Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Смирнов Денис Павлович

Нанесение водяных знаков на программное обеспечение

Курсовая работа

Научный руководитель:
Баклановский М. В.

Консультант:
Сибиряков А. Е.

Санкт-Петербург
2018

Содержание

Введение ... 3
1. Задачи ... 4
2. Обзор существующих техник нанесение ЦВЗ на программное обеспечение .. 5
2.1. Статические водяные знаки ... 5
2.1.1. Водяные знаки данных ... 5
2.1.2. Водяные знаки кода ... 5
2.1.3. Общие недостатки ... 5
2.2. Динамические водяные знаки ... 5
2.2.1. Пасхальные яйца ... 6
2.2.2. Водяные знаки встроенные в трассу ... 6
2.2.3. Водяные знаки встроенные в кучу ... 6
2.2.4. Общие недостатки ... 7
3. Обзор уже существующих решений ... 8
4. Портирование МАКа под Windows ... 9
5. Проектирование и реализация ЦВЗ ... 11
5.1. Требования ... 11
5.2. Типы ЦВЗ ... 13
5.3. Варианты реализации ЦВЗ ... 14
5.3.1. Связные списки ... 14
5.3.2. Решётка Кардано ... 16
5.3.3. Цепочка условных переходов ... 17
5.3.4. Лишние опкоды и «дырки» в инструкциях ... 18
5.4. Виды фрагментов водяного знака ... 19
5.4.1. Аргументы команд ... 19
5.4.2. Сами команды ... 19
5.4.3. Биты ... 20
5.5. Каркасный водяной знак ... 20
5.5.1. Исследование ... 20
5.5.2. Сравнение распределений ... 23
5.5.3. Спецификация ЦВЗ Каркас ... 24
Результаты ... 28
Список литературы ... 29

Введение

Не секрет, что повсеместное распространение интернета в последние несколько десятилетий способствовало значительному развитию общества. Однако, вместе с тем, такое быстрое расширение сети, существенно облегчив возможность передачи данных, в большой степени затруднило способность отдельно взятого человека защитить право своей интеллектуальной собственности, в связи с чем всё острее возникает потребность в современных технологиях, позволяющих бороться с компьютерным пиратством. Одной из таких технологий, становящейся всё более и более популярной в последнее время, является нанесение водяных знаков.

Термин «цифровой водяной знак» (далее ЦВЗ) был впервые использован Эндрю Тиркелем и др. [1]. – это некоторая дополнительная информация, созданная, как правило, с целью идентифицировать владельца, встраивающаяся в цифровой объект таким образом, чтобы не повредить его обычной эксплуатации. В роли цифрового объекта могут выступать электронные книги, фотографии, видео или аудио записи и др. Процесс работы с водяными знаками состоит из трех важных этапов: нанесения, эксплуатации и извлечения. На первом этапе, с помощью специализированного программного обеспечения, ЦВЗ наносится внутрь защищаемых им данных, таким образом, чтобы не ухудшить их качество. На втором этапе, цифровой объект выпускается на рынок точно так же, как если бы водяного знака в нем не было. Третий этап опционален и необходимость в нём возникает только в случае, когда происходит попытка кражи интеллектуальной собственности. Экземпляр цифрового объекта, который злоумышленник попытался использовать нелегально, снова пропускается через специализированное ПО, но теперь уже для того, чтобы извлечь из него водяной знак, который был нанесен ранее. В случае успешного завершения этой операции, ЦВЗ подтверждает право интеллектуальной собственности конкретного производителя и может использоваться им в суде, чтобы доказать свою правоту и выиграть процесс.

Такой подход обладает достаточной гибкостью и надежностью и может применяться для цифровых данных самых разных видов, в том числе и для программного обеспечения. Согласно недавним исследованиям [2] почти 40% программных продуктов, установленных на компьютерах по всему миру, являются нелицензионными, что, очевидно, приносит громадные убытки их производителям и делает защиту ПО с помощью водяного знака актуальной и нужной задачей, которая и рассматривается в данной курсовой работе.

1. Задачи

Главной целью курсовой работы является реализация прототипа программного обеспечения, позволяющего наносить ЦВЗ на секцию кода приложений на ассемблерном уровне за счет использования МАКа. Для этого были поставлены следующие задачи:

1) сделать обзор техник нанесения водяных знаков на ПО;
2) сделать обзор уже существующих решений;
3) реализовать простую версию МАКа под Windows;
4) провести декомпозицию ЦВЗ на различные виды и спроектировать их реализацию;
5) реализовать прототип ПО для нанесения ЦВЗ.

2. Обзор существующих техник нанесение ЦВЗ на программное обеспечение

При написании обзора удалось найти уже существующую курсовую работу [4] с полезной информацией по этой теме, в связи с чем удобно вести дальнейшее изложение с опорой на её структуру.

2.1. Статические водяные знаки

Статические водяные знаки – это ЦВЗ, которые можно полностью извлечь, не запуская программу. Самым простым примером такого водяного знака является информация, хранящаяся в строковых константах. Существует два типа статических водяных знаков: водяные знаки данных и кода.

1) 2.1.1. Водяные знаки данных

Это те водяные знаки, которые хранятся НЕ в секции кода. Например, можно представить ЦВЗ в виде глубоко спрятанной строки, которая возвращается, когда происходит вызов какой-то определенной, заранее созданной для этого функции.

2) 2.1.2. Водяные знаки кода

Эти водяные знаки хранятся в секции кода. Например, можно разбить ЦВЗ на части и занести их в аргументы некоторых ассемблерных инструкций в исполняемом файле, так, чтобы это не повредило семантике программы, а потом сохранить смещения до этих аргументов, чтобы их можно было найти.

3) 2.1.3. Общие недостатки

Самым существенным недостатком этого подхода является то, что такие водяные знаки могут быть найдены с помощью статического анализа кода или искажены равномерными обфускациями, сохраняющими семантику программы.

2.2. Динамические водяные знаки

Динамические водяные знаки, в отличие от статических, нельзя считать без непосредственного запуска ПО, они хранятся в некотором состоянии программы во время её работы. Общая идея состоит в том, что для того, чтобы получить доступ к конкретному ЦВЗ, программе нужно подать на вход некоторую заранее заданную последовательность данных, переводящих её в нужное состояние, из которого можно извлечь информацию.

4) 2.2.1. Пасхальные яйца

Самым известным и популярным типом динамического водяного знака являются пасхальные яйца. Они представляют собой часть кода, который

выполняется только в случае какого-то очень необычного и редкого ввода и в процессе свой работы каким-либо образом демонстрирует ЦВЗ.

Недостатки: В конечном счете, этот водяной знак тоже формируется некоторым кодом, который можно найти в статике, просто это труднее сделать. А поскольку этот код не имеет никакого отношения к остальному приложению, он может быть легко удален, без ущерба для семантики программы, как только удалось его локализовать, что ставит под сомнение стойкость этого ЦВЗ. Также, не все заказчики положительно относятся к нанесению кода, который реализует недекларированные возможности ПО, поскольку такие места в системе потенциально являются уязвимыми и могут быть атакованы злоумышленниками.

5) 2.2.2. Водяные знаки встроенные в трассу

Такой водяной знак использует трассу исполняемой программы, для какого-то заранее заданного входа. Данные могут быть спрятаны, например, в конкретной последовательности вызовов функций или обращений к стеку.

Недостатки: Хотя такой водяной знак на первый взгляд кажется очень надежным и неотделимым от программы, его использование довольно затруднительно. Во-первых, некоторые преобразования обфускаторов или компиляция с высокой оптимизацией, как и в случае со статическими ЦВЗ, способны существенно исказить поток исполнения. Во-вторых, даже если удалось найти место, которое исказить почти невозможно, следует учитывать, что любое современное ПО проходит долгий цикл разработки, в процессе которой код многократно изменяется, а значит, изменяется и трасса. Водяной знак в таком случае необходимо постоянно обновлять, внедряя в совершенно другие части программы, что делает практически невозможным разработку автоматической системы, которая может его извлечь.

6) 2.2.3. Водяные знаки встроенные в кучу

Эти водяные знаки содержатся в данных, которые создаются в динамической памяти при определенном вводе. Для извлечения необходимо проанализировать кучу с помощью специальной программы, которая сможет идентифицировать в ней ЦВЗ и вытащить из него информацию. Существует множество различных алгоритмов, работающих с разными динамическими структурами, самые распространенные из которых используют графы, чтобы скрыть в них ЦВЗ.

Недостатки: Если за генерацию динамических данных отвечает код, который ни для чего другого более не используется, то возникают все те же

проблемы, что и для пасхальных яиц. Если же модифицировать уже существующие динамические данные, то проблемы становятся очень похожими на проблемы связанные с трассой, поскольку водяной знак становится очень трудным в сопровождении.

7) 2.2.4. Общие недостатки

Помимо уже перечисленного ранее, все эти подходы содержат в себе ещё один недостаток. Возможна ситуация, при которой злоумышленник хочет украсть не всё приложение целиком, а только его часть, и в этом случае, водяные знаки будут эффективны только если они равномерно распределены по всему объему программы. Из этого соображения вытекает необходимость многократного дублирования ЦВЗ, что делает динамические реализации, являющееся более сложными и громоздкими, чем статические, ещё и куда более заметными.

3. Обзор уже существующих решений

Open source продуктов, которые реализуют нанесение ЦВЗ на код, практически нет. По-видимому это связано со спецификой данной тематики: для злоумышленника, желающего украсть программное обеспечение, защищенное водяным знаком, самым трудным является не процесс его извлечения, а попытки выяснить его вид и местонахождение. Если же существует возможность без труда получить доступ к алгоритму, с помощью которого ЦВЗ был нанесен, то эта задача становится практически тривиальной, и эффективность водяного знака падает до нуля.

Тем не менее, существует, например, проект SandMark, созданный в исследовательских целях, для защиты java-приложений. В нем реализовано несколько десятков алгоритмов обфускации и 15 разных способов нанесения водяных знаков, но, к сожалению, все они очень высокоуровневые и не подходят для использования на ассемблере. Кроме того, последняя версия SandMark выпущена в 2004 году и похоже проект больше не поддерживается.

Коммерческие продукты для нанесения ЦВЗ существуют, но их тоже немного. В пример можно привести DashO от компании Preemptive, имеющий возможность наносить водяные знаки в Java приложения или Dotfuscator, также от Preemptive, созданный для работы с .NET. Это ПО, направленное, в первую очередь, на обфускацию различных программ, но оно также может наносить водяные знаки на определенные файлы проекта, по выбору пользователя. К сожалению, информация о его алгоритмах закрыта, поэтому как-то воспользоваться ими в курсовой работе не представляется возможным.

4. Портирование МАКа под Windows

В проекте использовалась разработка кафедры под названием МАК. МАК – многоуровневая архитектура кода, ПО, позволяющее встраиваться в процесс компиляции программ и осуществлять обфускацию, профайлинг, запутывание графа потока исполнения и много чего ещё. Основная идея этой системы состоит в том, что сначала компилятор генерирует из исходника на C ассемблерный листинг в специальном формате, в котором отсутствуют команды, приводящие к нелинейному исполнению программы. Все эти команды удаляются из листинга, а участки последовательного выполнения (линейные участки), которые находятся между ними, оборачиваются в макросы начала и конца участка, которые содержат в себе информацию о командах перехода. После того, как это сделано, можно перемешивать участки, дробить их на участки меньшего размера, добавлять новые, мусорные участки, которых не было раньше, выставлять ложные условные переходы, которые будут динамически меняться на верные в ходе работы программы, а затем раскрыть макросы и скомпилировать листинг в исполняемый файл. То есть, МАК позволяет очень просто внедрять большие объемы данных прямо внутрь секции кода, не влияя на нормальную работу программы, и, вместе с тем, обфусцировать полученный код так, чтобы его было очень сложно исказить, оставив при этом работоспособным. Это означает, что с его помощью можно реализовать нанесение эффективных водяных знаков, которые будут устойчивы к большому число различных вредоносных преобразований. В рамках курсовой работы было произведено портирование небольшой части МАКа под Windows, реализующей функционал, необходимый для нанесения ЦВЗ.

С технической точки зрения, на данный момент МАК использует немного переписанный clang для генерации листинга из исходника, питоновские скрипты для его модификации, и несколько ассемблерных программ для получения из модифицированного листинга исполняемого файла. Для переноса части функционала МАКа под Windows, были написаны скрипты на питоне, обрабатывающие листинг ассемблера в Intel-синтаксисе, а также программа на ассемблере в Intel-синтаксисе, позволяющая корректно собирать PE-файлы из линейных участков, проводя обфускацию и искажая, где необходимо, поток управления. Позднее, программа была переписана на ассемблере в синтаксисе AT&T без макрокоманд компилятора gcc, используемых в версии под Linux, что позволило единообразно производить сборку clang-ом как PE, так и ELF-файлов (это не было сделано сразу, в связи с особенностями обработки ассемблерных

листингов препроцессором clang-a, с нюансами которого пришлось долго разбираться).

5. Проектирование и реализация ЦВЗ

Обзор существующих решений оказался несколько удручающим, в том смысле, что не нашлось никакой уже существующей реализации, которую можно было бы улучшить и встроить в МАК. Однако, обзор методов работы с водяными знаками в целом, оказался весьма полезным. Все последние исследования и публикации по этой теме направлены на совершенствование и усложнение алгоритмов, использующих динамические водяные знаки. Стало понятно, что статические ЦВЗ привлекают к себе мало интереса прежде всего потому, что они очень плохо себя ведут при равномерной деформации всего кода, сохраняющей его семантику. В остальном же, водяные знаки, которые многократно дублированы, раздроблены на маленькие части и разбросаны по всей программе, кажутся гораздо менее заметными и устранить их, непосредственно удалив каждый кусок, а не изменив всю программу целиком, довольно непросто. Именно это понимание и оказалось решающим для дальнейшей работы. Дело в том, что МАК позволяет организовывать код таким образом, что за разумное время, его практически невозможно деформировать, сохраняя при этом работоспособным, что открывает нам простор для изобретения и использования самых разных статических ЦВЗ.

5.1. Требования

В первую очередь нужно понимать, что водяные знаки могут эффективно выполнять свою роль только в случае, если они удовлетворяют ряду важных характеристик. Например, водяной знак в виде константной строки очень нагляден и прост для учебного примера, но совершенно не приемлем для серьезной реализации, потому что такой ЦВЗ легко найдет даже любитель. Кроме того, вообще говоря, разные виды водяных знаков могут использоваться с разными целями, некоторые из которых прямо противоречат друг другу, поэтому имеет смысл сформулировать список требований к реализации и сделать это таким образом, чтобы каждый конкретный ЦВЗ удовлетворял некоторому поднабору этих требований. Основные требования перечислены ниже.

1) Невидимость.

   Самое первое и самое очевидное требование – невидимость. Водяной знак должен быть таким, чтобы его было трудно найти, потому что, как уже указывалось ранее, это самая сложная задача на пути злоумышленника к устранению ЦВЗ.

2) Неустранимость.

   Это требование говорит о том, что водяной знак должен быть реализован

таким образом, чтобы его было затруднительно удалить различными вредоносными преобразованиями. Этот пункт во многом обеспечивается МАКом.

3) Устойчивость.
   Это требование говорит о том, что ЦВЗ должен сохраняться на протяжении всего этапа разработки программы, вне зависимости от того, какие ее части меняются, какие фрагменты добавляются, а что удаляется насовсем.

4) Защита от обвинения в мошенничестве.
   В применении водяных знаков существует один тонкий момент. В случае, если дело действительно доходит до суда, адвокат ответчика может обвинить истца в мошенничестве и попробовать доказать, что некая информация, которую преподносят как водяной знак, на самом деле им не является: это просто случайная последовательность байт и она намеренно интерпретирована производителем в свою пользу. Нужно заранее продумать формат водяного знака и/или методы нанесения и извлечения таким образом, чтобы исключить такую ситуацию.

5) Защита не только всего кода целиком, но и его частей.
   Как уже указывалось ранее, чтобы качественно защитить всё приложение, нужно покрыть его водяными знаками равномерно, а не просто записать их все в одно место.

6) Разумная трата ресурсов компьютера.
   В случае, если нанесенный водяной знак добавляет какие-то команды, которые процессор вынужден будет выполнить, это не должно существенно сказываться на производительности.

7) Возможность добавления, изменения и удаления водяных знаков.
   Водяной знак может являться не только способом идентифицировать личность владельца, но ещё и хранить внутри самой программы, например, историю ее развития. Для этого было бы очень полезно не просто разово добавлять ЦВЗ, но ещё и уметь их модифицировать.

8) Ортогональность.
   Последнее требование очень тесно связано с предыдущим. Под ортогональностью понимается независимость частей водяных знаков друг от друга. Если мы хотим нанести водяной знак на ПО, которое уже имеет водяной знак внутри себя, новый водяной знак не должен никак конфликтовать с уже имеющимся.

9) Хрупкость.
   Требование, обратное к неустранимости. Имеет смысл создавать некоторые водяные знаки так, чтобы они были максимально чувствительны к преобразованиям кода и ломались от малейшего воздействия, показывая, что программа была изменена.

5.2. Типы ЦВЗ

После того, как были сформулированы требования, можно сформулировать, какие именно типы водяных знаков мы ожидаем получить и каким требованиям они должны удовлетворять. Ниже представлен список из нескольких вариантов, с пояснениями, где необходимо.

1) Хорошо заметный ЦВЗ для отпугивания злоумышленников. Известно, что некоторые злоумышленники отказываются от идеи воровать что-либо, просто узнавая о том, что объект кражи находится под защитой. Имеет смысл создавать водяные знаки такого рода, просто для того, чтобы уменьшить количество потенциальных краж. Требования 3,5,6,7,8.

2) ЦВЗ заведомо небольшого объема, предназначенный исключительно для идентификации права собственности. Требования 1-6,8.

3) ЦВЗ, обладающий большой информационной емкостью, для сохранения истории развития проекта. Требования 1-8.

4) Хрупкий ЦВЗ, идентифицирующий попытку взлома. Требования 1,3,5,6,8,9.

5) Хрупкий ЦВЗ, “взрывающий бомбу” в случае взлома. Можно специально наносить в некоторые места программы код, чувствительный к вредоносным преобразованиям, удаление или изменение которого приводит к аварийному завершению работы программы. Это так же нужно для уменьшения количества потенциальных краж, за счет повышения сложности их осуществления. Требования 1,3,5,6,8,9.

6) Цифровая подпись.
   Хранит в себе небольшое количество информации о текущей сборке

(прим. версия компилятора, линковщика, дата компиляции). Похож на 2), но отличается тем, что 2 и 5 требования не так критичны, используется скорее самим разработчиком для поиска информации о происхождении конкретного файла.
Требования 1-3,5,7,8.

7) Каркасный водяной знак.
Наносится на файл неустранимым образом, так, чтобы с его помощью можно было эффективно нанести другие водяные знаки.
Требования 1-3,5-8.

5.3. Варианты реализации ЦВЗ

После того, как были сформулированы различные типы ЦВЗ, можно было приступить к техническим идеям по реализации. Ниже представлены некоторые из них.

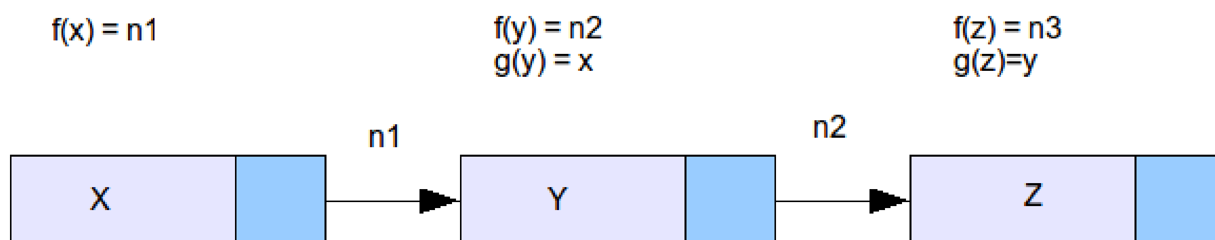
8) 5.3.1. Связные списки

Может реализовывать ЦВЗ 2) и 3).

Весь файл разбивается на n последовательных участков примерно равного размера. Мы идентифицируем начало каждого участка, а затем сохраняем несколько смещений от начала какого-нибудь из участков. Каждая пара (начало участка, смещение) указывает на голову списка, все списки дешифруются независимо друг от друга, каждый список целиком лежит в одном из n участков файла и отвечает за свой собственный набор данных. Каждый узел списка хранит внутри себя часть ЦВЗ и информацию о смещении до следующего узла. Извлечение происходит так: мы находим узел и извлекаем из него данные, часть данных это ЦВЗ, а другая часть -- новое смещение до следующего узла. Узел может быть расположен выше или ниже текущего, это либо хранится в самом узле, либо заранее придумывается несколько схем чередований, например:

1) идем с периодичностью вниз, вниз, вниз, вверх;
2) идем с периодичностью вниз, вниз, вверх, вверх;
3) идем с периодичностью вниз на величину, равную удвоенному смещению, вниз, вверх, вниз.

В этом случае для каждого списка хранится ключ, обозначающий конкретную схему. Это позволяет узлам списка пересекаться друг с другом, уменьшив количество узлов, потому что мы сможем по-разному интерпретировать смещения, которые в них находятся, а значит выбирать разные узлы после текущего. Это к тому же затрудняет распознавание списка в коде программы.



(рис. 1)

Список может быть двусвязным (рис.1), это можно организовать так: перед занесением ЦВЗ нам нужно выбрать две функции, f и g. Далее, мы выбираем произвольное значение Z и с помощью g генерируем цепочку Z, g(Z) = Y, g(g(Z)) = X и т.п.. В результате получается следующее: если X – данные, которые хранит в себе первый узел, мы применяем к нему f, чтобы получить расстояние n1 до второго узла, в котором лежит значение Y. Применив f к Y, мы снова получим расстояние n2 до следующего узла Z, однако применив к Y функцию g, мы получим X, поэтому композиция f(g(y)) = n1. Это позволяет нам идти по списку в обратном направлении, не сохраняя в коде дополнительных смещений, зная только функции f,g и значение узла.

Вдобавок, в каждом списке может храниться несколько узлов, которые содержат не ЦВЗ, а смещение от данного узла до какого-нибудь ближайшего узла другого списка (если список двусвязный, от любого узла можно совершить полный обход), как дополнительный уровень защиты на случай, если смещения для голов испортятся.

Недостатки: Добавление или удаление команд портит смещения, что частично компенсируется расположением списков в заранее отделенных друг от друга блоках так, чтобы испорченные смещения в одном из блоков не повлияли на остальные.

9) 5.3.2. Решётка Кардано

Может реализовывать ЦВЗ 3) и 4).

Sir John regards you well and spekes again that

(рис. 2)

Весь файл разбивается на n последовательных участков, мы как и в предыдущем случае, запоминаем начала участков так, чтобы их потом можно было найти. Следующая идея аналогична применению в стеганографии решётки Кардано (поворотной решётки) [3] (рис. 2). Смысл этого подхода в том, что на текст накладывается лист бумаги с вырезанными в нём отверстиями (решётка), через которые видно только некоторые буквы. Эти буквы являются секретным сообщением, прочитать которое можно, только если существует доступ к соответствующей решетке. Понятно, что с тем же успехом можно просто сохранить номер буквы, соответствующей началу каждого отверстия, и число букв, которые в него помещаются, чтобы использовать вместо решётки такие числовые значения, в чём и заключается основная идея.

Мы разбиваем наш водяной знак на части и для каждой части ищем в участке скомпилированного бинарника, соответствующего одному из n участков ассемблерного исходника, фрагмент, который побитово совпадает с частью ЦВЗ. Если такого фрагмента не найдется или их найдется недостаточно, мы находим все, которые можем, а потом возвращаемся на уровень ассемблера и искусственно добавляем кода так, чтобы фрагментов хватило. После этого мы снова компилируем файл и запоминаем смещения от начала участка до каждого из фрагментов. После этой процедуры, конкретный участок файла содержит в себе части водяного знака, который можно найти зная смещение до начала фрагмента. Также, помимо бит водяного знака, каждый фрагмент должен хранить биты контрольной суммы. В момент считывания ЦВЗ, мы сначала проверяем их, и если они не совпадают, значит произошло некоторое вредоносное преобразование. В этом случае, мы можем попробовать “сдвинуть решётку”, то есть, попытаться различными смещениями добиться того, чтобы контрольная сумма совпала, после чего произвести считывание. Если за разумное число сдвигов водяной знак извлечь не удается, он считается утраченным.

Кроме того, одна «решётка» может хранить в себе лишь небольшой фрагмент текста водяного знака, в этом случае отдельно должна сохраняться информация о том, в какой последовательности объединять фрагменты из

решёток. Процедуру объединения можно делать в несколько уровней: сначала объединим в пару две “решётки”, затем объединим две пары в четвёрку, затем две четвёрки в восьмерку и т.д. В “решётках” могут быть контрольные биты для каждого из этапов объединения, и водяной знак считается корректно считанным, если на каждом этапе все суммы сошлись. Если же произошла ошибка, мы возвращаемся на самый первый уровень, и пытаемся сместить решётку так, чтобы первая контрольная сумма снова совпала, после чего повторяем процедуру.

Недостатки: Добавления и удаления команд портят смещения, как и в первом случае, в данном подходе компенсировать это помогают контрольные суммы и разбиение программы на независимые участки.

10) 5.3.3. Цепочка условных переходов

Может использоваться для ЦВЗ 6). Необходимым условием является обфускация потока исполнения, в противном случае его неустранимость почти нулевая.

С помощью МАКа мы дробим весь ассемблерный текст на участки размера 5-7 команд, а затем перемешиваем их в определенном порядке, так, чтобы закодировать исходное сообщение командами условных переходов. Каждый переход между линейными участками кодирует один бит. Если переход был вверх от текущего линейного участка, это 1, если вниз, это 0. Можно добавить помимо направления перехода вторым параметром его длину, это позволит кодировать одним переходом не один бит, а 2, 3 и т. д. в зависимости от того, насколько большой разброс длин мы имеем возможность сделать. Использовать стоит в случае, если дополнительно, вместе с перемешиванием, происходит обфускация потока исполнения, добавляющая динамический патчинг условных переходов, иначе участки легко можно объединить друг с другом простым скриптом, и вся информация о ЦВЗ потеряется. Плюсом ЦВЗ является то, что в случае, если используется кодирование 1 бит – 1 переход, водяной знак совершенно не чувствителен к вставке нового кода.

Недостатки: Если используется кодирование n бит – 1 переход, добавление кода портит смещения и ЦВЗ искажается. В случае же 1 бит – 1 переход, с его помощью можно записать не так много информации.

11) 5.3.4. Лишние опкоды и «дырки» в инструкциях

Ранее уже говорилось о том, что водяные знаки применяются не только для того, чтобы сохранять в них данные. Одной из разновидностей водяных знаков является “хрупкий” водяной знак (fragile watermark в английской литературе). По

его названию можно понять, что он специально сделан так, чтобы ломаться при самом лёгком преобразовании, и используется для того, чтобы подтвердить целостность исходного цифрового контента. В случае, если производителю понадобится такой водяной знак, предлагается использовать следующий подход:

Как известно, компилятор переводит каждую ассемблерную инструкцию в последовательность определенных байт специального формата. Но так как трансляция должна быть универсальной, существует некоторый набор инструкций, которые переводятся в такие последовательности байт, что некоторые их биты не используются при интерпретации команды процессором. Иными словами, компилятор порождает последовательность байт, в часть битов которой можно записать что угодно, не испортив при этом команду. Иногда перед командой можно вставить несколько опкодов, которые также не повредят ее интерпретации. При дизассемблировании же такой программы, измененная команда прочитается как самая обычная ассемблерная инструкция, и после повторной компиляции приведется к стандартному виду. Таким образом, если мы внесем некоторые правки в инструкции сохраним смещения до этих исправленных мест, используя эту технику, мы можем выставить флаги, которые буду говорить нам о том, был в программе взлом или нет. Плюсом такого ЦВЗ является то, что он очень просто наносится.

Недостатки: Главный плюс может стать главным минусом, в случае, если злоумышленник поймет, что некоторые команды специально изменены. Это знание позволит ему легко имитировать такой ЦВЗ, стоит, однако, сказать, что заметить это довольно непросто, для этого нужно намеренно изучать программу на самом нижнем ее уровне – битовом.

5.4. Виды фрагментов водяного знака

Схемы нанесения определяют места, в которые внедряются части ЦВЗ, и способы, с помощью которых их можно собрать в исходное сообщение. Но помимо конкретных мест, нужно знать ещё и в каком представлении их туда сложить.

12) 5.4.1. Аргументы команд

Поскольку МАК позволяет встраивать дополнительные инструкции в исполняемый код, самым очевидным представлением ЦВЗ является хранение данных в аргументах ассемблерных команд. Для этого мы заранее (отдельно для каждого списка, например) определяем в каких конкретно битах аргумента будет лежать информация и просто помещаем её туда. Можно использовать формат

фрагмента, аналогичный юникоду, в котором определенный префикс указывает на количество байт, которыми кодируется символ. При таком подходе можно размещать ЦВЗ в нескольких командах подряд: найдя при считывании начало фрагмента с водяным знаком, мы можем считать его первую часть, и определить по ней, что в следующей команде будет продолжение.

Помимо мусорных команд, можно также изменять команды реального кода так, чтобы не ломать семантику, но добавлять водяной знак. Это сделает водяной знак похожим на часть логики программы и затруднит его идентификацию.

13) 5.4.2. Сами команды

Водяным знаком также может быть и сама команда. Существует, например, много разных видов команды mov, которые определяются типом операнда и его размером. Можно, например, сопоставить команде mov r64, r64 битовую последовательность 0000, а команде mov r64, m64 – 0001 и т.д. Проделав это для самых разных команд, мы получим много наборов, каждому из которых можем присвоить ключ, и сохранять его вместе с данными о конкретном списке, решётке и т.п. Тогда в момент считывания, мы просто будем знать, какой из наборов использовать, и сможем легко интерпретировать команды как конкретные цепочки битов.

14) 5.4.3. Биты

В редких случаях данные можно просто записывать побитово, не заботясь о том, какие команды они порождают, при условии, что фрагменты расположены в участке кода, на который никогда не передается управление. Но злоупотреблять таким не стоит, поскольку места со странными и нелогичными командами сразу будут привлекать к себе внимание.

5.5. Каркасный водяной знак

После того, как были спроектированы различные варианты ЦВЗ, стало понятно, что самой сложной частью каждой из реализаций является идентификация местоположения конкретного фрагмента водяного знака. Вообще говоря, совершенно непонятно, каким образом можно сохранять эту информацию. Злоумышленник может добавлять новые данные, менять части кода местами, модифицировать отдельные фрагменты, всё это негативно сказывается на сохраненных смещениях и они становятся бесполезны. Это проблема может быть частично решена на небольших фрагментах файла за счет независимости относительных смещений от остальных фрагментов, кодов исправления ошибок и множества попыток извлечения, однако в таком случае

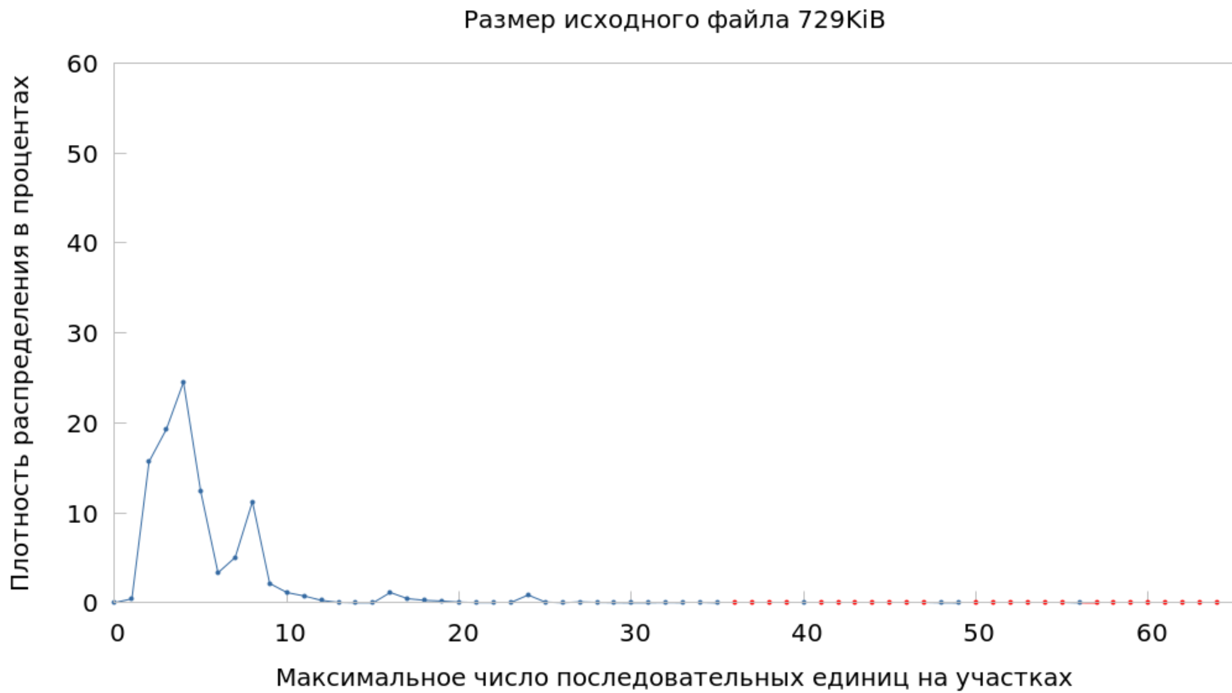
нужен способ, с помощью которого можно разбить файл на различные фрагменты и уметь однозначно их идентифицировать. Опять же, с учетом того, что части файла могут быть перемешаны, просто сохранять в определенных местах какие-то метки не получится, нужно было придумать способ маркировать фрагменты файла так, чтобы даже по небольшому кусочку уметь установить принадлежность к определенному фрагменту. Постепенно пришло понимание, что такая маркировка – это самый базовый, каркасный водяной знак, который должен быть максимально устойчивым к преобразованиям и поверх которого будут наноситься все остальные водяные знаки, без него эффективное нанесение осуществить не удастся. Этот ЦВЗ удалось реализовать после исследования структуры секций кода на битовом уровне.

15) 5.5.1. Исследование

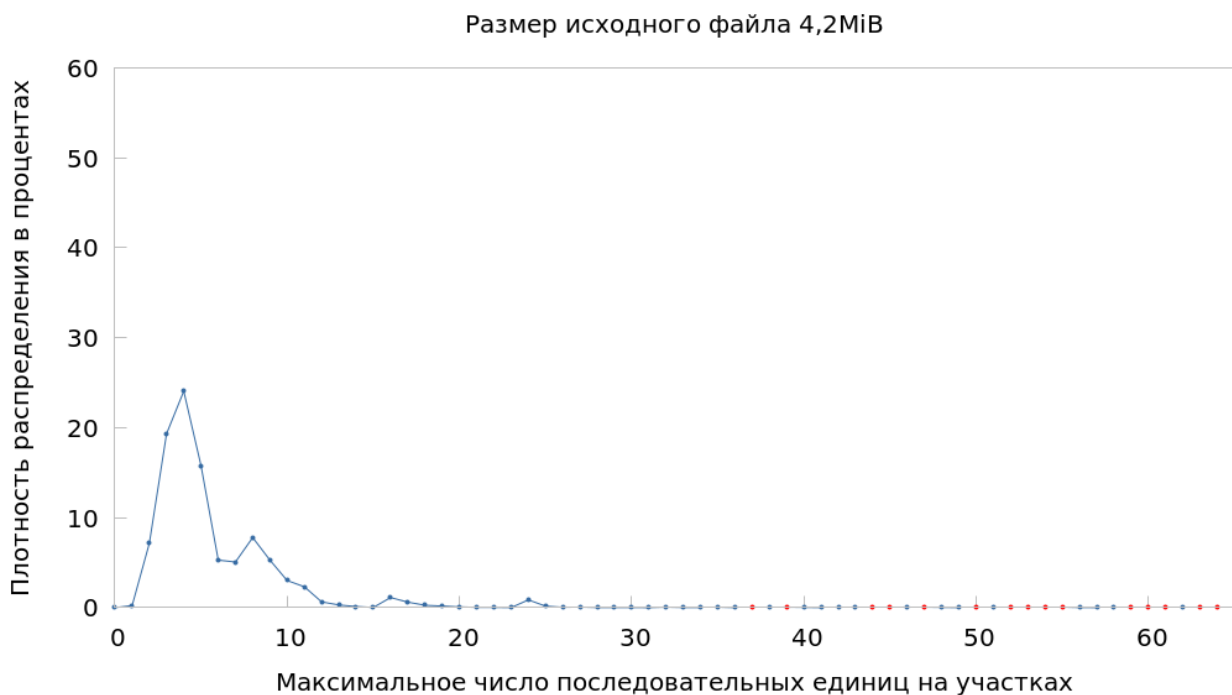
Идея реализации заключалась в следующем. Нужно взять фрагмент, который мы хотим идентифицировать, и разбить его на маленькие кусочки одинаковой длины, скажем, 8 байт. Затем, на каждом из этих кусочков мы посчитаем некоторые случайные величины, основанные на различных битовых комбинациях. (Примером такой случайной величины может являться, скажем, количество чередующихся единичных бит, максимальное количество идущих подряд единиц и др.). После этого мы получим выборку из значений каждой случайной величины на кусочках нашего фрагмента и сможем составить для каждой из величин её распределение. Далее эти распределения мы можем сравнить друг с другом и посмотреть, ведут ли они себя похожим образом. Если нет, никакой информации о том, как те или иные команды влияют на структуру файла получить не удастся, а значит, мы не сможем заранее предсказать, как изменится файл после некоторого вредоносного преобразования. Но если они ведут себя похожим образом, это будет означать, что самые часто встречающиеся команды генерируют такой битовый код, что влияние остальных команд оказывается несущественным и все файлы выглядят примерно одинаково, то есть, преобразование, состоящее из ассемблерных инструкций, не сильно исказит случайные величины.

Информации о том, как же есть на самом деле, нигде найти не удалось, поэтому было проведено самостоятельное исследование. Был осуществлен подсчет 8 разных случайных величин для 450 файлов разных архитектур на кусочках 7 разных длин. Исследование показало, что файлы ведут себя очень похожим образом и подчиняются определенным паттернам, например, на всех файлах есть определенные значения величин, которые встречаются крайне редко, вне

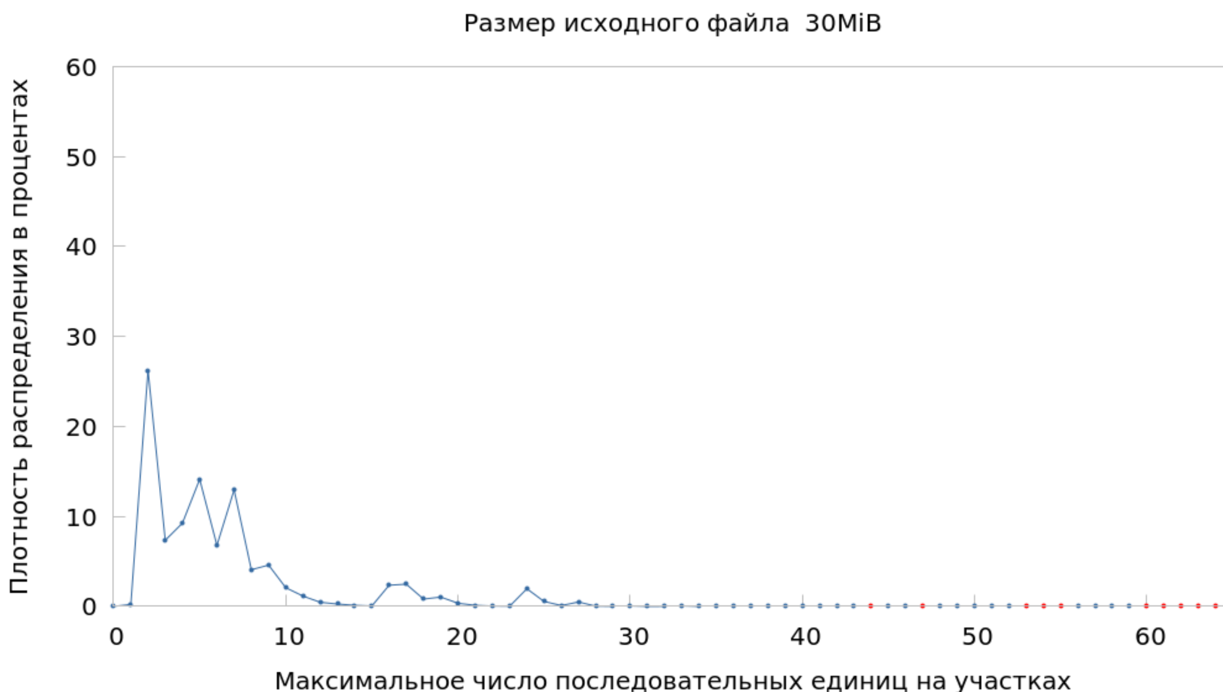
зависимости от архитектуры, размера, и характера исходника. Ниже для примера приведены несколько полученных графиков плотностей распределения, для одной конкретной случайной величины посчитанных на совершенно разных файлах, легко видеть, что для данной случайной величины характерны, например, спады и подъемы в определенных местах (это сохраняется на всех графиках, не только на приведенных тут).



(рис. 3)



(рис. 4)



(рис. 5)

После того, как были получены эти результаты, родилась идея реализации каркасного водяного знака.

5.5.2. Сравнение распределений

Перед непосредственной реализацией, стоит поговорить ещё об одном моменте. Идентификация фрагментов с помощью распределений предполагает, что мы умеем сравнивать распределения между собой. Значит, нам нужна некоторая метрика, позволяющая определять, насколько одно распределение похоже на другое. Задача о том, чтобы определить тип распределения по некоторой выборке решается методами математической статистики. Однако, стандартный для этой задачи критерий хи-квадрат [5] плохо работает с получающимися случайными величинами, во-первых, потому что после вредоносных преобразований выборка получается даже больше чем начальная, по которой строится теоретическое распределение для сравнения, а во-вторых, потому что некоторые из значений случайной величины встречаются в десятки раз реже, чем другие, и их вероятность получается очень маленькой, что вызывает затруднения. После нескольких неудачных попыток воспользоваться критерием для построения метрики между распределениями, был опробован другой подход, который, в итоге, сработал.

В качестве метрики была взята корреляция [6]. Понятно, что если две плотности распределения похожи друг на друга, то обе они будут увеличиваться и уменьшаться примерно одновременно, это наблюдение позволило использовать в качестве новой случайной величины значения плотностей распределения на каждом фрагменте файла. При считывании ЦВЗ мы идем по файлу и считаем случайные величины на разных его частях, после чего определяем значения плотностей распределения и вычисляем корреляцию этих значений между фрагментами файла для считывания и плотностями, которые мы сохранили при нанесении каркаса. Фрагменты, показывающие высокую корреляцию, с большой вероятностью и есть семантически одинаковые части кода, которые мы хотели идентифицировать.

5.5.3. Спецификация ЦВЗ Каркас

Ниже расположен текст подробной спецификации водяного знака, по которой была осуществлена реализация.

Примечание: под случайной величиной, используемой для нанесения каркаса, подразумевается значение некоторой битовой комбинации, посчитанное на произвольном фрагменте файла (например, максимальное число чередующихся битов на участке длиной 8 байт).

Нанесение

1. Исходник на C, на который планируется нанесение, транслируется в ассемблер, затем происходит дробление на линейные участки размером в 5 команд и сборка с помощью мака. В процессе сборки генерируется листинг с информацией о точных смещениях до линейных участков от начала файла.

2. Из бинарника извлекается информация о смещении до секции .text и ее размере.

3. Из листинга, полученного в первом пункте, извлекается информация о смещениях до начал линейных участков, лежащих в секции .text (на основе информации полученной в пункте 2). Затем генерируется набор пар, в котором в качестве первого элемента используется смещение от начала секции кода, а в

качестве второго -- номер линейного участка, после чего пары упорядочиваются по возрастанию по первому элементу. Далее, полученный список пар разбивается на k равных частей, где k – число фрагментов файла, на которые наносится каркас, после чего сохраняется информация о номерах линейных участков, лежащих в определенном фрагменте файла.

4. Формируется набор функций распределения N значений дискретных случайных величин, где N – количество значений случайных величин, которые будет искажать каркас. Поскольку каждая такая функция распределения полностью задается значениями на конкретных аргументах, этот пункт эквивалентен формированию набора из упорядоченных по возрастанию списков из N значений от 0 до 1 включительно, при этом последний элемент списка всегда равен – 1 (на практике его можно не хранить, подразумевая, что это значение всегда известно).

5. На каждый фрагмент файла происходит независимое нанесение каркаса.

Для каждого фрагмента:

Из сохраненной ранее информации извлекаются номера участков, находящихся во фрагменте, после чего между участками ассемблерного листинга с этими номерами происходит внедрение данных, генерируемых по алгоритму, описанному ниже.

1) Выбирается несколько случайных величин для нанесения каркаса и длина фрагментов файла, на которых будут считаться случайные величины.

2) Каждой случайной величине сопоставляется некоторая функция из набора функций распределения, так, чтобы выбранное множество функций было уникальным для каждого фрагмента файла. После этого каждый фрагмент файла будет однозначно идентифицироваться по конкретному набору распределений этих случайных величин.

3) Для каждой случайной величины выбирается ровно N значений, которые будут изменены, а затем для каждого значения i генерируется m битовых цепочек, значение случайной величины на которых равно i. Количество (то есть m) битовых цепочек для каждого значения

определяется функцией распределения сопоставленной данной случайной величине.

4) После этого битовые цепочки записываются между линейными участками, содержащимися в текущем фрагменте файла, тем самым меняя распределение конкретных случайных величин и идентифицируя тем самым конкретный участок файла. После этого происходит нанесение ЦВЗ на следующий фрагмент.

6. Полученный файл снова собирается маком.

Для каждого из k фрагментов файла:

Для каждой случайной величины:

1) фрагмент разбивается на последовательные участки длины, выбранной в пункте 5.1;

2) для каждого участка подсчитывается значение случайной величины на каждом участке;

3) составляется плотность распределения случайной величины и сохраняется в отдельный файл, который будет использоваться при считывании ЦВЗ.

Замечание: плотности N измененных значений, вероятно, будут немного отличаться от плотностей при нанесении, что вызвано неточным совпадением участков нанесения и участков считывания.

7. Водяной знак нанесен. Файл с информацией о плотности распределения является ключом, по которому можно извлечь цвз, без него извлечение невозможно.

Извлечение

1. Из файла извлекается секция кода.

2. Пусть P -- размер в байтах i-ого участка каркаса. Секция кода разбивается на P частей, на каждой из которых считаются случайные величины, тем же способом, что и при нанесении.

3. Считается корреляция между значениями распределений случайной величины в каждой из P секций и i-м участком.

4. Выбираются все части, с корреляцией больше 0,85, после чего границы частей раздвигаются, увеличивая секцию на 10%, и случайные величины пересчитываются снова. Если корреляция улучшилась, границы снова раздвигаются, и процесс происходит итеративно, пока корреляция не ухудшится. Затем проделывается та же процедура, но с сужением границ.

5. Результатом выполнения операции является такая область файла, которая имеет бОльшую корреляцию с i-м сектором на протяжении всей операции.

6. Проделав это для k участков, мы получим набор корреляций каждой области файла с некоторым поднабором участков каркаса. Выбрав максимум по корреляции для каждой области файла мы получим его предположительное разбиение на участки первоначального файла.

Результаты

Основные результаты курсовой работы:

1) сделан обзор техник нанесения водяных знаков на ПО;
2) сделан обзор уже существующих решений;
3) реализована упрощенная версия МАКа под Windows;
4) произведена декомпозиция ЦВЗ на различные виды и спроектированы конкретные реализации;
5) реализован прототип ПО, позволяющий наносить ЦВЗ, идентифицирующий местонахождение конкретных участков файла после вредоносных преобразований.

Дополнительные результаты, возникшие в процессе реализации ЦВЗ:

1) проведено исследование внутренней структуры секции кода исполняемых файлов, выявлены закономерности, порождаемые часто встречающимися командами;
2) получен инструмент сравнения исполняемых файлов друг с другом.

Список литературы

[1] A. Z. Tirkel, G. A. Rankin, R. M. V. Schyndel, W. J. Ho, N. R. A. Mee, and C. F. Osborne, “Electronic watermark,” in Digital Image Computing, Technology and Applications (DICTA’93), pp. 666–673, 1993.

[2] BSA Global Software Survey, May 2016, URL: http://globalstudy.bsa.org/2016/downloads/studies/BSA_GSS_InBrief_US.pdf. Дата обращения: 31.12.2017

[3] Введение в криптографию / Под общ. ред. В. В. Ященко. — 4-е изд., доп. М.: МЦНМО, 2012. — 348 с., страница 189

[4] Imran Ali, Software Watermarking, CSEP 590TU, 2006 URL: https://courses.cs.washington.edu/courses/csep590/06wi/finalprojects/ali.doc. Дата обращения: 31.12.2017

[5] V. Bagdonavicius, M. S. Nikulin Chi-square goodness-of-fit test for right censored data // The International Journal of Applied Mathematics and Statistics.— 2011— страницы 30-50

[6] Гмурман В. Е. Теория вероятностей и математическая статистика: Учебное пособие для вузов. — 10-е издание, стереотипное. — Москва: Высшая школа, 2004. — 479 с.