Государственное образовательное учреждение высшего профессионального образования «Московский физико-технический институт (государственный университет)»

Факультет управления и прикладной математики

Кафедра теоретической и прикладной информатики

Расширение возможностей проверяющей системы Ejudge по безопасной проверке решений с использованием технологий виртуализации

Выпускная квалификационная работа на степень бакалавра

студента 176 гр.

Дербышева Дмитрия Юрьевича

Научный руководитель

Подлесных Дмитрий Артурович, старший преподаватель кафедры информатики и вычислительной математки МФТИ

Долгопрудный,

2015 г.

Цель работы.

Кафедра информатики постоянно развивает техническую базу для проведения занятий. В частности, для работы студентам создаются персональные контейнеры, чтобы избавить их от необходимости самостоятельной установки и настройки средств разработки **Ошибка! Источник ссылки не найден.**. Для непредвзятой проверки задачи используются автоматизированные проверяющие системы ejudge и ElJudge. Все эти системы экономят время как студента, так и преподавателя, избавляя их от решения рутинных технических проблем. Для этого крайне полезна полная совместимость между средой разработки и средой тестирования: одинаковая архитектура, разрядность, операционная система, компилятор, набор библиотек, опции компиляции. В идеале тестирование программ студентов должно происходить на копии тех же контейнеров, что и их разработка.

Целью данной работы является повышение качества работы автоматической проверяющей системы Ejudge за счет расширения функциональности.

Более шести лет в Московском Физико-Техническом Институте и некоторых других высших учебных заведениях (например, МГУ) стабильно используется и совершенствуется известная система автоматического тестирования Ejudge, позволяющая проводить олимпиады по информатике для абитуриентов, межфакультетские контрольные работы и другие формы единомоментной массовой проверки по информатике. Данная система не является единственным аналогом, возможные альтернативы предложены в **Ошибка! Источник ссылки не найден.**.

Система автоматической проверки должна, как и любой инструмент измерения, обеспечивать достоверность результатов. Для этого она должна быть защищена от взлома, неумышленного повреждения или чрезмерного потребления ресурсов проверяемым кодом. За обеспечение этого уровня безопасности приходится расплачиваться либо производительностью, либо ограничением функциональности.

Задачи данной работы:

1. Создание нового механизма проверки, с устраненными ограничениями на проверяемые программы.
2. Достаточная производительность предложенного подхода (время проверки на каждом тесте должно возрасти не более, чем на 10 секунд).
3. Сохранение всех уже имеющихся механизмов проверки

В следующих пунктах будут подробнее описаны угрозы для проверяющих систем, уже имеющиеся в Ejudge механизмы защиты, их достоинства и недостатки.

Постановка проблемы безопасности

Необходимым условием признания корректности сдаваемой студентом программы является правильный вывод на тестовых примерах, при этом программа должна работать при заданных ограничениях на вычислительные ресурсы. Оценка может быть уточнена преподавателем вручную, но цель автоматизации (да и смысл вычислительной техники вообще) в сокращении затрат времени на рутинные операции.

Система проверки должна также предотвращать:

1. Случаи использования запрещенных (с точки зрения условия задачи) методов
    а) чтение готового правильного ответа;
    б) решение задачи на удаленном сервере;
    в) использование функций, запрещенных автором задачи.
2. Потенциально опасные уязвимости программ:
    а) выход за границы выделенной памяти
    б) использование неинициализированных переменных;
    в) утечка памяти.
3. Ухудшение качества работы самой проверяющей системы
    а) вердикт решения не должен зависеть от нагрузки сервера;

б) при любой расчетной нагрузке на сервер должна обеспечиваться своевременная проверка решений.

4. Работоспособная программа при некачественном исходном тексте

   а) несоблюдение стиля оформления программ;

   б) плагиат.

Помимо этого, желательно обеспечение некоторых организационных моментов, таких как автоматическое составление рейтинга участников, выбор студентов нужной группы из общего рейтинга, учет времени сдачи, возможность участников взаимодействовать с жюри и других. Их рассмотрение лежит за пределами данной работы.

Применяющиеся в настоящее время средства обеспечения защиты

Контроль кода на различных уровнях позволяет сделать проверяющую систему более гибкой.

Первым уровнем контроля может стать анализ текста решения на наличие определенных подстрок (команд, вызовов функций, имен файлов, URL и т. п.) с заменой (при необходимости) на безопасные аналоги. Например, проще автоматически удалить system("PAUSE"); чем объяснять впервые увидевшим автоматизированную проверку, что этой строки быть не должно. Впоследствии такую автоматическую замену нужно отключить (или выносить по таким решениям вердикт "нарушение правил оформления программ"), чтобы приучать студентов сразу писать правильно.

Вторым уровнем для программ на C/C++ являются специально сформированные (возможно, даже под конкретную задачу) заголовочные файлы, в которых, например, отсутствуют заголовки функций, которыми запрещено пользоваться в решении задачи. Недостатками этого подхода являются:

1. трудоемкость разработки задачи;
2. привязка к конкретному языку;
3. недопустимость размещения измененных заголовочных файлов вместо стандартных;
4. по умолчанию компиляторы собирают программу, даже если не все функции описаны в заголовочных файлах, но присутствуют в стандартной библиотеке.

Третьим уровнем являются опции компиляции, обнаруживающие потенциально опасное место в исходном коде и останавливающие компиляцию с ошибкой и выводом диагностики. В частности, сейчас на ejudge-кластере кафедры информатики МФТИ для C/C++ используются опции:

-Wall -Werror=format -Werror=return-type -Werror=vla -Werror=uninitialized -Werror=maybe-uninitialized -Werror-implicit-function-declaration.

Четвертым уровнем являются инструментальные средства, превращающие потенциально опасное поведение в гарантированный останов с определенным кодом возврата и диагностикой. Например, инструмент memcheck из набора Valgrind следит за освобождением динамически выделенной памяти, запрещает ее использование после освобождения, обнаруживает неинициализированные переменные (в том числе и ячейки массива), которые пропустил компилятор, а AddressSanitizer находит ошибки, связанные с выходом за границы стековых массивов.

Пятый уровень безопасности — ограничения на число процессов и оперативной памяти, заданные через стандартный механизм Linux — ulimit. К сожалению, многие интерпретаторы скриптовых языков и байт-кода требуют для своего запуска нескольких процессов и больших объёмов памяти, превышающих лимиты на задачу, используемые для компилируемых языков. При этом требуется модификация ограничений в зависимости от языка сдачи, а точное измерение потреблённой памяти и различение эффективного и неэффективного алгоритма становится затруднительным.

Шестой уровень безопасности основан на известной и многократно описанной идее перехвата функций (Function Interposition) стандартной библиотеки libc в Linux. Этот подход позволяет, например, запретить использование заданных библиотечных функций, причем возможна гибкая обработка перехватываемых вызовов, например, с выводом сообщения об ошибке и последующим завершением тестируемой программы с заданным кодом возврата. При этом возможно тестирование не только программ или модулей, написанных на языке C, но и статических и динамических библиотек. Редактор связей, входящий в состав GCC, позволяет скомпоновать программу, содержащую две сигнатуры функции с одним и тем же именем, если одна из них

описана в исходном модуле программы, а вторая – в стандартной библиотеке libc. В случае динамических библиотек используется следующая особенность загрузчика Linux: вызываемая из программы функция не привязана к конкретной разделяемой динамической библиотеке, т.е. загрузчик ищет требуемую функцию среди всех библиотек, подключенных в данный момент к программе.

Седьмым уровнем безопасности является модифицированное ядро Linux, предоставляющее тестируемому процессу только ограниченный «белый список» системных вызовов с ограниченным набором их допустимых параметров (например, ввод из входного файла или с консоли, вывод в файл или на экран, загрузка динамических библиотек).

Недостатки существующих подходов

Первые четыре из перечисленных механизмов не защищают саму проверяющую систему от взлома, а обеспечивают корректность решения задачи студентом. Не все они применимы к задачам на любом языке программирования, но это является их единственным недостатком: они не накладывают никаких функциональных ограничений на класс возможных для проверки задач. Последние же два пункта и являлись основной используемой в настоящее время защитой Ejudge.

Function Interposition повышает время на добавление новых задач и на добавление новых языков. Если бы этот метод был единственным, то затраты времени на его техническую поддержку оказались бы много больше времени, требуемого сейчас на добавление новых задач. Поэтому, применяется запуск под модифицированным ядром. Данное решение обеспечивает достаточный уровень надежности и не требует дополнительных временных затрат на составление новых задач, однако оно сильно ограничивает класс задач, которые можно проверять. Это же, в свою очередь, делает невозможным использование Ejudge во многих курсах и олимпиадах, таких как:

1. Сетевые технологии (7-8 семестр, ФУПМ МФТИ)
2. Прикладные физико-технические и компьютерные методы исследования: системное программное обеспечение (3 семестр, МФТИ, курс посвящен устройству операционных систем и межпроцессному взаимодействию)
3. Информационная безопасность
4. Работа с bash, shell, поддержка языков, которые могут использовать их при компиляции.

Запуск программ со многими потоками (и даже на нескольких узлах) поддерживается в системах управления кластером, например, PBS, Ganglia, Torque **Ошибка! Источник ссылки не найден.**. Но они не позволяют предоставить root-доступ, а также не проверяют результат, как чекеры ejudge.

Выбор инструментария, технические ограничения

Все решения, как-то ограничивающее вызываемые функции будут страдать теми же недостатками, что и предыдущие решения, поэтому требуется каким-то образом не ограничивать их, но сделать проверяющую систему изолированной от проверяемого кода. Для этого можно использовать технологии виртуализации.

При подходящем выборе технологии виртуализации компрометация сервера из изолированного приложения невозможна (при отсутствии уязвимостей в гипервизоре и микрокоде процессора). Для выбора конкретного решения вначале требуется уточнить, какие ограничения на решения уже имеются, и уже из подходящих выбрать наиболее заточенное под данную задачу. Как уже упоминалось ранее, обеспечение требуемого уровня безопасности влечет возможные потери в функциональности и в производительности. Рассмотрим производительность подробнее.

Основными ресурсами для сервера являются:

1. процессорное время (с нормировкой на текущую производительность, зависящую от модели процессора и текущей тактовой частоты);
2. оперативная память (локальная память для NUMA-систем);
3. место на системе хранения данных (твердотельный или жесткий диск, сетевое хранилище);
4. пропускная способность канала передачи данных.

Последняя не будет нами затронута, а место на жестком диске не является для Ejudge лимитирующим фактором. Процессорное же время является наиболее дефицитным ресурсом, как демонстрируется и из приблизительных расчетов, так и из уже имеющегося опыта использования.

Самое масштабное одномоментное использование Ejudge происходит во время олимпиад. В одной олимпиаде может участвовать порядка 2000 участников, каждый из которых, в худшем для нагрузки случае,

решит все задачи. Всего обычно порядка 10 задач, пускай они сдаются, в среднем, за 5 попыток. Каждая попытка сдачи представляет из себя последовательный запуск тестов, количество которых редко превышает 100. Каждый тест, в свою очередь, имеет ограничение по времени, указанное в задаче, и обычно измеряемое несколькими секундами

Оценка сверху машинного времени:

$$1\frac{с}{тест}\cdot 100\frac{тестов}{задача}\cdot 2000\ участников\cdot 10\ задач\cdot 5\frac{попыток}{задача}=10\,000\,000\ с=115\ суток\ машинного\ времени$$

Учитывая, что:

1. При неудачном тесте проверка чаще всего не выполняется на последующих тестах (при использовании групп тестов, правил “ACM” или “Olympiad”);
2. Среднее число тестов на задачу гораздо меньше (по одной из версий правил, во время олимпиады тестирование происходит только на небольшой подгруппе тестов, а на всём наборе перетестируется только последнее решение и уже после окончания тура олимпиады, например, ночью);
3. Не все участники приступают ко всем задачам;
4. Современные версии Ejudge позволяют тестирование на многоядерных процессорах в несколько потоков;
5. Во время олимпиады нагрузка распределяется на несколько (на практике - до 16) серверов,

это количество снижается до разумных пределов (например, только распараллеливание на 16 потоков снизит требуемое время до недели, что для олимпиады, длящейся несколько месяцев, приемлимо), и становится возможным проведение соревнований. Тем не менее, в процессе ее решения на сервере накапливается очередь посылок, и к концу требуется несколько часов, чтобы проверить все посланные решения. Также важно отметить, что нагрузка сервера неравномерна: если в начале он простаивает, то к концу контеста наступает пик нагрузки.

Таким образом, уже имеющиеся механизмы защиты почти не потребляют дополнительного процессорного времени, что и позволяет проводить столь масштабные контесты. Если отказаться от них и перейти к меньшему числу пользователей, одномоментно посылающих решения на проверку, то позволительны времена вплоть до 10 с, иначе необходимо уложиться в несколько секунд. Само средство безопасности должно увеличивать время работы программы не более чем в константу раз, причем желательно, чтобы эта константа не превышала 2.

Приведенные выше оценки показывают неэффективность использования полной виртуализации для данной цели, однако использование парализации является допустимым.

В данной работе рассматриваются две системы контейнерной виртуализации: OpenVZ и LXC — из-за достаточно малого времени запуска и остановки контейнеров, поддержки создания копии контейнеров на основе snapshot’ов (только в LXC; OpenVZ поддерживает лишь базовый механизм snapshot'ов), общей простоты установки, настройки и использования, свободной лицензии, совместимой с остальными компонентами Ejudge (GPL).

Для OpenVZ рассмотрены два подхода к созданию окружения: клонирование контейнера на основе скрипта, предложенного на официальном сайте **Ошибка! Источник ссылки не найден.**, или же использование одного и того же контейнера, восстанавливаемого в начальное состояние при помощи заранее сделанного snapshot'а.

Так как LXC поддерживает создание копии контейнеров на основе snapshot’ов, то другие варианты на основе этой платформы выглядят нецелесообразными.