Создаваемые тесты являются 32-битными пользовательскими приложениями и предназначены для запуска в ОС Linux.

Предполагается дальнейшее развитие описанной технологии с возможностью интегрирования ее в другие автоматизированные системы генерации тестов для верификации и тестирования СБК, отладочных средств и аппаратуры ВК в рамках проекта “Эльбрус” [1].

Список литературы

1. **Волконский В. Ю.** Оптимизирующие компиляторы для архитектур с явным параллелизмом команд и аппаратной поддержкой двоичной совместимости // Информационные технологии и вычислительные системы. 2004. № 3. С. 4-26.
2. **Intel Corporation**. IA-32 Intel Architecture Software Developer's Manual, Vol. 1, 2.
3. **Anshuman S. Nadkarni, Tom Kenville**. Transmeta Corporation. TiGeR, the Transmeta Instruction GEneratoR: A production based, pseudo random instruction, x86 test generator. Transmeta Corporation, 2004.
4. **Arbetman Ya.** et al. Genesys-x86 An automatic test program generator for x86 microprocessors. IBM Haifa Research Lab, 1997.
5. **Максименков Д. А.** Генератор тестов для бинарного компилятора // Современные информационные технологии и ИТ-образование. М.: Макс-пресс, 2005. С. 445-452.
6. **Максименков Д. А.** Метод создания самопроверяющихся тестов для системы бинарной компиляции // Высокопроизводительные вычислительные системы и микропроцессоры. Вып. 9. М.: ИМВС РАН, 2006. С. 26-37.
7. **Baraz L., Devor T., Etsion O. et al.** IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based system // Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36'03).
8. **Иванов А.А., Преснов Н.Ю.**, Технология динамического сравнения трасс для отладки систем двоичной трансляции // Информационные технологии. 2005. № 2. С. 49-54.

Список подрисуночных подписей

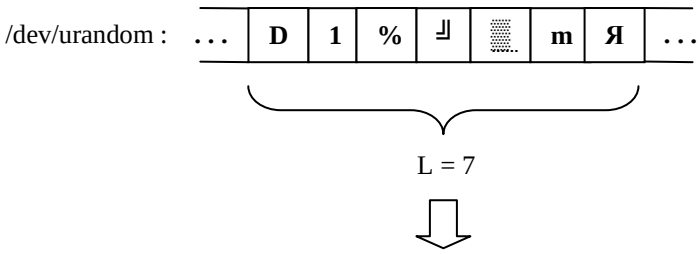
Исходный код теста на ассемблере (статическая часть)

```
.include  “rand.mac”
.section   .text
main:      …
           rand
           …
           exit (0)

.section  .data
           …
```

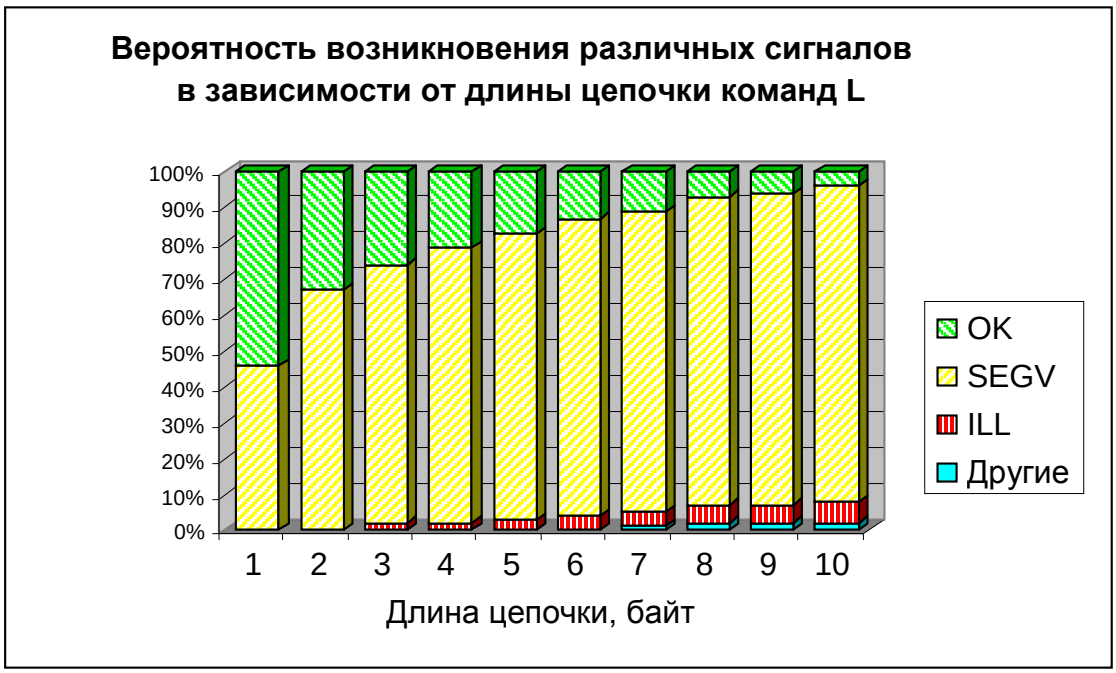
Подключаемый файл **rand.mac** с макросом **rand** (изменяемая часть)

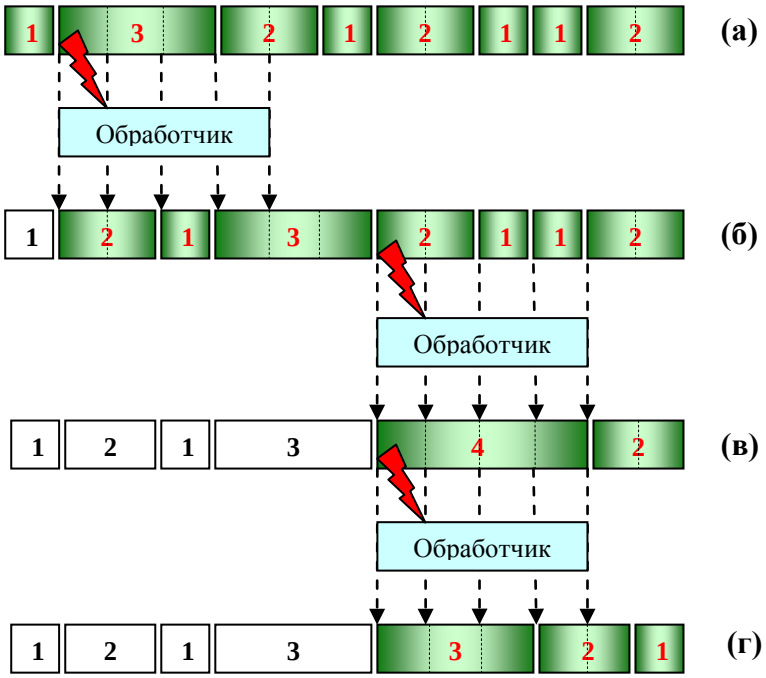
```
.macro rand
.string “D1%╝▒mЯ”
.endm
```



Макрос **rand**:

```
56 4F 1A    mulb  $0x4f, %ah
D1 C7       fcom  %st(5)
72 B3       push  %ecx
```


Вероятность возникновения различных сигналов
в зависимости от длины цепочки команд L
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
1
2
3
4
5
6
7
8
9
10
Длина цепочки, байт
OK
SEGV
ILL
Другие


1
3
2
1
2
1
1
2
(а)
Обработчик
1
2
1
3
2
1
1
2
(б)
Обработчик
1
2
1
3
4
2
(в)
Обработчик
1
2
1
3
3
2
1
(г)

Исходный код теста на ассемблере (статическая часть)

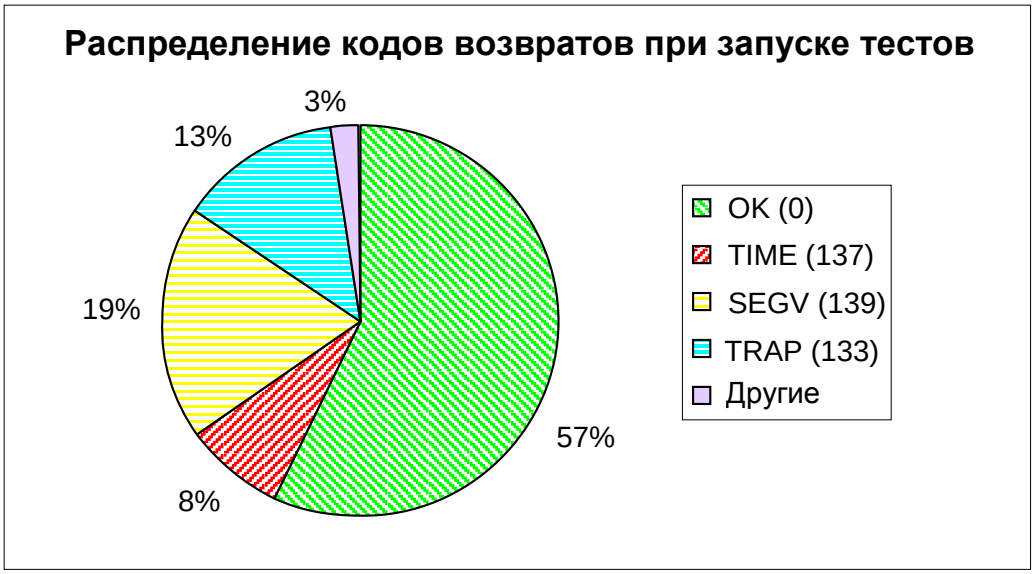
```
.include  “rand.mac”
.section   .text
main:      init Handler
           …
           movl mem, %eax
           movl mem+4, %ebx
           movl mem+8, %ecx
           …
loop:      rand
           nop (3)
           decl counter
           jnz loop
           exit (0)

.section  .data
counter: .long  32000
mem:       rand2
```

Подключаемый файл **rand.mac** с макросами (изменяемая часть)

```
.macro rand
.string “D1%╝▒mЯ”
.endm

.macro rand2
.string “Ы<#2+(sV.q~w¶”
.endm
```


Распределение кодов возвратов при запуске тестов
3%
13%
19%
57%
8%
OK (0)
TIME (137)
SEGV (139)
TRAP (133)
Другие