САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МАТЕМАТИКО-МЕХАНИЧЕСКИЙ ФАКУЛЬТЕТ
КАФЕДРА СИСТЕМНОГО ПРОГРАММИРОВАНИЯ

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
НА ТЕМУ:

«Сравнение потоков исполнения программ на основе списка системных вызовов»

Выполнил:
студент группы ИТ-661
Ханов А.Р.

Научный руководитель: Баклановский М.В.

Рецензент, д.т.н, профессор,
Зав. лабораторией информационных технологий
в управлении и робототехнике СПИИРАН Тимофеев А.В.

“Допущен к защите”
Заведующий кафедрой д.ф.-м.н., профессор Терехов А.Н.

Санкт-Петербург
2012 год

Оглавление

Введение 3
Обзор 4
Постановка задачи 8
Методы решения 8
Вычисление частной энтропии 10
Описание 10
Тесты 12
Корреляция n-грамм 14
Описание 14
Тесты 15
Выделение термов 17
Описание 17
Реализация 18
Тесты 20
Заключение 24
Список литературы 25
Приложения 27
Приложение 1. Оценка частной энтропии вызовов некоторых процессов . 27
Приложение 2. Идентификация процессов 28
Приложение 3. Дополнительный алгоритм фильтрации термов 29
Приложение 4. Псевдокод алгоритма фильтрации термов 30

Введение

Анализ программ является важнейшей областью исследований. Он включает в себя получение данных о программе и их интерпретацию.

Существует множество методик и инструментов для выполнения анализа программ на различных языках под всевозможные платформы. Одни из них встроены в свою среду разработки, другие доступны как системные утилиты. Они создаются для разных целей: это и поиск ошибок в программе, таких как ошибки памяти, переполнения, взаимоблокировки, и анализ производительности кода, и тестирование на соответствие формальной спецификации, и поиск вредоносных кодов.

Зачастую у нас имеется лишь бинарный файл с программой или даже просто процесс в системе, и мы не имеем возможности получить доступ к коду, на том языке, на котором была написана программа, или отладочной информации, не можем воспользоваться теми средствами анализа, которые были доступны в процессе разработки. В таком случае анализ программы усложняется тем, что его нужно проводить независимо от той технологии, с помощью которой она была создана. Приходится рассматривать программу как объект, работающий в системе.

Анализ программ как системных объектов производит как сама система для обеспечения своей работы, так и специальные утилиты. Но существующие решения, как правило, решают свои узконаправленные задачи и не позволяют решить главную проблему: мы не можем отличать программы друг от друга. Программа представляется для нас «черным ящиком», который взаимодействует с системой. Система собирает информацию, нужную ей для управления ресурсами: информацию для планировщика, для управления виртуальной памятью, и т.д. Но она не содержит никакой цельной модели исполняемой ею программы. Она не знает, выполняет ли программа полезную работу или находится в блокировке, не знает, какие действия может совершать программа, и какие потоки и трассы исполнения ей свойственны. В системе нет надежных средств идентификации программ, мы не можем установить, что два запущенных в ней процесса это одна и та же программа.

Эта проблема особенно остро влияет на безопасность. Существует множество публикаций на тему обнаружения вредоносных кодов, в которых строится модель, позволяющая различать программы.

В данной работе будут представлены алгоритмы построения моделей программ на основе информации о производимых ими системных вызовах. Заранее не было известно, можно ли вообще извлечь из этих данных какую-либо полезную информацию. Теперь мы можем точно сформулировать те

задачи, которые нам удается решать. Во-первых, это проверка эквивалентности двух процессов, то есть установление того, что два процесса являются одной и той же программой. Во-вторых, мы можем устанавливать, в какие периоды своей работы два потока производили одни и те же действия. В-третьих, нам удается выявлять те состояния потока, состояния активности, когда он совершал неожиданные, не свойственные ему действия. Сами алгоритмы будут испробованы на операционной системе Windows 7, хотя они никак не привязаны к данной платформе и могут быть использованы и на других.

Обзор

Программа – это не только процесс, запущенный из исполняемого файла, но и код для виртуальной машины, драйвер, задача в операционной системе, состоящая из нескольких процессов. С понятием программы связано большое число возможных объектов в системе. Одни программы работают в пакетном режиме, занимаются в основном вычислениями и их взаимодействие с системой не так важно, другие являются ее неотъемлемой частью и серьезно влияют на ее работу. Под программой мы будем понимать процесс, который она запускает. Одна программа может состоять из нескольких процессов, но мы не будем рассматривать такие случаи. Мы будем изучать не процессы в целом, а потоки, принадлежащие этим процессам. Программа рассматривается нами как множество потоков исполнения.

Поток включает в себя множество связанных с ним объектов, но мы будем рассматривать лишь код, который он исполняет. Каждый поток мы будем рассматривать как набор инструкций на машинном языке. Это исходный код потока. Когда мы исследуем этот код без его запуска в какой-либо системе, то мы проводим так называемый статический анализ. Если мы следим за кодом в процессе его исполнения, то мы выполняем динамический анализ. У этих видов исследования программ есть свои преимущества и недостатки.

Статический метод исследования обычно связан с изучением самого файла с программой. В качестве данных может быть взят не только код, но и метаинформация: отладочная информация, информация об импорте, поля инициализированных данных и т.д. Исследование программы, основанное на исходном коде, не может решать всех проблем. Например, мы не можем, имея два участка кода, сказать, что они функционально эквивалентны друг другу. Также мы не можем сказать, что код вообще закончит свою работу. Код может формироваться в процессе исполнения, поэтому возможности

статического анализа сильно ограничены. Но исследования в этом направлении все же ведутся. Так в [1] приведен метод построения моделей программ по коду с целью проверки их эквивалентности по нескольким группам преобразований: перестановка независимых команд, вставка фиктивных команд.

Динамическое исследование программ осложнено тем, что мы должны вмешиваться в работу программы в процессе ее исполнения, что может привести к ее неправильному функционированию или падению производительности. Существуют специальные техники, позволяющие получать различные данные о работающей программе. Мы можем получать доступ к ее адресному пространству и делать дампы памяти, исследуя ее код и данные в процессе работы самой программы. Для сбора более детальной информации нужно перевести программу в отладочный режим, в режим трассировки, ставить точки останова на память или код. Сама трассировка это очень медленный процесс, который, однако, дает нам всю информацию о ходе исполнения, которую мы можем получить в пространстве пользовательского процесса. Другой способ получения данных связан с внедрением в адресное пространство процесса внешнего модуля, который может, например, собирать информацию о вызываемых системных функциях. Такой модуль может даже проводить динамическую трансляцию кода программы и следить за его исполнением. Например, библиотека PIN [2] разрабатывается для упрощения отслеживания событий, которые происходят в программе в процессе ее исполнения, и позволяет собирать данные о вызовах системных функций, трассе исполнения, об обращениях к памяти,…

Чем более детальную информацию мы собираем о программе, тем больше мы замедляем ее исполнение. Программа может выполнять миллионы API-вызовов в секунду, но не за всеми из них действительно необходимо вести наблюдение. Поэтому при проведении динамического анализа стоит соблюдать баланс между точностью описания работы процесса и падением производительности из-за наших действий по сбору данных.

В [3] приведен метод, позволяющий перехватывать системные вызовы процессов. API-вызовы – это функции из системных библиотек. Сами эти вызовы нередко зависят друг от друга. Некоторые из этих функций взаимодействуют с системой с помощью вызовов, совершаемых в ядро (syscall). Такие вызовы может совершать и

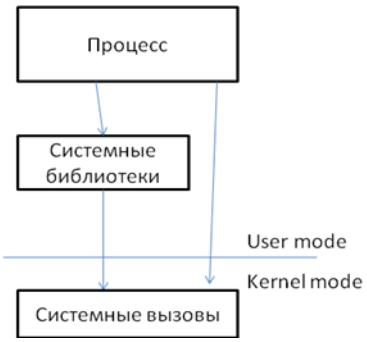


Рисунок 1. Схема вызовов

сама пользовательская программа, но, как правило, это делается через «обертку» из системных библиотек. Вызовы в ядро – это наиболее важные операции, совершаемые программой, способные влиять на систему. С помощью метода, описанного в [3], можно перехватывать системные вызовы, практически не замедляя работу системы. Таким образом, мы можем отслеживать работу сразу всех процессов и потоков. Если мы не изучаем работу исходного кода программы и следим лишь за тем, как процесс взаимодействует с системой, то такой вид анализа называется поведенческим.

На основе собранных данных строится модель программы. Она необходима для интерпретации этих данных. Модель программы – это некоторая структура, которая описывает существенные характеристики ее работы. Ее устройство зависит от назначения модели. Например, [4] исследуется возможность предсказания условных переходов по их истории. При этом строится контекстная модель PPM для описания каждого условного перехода. Существуют специальные модели для проверки блокировок в потоках, верификации, и т.д.

В данной работе изучается, прежде всего, проблема идентификации программ. Соответствующие модели следует искать в области, где эта проблема встает особенно остро, – в информационной безопасности.

Самые простые вредоносные программы можно идентифицировать, посчитав контрольную сумму по самим исходным файлам или по различным областям памяти. Традиционным методом идентификации является поиск характерных для вредоносных программ последовательностей инструкций [5]. Более сложные методы статического анализа связаны с верификацией моделей [6], символьным исполнением [7].

Но производители вредоносных программ создают все новые методы противодействия анализу кода.

Обфускация – это приведение программы к другому виду, который более сложен для анализа, но функционально эквивалентен исходной программе. Про методы обфускации можно узнать в [14].

Упаковка программ – это сжатие или шифрование программы с последующим дописыванием к ней специального кода, который расшифровывает и запускает ее. Некоторые упаковщики расшифровывают и зашифровывают код настоящей программы в процессе ее исполнения. Это позволяет сильно затруднить анализ программы, так как для доступа к оригинальному исходному коду необходимо вначале провести расшифровку. Вредоносные программы могут изменять свой запакованный образ после каждого заражения. Они называются полиморфными.[12]

Существуют такие преобразования кодов программ, которые сохраняют ее работоспособность, но видоизменяют ее код с помощью специальных операций, таких как вставка фиктивных инструкций или их замена на эквивалентные наборы команд. Такие вредоносные программы, которые при заражении изменяют свой код, называются метаморфными.

Динамический подход к построению моделей программ позволяет преодолеть многие сложности, возникающие при статическом анализе.

В [8] представлен метод построения графов зависимости API-вызовов по данным. Для проверки эквивалентности программ решается задача проверки изоморфизма соответствующих им графов, и поэтому метод обладает высокой вычислительной сложностью.

В [9] используется динамическое выделение определенных последовательностей машинных инструкций для формирования вектора признаков, по которым в дальнейшем происходит классификация программ. Но, как и большинство методов, основанных на анализе исходного кода, он не устойчив к преобразованиям программ к эквивалентному виду.

Поведенческий анализ программ позволяет избежать проблем с анализом исходного кода. В [10] представлен метод поиска ошибок, основанный на построении модели программы по итеративным шаблонам, которые являются аналогами программных шаблонов. Трасса исполнения программы – это последовательность генерируемых ею событий. Шаблон – это регулярное выражение для событий в трассе. Существует алгоритм, позволяющий собирать все шаблоны как признаки и с их помощью классифицировать трассы исполнения программ. Метод классификации трасс исполнения программ, основанный на этих шаблонах, изначально был создан для поиска ошибок и позднее успешно применен в [11] для распознавания вредоносных кодов. В качестве событий брались API-вызовы. Этим методом удалось достичь точности около 90%.

Последний из описанных методов является достаточно общим случаем, так как позволяет использовать различные события, которые могут произойти при исполнении программы. Он изначально рассчитан не только на опознание, но и на интерпретацию выделяемых особенностей для поиска ошибок в трассах исполнения.

Методы проверки эквивалентности программ уже достаточно развиты. Существующие модели имеют свои достоинства и недостатки. После их изучения были сделаны следующие выводы. Во-первых, время, необходимое для сбора данных о программе и вычислительная сложность построения модели, не менее важны, чем точность самой модели. Во-вторых, при построении модели программы нужно опираться на данные, которые в

меньшей степени зависят от ее кода. Данные о взаимодействии программы с системой важнее, чем логика ее работы, и развитие области распознавания вредоносных кодов указывает на высокую эффективность методов, основанных на анализе поведения программ.

Постановка задачи

Наша основная цель – разработать алгоритм идентификации программ на основе информации о системных вызовах, которые совершает запускаемый ею процесс. Это значит, что из списка системных вызовов нужно выделить такие признаки, которые позволят в дальнейшем обнаруживать списки системных вызовов, произведенных тем же самым потоком программы.

Сравнение программ включает не только идентификацию. Из списков системных вызовов можно получать различную информацию о ходе исполнения программы. Так в [10] трасса исполнения исследуется для поиска ошибок. При решении основной задачи были испробованы различные методы, исходные данные подвергались различным преобразованиям, по ним строились структуры данных, которые описывали те или иные характерные черты работы процесса. Второстепенная задача состоит в том, чтобы, исходя из получаемых данных о работе программы, интерпретировать их, выдвигая некоторые гипотезы о производимых процессом действиях. Другими словами, промежуточные модели могут быть проинтерпретированы как алгоритмы получения некоторой информации о ходе исполнения процесса.

Методы решения

Как уже было отмечено ранее, в качестве данных о работе программы в системе используются последовательности системных вызовов.

Последовательность API-вызовов, совершаемых программой, также предоставляет информацию о производимых ею действиях. В работе [10] идентификация происходит именно на основе этих данных. Но не все такие вызовы совершают важные действия и представляют интерес. В результате мы получаем огромные массивы информации, в которой большая часть элементов оказывается бесполезной. Необходимо перехватывать лишь некоторое их подмножество. API-функции из библиотеки ntdll.dll напрямую переводятся в системные вызовы. Но вместо их перехвата можно отслеживать сами системные вызовы. С помощью драйвера, работа которого описана в [3], можно перехватывать сразу все эти вызовы, совершаемые всеми процессами в системе.

Программы могут не использовать надстройку из библиотек для вызова API-функций и производить вызовы в ядро напрямую. Благодаря перехвату системных вызовов мы можем отслеживать и их работу. Но, отказавшись от перехвата высокоуровневых функций, мы теряем возможность интерпретировать последовательности вызовов. Мы также не знаем, настолько ли характерны полученные нами данные для потока и процесса, из которых они были получены, что мы можем на их основе производить идентификацию. Списки системных вызовов – это исходные данные, которые мы будем изучать.

Охарактеризуем исходные данные. Вначале была собрана статистика по номерам функций и их N-граммам разных длин. Распределение числа встречаемости каждой из функций и их пар, троек и четверок оказалось неравномерно. На графике показаны проценты их встречаемости для потока одной программы. Самый верхний график это гистограмма номеров функций, второй сверху – гистограмма пар и т.д. Номера функций, их пары, тройки и четверки отсортированы по убыванию частоты.

По графику видно, что самая частая функция встречается в 12,5%. Если же мы просуммируем проценты встречаемости первых 7 функций, то получим 51%. Ступенчатость графиков объясняется тем, что функции вызываются устойчивыми группами. Эта ступенчатость повторяется и при рассмотрении других отрезков графика, а не только его начала.

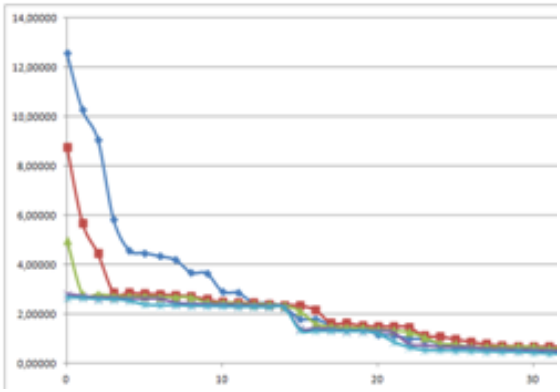


Рисунок 2.Гистограмма вызовов и их цепочек разно длины

Посчитаем энтропию последовательности исходных данных, основываясь на частоте функций. Энтропия изучаемого в данном случае потока, равная 4.867, указывает на то, что поток хорошо сжимаем. Кроме того, поток имеет энтропию близкую к энтропии естественных текстов, что указывает на возможность успешного применения алгоритмов анализа текстов. Действительно, номера системных вызовов не имеют численной семантики и их последовательность можно рассматривать как последовательность символов.

Запустив одну и ту же программу, мы можем получать различные последовательности системных вызовов. Это связано с тем, что программа взаимодействует с системой, обрабатывает события, которые могут возникать в произвольные моменты времени. Это вносит зашумленность в

потоке вызовов. Другими словами, рассмотрев достаточно длинные последовательности системных вызовов, мы можем рассчитывать лишь на совпадение отдельных участков, но не всех последовательностей, даже если сами эти последовательности будут взяты из одного и того же потока.

Вычисление частной энтропии

Описание

Энтропия – это мера неопределенности информации. Для некоторой последовательности она вычисляется по формуле:

$$H(x) = -\sum_{i=1}^{n} p(i) \log_2 p(i)$$

, где p(i) – это вероятность символа с номером i. Величина 1/p(i) называется частной энтропией (иногда log (1/p(i))), а H(x) – общей энтропией. Частная энтропия – это мера неопределенности, которая устраняется при получении i-го символа. Общая энтропия – это оценка количества информации, приходящейся на все сообщение.

По формуле видно, что основную роль в количестве информации играет вероятность каждого из символов. Но саму эту вероятность можно вычислять по-разному. Традиционной является модель, в которой вероятность символа зависит от его частоты. В таком случае нам важен лишь факт появления символа в последовательности, а не его контекст. Ранее нами была подсчитана энтропия последовательности вызовов в традиционной модели вероятностей. Действительно, как и в естественных текстах, она довольно высока. При изучении других последовательностей она колебалась от 4 до 12.

Высокая энтропия говорит о хорошей сжимаемости последовательности. Ступенчатость на гистограмме цепочек функций различных длин указывает на то, что в последовательности заметную роль играет контекст вызова. Большая часть вызовов несет мало информации и может быть предсказана по контексту своего появления.

Контекст – это некоторая модель, которая содержит в себе условия, в которых появился символ. Для оценки количества информации, которую несет вызов, был применен один из самых совершенных алгоритмов контекстного моделирования - алгоритм PPM. Об алгоритме PPM можно узнать здесь [12]. Он позволяет последовательно в поточном режиме оценивать вероятность символа в контексте его префикса. При получении нового символа он может быть учтен в модели в соответствии с его

контекстом. Благодаря этому модель постепенно изучает весь поток символов.

В [4] дан метод предсказания ветвлений на основе алгоритма PPM. Как оказалось, информацию о ветвлениях в потоке можно предсказывать по контексту. Более того, эта информация тесно связана с нашей информацией о системных вызовах, так как их последовательность определяется именно ветвлениями.

Опишем кратко алгоритм PPM. Контекстная модель порядка N - статистика символов, стоящих после каждой последовательности длины N. Модель 0 порядка, называемая моделью свободного контекста, содержит статистику символов, а модель -1 порядка, называемая статической моделью, имеет равные вероятности для каждого символа. Фиксируем N – длину контекста, являющуюся наибольшим порядком контекстной модели. В модели введен специальный символ, символ ухода, вероятность которого равна вероятности перехода к контекстной модели меньшего порядка. Когда мы получаем символ и его префикс, то на основе всех этих моделей мы получаем вероятность символа при данном префиксе. При этом большее значение для нас имеет самый длинный префикс и если символ может быть оценен в модели более высокого порядка, то модели меньшего порядка не учитываются.

Рассмотрим последовательность чисел, оценивающих частную энтропию каждого вызова по алгоритму PPM. Из частной энтропии мы можем получить общую энтропию последовательности, используя вероятности символов. Но величина общей энтропии не настолько важная для нас характеристика, как то, как частная энтропия распределена между вызовами в различные моменты времени работы программы.

Сама частная энтропия имеет очень важную интерпретацию. В случае если программа не получает никаких внешних воздействий, то каждый следующий ее вызов достаточно предсказуем. Назовем такое состояние потока «пассивным». «Активность» потока в таком случае обозначает те периоды его работы, когда он совершал непредсказуемые вызовы. Например, в случае оконного приложения «активностью» будет нажатие мышью по кнопке, в случае браузера – переход на другой сайт. Если мы имеем некоторую историю наблюдений за вызовами потока, то мы можем явно выявить те действия, которые поток за наблюдаемый период не совершал или совершал очень редко. Активность потока определяется тем, насколько неожиданными были его вызовы в контексте других вызовов. Активность также можно определить как совершение потоком действий, ранее не

производимых за исследованный к определенному моменту период его работы, или совершаемых им достаточно редко.

«Активность» и «пассивность» это нечеткие характеристики, которые являются нашей интерпретацией данных о частной энтропии вызовов.

Тесты

Тесты проводились на программах Internet Explorer(IE) и Mozilla Firefox(MF). Целью было не только подсчитать частную энтропию вызовов и таким образом оценить возможность их предсказания по контексту, но и выявить состояния «активности» программ. Активность создавалась искусственно через переходы по ссылкам в браузере.

В методе имеется настраиваемый параметр – длина контекста. Были испробованы различные значения. Для IE в пассивном состоянии было посчитано среднее значение частной энтропии вызовов. При длине контекста около 9 она была минимальной. Такой же тест для других процессов показывает, что при длине контекста от 7 до 10 частная энтропия минимальна.

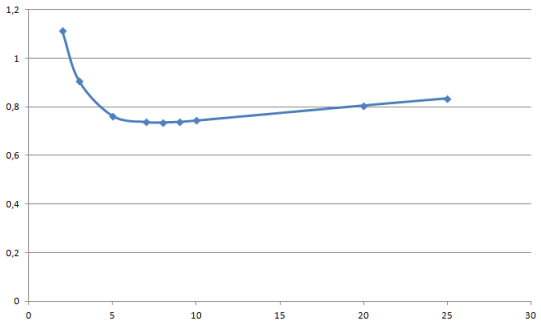


Рисунок 3. Среднее значение энтропии вызовов при контекстах PPM различных длин

На следующем графике показано значение частной энтропии для каждого вызова самого длинного потока процесса IE в пассивном состоянии. Такой вид графика был для нас неожиданным. Оказалось, что в вызовах часто встречаются такие функции, которые не свойственны своему контексту. Хотя вызовов с малым значением частной энтропии намного больше, видно, что сам поток сильно зашумлен. В пассивном состоянии вызовы должны быть предсказуемы, так как мы не производим искусственно никаких действий, которые могут вызвать повышение энтропии. Но наш процесс обрабатывает события, которые возникают в произвольные моменты времени. Это вносит свою неопределенность в ход его исполнения.

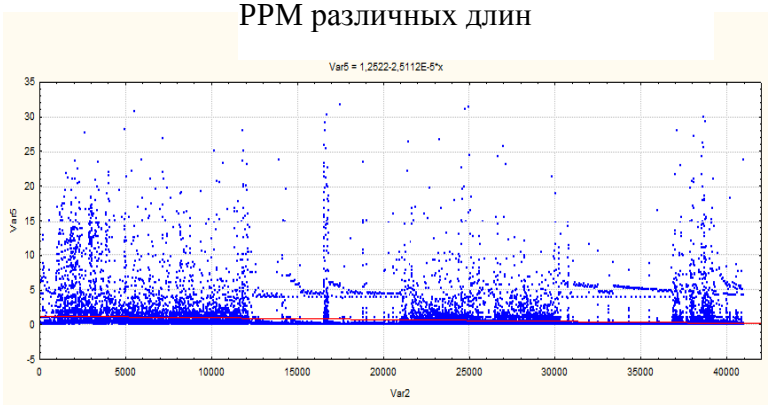


Рисунок 4. Частная энтропия для каждого вызова в потоке IE

Чтобы избавиться от этой зашумленности и для оценки активности потока, было проведено размытие. Разобьем всю последовательность на пересекающиеся интервалы, просуммируем значения в каждом из них.

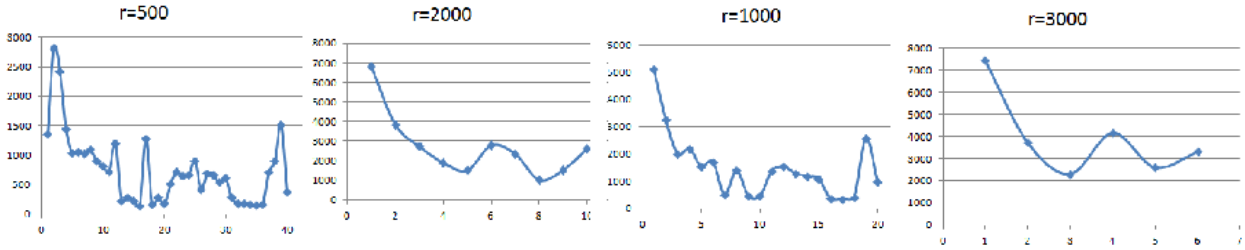


Рисунок 5. Размытия последовательности частных энтропий вызовов

На графиках выше значения энтропии сглаживаются постепенным увеличением окрестности суммирования. В начале значение энтропии относительно велико - модель только изучает поток. В дальнейшем значение энтропии колеблется - процесс обрабатывает события.

Аналогичный тест был проведен для IE в активном состоянии. Были сделаны переходы по трем ссылкам, в процессе чего была оценена частная энтропия для каждого вызова. Было проведено размытие полученной последовательности. На графике ниже изображено изменение частной энтропии этого потока в пассивном и активном состоянии.

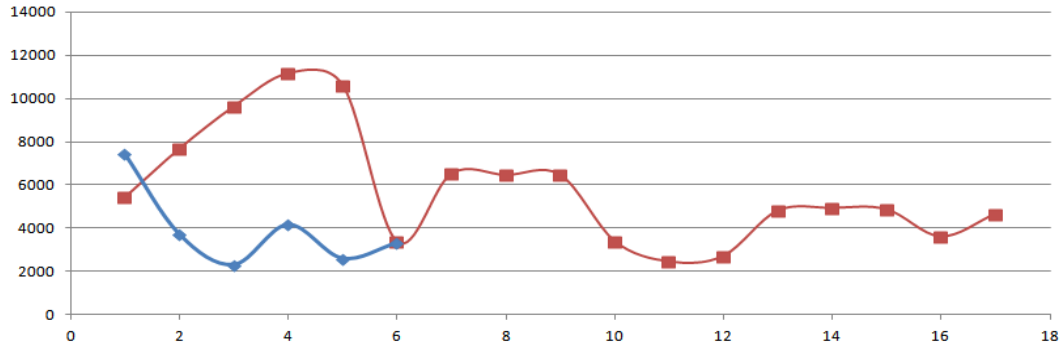


Рисунок 6. Сравнение размытых значений частной энтропии у IE в пассивном и активном состоянии

Моменты переходов по ссылкам можно явно увидеть по трем участкам с наибольшими значениями энтропии. Эти участки, как видно из графика, гораздо больше, чем колебания энтропии при естественной активности потока, когда он занимается обработкой сообщений. Кроме того, величина этих участков из раза в раз уменьшается, что говорит о том, что модель PPM запоминает контексты и учится справляться с неопределенностью, которую вносит наша искусственная активность.

В приложении 1 приведены графики оценки энтропии для MF.

Оценка контекстной энтропии показала, что последовательность вызовов сильно зашумлена. Под шумом мы будем подразумевать вызовы, не

свойственные своему контексту. При создании алгоритма идентификации процессов это нужно будет учитывать.

Корреляция n-грамм

Описание

Гистограммы цепочек вызовов фиксированной длины содержат в себе информацию о контексте вызовов. На основе этих гистограмм можно сравнивать последовательности и искать в них общие части.

Сравнивая на схожесть две строки, мы сравниваем их посимвольно. Но в случае наших последовательностей говорить о точном соответствии нельзя. Для сравнения последовательностей нужна оценка степени их схожести. Таких оценок существует множество, например редакционное расстояние или расстояние Левенштейна. Но нами был применен другой алгоритм.

Рассмотрим две последовательности вызовов. Для каждой последовательности составим гистограммы цепочек вызовов длины N. Найдем пересечение этих гистограмм по этим цепочкам. Теперь мы имеем множество N-грамм, для которых известна частота встречаемости в первой и второй последовательностях. Посчитаем линейный коэффициент корреляции между последовательностями частот по стандартной формуле:

$$r_{XY} = \frac{\sum(X - \bar{X})(Y - \bar{Y})}{\sqrt{\sum(X - \bar{X})^2 \sum(Y - \bar{Y})^2}}$$

, где X – частота N-граммы в первой последовательности, Y – частота N-граммы во второй последовательности, X с чертой – среднее арифметическое частот X.

Эту величину можно проинтерпретировать как степень схожести между последовательностями. Если она близка к 1, то последовательности похожи, если к 0 или меньше 0, то нет.

N – длина цепочки вызовов – это параметр, который подбирается в зависимости от точности. Если N равен 1, то мы просто сравниваем гистограммы вызовов. Но при этом коррелировать могут даже явно не связанные друг с другом последовательности вызовов. При увеличении N мы улучшаем точность сравнения, но зашумленность последовательности вызовов начинает играть свою роль, и при больших значениях N мы уже не обнаруживаем связи между теми последовательностями, в которых ожидаем ее увидеть, например, в цепочках самого длинного потока одной и той же программы.

При обнаружении корреляции в двух последовательностях мы можем проводить более точный поиск коррелирующих участков. Разобьем каждую из последовательностей на участки фиксированной длины. Для каждого

участка одной последовательности посчитаем его корреляцию с каждым участком другой. Так как сравниваемые последовательности намного меньше исходной, то, возможно, нам придется уменьшить параметр N, чтобы отрегулировать точность сопоставления. Но таким образом мы сможем для двух потоков вызовов последовательным уточнением обнаружить схожие участки, на которых они, скорее всего, производили одинаковые действия.

Тесты

Были написаны две программы на языке Perl. Они загружали пять страниц из сети и сохраняли их в файлы. Первая производила сохранение после каждой загрузки, вторая производила все сохранения в конце работы. Также эти программы были написаны на Python. Мы ожидаем, что на основе корреляции гистограмм мы сможем обнаружить интервалы вызовов, на которых эти программы производили одни и те же действия.

Было обнаружено, что у потоков со схожей функциональностью действительно сильно возрастает корреляция по гистограммам: между двумя программами на Perl корреляция была близка к 1. При поиске участков со схожей функциональностью у программ на Perl и Python эти участки также были обнаружены.

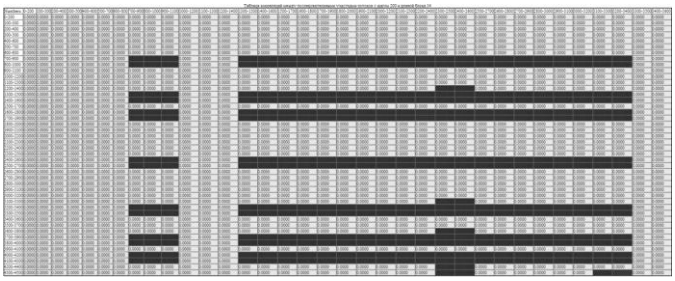


Рисунок 7. Сравнение корреляции гистограмм у программ на Perl и Python

На рисунке показано одно из сравнений. По вертикали отложены интервалы номеров вызовов первой программы, написанной на Perl с шагом 200, по горизонтали номера такой же программы, написанной на Python. Выделенные ячейки таблицы указывают на высокую корреляцию между участками при N=14.

Проверка возможности установления такого рода соответствий нужна для того, чтобы иметь возможность на столь низком уровне, как системные вызовы, быстро находить схожие участки. На уровне API-вызовов операции загрузки страницы и ее сохранения можно выполнить всего лишь несколькими API-функциями, которые в итоге произведут какой-то набор системных вызовов. Их последовательности интерпретируются не так легко, как API-вызовы. Помимо этого, на сами последовательности влияет зашумленность. Теперь мы имеем алгоритм, который обнаруживает выполнение схожих участков кода.

Все это показывает, что, несмотря на зашумленность, цепочки вызовов все еще несут в себе полезную информацию и их можно сравнивать, находя схожие участки.

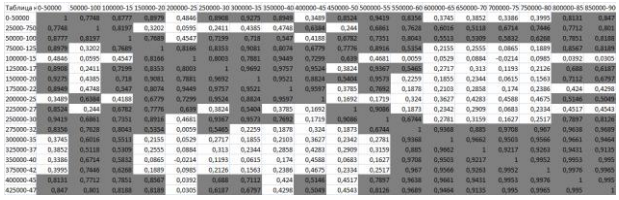


Рисунок 8. Корреляция между различными участками в IE

Как проявляет себя этот алгоритм на более зашумленных потоках? Для теста был взят самый длинный поток IE в пассивном состоянии. Разобьем этот поток на непересекающиеся участки и проверим, насколько они коррелируют друг с другом. В таблице указана корреляция различных участков длины 50000 по гистограммам последовательностей длины 10. Серым цветом обозначена корреляция более 0,5. По этой таблице видно, что статистика цепочек лишь примерно в половине случаев повторяется в других цепочках.

Можно ли применять этот алгоритм для идентификации процессов? При построении гистограмм и вычислении корреляции большее значение имеют частые цепочки вызовов, в то время как идентификация предполагает выявление особенностей, которые могут проявляться не настолько часто. Кроме того, гистограммы в различных потоках у разных процессов очень похожи. В таблице справа указаны значения корреляции между различными участками IE и MF в пассивном состоянии. Как и раньше, были взяты только самые длинные потоки. Серым цветом указаны участки с корреляцией больше 0,5. По данному рисунку видно, что потоки различных процессов имеют высокую корреляцию на отдельных участках.

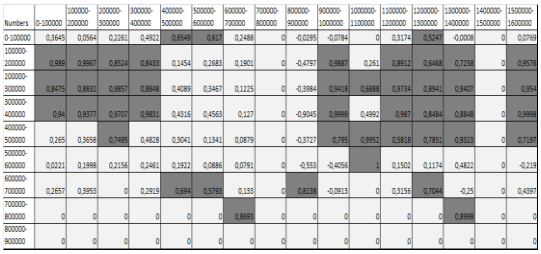


Рисунок 9. Корреляция между гистограммами цепочек длины 10 у потоков IE и MF

Таким образом, мы можем находить схожие участки в последовательностях. Но сами гистограммы цепочек фиксированной длины не позволяет идентифицировать потоки с достаточной точностью. Причина в том, что мы строим модель всего потока целиком. При этом особенности работы, которые и позволили бы идентифицировать поток, не проявляются.

Выделение термов

Описание

Идентификацию программ по спискам системных вызовов можно сравнить с индексацией текстов. Самые частые слова несут нам мало информации, так как встречаются в большинстве текстов. Обычно при индексации из текста выделяются наиболее характерные для него выражения. Значит, при построении признаков программного потока мы должны находить последовательности, в наибольшей степени характеризующие данный поток. Для этого было введено понятие терма.

Терм – это последовательность вызовов, которая повторяется в потоке более одного раза. Другими словами, термом является любой дубликат в последовательности. Так как терм повторяется не менее двух раз, то его появление может не быть случайным. Либо два вызова стояли рядом в самом коде, либо они имеют какую-то связь друг с другом через другие вызовы, происходящие в коде программы, либо программе срабатывают условные переходы, которые формируют устойчивые, повторяющиеся из раза в раз цепочки вызовов.

В последовательности вызовов встречаются элементы множества термов. Они могут пересекаться друг с другом, включаться один в другой, конкатенировать и порождать новый терм. Некоторые термы являются степенями более коротких термов. Особенности потока исполнения мы будем искать именно во множестве термов, а признаки будут формироваться нами как подмножества множества термов. У каждого терма можно выделить следующие свойства:

1) Длина.
2) Число появлений в последовательности вызовов
3) Периоды появления: средний, минимальный, максимальный

Для того чтобы выделить из всего множества термов некоторое подмножество, были разработаны специальные эвристики:

1) Чем длиннее терм, тем больше он несет информации.
2) Чем чаще встречается терм, тем более он характерен для потока и тем вероятнее он встретится в последовательности вызовов того же самого потока.

Эти эвристики друг другу противоречат и между ними нужно искать баланс: длинные термы встречаются реже коротких. Кроме того длинные термы могут обладать низкой энтропией и нести мало информации. Исходя из этих эвристик, для нас наибольший интерес представляют длинные и частые термы.

На основе этих эвристик был разработан следующий алгоритм фильтрации термов: перебираем термы, начиная с самых длинных, и выбираем те из них, которые встречались не менее фиксированного числа раз. Может возникнуть большое число коротких термов. Поэтому в данном алгоритме лучше сразу фиксировать еще и максимальное число термов, которое мы хотим получить и их минимальную длину.

Период встречаемости каждого терма дает нам оценку размера последовательности вызовов, на которой можно ожидать встретить этот терм. Для того чтобы выявить степень схожести между тестируемым потоком и ранее изученным, нужно искать в первом потоке термы второго. Выделение термов нужно проводить на достаточно больших последовательностях, чтобы характерные потоку цепочки появились несколько раз. Распознавание потока можно проводить по более короткой последовательности вызовов, размер которой можно оценить из периодов встречаемости термов или подобрать экспериментально.

По аналогии с индексацией текстов, термы можно считать ключевыми последовательностями для списка вызовов. Наши эвристики направлены на то, чтобы выделить такие термы, которые будут встречаться достаточно часто на протяжении всего исследуемого периода работы. В таком случае высока вероятность встретить эти термы и в любой другой период работы этого потока, а значит опознать поток по этим термам. Алгоритм фильтрации термов был выбран так, чтобы этап фильтрации работал за линейное время относительно числа вызовов. Чтобы создать более сложные алгоритмы, нужно сначала изучить структуру самого множества термов.

Реализация

Для поиска дубликатов в последовательностях был взят соответствующий строковый алгоритм. В нашей реализации для этого использовались суффиксные массивы [13]. Построение множества термов происходит в два этапа:

1) Построим массив суффиксов, отсортированных в лексикографическом порядке - суффиксный массив. Алгоритм построения позволяет одновременно подсчитывать величину наибольшего общего префикса (LCP) между соседними в массиве суффиксами. Благодаря этому можно посчитать величину наибольшего общего префикса между произвольными суффиксами как минимум из этих чисел между позициями суффиксов в массиве. Построение всей структуры данных занимает $O(n \log n)$, где n – размер последовательности вызовов. При этом сама структура линейна по памяти, так как хранит в каждом элементе массива

всего лишь позицию в строке и длину наибольшего общего префикса со следующим в массиве элементом. Если существует строка, которая повторяется в последовательности хотя бы два раза, то наибольший общий префикс между позициями, в которых она встречается, имеет длину не меньше, чем длина этой строки. Таким образом, если мы рассмотрим множество наибольших общих префиксов между соседними в суффиксом массиве элементами, то все они будут дубликатами, а все дубликаты будут встречаться в них как подстроки. На этом шаге мы получаем суффиксный массив и множество наибольших общих префиксов между соседними элементами. В последнем множестве мы и будем искать нужные термы.

2) Теперь нужно отфильтровать множество наибольших общих префиксов, используя описанные ранее эвристики.

2.1) Построим массив Size. В каждом его элементе находится список позиций суффиксного массива, в которых длина LCP со следующим элементом совпадает с номером элемента в массиве. Само построение происходит за один проход по суффиксному массиву.

2.2) Теперь необходимо провести выбор термов по следующему алгоритму:

Алгоритм фильтрации термов.

1. MinFreq – минимальное число встречаемости терма, MaxCount – максимальное число термов, которое нужно получить, MinLen – минимальная длина терма.
   Hits – массив занятых номеров вызовов
2. Для всех списков S из Size, начиная с последнего
3. Для всех элементов E текущего списка S
4. Запомним в массиве L все вхождения наибольшего общего префикса между элементом E и следующим за ним в массиве суффиксом без взаимных пересечений этих вхождений. Все вхождения данного LCP в суффиксном массиве будут стоять рядом в силу упорядоченности элементов массива.
5. Если хотя бы один элемент L уже отмечен в Hits, перейти к следующему элементу E.
6. Если количество элементов в L меньше MinFreq, то перейти к следующему элементу E.
7. Запомнить наибольший общий префикс E, число его встречаемости без взаимных пересечений (число элементов в L), минимальный, максимальный и средний периоды встречаемости, заполнить Hits вхождениями L.

8. Если мы получили MaxCount термов или длина терма меньше MinLen, то завершаем работу.

Массив Hits – это битовый массив, в котором обозначены позиции в последовательности, которые уже входят в один из уже выбранных термов. Сложность самого алгоритма не превышает O(n log n). После проведения фильтрации мы получаем множество последовательностей – термов, каждый из которых встречается без взаимных пересечений своих вхождений не менее MinFreq раз. Все множество термов не превышает по размеру MaxCount, каждый его элемент имеет длину не меньше MinLen.

В приложении 4 дан псевдокод описанного алгоритма.

Пусть T – множество термов. F – последовательность вызовов. Выделим следующие значения:

1. UniqCount – число последовательностей из T, которые встретились в F.
2. MaxTerm – максимальная длина терма из T, который встретился в F.

Опираясь на эти значения, мы будем оценивать степень схожести тестируемого потока вызовов и изученного.

Тесты

Для проверки того, что выделенные термы позволяют идентифицировать потоки были записаны последовательности вызовов 10 программ, типичных для Windows 7: calc.exe, devenv.exe, explorer.exe, Internet Explorer, minesweeper.exe, Mozilla Firefox, notepad.exe, Opera, VLC Player, WMPlayer. Они производят системные вызовы с различной интенсивностью. В таблице показано число вызовов, производимых самым длинным потоком программ в секунду.

Таблица 1. Интенсивность вызовов самых длинных потоков некоторых программ

Проц.	Opera	IE	Notepad	Mozilla	Calc	Devenv	VLC	WMPlayer	Explorer	Mines
Выз./сек	2551	872	560	4526	1259	2609	1771	8637	392	1700

Из всех потоков, которые совершали вызовы, выделим самый длинный поток и для него проведем выделение термов по описанному ранее алгоритму. Запишем списки системных вызовов тех же программ на новых экземплярах процессов. В этих списках выделим самый длинный поток. Попробуем распознать процесс по его самому длинному потоку. Для выделения термов были взяты около миллиона вызовов из каждого потока,

для распознавания были взяты 50000 из тестируемого потока. В таблице для каждого набора термов процесса из левой колонки и для каждого тестируемого процесса справа указано число UniqCount и в скобках MaxTerm. Здесь приведена часть результатов, куда были включены случаи наиболее неточного распознавания. Минимальное число встречаемости в данных тестах равно 3. Максимальное число взятых термов равно 2000. В приложении 2 приведены результаты подобных тестов для других значений количества тестовых вызовов.

Таблица 2. Число найденных термов и их наибольшая длина для некоторых программ

Процесс, Термы	Calc	Devenv	Explorer	IE	Mines	Mozilla	Notepad
CALC	964 [360]	103 [93]	37 [11]	77 [32]	61 [134]	32 [11]	86 [103]
DEVENV	3 [62]	145 [585]	2 [45]	3 [47]	6 [71]	0 [0]	23 [98]
EXPLORER	105 [55]	192 [49]	85 [28]	176 [45]	62 [23]	42 [21]	87 [41]
IE	3 [29]	8 [34]	1 [27]	322 [110]	1 [23]	0 [0]	0 [0]
MINES	1 [71]	0 [0]	0 [0]	0 [0]	400 [722]	0 [0]	0 [0]
Mozilla	0 [0]	0 [0]	0 [0]	0 [0]	0 [0]	174 [1426]	0 [0]
NOTEPAD	105 [80]	148 [65]	14 [47]	50 [39]	45 [60]	13 [18]	224 [165]

Возьмем из названия колонки программу, список системных вызовов мы хотим распознать. Для каждой из уже изученных программ проведем поиск ее термов в тестируемой последовательности. Та программа, чьих термов окажется больше, и является программой, запустившей этот поток.

В данной таблице почти в каждой колонке наибольшее значение UniqCount в столбце находится на диагонали, что говорит об успешности распознавания. В случае Devenv и Explorer мы можем провести различие, если учтем величину MaxTerm.

Практически во всех случаях мы можем выполнять и обратное сравнение, то есть, имея набор термов для некоторого потока, устанавливать,

какой из тестируемых потоков ему соответствует. Исключением вновь является Explorer. Этот важный системный процесс плохо идентифицируется по своему главному потоку. Чтобы выяснить, почему идентификация по термам может быть неточной, нужно лучше понять, на чем она основана. Ранее было отмечено, что последовательность вызовов сильно зашумлена. При этом под шумом мы подразумевали те вызовы, которые не свойственны своему контексту, то есть вызовы с высокой частной энтропией, оцененной по алгоритму PPM. Так как мы ищем термы, повторяющиеся несколько раз последовательности, то содержащиеся в них вызовы должны быть ожидаемы и обладать низкой энтропией. То, насколько частная энтропия в потоке вызовов отклоняется от среднего значения, может быть оценено через среднеквадратическое отклонение. Оно считается по формуле

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x(i) - xs)^2},$$

где x(i) – значение частной энтропии вызова под номером i, xs – среднее значение частной энтропии вызовов. В таблице приведены значения этой величины для самых длинных потоков различных процессов.

Таблица 3. Среднеквадратичное отклонение частной энтропии

Процесс	Explorer	IE	Devenv	Notepad	Opera	MF	Calc	Mines	WMPlayer	VLC
Ср.откл.	2,0018	1,90	1,78	1,766	1,64	1,46	1,27	1,23	0,995	0,86

Из данных этой таблицы и по результатам сравнения из предыдущего примера можно сделать следующий вывод: высокое среднеквадратическое отклонение частной энтропии вызовов отрицательно сказывается на распознавании потока при помощи термов. Лучше всего были распознаны те потоки, у которых эта величина минимальна.

Что будет, если мы начнем сравнивать и другие потоки? Был проведен тест: были выделены термы у пяти самых длинных потоков тех же процессов, что и в предыдущем примере, потом из тех же потоков были взяты по 50000 вызовов. Здесь для тестов брались вызовы того же самого экземпляра процесса, чтобы мы могли при сравнении ставить потоки в соответствие своим же экземплярам. Результаты показаны на рисунке.

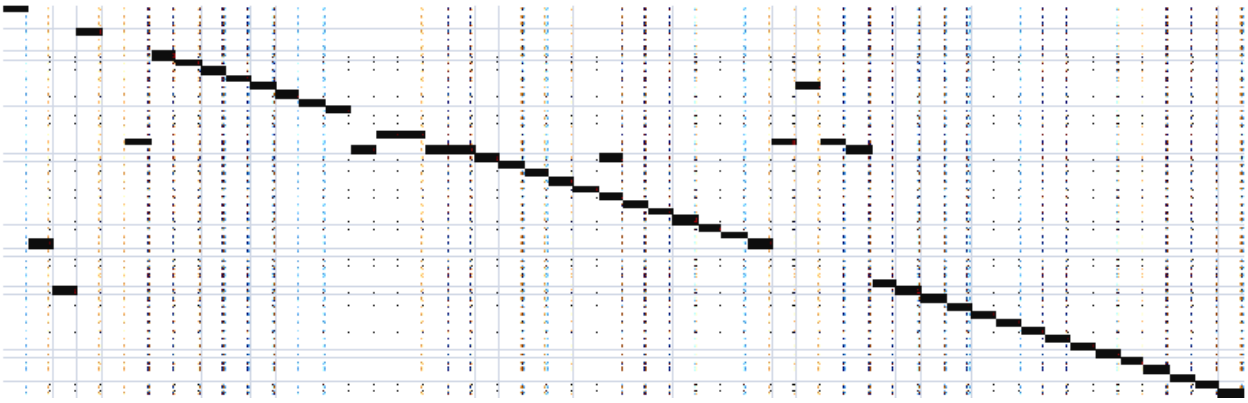


Рисунок 10. Идентификация нескольких потоков

По вертикали отложены потоки, из которых брались термы, по горизонтали – тестируемые потоки. Черным цветом отмечены наибольшие в данном столбце значения UniqCount. Видно, что почти во всех случаях потоки распознаются правильно. Неверные результаты при опознавании указывают, на схожесть в работе соответствующих потоков у различных процессов.

По результатам тестов видно, что термы позволяют проводить идентификацию потоков. К сожалению, эвристики, с помощью которых выбираются термы, не всегда хорошо срабатывают. В идеале мы хотим получить такое множество последовательностей, которые в совокупности встречаются одинаково часто на протяжении всего потока. Но так бывает не всегда.

Проведем еще один эксперимент: выделим 1000 термов с минимальным числом встречаемости 3 из самого длинного потока процесса IE, затем разделим этот поток на непересекающиеся последовательные участки длинной по 50000 вызовов. Посчитаем число UniqCount для каждого участка, то есть определим число уникальных термов в каждом из них. Результаты приведены на графике. Видно, что эта характеристика сильно колеблется в зависимости от взятого участка. Это значит, что если мы неудачно выберем участок для теста на принадлежность данного потока процессу, то получим худший результат.

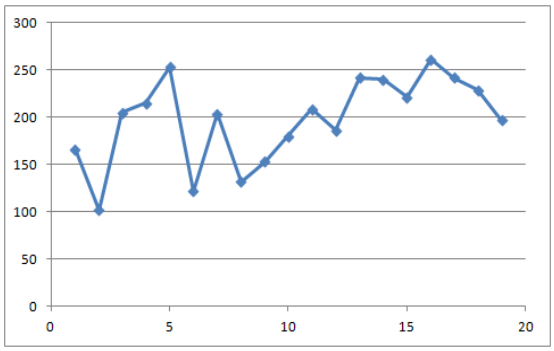


Рисунок 11. Число уникальных термов на различных участках потока

Таким образом, мы можем различать большую часть процессов только по самому длинному потоку. На качество распознавания оказывает влияние

предсказуемость вызовов по контексту, которую можно выразить с помощью среднеквадратического отклонения частной энтропии вызовов.

Заключение

Задача идентификации процессов была сведена нами к сравнению потоков исполнения программ по спискам системных вызовов, которые они производят. Был предложен алгоритм выделения признаков, множества термов, которые позволяют идентифицировать потоки. На тестах было показано, что идентифицировать программы можно лишь по данным о работе самого длинного потока. Для оценки того, насколько качественно может отработать наш алгоритм, был также предложен формальный критерий - оценка среднеквадратического отклонения частных энтропий вызовов, посчитанных по алгоритму PPM. Полученные нами объекты, термы, требуют более глубокого изучения. Работа с ними происходила на основе эвристик, которые не всегда давали точный результат.

Алгоритм построения множества термов имеет довольно низкую вычислительную сложность O(n log n). Благодаря этому он может быть использован для идентификации всех процессов в системе. Теперь мы можем, например, построить признаки тех процессов, которые могут исполняться в системе, а при запуске нового проверять, может ли он исполняться в нашей системе, основываясь лишь на информации о последовательности системных вызовов. Еще один возможный сценарий применения алгоритма – отслеживание запуска в системе определенного процесса, модель которого была ранее построена. После обнаружения мы можем закрыть процесс, либо записать в журнал, что данный процесс был обнаружен.

Наш алгоритм основан на системных вызовах, взаимодействовать с системой в обход этого механизма нельзя.

К недостаткам нашего алгоритма можно отнести неточность работы эвристик по получению термов. Кроме того алгоритм был протестирован на обычных процессах, а не на процессах, которые противодействуют своему обнаружению.

В процессе решения задачи идентификации был также получен алгоритм оценки «активности» в поведении потока через оценку средней частной энтропии вызовов. Также был получен алгоритм для поиска общих участков кода в потоках через оценку линейного коэффициента корреляции между гистограммами цепочек вызовов фиксированной длины. Оценка активности потоков может быть применена там, где следует выявлять

неожиданные поведения программы, например, при поиске ошибок, при обнаружении сетевых атак.

Наш алгоритм идентификации следует в будущем проверить не только на обычных процессах, но и на вредоносных программах. Кроме того, алгоритм разрабатывался для достаточно длинных потоков, в то время как в процессе могут быть запущены потоки, работающие всего несколько секунд и не производящие большое количество вызовов. В плане теории следует более тщательно изучить термы, выработать и обосновать новые эвристики для фильтрации термов.

Данная работа была представлена на конференции СПИСОК 2012 [15][16][17].

Список литературы

[1] Р. И. Подловченко, Н. Н. Кузюрин, В . С. Щербина, В . А. Захаров «Использование алгебраических моделей программ для обнаружения метаморфного вредоносного кода» //Фундаментальная и прикладная математика, 2009, том 15, №5, с. 181—198.

[2] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, Moshe Bach , «Dynamic Program Analysis of Microsoft Windows Applications»

[3] Одеров Р.С., Тенсин Е.Д., «Способы размещения своего кода в ядре ОС Microsoft Windows Server 2008». // Сборник трудов межвузовской научно-практической конференции «Актуальные проблемы организации и технологии защиты информации», СПб, 2011 год. Стр.100-102

[4] Pierre Michaud «A PPM-like, tag-based branch predictor» // Journal of Instruction-Level Parallelism 7 (2005) 1-10

[5] Szor, P. The Art of Computer Virus Research and Defense. Addison Wesley, 2005.

[6] Kinder, J., Katzenbeisser, S., Schallhart, C., and Veith, H. Detecting Malicious Code by Model Checking. //In Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) (2005).

[7] Kruegel, C., Robertson, W., and Vigna, G. Detecting Kernel-Level Rootkits Through Binary Analysis. //In Annual Computer Security Applications Conference (ACSAC) (2004).

[8] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang «Effective and Efficient Malware Detection at the End Host» // presented at the 18th Usenix Security Symposium, 2009

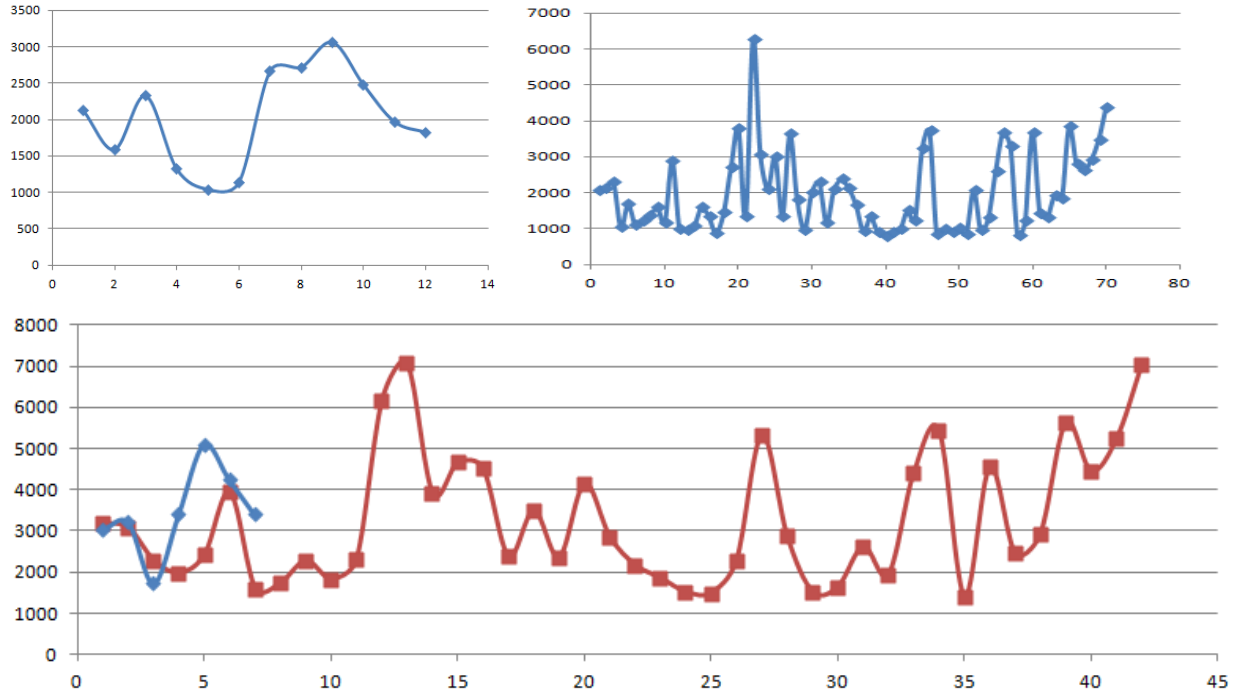
[9] Jianyong Dai, Ratan Guha, Joohan Lee «Efficient Virus Detection Using Dynamic Instruction Sequences» //Journal of Computers, 2009

[10] David Lo, Hong Cheng,, Jiawei Han, SiauCheng Khoo, Chengnian Sun «Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach» // KDD'09, June 28–July 1, 2009, Paris, France.
[11] M. Ahmadi, A. Sami, H. Rahimi, B. Yadegari, «Iterative System Call Patterns Blow the Malware Cover»// Security for The Next Generation 2011.
[12] John G. Cleary, Ian H. Witten «Data Compression Using Adaptive Coding and Partial String Matching» // IEEE Transactions on communications, vol. com - 32,№. 4, April 1984
[13] Juha Karkkainen, Peter Sanders, «Simple Linear Work Suffix Array Construction» //J.C.M. Baeten et al. (Eds.): ICALP 2003, LNCS 2719, p. 943–955, 2003
[14] Чернов А.В. Анализ запутывающих преобразований программ.// Сб. "Труды Института системного программирования", под. ред. В. П. Иванникова. М.: ИСП РАН, 2002.
[15] Ханов А.Р. Идентификация процессов программ по внешним признакам // Материалы конференции СПИСОК 2012
[16] Ханов А.Р. Распознавание программных потоков // Матералы конференции СПИСОК 2012
[17] Ханов А.Р. Идентификация процессов для борьбы с вредоносными программами // Материалы конференции СПИСОК 2012

Приложения

Приложение 1. Оценка частной энтропии вызовов некоторых процессов

Частная энтропия самого длинного потока процесса Mozilla Firefox при размытии по окрестности 3000. Слева – в пассивном состоянии, справа – в активном состоянии, снизу – сопоставление этих графиков.



Тесты проводились при длине контекста 9. Среднее значение энтропии вызова процесса IE в пассивном состоянии было равно 0,73, у процесса MF это значение равно 0,39. Оказалось, что каждый вызов MF несет меньше информации и более предсказуем по контексту.

Далее приведены значения средней частной энтропии и среднеквадратическое отклонение энтропии для списков системных вызовов некоторых программ. Взяты списки системных вызовов большего размера, чем в предыдущих тестах. Энтропия считалась по алгоритму PPM с длиной контекста 9 символов.

Процесс	среднее	отклонение
Internet Exlorer	0,635526	1,903342
Opera	0,58103	1,627053
Explorer	0,562794	1,992171
Devenv	0,528879	1,775151
Notepad	0,482335	1,760748
Mozilla Firefox	0,468603	1,448088
Calc	0,383857	1,253339

WMPlayer	0,347926	0,97046
VLC	0,332954	0,802516
Minesweeper	0,321938	1,221278

Приложение 2. Идентификация процессов

В таблице приведены результаты идентификации процессов с помощью термов. Из самых длинных потоков левой колонки было извлечено 2000 термов с минимальным числом встречамости 3, из самых длинных потоков других экземпляров этих процессов были извлечены первые 50000 вызовов. В первой таблице дано число встретившихся термов, во второй - их максимальная длина.

>тесты <термы	Calc	Devenv	Explorer	IE	Mines	Mozilla	Notepad	Opera	VLC	WMP
calc	1078	109	64	84	157	32	90	50	0	62
devenv	3	217	3	3	7	0	40	0	0	20
explorer	106	222	152	179	147	44	87	54	0	89
IE	3	13	4	502	4	0	0	3	0	2
mines	1	0	0	0	510	0	0	0	0	0
Mozilla	0	0	0	0	0	250	0	1	0	0
notepad	107	170	38	54	115	14	294	23	0	46
Opera	0	0	0	0	0	1	0	324	0	0
vlc	0	0	0	0	0	0	0	0	953	0
WMPlayer	39	54	27	40	49	23	30	23	0	812

>тесты <термы	Calc	Devenv	Explorer	IE	Mines	Mozilla	Notepad	Opera	VLC	WMP
Calc	360	93	21	32	134	11	103	16	0	19
Devenv	62	585	57	47	71	0	98	0	0	67
Explorer	55	49	28	45	42	21	41	28	0	45
IE	29	34	27	480	31	0	0	27	0	20
Mines	71	0	0	0	722	0	0	0	0	0
Mozilla	0	0	0	0	0	1426	0	183	0	0
Notepad	80	65	76	39	60	18	165	23	0	40
Opera	0	0	0	0	0	59	0	441	0	0
vlc	0	0	0	0	0	0	0	0	128	0
WMPlayer	22	47	17	13	22	22	22	13	0	326

Тот же самый тест, но из опознаваемых потоков брались первые 10000 вызовов. В первой таблице – число встретившихся термов, во второй – их максимальная длина.

>тесты <термы	Calc	Devenv	Explorer	IE	Mines	Mozilla	Notepad	Opera	VLC	WMP
Calc	532	99	18	72	0	29	77	42	0	55
Devenv	3	107	2	0	0	0	6	0	0	18
Explorer	101	170	69	141	0	39	73	49	0	65
IE	3	8	1	180	0	0	0	3	0	1
Mines	1	0	0	0	271	0	0	0	0	0
Mozilla	0	0	0	0	0	58	0	0	0	0
Notepad	98	108	10	42	0	13	119	19	0	34
Opera	0	0	0	0	0	0	0	78	0	0
VLC	0	0	0	0	0	0	0	0	437	0
WMPlayer	36	47	10	30	0	21	29	20	0	458
>тесты <термы	Calc	Devenv	Explorer	IE	Mines	Mozilla	Notepad	Opera	VLC	WMP
Calc	360	93	7	30	0	11	103	14	0	19
Devenv	62	272	45	0	0	0	98	0	0	67
Explorer	55	49	28	30	0	21	23	28	0	45
IE	29	34	27	102	0	0	0	27	0	20
Mines	71	0	0	0	142	0	0	0	0	0
Mozilla	0	0	0	0	0	144	0	0	0	0
Notepad	80	48	47	32	0	18	165	23	0	40
Opera	0	0	0	0	0	0	0	129	0	0
VLC	0	0	0	0	0	0	0	0	118	0
WMPlayer	22	47	10	13	0	22	22	13	0	253

Приложение 3. Дополнительный алгоритм фильтрации термов

В разделе «Выделение термов» приведен алгоритм фильтрации термов, который работает за линейное время. Но такой алгоритм появился не сразу. После получения суффиксного массива и множества наибольших общих префиксов можно выделять термы по-разному. Далее приведен алгоритм, который, ввиду высокой вычислительной сложности, так и не был протестирован на больших примерах.

В суффиксном массиве для каждого элемента мы имеем два числа: позиция в строке и длина наибольшего общего префикса со следующим элементом массива. Так как элементы массива отсортированы в лексикографическом порядке, то, зная длину наибольшего общего префикса между соседними элементами, мы можем посчитать ее между любыми элементами как минимум по этим числам между позициями в массиве. Поэтому если мы зафиксируем некоторую минимальную длину наибольшего общего префикса(LCP), то получим разбиение всего массива на блоки, между каждым элементом которых LCP будет не меньше, чем фиксированное значение. Для того чтобы найти для некоторого LCP все его вхождения, нужно перебрать все элементы суффиксного массива, стоящие рядом с ним, у

которых размер LCP не меньше. Все такие элементы мы будем называть блоком этого LCP.

По определению LCP – это максимальная по длине совпадающая строка между двумя позициями. Однако если символ, стоящий перед каждой из позиций, совпадает, то между позициями на единицу меньше длина наибольшего общего префикса на единицу больше, чем в данных позициях. Другими словами, LCP может быть не максимален влево.

Данный алгоритм фильтрации можно разбить на два этапа.

На первом этапе мы фиксируем минимальную длину LCP, разбиваем суффиксный массив на блоки, между каждым элементом которых LCP будет не меньше этого значения. Для каждой пары элементов в каждом блоке проверяем, что эта пара максимальна влево путем проверки, что перед их позициями в последовательности стоят разные символы. Проверяем, что LCP у позиций этих элементов не пересекаются друг с другом. Если все условия выполнены, запоминаем эту пару.

На втором этапе мы работаем со всеми этими парами. Каждому LCP мы теперь можем поставить в соответствие множество его пар, откуда получим множество позиций его вхождений. Отсюда можно выделить число вхождений этого LCP без самопересечений. Получившееся множество LCP и будет множеством термов для данного списка вызовов.

Алгоритм имеет кубическую сложность. Суффиксный массив делится на блоки, каждый блок обрабатывается за квадрат, от его размера. Поэтому крупные блоки обрабатывались долгое время. Описанный в основной части алгоритм работал не хуже данного. Оба этих алгоритма основаны на эмпирических соображениях о том, какие термы нужно выбрать, и не опираются на структуру самого множества термов, которая требует детального изучения.

Приложение 4. Псевдокод алгоритма фильтрации термов

В данном приложении приведен псевдокод основного алгоритма фильтрации, описанного в разделе «Выделение термов». В результате его работы получается массив термов FEATURES, содержащий в себе отфильтрованное множество термов.

1. SM[0..N] – суффиксный массив для последовательности длины N
   TEXT[0..N] – исходная последовательность.
   LCP[i] – длина наибольшего общего префикса между суффиксами в позициях SM[i] и SM[i+1].
   HIT[0..N] – массив занятых номеров, изначально заполнен 0.
   SIZE[0..C] – массив списков, C = max(LCP(i)).

FEATURES:=() – массив для записи реультата
MinFreq, MaxCount, MinLen – параметры алгоритма.
Count: = MaxCount;

2. For I in 0 to N-1
3. Push SIZE[LCP[I]], I
4. For I in C to MinLen
5. For E in SIZE[I]
6. (imin,imax) :=(E,E); L := (); # Позиции вхождений и список вхождений
7. While (imin > 1 and LCP[imin-1] >=LCP[E]) imin—
8. While (imax < N-1 and LCP[imax+1] >=LCP[E]) imax++
9. For T in imin to imax
10. If (HIT[SM[T]] == 1 or HIT[SM[T]+LCP[T]-1] == 1 or HIT[SM[T+1]] == 1 or HIT[SM[T+1] +LCP[T]-1] == 1) next
11. For J in SM[T] to SM[T]+LCP[T]-1 do HIT[J]:=1;
12. For J in SM[T+1] to SM[T+1]+LCP[T]-1 do HIT[J]:=1;
13. Push L, T;
14. If (size(L) < MinFreq)
15. For f in L
16. For pos in SM[f] to SM[F]+LCP[F]-1do HIT[pos]:=0;
17. For pos in SM[f+1] to SM[F+1]+LCP[F]-1do HIT[pos]:=0;
18. Next
19. Symbols := TEXT[SM[E]..SM[E]+LCP[E]-1]
20. Push FEATURES, Symbols