Московский государственный университет имени М. В. Ломоносова
Факультет Вычислительной математики и кибернетики
Кафедра Автоматизации систем вычислительных комплексов

Дипломная работа

Создание параллельного ядра анализа событий для системы обнаружения и предотвращения атак

студент 522 гр. Казачкин Д. С.

Научный руководитель:
к.ф.-м.н. Гамаюнов Д. Ю.

Москва, 2009г.

Аннотация

В данной работе рассматривается задача высокоскоростной обработки трафика в контексте задачи обнаружения атак. Разработана и реализована архитектура параллельного ядра анализа для экспериментальной среды высокоскоростного анализа событий AURA (Automata for Recognition and Analysis – Автоматы для распознавания и анализа). Данная архитектура позволяет организовать быструю обработку асинхронных потоков событий за счет использования возможностей современных многоядерных процессоров. Для оценки эффективности параллельного ядра анализа событий разработан автоматизированный тест производительности, который включает в себя как синтетические тесты, так и реальные тесты для типовых вариантов использования среды анализа. Результаты экспериментов показывают близкий к линейному рост скорости обработки событий для типовых вариантов использования при увеличении числа используемых процессорных ядер.

Содержание

1 Введение

В настоящее время создание эффективных систем защиты информационных систем сталкивается с нехваткой вычислительной мощности. Гигабитные каналы Ethernet уже несколько лет являются основой построения локальных вычислительных сетей и подключения сетей организаций к сетям провайдеров Интернет, 10-гигабитный Ethernet также становится нормой, в ближайшее время ожидается стандартизация 100-гигабитного Ethernet. При этом можно отметить две основные тенденции, наблюдаемые в области вычислительной техники и компьютерных сетей – закон Мура и закон Гилдера. Закон Мура гласит о ежегодном удвоении вычислительных мощностей, доступных за фиксированную стоимость, закон Гилдера – об утроении пропускной способности каналов связи за тот же период. Таким образом, рост вычислительной мощности узлов сети отстает от роста скорости передачи информации в сетях, что с каждым годом ужесточает требования к вычислительной сложности алгоритмов систем защиты информации, к которым относятся и системы обнаружения атак (далее – СОА).

Первые СОА, получившие широкое распространение, появились в середине 90-х годов XX века [1]. Среди них следует выделить систему Snort [2], которая в настоящее время активно используется, развивается и, благодаря открытой архитектуре, часто используется в качестве основы для некоммерческих (Prelude IDS [4]) и коммерческих систем, таких как Форпост [46], Sourcefire 3D [47], NitroGuard [48]. Среди коммерческих систем можно отметить RealSecure [3], Arbor Peakflow [5], Cisco Guard [6].

Можно выделить следующие задачи, которые являются наиболее сложными при построении систем анализа трафика на типовой аппаратуре с использованием операционных систем общего назначения:

- сбор сетевого трафика без потерь;
- анализ полученных данных в режиме реального времени;
- реагирование с целью минимизации потенциального ущерба.

Существующие средства захвата трафика демонстрируют крайне низкую эффективность – в одном из тестов библиотеки libpcap (стандартного средства захвата трафика в современных ОС) под операционной системой на базе ядра Linux 2.6.1 успешно захватывались лишь от 1% до 34% в зависимости от размера пакета от всего сетевого трафика в ~100Мбит/с [7]. Данная проблема может быть решена модификацией сетевых драйверов, а также перемещением анализа пакетов на ядерный уровень. К таким модификациям относится запрет прерываний на сетевых устройствах [8], использование

разделяемых областей памяти между ядром ОС и анализирующим трафик приложением [9]. Данные методы позволяют практически без потерь осуществлять захват всех сетевых пакетов при загрузке канала в 970 Мбит/с [7].

Приблизив пропускную способность системы захвата пакетов к 1 Гбит/с, возникает необходимость поддерживать соответствующую скорость анализа событий. Существуют различные подходы к снижению среднего времени на обработку отдельного события с целью достижения анализа в реальном времени. Скорость анализа, прежде всего, зависит от вычислительной сложности используемого метода обнаружения атак. Методы обнаружения атак, используемые в СОА, обычно делят на два больших класса [10].

Первый класс методов – это обнаружение злоупотреблений (как правило, это сигнатурные методы). Сигнатура – это описание значений полей заголовков и содержимого сетевых пакетов, характерных для известной атаки. В качестве сигнатуры может применяться строка символов, набор строк в заданной последовательности, регулярное выражение. Помимо сигнатур атак, в последнее время получили активное развитие сигнатуры уязвимостей, покрывающие целый класс возможных атак на одну уязвимость [11, 12].

Другой класс методов – обнаружение аномалий [45]. Эти методы сопоставляют наблюдаемую активность с нормальной, безопасной активностью, и в случае значительных различий, генерируют сообщения об аномалии. К данному классу относятся статистические, иммунные методы, а также различные технологии извлечения знаний и машинного обучения [5, 6].

Подавляющее большинство существующих в настоящее время СОА используют именно сигнатурные методы, а число сигнатур в современных системах достигает нескольких тысяч (в системе Snort – около 7 тысяч [13]).

Типичная сигнатура СОА представляет собой упорядоченный набор проверок над заголовком и содержимым сетевого пакета – значения характерных полей заголовков, характерные текстовые подстроки, соответствие регулярным выражениям и т.п. Обнаружение аномалий также подразумевает множество тяжеловесных операций над полученным событием, направленных на сопоставление данного события с ожидаемыми событиями. Таким образом, анализ одного сетевого пакета выливается в сотни тысяч машинных операций, что является одним из факторов, препятствующих широкому использованию таких систем на высокоскоростных сетевых каналах – до 1 Гбит/с и выше.

Наиболее распространенный метод повышения эффективности СОА – расщепление потока событий. Суть метода заключается в разделении потока событий на множество потоков меньшего объёма, которые независимо анализируют несколько

анализаторов. При анализе сетевого трафика этого можно достичь, используя аппаратный «секатор» (slicer), получающий весь трафик сетевого канала, и распределяющий его в равной пропорции по отдельным узлам-анализаторам. Существенные недостатки данного метода – неполнота данных у всех датчиков, что может не позволить выявить комплексную атаку, а также низкая надежность при высокой стоимости эксплуатации. В общем случае распараллеливание анализа трафика данным способом неэквивалентно по результатам анализа непараллельной версии анализатора.

Другое актуальное направление – снятие нагрузки с процессоров общего назначения путем переноса вычислений на другую, специализированную аппаратуру. В частности, многие исследователи рассматривают графические карты как относительно дешевый и удобный способ организации параллельных вычислений [14, 15]. Данный метод обычно используется для выполнения специфических задач, предполагающих однотипную обработку всех событий. В частности, в работе [15] пропускная способность модуля поиска множества подстрок в пакете была увеличена с 600 Мбит/с до почти 1400 Мбит/с за счёт вынесенной на GPU реализации алгоритма Ахо-Корасик поиска множества подстрок в теле пакете. Однако, данные оптимизации неприменимы к СОА, которая анализирует трафик множеством различных методов.

В рамках работы [16] нами была оптимизирована обработка трафика в рамках среды AURA путем построения дерева предикатов проверок на этапе компиляции правил анализа, за счёт чего каждый предикат вычислялся только один раз, и общее количество проверок было снижено в несколько раз. В настоящей работе продолжена работа по ускорению анализа трафика в среде AURA за счёт использования многоядерных процессорных архитектур и распараллеливания ядра анализа с сохранением эквивалентности поведения параллельного ядра поведению однопоточного ядра AURA.

2 Постановка задачи

Цель работы: Повышение производительности средства анализа сетевого трафика за счёт использования параллелизма при сохранении программной и функциональной совместимости с существующей среды прогона (RTS - RunTime System) языка AURA.

Постановка задачи: Задачей дипломной работы является создание параллельного аналога RTS специализированного языка AURA для выполнения на многоядерных процессорах архитектур IA32, AMD64, разработка автоматизированного теста (бенчмарка) для численной оценки производительности RTS и экспериментальная оценка производительности параллельной RTS.

Для выполнения данной задачи необходимо решить следующие подзадачи:

1. Разработать и обосновать архитектуру RTS языка AURA, включая алгоритмы планирования, организацию очередей задач, механизмов IPC и синхронизации.
2. Реализовать параллельную RTS языка AURA для ОС Linux, Windows 2000/XP.
3. Разработать автоматизированный тест производительности RTS с учётом особенностей существующей базы программ на языке AURA – использование регулярных выражений, алгоритма Ахо-Корасик в предикатах и т.д. Оценить производительность существующей и разработанной реализации RTS с помощью данного теста и сравнить производительность реализаций.

3 Обзор средств параллельного анализа асинхронных потоков событий

В данном разделе представлен обзор предметной области. Основные объекты обзора и сравнительного анализа – специализированные параллельные языки программирования и параллельные расширения языков общего назначения, которые могут быть использованы для организации высокоскоростного событийно-ориентированного анализа. Результаты обзора использованы в дальнейших разделах для обоснования принятых решений при разработке параллельного ядра анализа среды AURA.

3.1 Критерии обзора

В качестве основных критериев для сравнения параллельных систем анализа событий в данном обзоре приняты следующие:

1. Событийная ориентированность. Так как сетевой трафик в общем случае состоит из пакетов, то поток событий является наиболее естественным его представлением. Возможные значения критерия: поддержка на уровне языка, на уровне IPC, отсутствие поддержки.
2. Неудобства программиста при разработке анализаторов сетевого трафика. Данный критерий отражает объем накладных расходов на рутинные операции – обеспечение корректной работы параллелизма, передача событий обработчикам и другие. Значения критерия: перечисление неудобств.
3. Поддержка реального времени. Задача анализа трафика в режиме реального времени обладает свойствами задачи реального времени, поэтому поддержка механизмов RT является важным критерием. При оценке средств по данному критерию мы также будем оценивать механизмы работы с памятью и связанные с этим ограничения на функционирование в режиме ядра ОС. Возможные значения критерия: «жёсткое реальное время», «мягкое реальное время», «поддержка отсутствует».
4. Используемые механизмы IPC. К ним относятся сообщения, общая память, синхронизация и другие. Возможные значения: перечисление механизмов.
5. Переносимость. В настоящее время существует большое количество многоядерных процессоров, отличающихся архитектурой: IA32, IA64, AMD64, SPARCv9, MIPS и т.д. Переносимость кода анализатора между аппаратными платформами является важной с точки зрения выбора наиболее подходящей для конкретной задачи аппаратной платформы. Возможные значения критерия:

низкая переносимость (наличие одной или нескольких поддерживаемых платформ), реализации под различные платформы (когда для каждой из платформ система анализа реализуется заново, а приложения требуют перекомпиляции при смене платформы), байт-код (единожды скомпилированный анализатор работает всюду, где обеспечена работа байт-кода), трансляция в C++, интерпретация кода (набор платформ, для которых реализован интерпретатор).

Рассматриваемые в обзоре средства упорядочены в порядке снижения описательной мощности: от высокоуровневых языков общего назначения и их параллельных расширений к проблемно-ориентированным языкам, которые не являются полными по Тьюрингу и ориентированных исключительно на частную задачу анализа потока байтов. Сначала будут рассмотрены библиотеки обеспечения параллелизма для C/C++ и языки общего назначения, ориентированные на параллельное выполнение. Затем проблемно-ориентированные языки, относящиеся к той же области, что и язык AURA. В последнюю очередь будут рассмотрены языки средств анализа сетевого трафика.

3.2 Библиотеки C/C++, обеспечивающие параллелизм

3.2.1 POSIX Threads

POSIX Threads – POSIX стандарт [22], описывающий API управления нитями (легковесными процессами, разделяющими общую память, опционально имея при этом собственную) в многопоточных приложениях.

Стандарт описывает широкий диапазон возможностей по части IPC: мьютексы, блокировка на чтение/запись, ожидание барьеров, завершения других нитей, выполнения условий.

POSIX Threads имеет несколько уровней [23]:

- Стандартный API, необходимое для работы с нитями,
- API, необходимый для поддержки реального времени, включающее в себя управление планировщиком нитей и приоритеты на блокировках,
- Расширенный API реального времени.

Все реализации POSIX Threads обязаны поддерживать стандартный API, остальные же уровни опциональны и являются расширениями.

Механизмы отправки сигналов, conditional variable и другие позволяет говорить о возможности реализации событийно-ориентированного средства анализа с

использованием библиотеки, реализующей данный стандарт даже на минимальном уровне.

При этом распараллеливание довольно эффективно, а общая память всех нитей существенно снижает издержки при обмене информацией между нитями. Однако для обеспечения стабильной и эффективный работы программист должен позаботиться о блокировке областей памяти при одновременной работе с ними нескольких нитей, а также об отсутствии ряда весьма специфических проблем, в частности race condition, dead lock и других.

Использующий интерфейс POSIX Threads алгоритм на языке C/C++ является широко переносимым, благодаря наличию реализаций POSIX Threads под множество различных платформ.

3.2.2 Нити в WinAPI

Нити в WinAPI во многом повторяют функциональность реализаций стандарта POSIX Threads. Поэтому отметим лишь основные отличия.

WinAPI реализовано только для платформ, совместимых с x86. Изначально реализация WinAPI существовала только под ОС семейства Windows, будучи частью данной системы, однако сейчас получил развитие проект Wine – open-source реализация WinAPI, позволяющая использовать данное API также и под UNIX-подобными операционными системами, но опять же только на совместимых с x86 платформах.

Кроме того, WinAPI не предоставляет интерфейсов, позволяющих обеспечить реальное время.

3.2.3 Intel TBB

Intel TBB (Threading Building Blocks) [24] – библиотека языка C++, содержащая множество инструментов эффективной параллельной обработки.

Часть этой библиотеки покрывает возможности базового стандарта POSIX Threads (нити, средства блокировки и т.д.), используя при этом родную для платформы реализацию нитей – либо реализацию POSIX Threads, либо WinAPI. Таким образом, для данной библиотеки применимо все сказанное выше.

Другая часть содержит шаблоны: алгоритмы параллельной работы (parallel_for и другие) и thread-safe контейнеры (concurrent_queue и т.п.). Данные инструменты

существенно упрощают реализацию алгоритмов анализа потоков событий, однако инфраструктура потока по-прежнему должна быть разработана.

3.2.4 OpenMP

OpenMP (Open Multi-Processing) [43, 44] – открытая спецификация программного интерфейса для использования параллельных вычислений с общей памятью, представляющий из себя набор переменных окружения, библиотечных функций и директив компилятора для языков C/C++ и Fortran.

Реализации OpenMP существуют под множество различных платформ и поддерживаются множеством компаний, включая Microsoft, Sun, Intel, IBM и HP.

Используя OpenMP, можно создавать потоки, как для различных задач, так и для работы над единым циклом for. Также имеется широкий набор синхронизационных примитивов: критические секции, барьеры, атомарные секции и другие.

На базе этих примитивов можно реализовать анализ потока событий, в частности сетевого трафика, однако это подразумевает работу по обеспечению статуса thread-safe.

3.2.5 MPI

MPI (Message Passing Interface) [38] – стандартизованное API для передачи информации, который позволяет обмениваться сообщениями между процессами, в том числе на разных компьютерах. Данный интерфейс реализует подход к параллелизму, ортогональный рассмотренному выше подходу.

Вместо разделяемой памяти предложена модель, основанная на передачи необходимых данных от узла к узлу в виде сообщений. Стандарт описывает множество различных способов взаимодействия процессов: различные варианты передачи сообщений, барьеры, запросы на взаимодействия.

Стандарт MPI имеет расширение MPI/RT [39], посвященное достижению работы в реальном времени. Кроссплатформенность достигается множеством реализаций для различных платформ и различных языков программирования, включая C++, Fortran, Python и Java.

Являясь лишь механизмом обмена сообщениями, MPI может быть использован для построения событийно-ориентированного анализатора. Однако данная архитектура все же не слишком подходит для задачи анализа трафика, где каждое копирование данных – весьма дорогостоящая процедура.

3.3 Параллельные языки общего назначения

3.3.1 Erlang

Язык Erlang [25, 26] разработан компанией Ericsson и назван в честь Агнера Крарупа Эрланга. Первая реализация языка была в 1986 году. Язык разрабатывался как язык распределенных программ, работающих в режиме «мягкого» реального времени.

Основным механизмом IPC для языка Erlang является асинхронный обмен сообщениями между процессами, что позволяет говорить о событийной ориентированности языка.

Кроме того, процессы могут устанавливать связь (link) с другими процессами и по выбору либо получать сообщение об их заверении с указанием причины, либо также завершаться.

Накладные расходы на порождение процессов и их взаимодействие незначительны.

Функциональная парадигма программирования, благодаря отсутствию присваиваний, с одной стороны позволяет писать код, не задумываясь о традиционных для императивных языков проблемах: синхронизации, ситуаций dead-lock и race condition. Однако использование исключительно функциональной парадигмы может быть сопряжено и с трудностями при реализации алгоритма анализа сетевого трафика.

Кроссплатформенность в Erlang обеспечивается наличием байт-кода, который затем выполняется виртуальной машиной.

3.3.2 NESL

Язык NESL [27] разработан в Университете Карнеги—Меллон и впервые реализован в 1993 году.

Язык не является событийно-ориентированным. В качестве модели параллелизма используется параллелизм по данным, при полном отсутствии взаимодействия между процессами. В тоже время, язык позволяет необычайно просто разрабатывать параллельные алгоритмы, обладающие неплохой производительностью.

Далее приведен пример реализации алгоритма быстрой сортировки с использованием данного языка:

```
function QUICKSORT(S) =
if (#S <= 1) then S
else
  let a   = S[rand(#S)];
```

```
    S_1 = {e in S | e < a};
    S_2 = {e in S | e == a};
    S_3 = {e in S | e > a};
    R  =  {QUICKSORT(v): v in [S_1, S_3]};
  in R[0] ++ S_2 ++ R[1];
quicksort([8, 14, -8, 5, -9, -3, 0, 17, 19]);
```

Возможна работа в режиме «жесткого» реального времени, т.к. язык позволяет производить оценки сложности и времени выполнения любого кода, т.е. worst case execution time.

Переносимость языка низкая, поскольку текущие реализации требуют использования UNIX-подобной системы. Однако перед запуском код компилируется в промежуточное представление, называемое VCODE, затем интерпретируемое.

Неудобство для задачи анализа сетевого трафика в силу слабых выразительных способностей языка, в котором отсутствуют даже структуры.

3.3.3 Oz

Язык Oz [36, 37] – мультипарадигмальный язык программирования, сочетающий в себе функциональную, процедурную и декларативную семантики. Язык разработан в 1991 году в Католическом Университете Лёвена. Имеет открытую кроссплатформенную реализацию в рамках среды разработки Mozart, портированной на различные архитектуры.

Язык ориентирован на параллелизм, который легок в использовании и при этом эффективен. Основная абстракция параллелизма в языке – thread, имеющий минимальные накладные расходы на создание поток. Потоки могут обмениваться сообщениями произвольного вида через каналы (port). Интересной особенностью языка, касающейся взаимодействия потоков, является временная приостановка выполнения выражения в случае неинициализированного аргумента, что в совокупности с другими конструктивными особенностями языка позволяет избежать необходимости в синхронизации, а также исключает возникновение «условий гонки». Далее приведен полностью корректный пример пары потоков, один из которых генерирует последовательные числа, а другой их суммирует.

```
fun {Ints N Max}
        if N<Max then
                {Delay 1000}
```

```
                N|{Ints N+1 Max}
        else nil end
end
fun {Sum S Xs}
        case Xs of X|Xr then
                S|{Sum S+X Xr}
        [] nil then
                 Nil
        end
end
local Xs Ys in
        thread Xs={Ints 1 1000} end
        thread Ys={Sum 0 Xs} end
end
```

Несмотря на отсутствие в языке событийной ориентированности, она эмулируется при помощи механизмов взаимодействия потоков. Язык имеет интерфейс к библиотекам C/C++, что делает его еще более привлекательным.

3.3.4 Smalltalk

Smalltalk [41, 42] – объектно-ориентированный язык с динамической типизацией, разработанный Xerox PARC. Основу языка составляют объекты и передача сообщений между ними. Параллелизм достигается за счет введения объекта Process, порождаемого для выполнения заданного кода. Создать процесс можно, отправив сообщение fork оператору блока:

```
[Float computeFunc] fork
```

Для синхронизации процессов используются семафоры, кроме того есть возможность задавать их приоритет, приостанавливать и продолжать из других процессов (также при помощи отправки сообщений). Защита от ситуаций гонки и других классических проблем параллельного программирования отводится на долю программиста.

Язык компилируется в кроссплатформенный байт-код, что делает его приложения переносимыми.

3.3.5 Occam

Occam [40] – разработка компании INMOS Ltd. в рамках работ по созданию транспьютеров (элемент построения многопроцессорных систем). Предполагалось использовать данный язык в качестве полноценной замены ассемблера на разрабатываемой архитектуре.

Язык ориентирован на параллелизм, и программа представляет собой комбинацию простых процессов (присваивание, ввод/вывод, бесконечное ожидание, завершение) при помощи различных связок (последовательное выполнение, параллельное выполнение, условия, циклы). Единственный механизм взаимодействия процессов – передача сообщений, используя обмен информацией без буфера типа «рандеву».

Далее приведен пример мультиплексора, бесконечно читающий из массива каналов in[] и передающий в общий канал out, используя промежуточную переменную temp.

```
WHILE TRUE
    INT temp :
    ALT i=0 FOR N
        in[i] ? temp
        out ! temp
```

Хотя существуют реализации Occam и для не-транспьютерных архитектур, переносимость программ на нем довольно низка.

3.4 Проблемно-ориентированные языки

3.4.1 STATL

STATL [19, 28, 29] – событийно-ориентированный язык, предназначенный для описания сигнатур атак, как сетевых, так и локальных.

В основе языка – модель конечных автоматов специального вида. Единичная программа языка STATL – сценарий, который представляет собой описание автомата над заданным подмножеством событий в качестве алфавита. Каждый сценарий способен при получении события породить собственную копию в измененном состоянии. Данный подход весьма удобен в контексте задачи обнаружения атак, позволяя, к примеру, иметь наблюдающий за первыми пакетами всех сетевых соединений сценарий, который в случае подозрительных данных порождает поток анализа, наблюдающий исключительно за данной сессией. Единственный механизм обмена сообщениями между сценариями – отправка событий.

Механизмов обеспечения реального времени язык не имеет. Кроме того, необходимо отметить низкую переносимость языка – сценарии необходимо собирать отдельно под каждую архитектуру в качестве динамической библиотеки. К поддерживаемым системам относятся ОС Linux, Solaris и Windows NT. Далее приведен пример сценария на языке STATL, отслеживающего создание файлов.

```
use ustat;
scenario ftp_write
{
        int inode;
        initial state s0 { }
        transition create_file (s0 -> s1)
        nonconsuming
        {
                [WRITE w] : (w.euid != 0) &&
                (w.owner != w.ruid)
                { inode = w.inode; }
        }
        state s1 { }
}
```

3.4.2 Esterel

Язык Esterel [30, 31] разрабатывается как язык систем реального времени. Компилятор данного языка создает код аналогичной функциональности на языке C или на языке описания аппаратуры (VHDL либо Verilog).

Язык описывает множество конечных автоматов, меняющих состояния по получении сигналов, т.к. является автоматным событийно-ориентированным языком. Помимо обмена сигналами, в качестве IPC язык поддерживает посылку “valued” сигналов, представляющих собой связку сигнал-данные.

Автоматная семантика языка позволяет реализовывать системы жесткого реального времени, однако использование исключительно детерминированных конечных автоматов, состав которых предопределен при запуске, может существенно ограничить программиста в специфических задачах, например при наблюдении за несколькими сетевыми сессиями.

Тем не менее, для отдельно взятых задач анализа данный язык исключительно прост и удобен. Далее представлен пример, программного модуля, который посылает событие O по получении событий A и B, обнуляя свое состояние по приходу события R:

```
module ABRO:
input A, B, R;
output O;
loop
    [ await A || await B ];
    emit O
each R
end module
```

3.5 Языки анализа сетевого трафика

3.5.1 VESPA

VESPA [32, 33] – ядро системы разбора протоколов, проверяющей сигнатуры уязвимостей. В основе языка лежат процедуры разбора бинарных и текстовых протоколов, вызываемые при получении пакета соответствующего протокола, таким образом можно говорить о событийной ориентированности, ограниченной пакетами. С процедурами разбора ассоциированы вызываемые после них обработчики. Пример разборщика и обработчика:

```
bool is_post = str_matcher “POST” handler handle_post() %{
            is_post = true;
    }%
handle_post() %{
    if(is_post)
            deploy(content_length);
}%
```

Переносимость обеспечена конвертацией модуля на языке VESPA в код на языке C++. Платформа VESPA находится в разработке и пока не обладает средствами обеспечения параллелизма, однако параллельная проверка сигнатур заявлена как один из основных векторов развития.

3.5.2 Bro Language

Bro IDS [34, 35] использует собственный событийно-ориентированный язык сценариев. Набор событий ограничен встроенными событиями сенсоров системы, однако может быть расширен в силу открытости системы.

Система Bro IDS портирована на различные архитектуры, а сценарные файлы являются интерпретируемыми, что обеспечивает их работу на различных платформах.

В данный момент Bro IDS выполняет обработку сценариев в одном потоке, однако сценарии могут обмениваться информацией через глобальные переменные. Механизмами поддержки реального времени язык не обладает.

Далее приведен пример сценария, собирающего список узлов и портов, к которым обращаются узлы внутренней сети. Один из сценариев собирает статистику по мере открытия сессий, другой печатает собранные данные в конце работы системы. Ключевое слово persistent означает, что данные прошлых запусков системы сохраняются.

```
global destinations: set[addr, port] &persistent;
event connection_established(c: connection)
{
        local dst_host = c$id$resp_h;
        local dst_port = c$id$resp_p;
        if ([dst_host, dst_port] in destinations)
                return;
        add destinations[dst_host, dst_port];
}

event bro_done()
{
        print "Outgoing traffic ";
        for ( [host, p] in destinations )
                print fmt("%20s %s", host, p);
}
```

3.6 Результаты обзора

В таблице 1 кратко описаны результаты сравнительного анализа рассмотренных систем и языков.

	Событийная ориентированность	**Неудобства для задачи анализа трафика**	**Поддержка реального времени**	**Разновидности IPC**	**Переносимость**
POSIX Threads	можно реализовать средствами IPC	рутинная работа по обеспечению статуса thread-safe	в некоторых расширенных реализациях	множество способов	реализации под различные платформы
WinAPI	можно реализовать средствами IPC	рутинная работа по обеспечению статуса thread-safe	нет	множество способов	низкая

Intel TBB	можно реализовать средствами IPC	необходимость реализовывать событийную ориентированность	нет	множество способов	реализации под различные платформы
OpenMP	можно реализовать средствами IPC	рутинная работа по обеспечению статуса thread-safe	нет	множество способов	реализации под различные платформы
MPI	можно реализовать средствами IPC	копирование памяти влечет накладные расходы	в некоторых расширенных реализациях	множество способов	реализации под различные платформы
Erlang	да	только функциональная парадигма программирования	нет	сообщения и связи	байт-код
NESL	нет	отсутствие взаимодействия между параллельными потоками	есть	нет	байт-код
Oz	можно реализовать средствами IPC	необходимость реализовывать событийную ориентированность	нет	множество способов	реализации под различные платформы
Smalltalk	да	рутинная работа по обеспечению статуса thread-safe	нет	сообщения, семафоры и управление	байт-код
Occam	можно реализовать средствами IPC	необходимость реализовывать событийную ориентированность	нет	сообщения	низкая
STATL	да	нет	нет	отправка сообщений	низкая
Esterel	да	ограниченность детерминированными конечными автоматами	есть	сигналы, сообщения	трансляция в C++
VESPA	набор событий определяется сенсором системы	нет	нет	параллелизм в разработке	трансляция в C++
Bro Language	набор событий определяется сенсором системы	нет	нет	параллелизм в разработке	интерпретация

Таблица 1. Сводная таблица результатов обзора

Средства, разработанные специально для анализа сетевого трафика (STATL и языки анализа сетевого трафика), не обеспечивают поддержку реального времени и многопоточную обработку (существующие в них сценарии анализа выполняются последовательно в едином процессе), обладают низкой переносимостью. Способ реализации событий в языках анализа сетевого трафика (VESPA, Bro Language) не позволяет генерировать собственные события, что затрудняет их иерархическую обработку, когда результатом работы одного анализатора являются события, подаваемые на вход другому анализатору более высокого уровня.

Поддерживающие работу в реальном времени средства анализа отличаются либо описательной бедностью языка (Esterel ограничен конечными автоматами, а в NESL отсутствуют взаимодействия между параллельными потоками во время их выполнения), либо отсутствием встроенной событийной ориентированности (в случае специализированных библиотек C/C++).

В языках общего назначения отсутствует поддержка модели событий на уровне языка, а её организация требует много рутинной работы по обеспечению статуса thread-safe (борьба с условиями гонки, взаимными блокировками и т.п.), что ведёт к большому объёму накладных расходов на решение задачи анализа трафика при использовании

библиотек обеспечения параллелизма в C/C++ и параллельных языках (Oz, Occam, SmallTalk).

Среди рассмотренных средств лучше всего критериям сравнения соответствует язык Erlang, но данный язык использует неявные механизмы управления динамической памятью (включая сборку мусора) и в настоящее время не имеет встроенных механизмов обеспечения режима реального времени, что затрудняет или не позволяет вынести анализаторы трафика, использующие Erlang, на уровень ядра ОС.

Таким образом, ни одно из рассмотренных средств не удовлетворяет в полной мере всем критериям обзора. При разработке среды анализа событий и языка программирования AURA, в рамках которых выполнена данная работа, учитываются следующие ключевые требования:

1) язык среды должен использовать автоматную парадигму;
2) код анализатора должен быть переносим;
3) среда прогона должна обеспечивать планирование выполнения анализаторов на многопроцессорных системах прозрачно для программиста анализаторов;
4) анализаторы не могут явно управлять памятью, при этом динамическое выделение памяти возможно только через стандартные структуры данных, управляемые системой прогона.

В следующем разделе приводится краткое сравнение среды AURA с рассмотренными в обзоре средствами.

3.7 Сравнение AURA с рассмотренными средствами

3.7.1 Описание языка

В среде AURA используется проблемно-ориентированный язык AURA [18] для описания алгоритмов анализа поведения объектов в сети и на узлах в терминах конечных автоматов. Язык AURA основан на идеях работ Энкмана, Вигны и Кеммерера, языке STATL [19], а также включает в себя элементы языков C и C++.

Единица языка – сценарий – представляет собой реализацию некоторого алгоритма анализа входных событий в терминах альтернирующих автоматов, с описанием множества состояний и переходов между ними по событиям. В процессе выполнения каждому сценарию соответствует некое дерево экземпляров, которое всегда зависит от реальной истории событий. Формально сценарий представляет собой структуру следующего вида:

$$Scenario : (S, P_S, T, P_T, s_0, I, g, q)$$, где

S – множество состояний;

P_S – множество логических предикатов состояний;

T – множество переходов;

P_T – множество логических предикатов переходов;

s_0 – начальное состояние;

I – множество экземпляров автомата;

g – глобальное окружение;

q – глобальная очередь таймера.

Множество экземпляров $I = \{i\}, i = \sum^{+}$ – непустая последовательность срезов. Каждый срез $\sigma \in \sum : N_K \times L \times Q \times E$ характеризует текущее состояние автомата и содержит имя текущего состояния (элемент пространства имен состояний), локальное окружение, локальную очередь таймера и локальную очередь событий (строка входного алфавита). По сути, срез отражает историю выполнения дерева экземпляров сценария в конкретной точке.

Программа на языке AURA с помощью компилятора транслируется в аппаратно-независимый байт-код LLVM [20], который транслируется в машинный код целевой аппаратной платформы при первом выполнении в теле анализатора с помощью компилятора LLVM.

Когда сценарий загружается системой, создается соответствующий ему корневой экземпляр сценария, принимающий поочередно события и выполняющий переходы, в случае их соответствия условиям перехода. Переходы бывают трех типов – consuming (выполнить данный переход в текущем экземпляре сценария), nonconsuming (породить дочерний экземпляр сценария, точную копию текущего, и выполнить переход в нем) и unwinding (после выполнения тела перехода, уничтожить текущий экземпляр сценария и все дерево его потомков). Таким образом, каждому сценарию соответствует дерево экземпляров сценария.

Каждому переходу соответствует тип перехода, тип принимаемого события, начальное и конечное состояние, логический предикат (условие перехода) и полезная нагрузка (тело перехода). Каждое событие – структура данных определенного типа, соответствующего типу события. При получении очередного события экземпляром сценария перебираются те переходы, которые выходят из данного состояния и принимают события данного типа. При этом вычисляется соответствующее условие перехода. В случае его истинности, осуществляется реализация семантики типа перехода (т.е.

возможно создание новых экземпляров сценария), после чего выполняется код, записанный в теле перехода.

Дальнейший просмотр списка переходов не осуществляется – одно событие не может вызвать более одного перехода в одном экземпляре сценария. Порядок обхода экземпляров явно не специфицирован, однако гарантируется, что родительский экземпляр сценария получает событие лишь после его обработки всеми дочерними экземплярами. Кроме того, имеется функция языка, прекращающая распространение события по экземплярам сценария.

Далее рассмотрен пример сценария, осуществляющего слежение за сетевыми сессиями. Сценарий имеет 2 состояний – начальное, ожидающее новых сессий, и рабочее, наблюдающее за определенной сессией. При получении очередного события о сетевом пакете проверяется, не относится ли он уже к какой-нибудь из замеченных ранее сессий. Если относится, производится увеличение счетчика, иначе создается новый экземпляр, предназначенный для слежений за данной сессией. По завершению сессии печатается число пакетов в ней.

Пример сценария на языке AURA:

```
scenario TCPGather ( NetTCPStreamEvent
streamEv, NetTCPStreamCloseEvent closeEv )
{
   u_int_32       sess;
   u_int_32       n;
   string TCPStateToString ( TCPState tcpState )
   {
       return IPtoa(tcpState.srcAddress) + ":" +
           utoa(tcpState.srcPort) + "->" +
           IPtoa(tcpState.destAddress) + ":" +
           utoa(tcpState.destPort);
   }
   initial state state0 {}
   state state1 {}
   nonconsuming transition state0->state1
   event NetTCPStreamEvent ( true )
   {
      sess = streamEv.tcpStreamId; n=1;
   }
   consuming transition state1->state1
   event NetTCPStreamEvent
   ( streamEv.tcpStreamId == sess )
   {
      n++; StopEventProcessing ();
   }
   unwinding transition state1->state1
   event NetTCPStreamCloseEvent
   (closeEv.tcpStreamId==sess&&closeEv.tcpStreamDirection==IDS_TCP_FORWARD)
```

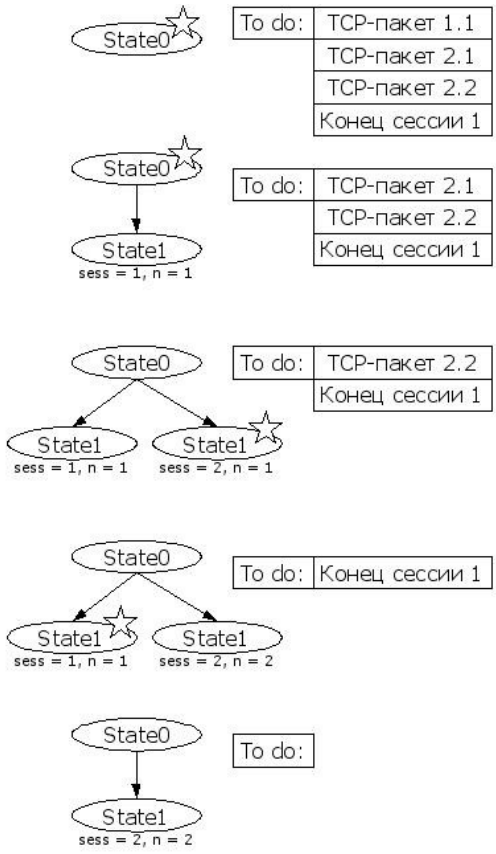

Рисунок 1. Порождение экземпляров сценария при обработке потока событий сценарием

```
    {
        DebugPrint (TCPStateToString(closeEv) + " closed\n" + itoa(n)+ " packets\n");
    }
};
```

Рисунок 1 иллюстрирует изменения состояния дерева экземпляров сценария, происходящие при обработке событий. Изначально в очереди на обработку 4 события. Первое из них применимо (применимость отмечена звездочкой) к корневому экземпляру (предикат true). Это приводит к nonconsuming-переходу, порождающему экземпляр-потомок. Следующее событие может не применимо к экземпляру-потомку (предикат streamEv.tcpStreamId == sess ложен), но применимо к корневому (предикат true), что порождает еще один экземпляр-потомок. Третье событие не применимо к первому потомку, но применимо ко второму. В ходе обработки производится вызов StopEventProcessing(), что прекращает анализ события экземплярами сценария. Последнее событие применимо к первому экземпляру-потомку, который уничтожается после выполнения unwinding-перехода.

В состав любого сенсора AURA входит набор сценариев на данном языке и система прогона. Задача системы прогона – формирование входного потока событий для сценариев и планирование порядка выполнения предикатов и переходов из состояния в состояние для каждого сценария.

На момент начала выполнения настоящей дипломной работы система прогона была условно-параллельна – поток событий и выполнение сценариев выполняются независимо. Но при этом в каждый момент времени, как и во всех других современных СОА, обрабатывается только одно событие, независимо от числа доступных процессоров. Работа «старой» системы прогона сводится к взятию очередного события из очереди и последовательному применению данного события ко всем сценариям, его принимающим (для каждого экземпляра такого сценария последовательно ищется переход с истинным предикатом).

3.7.2 Достоинства и недостатки по сравнению с рассмотренными средствами

Язык AURA как развитие языка STATL обладает всеми его достоинствами, в частности, удобной для обработки событий сетевого трафика семантикой альтернирующих автоматов. Вместе с тем, сценарии AURA обладают высокой переносимостью благодаря компиляции в платформо-независимый байт-код LLVM. Кроме того, сценарии AURA полностью реализуют событийно-ориентированную

семантику, т.е. обработчик событий может порождать собственные события. Помимо генерации событий, сценарии имеют дополнительную возможность по межсценарному взаимодействию в виде статических переменных.

В то же время, на момент начала выполнения данной дипломной работы, язык AURA не обладал механизмами обеспечения параллельного выполнения сценариев и механизмов обеспечения режима мягкого реального времени. В настоящей работе система прогона AURA дополнена возможностью параллельного выполнения сценариев на многопроцессорных и многоядерных системах, при которой результат выполнения эквивалентен непараллельному варианту. Обеспечение реального времени частично выполняется в данной работе за счёт введения приоритетов сценариев, более полная поддержка планируется в будущем, как за счёт профилировки на типовых данных, так и за счёт введения оценки наихудшего времени выполнения в язык.

4 Исследование и построение решения задачи

4.1 Требования к модели параллельного выполнения

В данном разделе производится поиск наиболее удачной модели параллельного выполнения сценариев в ядре среды AURA. Критерием сравнения будет являться наиболее полное выполнение приведенных ниже требований.

Требуется такая модель параллельного выполнения, которая позволила бы параллельно обрабатывать одно событие множеством сценариев, а также множеству экземпляров одного сценария также обрабатывать события параллельно, если это не нарушает правил получения событий экземплярами («потомок получает события раньше родителя», «события приходят в порядке их генерации»). Кроме того, от системы параллельного прогона сценариев также требуется поддержка приоритетов сценариев, в том числе приоритета мягкого реального времени.

При построении такой модели, нужно иметь в виду ряд особенностей языка:

- Порядок обработки события различными сценариями не детерминирован, возможна и обработка одного события в параллель. Если данная обработка допустима, необходимо введение в язык защищаемых секций кода.
- Экземпляр сценария не может начать обрабатывать событие, пока не обработал все прошлые события. Таким образом, события нельзя переупорядочивать.
- Экземпляр-родитель не может начать обрабатывать событие, пока его не обработали все его потомки. Таким образом, зависимости между экземплярами имеют древовидный характер, и дерево может быть сколь угодно ветвистым.
- Если в процессе обработки события некоторым сценарием был порожден новый экземпляр сценария, то он должен обработать все последующие события, т.е. включиться в планирование немедленно. Таким образом, появление экземпляра должно автоматически гарантировать его добавление в очередь на обработку уже поставленных в очередь событий.
- Обработка некоторого события некоторым экземпляром сценария занимает непрогнозируемое время, элемент очереди не может иметь никаких существенных для планирования статических параметров, за исключением назначенного извне приоритета. Таким образом, невозможно планирование по принципу Shortest-Job-First и ему подобные.
- Типичные сценарии подразумевают обработку события либо всем деревом экземпляров, либо некоторым поддеревом, либо одной вершиной. При этом

порядок обработки потомков произволен и все потомки должны обработать события.

Помимо особенностей непосредственно языка AURA, следует учесть и особенности существующих модулей сценариев, написанных на данном языке. Для этого проанализируем имеющееся множество сценариев и выделим несколько наиболее характерных случаев:

- Модуль сигнатурного анализа сетевых пакетов. Данный модуль содержит несколько сотен сценариев, каждый из которых во время работы состоит всего из одного экземпляра сценария. Эффективное выполнение данного модуля сценариев требует возможность выполнять различные сценарии параллельно.
- Модуль реконструкции дерева процессов и сопоставления сетевой активности с группой процессов. Данный модуль состоит из одного сценария, который во время работы может порождать множество экземпляров сценария. Эффективное выполнение данного модуля сценариев требует возможность выполнять различные экземпляры сценария одновременно. Кроме того, данный модуль использует события разной природы, что также необходимо учитывать.
- Модуль профилирования сетевых сессий. Данный модуль состоит из одного сценария, во время выполнения которого порождается множество экземпляров сценария, обменивающихся информацией посредством статических переменных.

Все рассмотренные схемы предполагают некоторое количество поставщиков событий и некоторое количество обработчиков. Поставщики событий формируют задания и добавляют их в очередь (одну из очередей), а обработчики забирают из очереди назначенные им задачи и производят их обработку.

4.2 Расщепление потока событий

Одним из возможных путей достижения параллелизма может быть простое расщепление потока событий на домены. На каждом вычислителе одинаковый набор сценариев, но со своим корневым экземпляром сценария и деревом потомков. Тэг события выставляется его поставщиком, каждый вычислитель имеет набор тэгов, выполняемых на нем. При появлении нового события, его отправляют в очередь на обработку исполнителя, имеющего ту же метку, если такой есть, иначе в очередь наименее загруженного. Событие обрабатывается сценарием в соответствии с указанными в пункте 4.1 особенностями.

В такой схеме элементом очереди является событие, тэг специфичен для события и определяется поставщиком, параллельность лишь по событиям (разные данные на разных процессорах).

Обращаясь к типовым сценариям, заметим, что схема ведет себя довольно плохо: она не позволяет обрабатывать одно событие множеством сценариев одновременно, т.к. событие жестко привязано к одному исполнителю. Расщепление событий на различные потоки, согласно описанной выше технике, затруднительно, т.к. среда AURA оперирует множеством разнотипных событий, для которых подчас невозможно найти общего признака для расщепления, например, сообщения о системных вызовах, сетевых пакетах, операциях с файловой системой. И в довершение всего, данная схема существенно затрудняет обмен информацией между сценариями на различных исполнителях. Все это делает ее неприменимой для среды AURA.

Тем не менее, эта простая схема могла бы быть использована для распараллеливания, например, ядра анализа COA Snort, анализирующего лишь сетевые пакеты, и поддерживающего память лишь в рамках анализа отдельной сессии.

4.3 Разделение сценариев на домены

Другой простейший вариант – параллельный анализ несколькими неполными наборами сценариев. На каждом вычислителе свой уникальный набор сценариев. При появлении нового события, оно добавляется во все очереди. Событие обрабатывается сценарием в соответствии с указанными в пункте 4.1 особенностями.

В такой схеме элементом очереди является событие, параллельность лишь по сценариям (разные сценарии на разных процессорах).

Данная схема позволяет параллельно обрабатывать одно событие множеством сценариев, однако не позволяет эффективно работать с модулями, содержащими лишь один сценарий. Кроме того, возникают проблемы с балансировкой нагрузки, которые не могут быть решены простым перемещением сценариев между исполнителями в силу различного состояния очередей.

Таким образом, разделение сценариев на домены, хотя и позволяет решить некоторые проблемы, не является вполне удачным решением.

4.4 Планирование на основе экземпляров сценария

Анализ существующего множества сценариев выявил важную особенность – часто условием перехода является выражение, включающее в себя лишь условия над полями события и не имеющие побочных эффектов. Данная схема существенно использует этот факт.

Данная схема предлагает подход, в некотором смысле противоположный описанным выше схемам. Если ранее элементами очереди служили отдельные события, то теперь планирование производится на основе совсем мелких единиц – отдельных экземпляров сценария и даже переходов.

Схема оперирует двумя очередями. В первой очереди находятся потенциальные задачи – пары <Событие, Экземпляр Сценария>. Во второй очереди находятся ожидающие выполнения задачи – тройки <Событие, Экземпляр Сценария, Переход>. При появлении нового события для каждого экземпляра каждого сценария просматриваются таблицы переходов на предмет перехода, который мог бы выполниться при получении данного события. Пусть переход найден и его условие может быть вычислено сразу, то есть условие перехода тождественно равно истине или сводится к серии проверок полей события без побочных эффектов. Тогда тройка <Событие, Экземпляр Сценария, Переход> добавляется в очередь ожидающих выполнения задач. Если для некоторого экземпляра сценария из-за зависимости по данным не удается заранее определить истинность условия перехода, то пара <Событие, Экземпляр Сценария> добавляется в очередь потенциальных задач.

Для согласования со спецификой выполнения сценария следует принять дополнительные меры. Для обеспечения древовидной связи между экземплярами сценариев поставленные в очередь задачи должны иметь связи, а для сохранения порядка следования событий требуется не допускать выполнения переходов из очереди ожидающих выполнения задач, если в очереди потенциальных задач имеются задачи, связанные с более ранними событиями.

Consuming-переходы хорошо укладываются в эту схему. Если переход попадает в очередь ожидающих выполнения задач, экземпляр сценария сразу меняет свое состояние, что по приходу следующего события позволит просматривать таблицу переходов для уже нового состояния.

Для реализации unwinding-переходов достаточно поддерживать флаг удаления и счетчик ссылок на экземпляр сценария. Экземпляр сценария перестает обрабатывать события, если флаг удаления взведен, а уничтожается тогда, когда счетчик ссылок, соответствующий числу включающих этот экземпляр задач в обеих очередях, достигает

нуля. Помимо unwinding-перехода в стандартной библиотеке языка AURA присутствует функция Unwind(), обладающая схожим эффектом. Для её реализации достаточно взвести флаг удаления.

Серьезные проблемы начинаются при реализации nonconsuming-переходов. Если переход помещается в очередь ожидающих выполнения задач, то порождение нового экземпляра выполняется сразу, и оба экземпляра учитываются по приходу следующего события. Если же экземпляр сценария отложен в очередь потенциальных задач и может выполнить nonconsuming-переход, необходимо учитывать его потомка как некоторый виртуальный экземпляр. Фактически, все события, приходящие после такой потенциальной задаче необходимо обрабатывать так, чтобы была возможность применить их к новому экземпляру.

В итоге получаем множество довольно громоздких проверок:

- может ли событие быть обработано каким-то из переходов данного экземпляра сценария?
- может ли что-то из запланированных тел переходов изменить значение предикатов в условиях переходов в данном состоянии?
- Имеются ли связанные с данным экземпляром сценария задачи в очереди потенциальных задач

Обратимся к особенностям выполнения имеющихся модулей сценариев при использовании данной схемы. В случае модуля сигнатурного анализа пакетов, на который и была нацелена данная схема, имеем параллельную обработку события множеством сценариев. В случае модуля реконструкции дерева процессов и модуля профилировки сетевых сессий – параллельную обработку события множеством экземпляров сценария.

Однако в последних двух случаях имеем неоправданно большой размер очередей. Следующая схема решает и эту проблему.

4.5 Планирование на основе обхода дерева экземпляров

Поскольку при обработке задачи могут порождаться новые экземпляры сценария, планирование на основе отдельных экземпляров не представляется перспективной идеей. Если же взять в качестве элемента планирования целый сценарий со всеми его экземплярами, теряется возможность одновременно обрабатывать экземпляры сценария, даже на различных событиях.

В данной схеме предложен некоторый компромиссный вариант – элементом очереди задач является задача вида <Событие, Сценарий, Итератор обхода дерева

экземпляров, Финал обхода дерева экземпляров>. Итератор – текущее состояние обхода дерева, Финал – значение итератора, после которого обход следует прекратить. После каждого выполнения итератор получает значение, соответствующее вершине, которая была обработана последней. Также имеется некоторое количество исполнителей, каждый из которых берет из очереди очередную задачу и продвигает итератор, обходя экземпляры сценария, выполняя в них всю работу.

Для того чтобы была возможной одновременная обработка экземпляров одного сценария, фиксируем порядок обхода. Любой сценарий имеет дерево экземпляров: корень и созданные из него при помощи nonconsuming-переходов дети, образующие древовидную иерархию.

Зададимся вопросом: как можно обойти это дерево, учитывая, что одновременно его может обходить кто-то другой? Кроме того, алгоритм обхода должен быть не чувствительным к добавлениям и удалению экземпляров сценария в оставшейся к обходу области, а также содержать минимум информации для продолжения обхода в случае его прерывания.

Данным условиям удовлетворяет следующее правило обхода дерева: обрабатывать все деревья рекурсивно, передавая управление сначала самому старшему дочернему экземпляру, затем самому старшему из оставшихся братьев, в последнюю очередь – родителю.

Формально продвижение итератора производится следующим образом:

1. Если значение итератора не инициализировано (обход еще не был начат), то перейти в корневой экземпляр дерева и перемещаться в самого старшего потомка до тех пор, пока потомки существуют.
2. Иначе, если у экземпляра существуют младшие братья, применить описанную выше процедуру к самому старшему из них.
3. Иначе, переместиться в родительский экземпляр сценария.

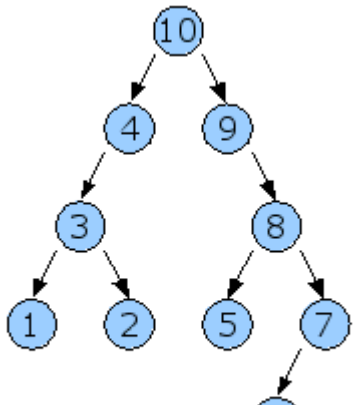

Рисунок 2. Дерево экземпляров и предложенный порядок их обхода. Старшие потомки располагаются левее, младшие – правее.

Рисунок 2 отображает порядок обхода дерева экземпляров, соответствующий данному алгоритму. Сначала итератор не инициализирован, затем переходит в вершину 1 в соответствие с шагом 1 алгоритма, затем в вершину 2, являющуюся старшим из имеющихся братьев вершины 1, затем в вершину 3, поскольку у вершины 2 нет младший братьев, и т.д.

Данное правило выполняет требование относительно порядка обработки события экземплярами сценария: родительский экземпляр получает событие лишь тогда, когда его обработали все его потомки. Данный факт также иллюстрируется Рисунком 2.

Запретим также обработку события экземпляром сценария, если он уже занят обработкой какого-либо из прошлых событий. В таком случае данный порядок обхода сохраняет единый порядок получения событий. Также стоит отметить, что порядок обхода не чувствителен к вставкам экземпляров сценария.

Рисунок 3 демонстрирует это свойство: продолжая обход дерева из узла, помеченного цифрой 5, помимо запланированных изначально узлов, будет также пройден узел 6', но не 3' и не 4'. В то же время, поскольку итераторы не могут обогнать друг друга, узел 6' мог возникнуть только при обработке предыдущего события, а 3' и 4' – лишь следующих событий. Таким образом, использование данной схемы позволяет автоматически учитывать nonconsuming-переходы, единственным накладным расходом является проверка того, не занят ли экземпляр сценария другим обработчиком.

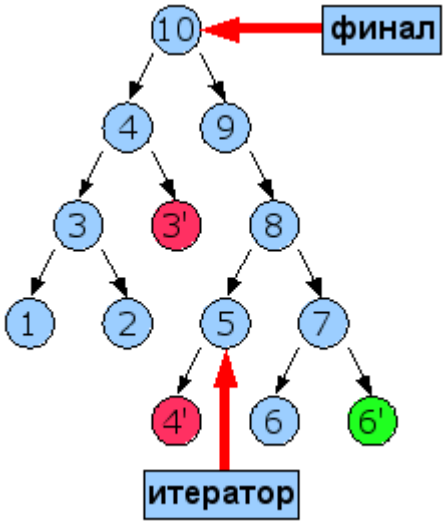

Рисунок 3. Дерево экземпляров и предложенный порядок их обхода

Поскольку дерево экземпляров может обходиться одновременно несколькими процессами, unwinding-переход не вправе осуществлять немедленное удаление поддерева. Вместо этого выставляется флаг удаления, и оно произойдет лишь тогда, когда счетчик ссылок на экземпляр сценария все его потомки достигнет нуля.

Все эти важные свойства выполняются с возможностью приостановить выполнение сценария: если исполнитель пытается перейти к экземпляру сценария, который уже задействован другим исполнителей, то первый исполнитель возвращает свою задачу в очередь, взяв оттуда другую. Во избежание нарушения порядка следования событий, в таких случаях следует выгружать также и все последующие задачи, использующие данный сценарий.

Данная схема поддерживает помимо обычных приоритетов, определяющих частоту взятия событий из различных очередей, приоритет реального времени – отдельную очередь, при появлении чего-либо в которой немедленно возвращается в очередь одна из выполняемых задач.

Если обратиться к имеющимся модулям сценариев на языке AURA, то видно, что в случае модуля сигнатурного анализа пакетов имеет место параллельная обработка пакета

множеством сценариев одновременно. В случае модулей реконструкции дерева процессов и модуля профилировки сетевых сессий возможна параллельная работа различных экземпляров одного сценария. Единственный потенциальный недостаток – большой размер очереди в случае модуля сигнатурного анализа пакетов.

4.6 Выводы

Описанная в пункте 4.5 модель параллелизма для языка AURA позволяет реализовать параллельное независимое выполнение сценариев на разных процессорах, одновременно сохраняя частичный порядок выполнения взаимозависимых экземпляров сценария, характерный для непараллельной системы прогона. При этом обеспечивается минимальный объём накладных расходов, а также учитывает особенности существующих модулей сценариев. Незначительность накладных расходов подтверждается результатами экспериментов (см. раздел 6).

5 Описание практической части

5.1 Обеспечение кроссплатформенности

Для реализации параллельных потоков управления был использован механизм нитей (threads), поскольку (по сравнению с использованием отдельных процессов) их использование существенно снижает накладные расходы на переключение контекста в ОС, обмен информацией между обработчиками и поставщиками событий, а также между обработчиками друг с другом, что крайне важно для задачи высокоскоростного анализа.

В качестве библиотеки, предоставляющей данную функциональность, была выбрана Native POSIX Thread Library (NPTL), поскольку требовалась высокая переносимость ядра анализа на различные платформы, а NPTL – реализация POSIX-стандарта. Под операционными системами семейства Windows, не поддерживающими данный стандарт, были написаны обёртки процедур pthreads с использованием процедур WinAPI.

Во избежание излишних блокировок, в коде используются атомарные операции, в частности операция atomic_xadd(addr,value). Данная операция неделима и выполняет следующие действия: прибавляет value к переменной по адресу addr, возвращая как результат работы старое значение переменной addr. В ОС семейства Windows данная операция реализуется вызовом InterlockedExchangeAdd(), а в семействе Linux – __exchange_and_add().

Данные операции удобны для организации счетчика ссылок: при создании новой ссылки прибавляем единицу, при уничтожении вычитаем единицу, и если предыдущее значение равнялось единице, то это была последняя ссылка и объект может быть уничтожен.

На этих же операциях строятся и спинлоки – легковесные блокировки, ограждающие небольшие фрагменты кода.

Для обеспечения работы скомпилированных модулей сценариев на множестве различных архитектур используется инфраструктура компилятора LLVM (Low Level Virtual Machine).

5.2 Архитектура системы

5.2.1 Компоненты ядра анализа

В силу относительно малого числа исполнителей (десятки) в рассматриваемых целевых архитектурах, в настоящее время нет необходимости выделять планировщик в отдельную нить. Вместо этого достаточно использовать структуру общей очереди, поддерживающую захват в монопольный доступ. Таким образом, основными активными компонентами являются поставщики событий (находящиеся вне ядра анализа, обращающиеся к нему лишь через интерфейс постановки события в очередь на обработку) и нити-исполнители, занимающиеся выполнением задач. Взаимодействие данных компонентов производится через структуры, к которым относится множество имеющих различный приоритет очередей задач, а также таблица исполнителей, отражающая текущую задачу и её состояние для каждого исполнителя. Рисунок 4 отражает связи между компонентами.

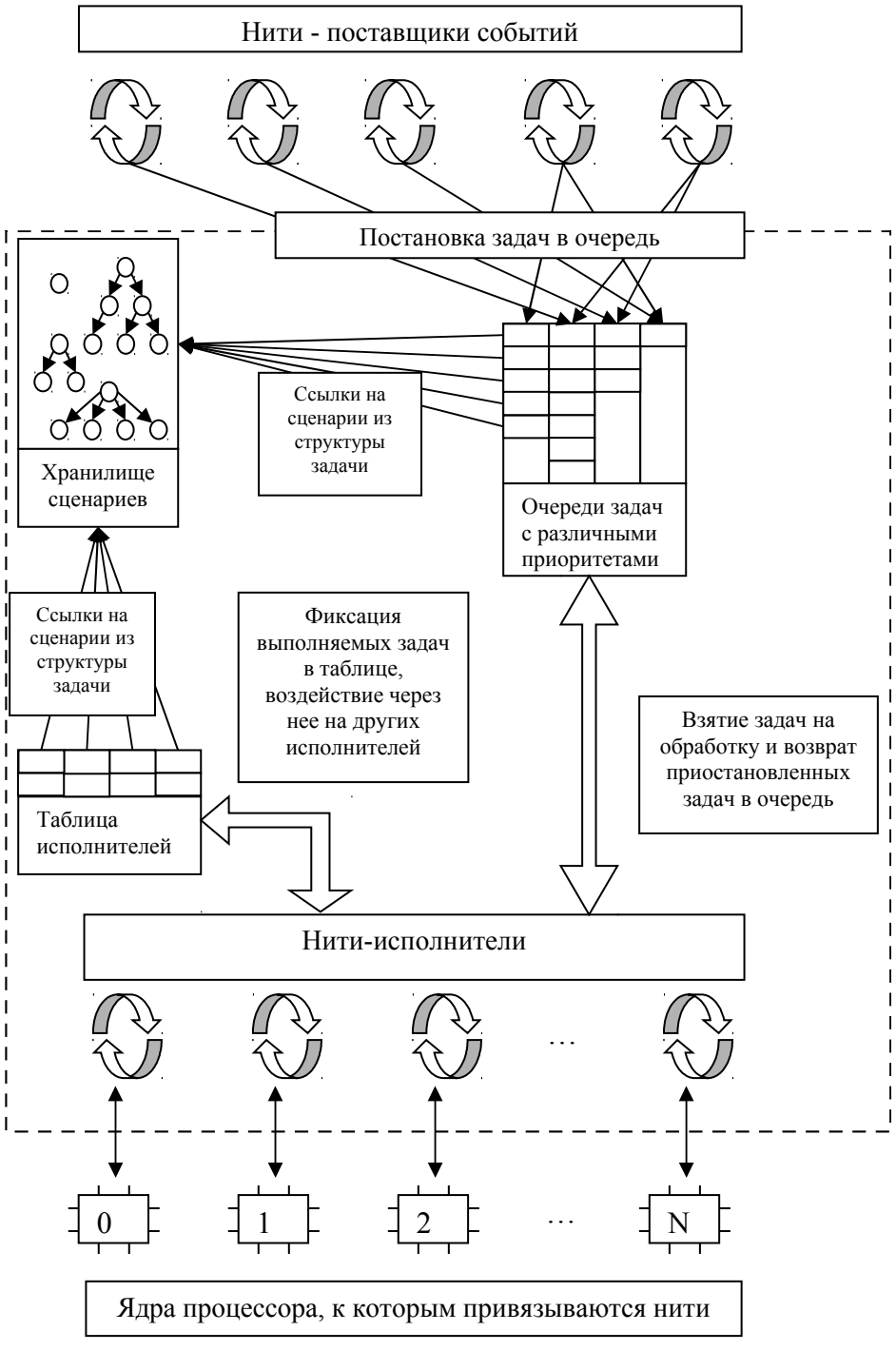

Рисунок 4. Основные компоненты архитектуры и их взаимодействия.

Хранилище сценариев реализовано как отображение типа события на список корневых экземпляров сценариев, данный тип событие обрабатывающих. Намного более интересной задачей является представление самого дерева экземпляров сценария.

В пункте 4.5 была описан алгоритм обхода дерева, а также обосновано преимущество данного фиксированного способа обхода дерева. В реализованной архитектуре дерево экземпляров сценария организовано в полном соответствии с данным алгоритмом. Для этого, каждый экземпляр имеет пять указателей: на родительский экземпляр, на своих братьев, на самого младшего и самого старшего потомка. Рисунок 5 иллюстрирует связь между экземплярами сценария.

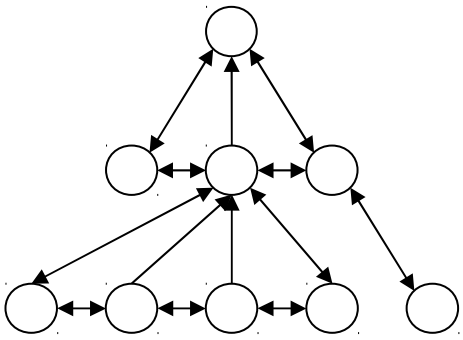

Рисунок 4. Связи между экземплярами сценариев.

Итератор обхода дерева реализован в виде указателя на вершину, которая была обработана последней (начальное значение равно NULL). Для работы с итератором имеется пара примитивов:

```
IDSContext* IDSContext::GetFirstInTree()
{
   IDSContext* ret = this;
   while(ret->firstChild != NULL) ret = ret->firstChild;
   return ret;
}
IDSContext* IDSContext::GetNextInTree()
{
   if( next != NULL) return next->GetFirstInTree();
   else return parent;
}
```

Далее приведен способ вычисления следующего значения итератора (экземпляра, который будет обработан следующим). Если итератор равен NULL, то следующее значение равно результату метода GetFirstInTree() вызванного из корневого экземпляра сценария, т.е. выбираем старшего потомка, пока они существуют, начиная двигаться от экземпляра сценария, которым должны закончить обработку. Иначе из текущего экземпляра сценария вызывается метод GetNextInTree(). Данная пара методов реализует необходимый порядок обхода экземпляров сценария.

В случае порождения новых экземпляров (при nonconsuming переходе) новый экземпляр рождается с указателем на родителя, а также на самого младшего из потомков родителя в качестве брата, все остальные ссылки устанавливаются в NULL. Самый

младший из потомков родителя получает ссылку на новый экземпляр в качестве младшего брата, родитель меняет ссылку на новый экземпляр в качестве самого младшего потомка. Если же у родителя до этого потомков не было, то новый экземпляр указывается также и как самый старший потомок тоже.

В случае уничтожения экземпляров (при unwinding переходе) в экземпляре сценария и всех его потомках взводится флаг удаления. Само удаление происходит в момент, когда счетчик ссылок в экземпляре достигает нуля. Счетчик ссылок считает число запланированных задач, включающих данный экземпляр сценария и его потомков в качестве итератора или финала обхода. Этот момент наступит, т.к. задачи не хранятся в очереди вечно, а новые задачи такие экземпляры при обходе пропускают.

Каждый экземпляр сценария при создании получает метку-тэг, определяемую текущим значением постоянно увеличивающейся на единицу статической переменной. Это значение использовано для защиты от одновременной обработки одного экземпляра сценария двумя исполнителями.

5.2.3 Задача и очередь задач

События при постановке в очередь задач помещаются в структуру-обёртку, поддерживающую счетчик ссылок на данное событие. Как только последняя задача была выполнена, событие удаляется при помощи callback-функции, передаваемой вместе с событием.

Задача, которой в модели соответствовала четверка <Событие, Сценарий, Итератор обхода, Финал обхода> в реализованной архитектуре описывается следующей структурой:

- IDSPPQueueGroup* group – указатель на группу задач, связанных с тем же событием и имеющих тот же приоритет
- IDSPPQueueElem* prev – указатель на предыдущую задачу из группы
- IDSPPQueueElem* next – указатель на следующую задачу из группы
- u_int priority – приоритет задачи (описаны ниже)
- u_int type – тип задачи (поддерево или вершина)
- EventWrapper* event – обёртка события, осуществляющая подсчет ссылок

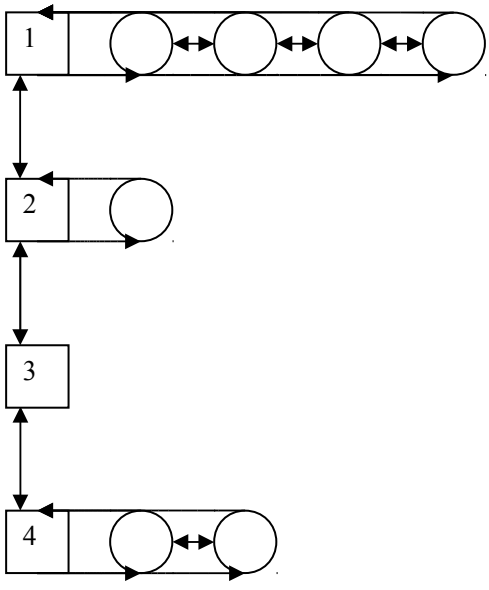

Рисунок 6. Схема организации очереди задач. Квадратами обозначены группы задач (числа – их

- IDSContext* curContext – итератор обхода дерева экземпляров
- IDSContext* finContext – финал обхода дерева экземпляров (через него доступна непосредственно структура сценария)
- bool bToUnload – флаг необходимости приостановить обработку
- bool bShouldStopProcessing – флаг прекращения обработки

Оперируя задачами, требовалось выбрать алгоритм планирования, позволяющий достигнуть справедливости (гарантии выполнения каждой задачи рано или поздно), эффективности (отсутствия, если возможно, излишнего простоя вычислителя), а в случае отдельных сценариев – сокращения времени ожидание (промежуток времени от постановки в очередь до окончания обработки).

Реализованный алгоритм планирования представляет собой гибридную версию алгоритмов вытесняющего приоритетного планирования с абсолютными приоритетами (приоритеты подразделяются на приоритет реального времени, вытесняющего с обработки остальные задачи, и все остальные приоритеты) и планирования с выбором очередного приоритета по принципу round-robin. При этом задачи с одной временной меткой (связанные с одним событием) обрабатываются по принципу First Come First Served. Кроме того, если задача блокируется вследствие конфликта с другой задачей (если экземпляр сценария уже занят другой обработкой), происходит выгрузка задачи в конец списка, соответствующего её временной метке.

Задачи, имеющие одинаковый приоритет и связанные с общим событием, связаны с группой задач, структурой, включающей в себя:

- int refCounter – счетчик ссылок на данную группу
- IDSPPTimeStamp timestamp – временная метка данной группы
- IDSPPQueueElem* first – указатель на первую задачу из группы
- IDSPPQueueElem* last – указатель на последнюю задачу из группы
- IDSPPQueueGroup* prev – указатель на предыдущую группу
- IDSPPQueueGroup* next – указатель на следующую группу

Временная метка – объект, к которому применимо отношение полного порядка. Данными метками помечаются группы при добавлении в очередь, что обеспечивает упорядоченность групп по меткам. Метки реализованы в виде 64-битных беззнаковых целых (unsigned long long), оператор сравнения сравнивает не абсолютные значения, а соотношения разностей:

```
bool operator < (IDSPPTimeStamp& a)
{
```

```
        return (a.i-i)<(i-a.i);
}
```

Приоритеты реализованы в виде нескольких отдельных очередей, для каждой из которых хранятся указатели на первую и последнюю группу. Одна из очередей реализует приоритет реального времени – при попадании задачи в нее выполняемые задачи могут выгружаться, чтобы уступить вычислитель. При инициализации очередей, каждая из них заполняется фиксированным количеством пустых групп. Во время работы с очередью отслеживается, чтобы число групп не опускалось ниже данного порога. Это нужно для удобной организации блокировок, которых в очередях две:

- блокировка на добавление – блокирует запись в нижний конец очереди, во время добавления пустых групп или постановки новых групп задач
- блокировка на взятие – блокирует верхушку очереди, во время взятия оттуда головного элемента или возврата выгружаемого

Добавление в очередь выполняется поставщиком событий и осуществляется следующим образом:

1. Создаётся обёртка события, включающая в себя счетчик ссылок и callback-функцию с параметром, вызываемую перед удалением. Счетчик ссылок увеличивается на 1.
2. У хранилища сценариев запрашивается список корневых экземпляров сценариев, обрабатывающих данное событие.
3. Если корневой экземпляр сценария помечен к удалению, сценарий игнорируется.
4. Счетчик ссылок на корневой экземпляр сценария увеличивается на 1.
5. Для каждого сценария формируется задача, в которой прописывается событие, итератор обхода устанавливается в NULL, а финал обхода в корневой экземпляр сценария. В качестве типа задачи указывается IDS_SUBTREE.
6. Задачи формируются в группы, в соответствие с приоритетами сценариев.
7. Устанавливается блокировка на добавление.
8. Генерируются временные метки и приписываются группам, после чего группы добавляются в соответствующие их приоритетам очереди.
9. Снимается блокировка на добавление.
10. Счетчик ссылок на событие уменьшается на 1. Если он достиг нуля (а такое возможно, если не было найдено ни одного сценария или все сценарии были помечены к удалению), происходит уничтожение события. В остальных случаях событие будет уничтожено уже после обработки.

11. Если имело место произведено добавление задачи в сверхприоритетную очередь, то необходимо выставить флаг bToUnload для одной из находящихся в таблице исполнителей задач.

Текущая организация очереди позволяет расширить язык, введя в нем возможность обработки события не всем деревом экземпляров, а отдельным поддеревом или отдельным экземпляром сценария. Для этого, в очередь необходимо помещать задачи следующего вида (изменения в пункте 5):

- Для поддерева – указать корень поддерева в качестве финала обхода, NULL в качестве исходного значения итератора. Указать IDS_SUBTREE в качестве типа задачи.
- Для отдельной вершины – указать вершину в качестве финала обхода, NULL в качестве исходного значения итератора. Указать IDS_POINT в качестве типа задачи.

Взятие из очереди (запись задачи в таблицу исполнителей с удалением из очереди) выполняется нитью-обработчиком и осуществляется следующим образом:

1. Если все очереди пусты, взятие завершается неудачей.
2. Устанавливается блокировка на взятие
3. Происходит ротация приоритетов: для каждого приоритета разрешается брать подряд не более фиксированного для данного приоритета числа задач. Данный лимит возрастает со значимостью приоритета: для низкоприоритетной очереди это 1, далее идет 2 и 4. Пустые очереди при ротации игнорируются. Если сверхприоритетная очередь не пуста, то выбирается именно она.
4. Рассматривается головная группа очереди с выбранным приоритетом. Если в группе более одного элемента, то первый элемент головной группы забирается на обработку, отделяясь от группы (первым элементом головной группы становится второй элемент головной группы, но связь задача-группа остается), осуществляется переход к пункту 7.
5. Если число групп в очереди равно минимальному возможному числу групп, то в конец очереди дописывается пустая группа (с использованием блокировки на добавление).
6. Поскольку в головной группе ровно один элемент, группа отделяется от очереди, а её элемент забирается на обработку. При этом задача отделяется от группы, остается только связь задача-группа.
7. Снимается блокировка на взятие и взятие заканчивается успехом.

Возврат в очередь (удаление задачи из таблицы исполнителей с возвратом в очередь) выполняется нитью-обработчиком и осуществляется следующим образом:

1. Устанавливается блокировка на взятие (именно она, поскольку производится операция с головой очереди)
2. Если группа, к которой относится выгружаемое событие, находится в очереди, то событие помещается в конец списка задач, которые включает в себя данная группа. Осуществляется переход к пункту 5.
3. Иначе, производится просмотр очереди сверху вниз на предмет места, куда должна быть вставлена данная группа, чтобы сохранить упорядоченность групп по временным меткам.
4. Группа вставляется в найденную позицию, к ней прикрепляется выгружаемая задача.
5. Блокировка снимается, событие удаляется из таблицы исполнителей.

Для индикации пустоты очередей и оценки числа групп в них, реализованы целочисленные счетчики, модифицируемые при помощи атомарных операций. Данные счетчики показывают число, не превосходящее реального числа задач и групп в очереди, и равное ему, когда не выполняется ни одна из операций работы с очередями. Данные счетчики увеличиваются всегда после добавления, а уменьшаются всегда перед удалением, что придает им описанную выше семантику.

Сценариев с максимальным приоритетом крайне мало и они подразумевают быструю обработку редких событий, в то время как остальных сценариев множество и они оперируют постоянным потоком событий, таким как сетевой трафик. Учитывая это, реализованные алгоритмы планирования достигают поставленных целей: справедливости и эффективности планирования, а также сокращения времени ожидания в случае экстренной обработки.

5.2.4 Таблица исполнителей

Таблица исполнителей реализована в виде массива указателей на задачи. Помещение флагов обработки в структуру события помогает избежать копирований и выделения памяти на этапе взятия из очереди.

Если исполнитель свободен, то в таблице хранится NULL, иначе – указатель на обрабатываемую в данный момент задачу. Модификация таблицы носит упреждающий характер – исполнитель сначала фиксирует в таблице факт обработки некоторого экземпляра, продвигая итератор, и лишь затем приступает непосредственно к обработке.

5.2.5 Нить-исполнитель

На Рисунке 7 изображена блок-схема работы нити-исполнителя. Все нити начинают из свободного состояния, не имея никаких задач. Затем, нить обращается к очереди, пытаясь взять из нее задачу на обработку. Если очередь пуста, данная операция заканчивается неуспехом. В таком случае, нить ждет небольшой квант времени и повторяет обращение к очереди.

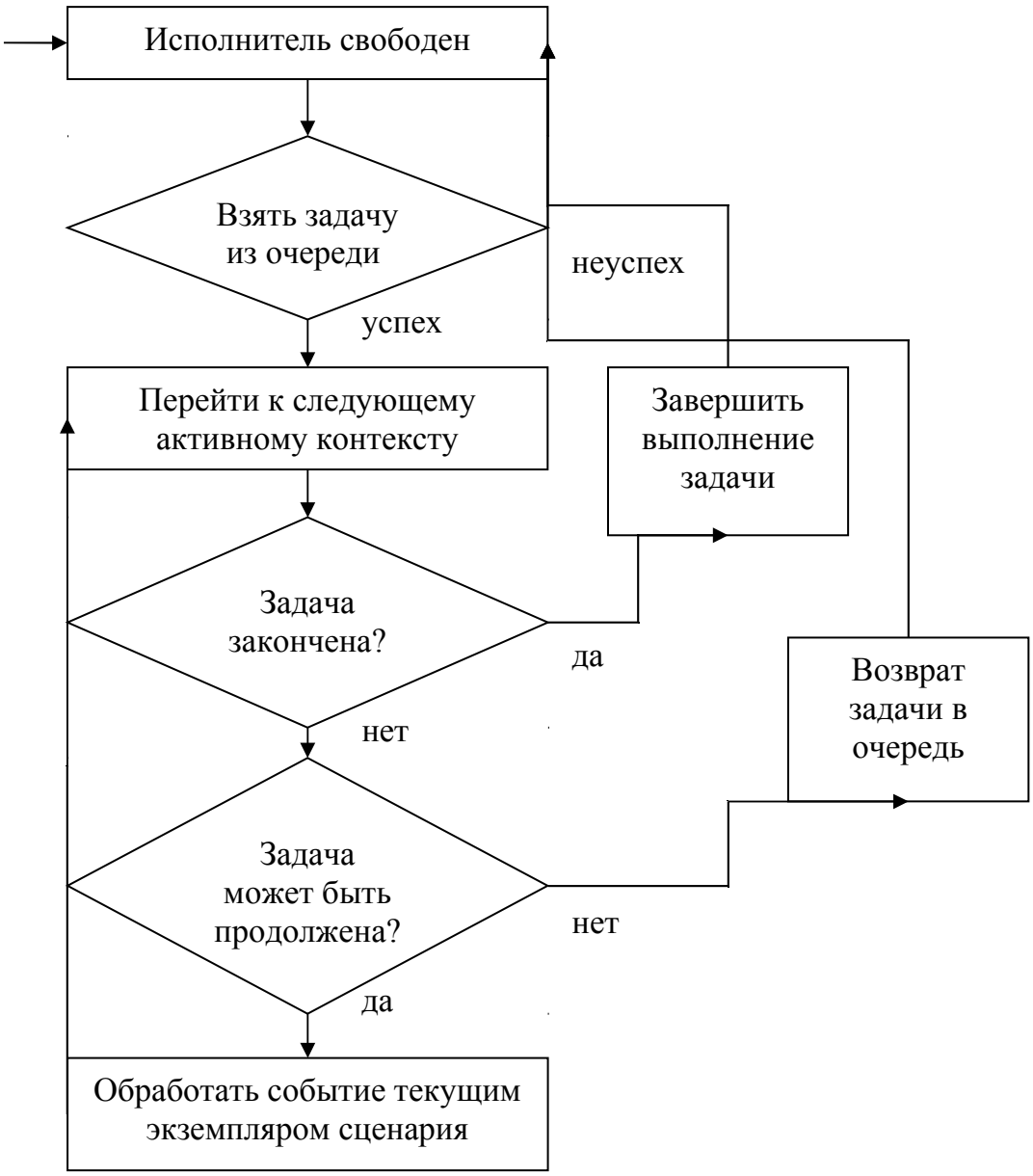

Рисунок 5. Блок-схема работы нити-исполнителя

1. Вычисляется следующее значение итератора next.
2. Если next равен NULL, что возможно в результате вычисления родительского экземпляра от корневого, то прекратить обход.
3. Если экземпляр сценария, на которого указывает next, помечен к удалению, проверить счетчик ссылок на данный экземпляр и если он равен нулю, удалить, после чего перейти к шагу 1, иначе завершить обход, вернув next.

Проверка «задача закончена» включает в себя несколько альтернатив:

- Выставлен флаг bShouldStopProcessing
- Следующее значение итератора next равно NULL
- Записанный в таблицу исполнителей итератор равен финалу обхода

В случае если задача полностью выполнена, уменьшаются счетчики ссылок на обрабатываемое событие и на группу задач, если счетчик достигает нуля, объект удаляется. Счетчик ссылок на итератор и финал обхода также уменьшается, и если они помечены к удалению, то удаляются в случае обнуления счетчика ссылок.

Проверка «задача может быть продолжена» включает в себя несколько альтернатив:

- Выставлен флаг bUnload
- Другая нить использует экземпляр сценария, соответствующий следующему значению итератора next

Если одна из этих альтернатив выполняется, задача возвращается в очередь, а значение итератора остается прежним. Иначе, происходит обработка события текущим экземпляром сценария:

1. Итератору присваивается значение next. Счетчик ссылок на прошлое значение итератора уменьшается на 1, а счетчик ссылок на next увеличивается на 1.
2. Для данного экземпляра сценария берется очередной переход из списка переходов из текущего состояния. Если переходы кончились, выполнение завершается.
3. Если переход связан с событиями другого типа, он пропускается.
4. Иначе выполняется код, связанный с условием перехода. Если условие вернуло True, выполняется код, связанный с телом перехода. Иначе – переход к шагу 2.

5.2.6 Пример обработки последовательности событий

Рисунок 8 суммирует вышесказанное и иллюстрирует работу системы. Предположим, в системе работают 3 нити системы прогона и загружено 2 сценария – сценарий S_1 с большим числом состояний и высокоприоритетный сценарий S_2. Пусть последовательно приходят события A, B, C, D, обрабатываемые сценарием S_1, а позже приходит событие E, принимаемое сценарием S_2, т.е. требующее срочной обработки.

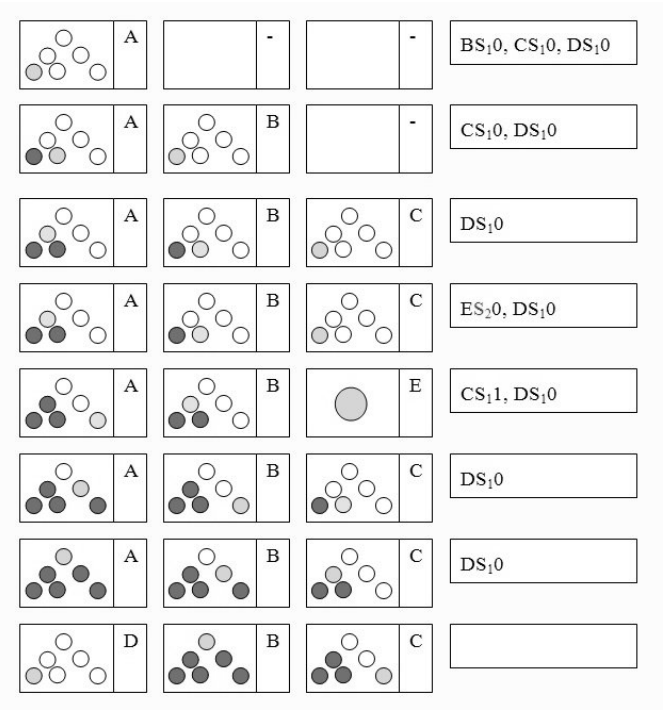

Рисунок 6. Работа многопоточной системы прогона сценариев. Каждой строке соответствует временной срез. Справа в прямоугольнике расположен список необработанных задач, слева – таблица из трёх исполнителей

Первая нить занимается обработкой события A в самом первом экземпляре. Остальные нити ждут, т.к. первый экземпляр сценария S_1 занят. Когда первая нить переходит к следующему экземпляру, вторая нить занимается обработкой события B в рамках первого экземпляра. Точно также, когда появляется возможность, нить захватывает первый экземпляр для обработки события C.

Пусть теперь пришло событие E, предназначенное для сценария S_2. Тогда, третья нить, как обрабатывающая самое позднее событие, обязана прекратить обработку события C ради срочной обработки события E. В очередь помещается элемент расписания, предполагающий продолжение обработки события C со второго экземпляра.

После обработки события E третья нить снова свободна и берет из очереди отложенную ранее задачу, после чего работа продолжается в нормальном режиме.

5.3 Стандартная библиотека языка

5.3.1 Изменения в стандартной библиотеке языка

С внедрением данной схемы различные сценарии языка AURA и даже различные экземпляры одного сценария получают возможность работать параллельно. В то же время, некоторые сценарии оперируют static-переменными таким образом, что при параллельной работе возможны ошибки. Поэтому необходимо предоставить программисту сценариев возможность создавать защищаемые секции кода. В частности, предлагается добавить мьютексы в стандартную библиотеку языка.

Кроме того, модельные сценарии имеют приоритеты, в то время как сам язык их никак не поддерживает. В качестве решения данной проблемы в стандартную библиотеку языка вводится пара функций GetPriority и SetPriority, отвечающих за установку приоритета сценария и проверку его текущего значения. Архитектура ядра анализа не нацелена на динамическую смену приоритетов сценариев, они учитываются лишь при постановке задачи в одну из очередей, поэтому в момент переключения приоритета возможно нарушение порядка получения событий. В связи с этим, в руководстве программиста необходимо указать рекомендацию использования данных функций исключительно в конструкторе сценария.

5.3.2 Реализация функций стандартной библиотеки

Некоторые функции стандартной библиотеки языка являются прямыми обращениями к ядру анализа, поэтому со сменой ядра анализа необходима и повторная реализация данных функций.

Далее представлен список данных функций:

- StopEventProcessing() – прекращает обход дерева экземпляров для текущего сценария
- Unwind() – уничтожает данный экземпляр и все его потомки для данного сценария
- GetCurrentState() – возвращает порядковый номер текущего состояния данного экземпляра сценария

Единая особенность всех этих функций – необходимость доступа к текущей задаче или к текущему экземпляру сценария. В то же время, данные функции вызываются из языка AURA, и вызвавший экземпляр сценария не имеет информации о номере нити и соответствующей ей записи в таблицы исполнителей.

В связи с этим, было добавлено отображение дескриптора потока на номер его записи в таблице исполнителей. Для работы с данным отображением были реализованы примитивы SetThreadID() и GetThreadID(). Дескрипторы, не относящиеся к нитям-исполнителям, отображение переводит в константу IDSPP_INIT_THREAD. Дело в том, что часть кода на языке AURA выполняется еще до инициализации нитей: во время загрузки модуля сценариев выполняются их конструкторы. Для доступа к экземплярам сценария, конструкторы которых выполняет загрузчик, загрузчиком выставляется переменная IDSContext::curContext. Это решает проблему доступа к экземпляру сценария в данном особом случае, хотя и предполагает, что конструкторы работают в одном потоке.

С использованием данного отображения, реализация этих функций стандартной библиотеки была произведена следующим образом:

- StopEventProcessing() – если GetThreadID() возвращает IDSPP_INIT_THREAD, не делать ничего, иначе в таблице соответствующего исполнителя выставить флаг bShouldStopProcessing.
- Unwind() – если GetThreadID() возвращает IDSPP_INIT_THREAD, вызвать метод Unwind() для IDSContext::curContext, иначе вызвать его для threadData[GetThreadID()]->curContext (итератор текущей задачи в таблице исполнителей для данной нити).
- GetCurrentState() – если GetThreadID() возвращает IDSPP_INIT_THREAD, вызвать метод GetCurrentState() для IDSContext::curContext, иначе вызвать его для threadData[GetThreadID()]->curContext.

5.3.3 Отступление от оригинальной семантики языка

В «старой» системе прогона среды AURA использовалась специфичная реализации функции BroadcastEvent(). Даная функция выполняет вещательную рассылку события. При этом событие обработают все сценарии, кроме сценария-источника. В старой системе прогона при данном вызове производилось inline-обработка события всеми существующими сценариями, кроме текущего – в момент выхода из функции обработка сценария-источника обработка события всеми сценариями-приёмниками была уже завершена. Данная семантика вещательной рассылки, по сути, является «хаком» и не годится для параллельной системы прогона.

Следует отметить, что существующие сценарии, использующие BroadcastEvent(), не используют побочные эффекты её реализации в старой системе прогона. В частности, сценарии вещают события, на которые сами не подписаны, и никак не учитывают

скорость обработки. Это позволяет заменить inline-обработку простой постановкой в очередь с последующей обработкой всеми сценариями, возможно даже тем, который вещал данное событие.

Тут возникают определенные трудности, необходимо копирование события, созданного в процессе выполнения кода на языке AURA. Данный язык не поддерживает виртуальных методов и динамической памяти, а самое главное, не все поля события являются pod (т.е. скопированы при помощи простого копирования памяти). При этом функция BroadcastEvent языка C++ получает указатель на общего предка всех событий, объект типа Event.

Объект Event содержит единственное поле:

- u_int_32 eventType

На текущий момент в среде AURA присутствует около 70 типов событий, пользовательские события добавляются крайне редко и, как правило, сопряжены с одновременными модификациями в самой среде: добавление новых поставщиков событий, средств реагирования и т.п., что подразумевает перекомпиляцию проекта.

На основании данного факта было принято решение об отказе от вещания пользовательских событий, не входящих в библиотеки языка и подключаемых модулей. С учетом данного допущения, проблема копирования была решена.

В исходные коды ядра анализа была добавлено отображение типа события на функцию копирования событий данного типа: map<u_int_32,Event* (*)(Event*)>.

Во время вызова BroadcastEvent() происходит копирование события при помощи функции, полученной данным отображением из типа копируемого события:

```
Event* IDSParaPlanner::CloneEvent(Event* ev)
{
        return StaticClonningMap.map[ev->eventType](ev);
}
```

Поскольку различных типов событий несколько десятков и их число увеличивается по мере написания новых модулей среды, был реализован сценарий на языке sh, автоматически заполняющий данное отображения. Сценарий принимает на вход список заголовочных файлов, содержащих объявления структур событий, а на вывод печатает код на языке C++, соответствующий объявлению функций копирования для структур событий и прописыванию данных функций в отображение StaticClonningMap.

Реализация сценария подразумевает некоторое соглашение в оформлении структур событий, в частности первая строка в объявлении каждого нового события должна иметь строго следующий вид:

struct <Имя события> : <Имя предка события> IDS_EVENTID(<ID события>)

- <Имя события> – идентификатор, имя типа определяемого события
- <Имя предка события> – идентификатор, имя типа события-предка
- <ID события> – уникальная числовая константа, либо макрос в нее разворачивающийся

Автоматически сгенерированная функция копирования создается применением к структуре события следующего макроса:

```
#define IMPLEMENT_EVENT(EventType,id) \
static Event * Clone_##EventType (Event* src) \
{ \
   EventType *newEvent = new EventType; \
   *newEvent = *((EventType*)src); \
   return newEvent; \
}
```

5.4 Выводы

Разработанная RTS языка AURA использует гибридный алгоритм планирования выполнения сценариев на основе алгоритмов вытесняющего приоритетного планирования с абсолютными приоритетами и планирования с выбором очередного приоритета по принципу round-robin. Задачи с одной временной меткой и одним приоритетом обрабатываются по принципу First Come First Served. Кроме того, если задача блокируется вследствие конфликта с другой задачей, происходит выгрузка задачи в конец списка, соответствующего её временной метке.

Самым серьезным недостатком данной реализации является невозможность производить вещательную рассылку пользовательских событий (ограничение на IPC). Незначительным недостатком RTS является необходимость загружать сценарии в одном потоке, что влияет только на время запуска системы. Поскольку это время оценивается секундами или их долями, решение данной проблемы не представляет важной задачи.

Несмотря на эти проблемы, данная реализация позволяет обрабатывать сценарии на языке AURA параллельно, причем:

- достигнут параллелизм по коду – одно событие может обрабатываться множеством сценариев в разных нитях-исполнителях одновременно;

- достигнут параллелизм по данным – различные экземпляры одного и того же сценария в различных нитях-исполнителях одновременно способны обрабатывать последовательно события;
- и при этом полностью сохранена семантика последовательности обработки событий экземплярами сценария.

6 Экспериментальная оценка эффективности

6.1 Синтетический тест

Цель данного теста – оценка эффективности распараллеливания нескольких тяжеловесных обработчиков событий, не способных выполняться в параллель.

Тест проведен на сервере под управление ОС GNU Linux, оснащенном 2Gb оперативной памяти, двумя процессорами Intel(R) Xeon(TM) с частотой 2.40GHz и, благодаря технологии Hyper Threading, имеющем 4 виртуальных ядра.

Для данного теста был использован искусственный модуль из простых сценариев, по получении события выполняющих consuming-переход. В модуле содержится 4 идентичных сценария. Каждый сценарий, получив событие, безусловно выполняет цикл for, имитируя тяжеловесную обработку события. Далее приведен фрагмент, отвечающий телу данных сценариев на языке AURA:

```
initial state s0 {}
consuming transition s0->s0 event aEvent(true)
{
   int i; for(i=0;i<1000000;i++);
}
```

Количество сценариев выбрано таким, чтобы превосходить число нитей-анализаторов, поскольку каждый сценарий может обрабатываться лишь одной нитью.

Объект анализа в данном тесте – поток из фиксированного числа идентичных событий одного типа, генерируемый на максимально возможной скорости. Для повышения пропускной способности, в качестве события была выбрана пустая структура. Всего в рамках теста генерировалось 500 таких событий.

При тестировании был использован полностью автоматический тест ядра анализа, позволяющий оценить максимальную пропускную способность (число событий, обрабатываемых в единицу времени) ядра анализа при различных наборах загруженных сценариях.

Измерение времени работы производилось путем взвода таймера перед началом генерации событий и добавления функции остановки таймера в качество callback-функции, вызываемой при обработке последнего события. Таким образом, измеряется время между генерацией первого и обработкой последнего из 500 событий.

Тест был проведен для четырех версия ядра анализа системы AURA: для исходного непараллельного ядра, а так же для параллельного ядра с одной, двумя и тремя нитями-

обработчиками. Ограниченность тремя потоками связана с необходимостью оставить процессорное время и для генерирующего события потока. Для каждой версии ядра тест проводился несколько раз и результаты усреднялись.

Рисунок 9 демонстрирует результаты данного теста. В результате теста показан выигрыш в ~N раз при N>1 потоках по сравнению с исходной однопоточной RTS в случае тяжеловесной обработки, не допускающей одновременную работу экземпляра сценария над различными событиями.

Кроме того, однопоточная версия параллельного ядра работает чуть медленнее непараллельного, что объясняется дополнительными издержками по поддержанию очереди задач.

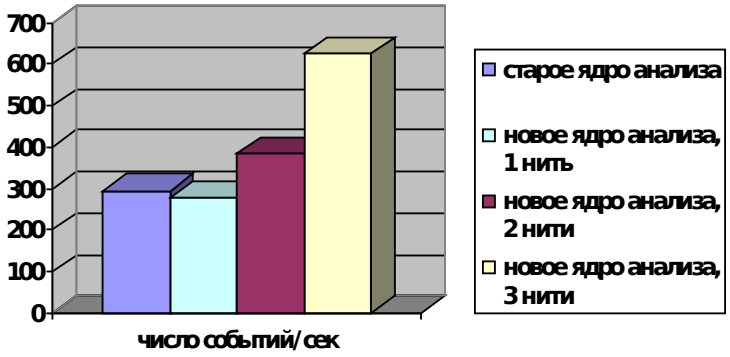

Рисунок 7. Скорость обработки событий для различных версий ядра анализа

6.2 Тестирование модуля анализа сетевого трафика

Цель данного теста – оценка эффективности распараллеливания большого объема сценариев, а также демонстрация идентичности деятельности различных версий ядра анализа (отсутствия пропущенных или дублированных задач в очереди).

Тест также проведен на сервере под управление ОС GNU Linux, оснащенном 2Gb оперативной памяти, двумя процессорами Intel(R) Xeon(TM) с частотой 2.40GHz и, благодаря технологии Hyper Threading, имеющем 4 ядра, а также сетевой интерфейс с бездействующим каналом, в который производилась запись фиктивных пакетов.

В тесте использовался модуль сигнатурного анализа сетевого трафика, полученный путем конвертации 7176 правил СОА Snort для протокола TCP, состоящий из 485 сценариев.

В качестве анализируемого потока событий использовался фиктивный сетевой трафик, полученный воспроизведением 18 записей реальных атак на сервис FTP при помощи tcpreplay. Записи содержат в себе реальные атаки и суммарно состоят из 1127

сетевых пакетов общим объемом 578кб. За время теста происходит 18 перезапусков утилиты tcpreplay, в связи с чем общий процесс воспроизведения трафика занимает около 2 секунд.

Тест был проведен для четырех версия ядра анализа системы AURA: для исходного непараллельного ядра, а так же для параллельного ядра с одной, двумя и тремя нитями-обработчиками. Ограниченность тремя потоками связана с необходимостью оставить процессорное время и для потока, производящего захват сетевого трафика и формирование событий. Для каждой версии ядра тест проводился несколько раз и результаты усреднялись.

В результате теста показана идентичность наборов сообщений о сетевых атаках, а также отмечено существенное увеличение скорости анализа при добавлении каждого дополнительного потока анализа. Данное ускорение отображено на рисунке 10.

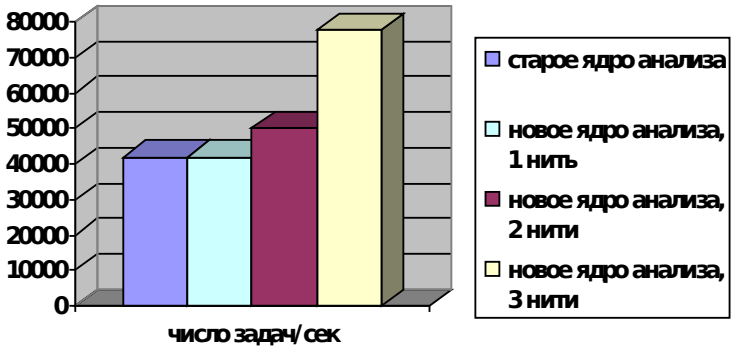

Рисунок 8. Усредненные результаты серии экспериментов в рамках тестирования на сетевом трафике.

В ходе теста в очередь помещается порядка 546 тысяч задач (число пакетов в записях, умноженное на число сценариев языка AURA, поскольку в данном тесте каждый пакет обрабатывается всеми сценариями). Таким образом, при сокращении средней длительности теста с 13 до 7 секунд, средняя скорость анализа была повышена с 42 тыс. задач/сек до 78 тыс. задач/сек.

Результаты экспериментов показывают, что параллельная RTS языка AURA позволяет значительно увеличить скорость обработки сетевого трафика практически без увеличения накладных расходов. При тестировании использовались реальные данные, что позволяет экстраполировать результаты эксперимента на условия функционирования системы обнаружения атак, построенную на основе среды AURA.

Заключение

1. Разработана архитектура RTS языка AURA, использующая очередь с гибридной системой приоритетов. Используемые алгоритмы планирования и архитектура RTS разработаны с учетом требований совместимости: строгий FIFO-порядок обработки событий экземпляром сценария, отношение частичного порядка на множестве экземпляров — дочерние экземпляры обрабатываются раньше родительских.
2. Разработанная архитектура реализована в виде RTS языка AURA для операционных систем Linux, Windows 2000/XP.
3. Разработано и реализовано несколько автоматизированных тестов производительности: тест оценки максимальной пропускной способности ядра анализа при различных наборах загруженных сценариев, тест качества обнаружения атак и времени анализа при использовании модуля сигнатурного анализа. Результаты оценки эффективности на данных тестах показывают устойчивый рост скорости обработки событий для типовых вариантов использования при увеличении числа используемых процессорных ядер.

Эксперименты продемонстрировали значительный рост производительности ядра анализа на реальных задачах без существенного увеличения накладных расходов на синхронизацию, что позволяет сделать вывод о возможности замены «старого» однопоточного ядра анализа среды AURA на параллельную RTS, разработанную и реализованную в данной работе.

Представляется целесообразным продолжить исследования в области обеспечения режима мягкого реального времени в среде AURA за счёт оценки наихудшего времени выполнения на уровне языка, а также профилировки существующих анализаторов данной среды на типовых данных.

Литература

1. Кеммерер Р., Виджна Д. Обнаружение вторжений: краткая история и обзор [HTML] (http://www.osp.ru/text/302/181714.html)
2. Пировских А. Snort: инструмент выявления сетевых атак [HTML] (http://www.thg.ru/network/20051020/index.html)
3. Система обнаружения атак RealSecure [HTML] (http://citforum.ru/security/internet/real_scan.shtml)
4. Prelude User Manual [HTML] (https://trac.prelude-ids.org/wiki/ManualUser)
5. Pervasive Network Visibility, Deep Application Insight, Security and Profitable Managed Services [PDF] (http://www.arbornetworks.com/index.php?option=com_docman&task=doc_download&gid=6)
6. Defeating DDOS Attacks [PDF] (http://www.cisco.com/en/US/prod/collateral/vpndevc/ps5879/ps6264/ps5888/prod_white_paper0900aecd8011e927.pdf)
7. Deri L. Improving Passive Packet Capture: Beyond Device Polling [PDF] (http://luca.ntop.org/Ring.pdf)
8. Rizzo L. Device Polling Support for FreeBSD [HTML] (http://info.iet.unipi.it/~luigi/polling/)
9. Deri L. nCap: Wire-speed Packet Capture and Transmission [PDF] (http://luca.ntop.org/nCap.pdf)
10. Mikolasek V. Intrusion Detection Systems - State of the Art Report [PDF] (http://www.vmars.tuwien.ac.at/php/pserver/extern/download.php?fileid=1524)
11. Wang H.J., Guo C., Simon D.R., Zugenmaier A. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. [PDF] (http://research.microsoft.com/~helenw/papers/shieldSigcomm04.pdf)
12. Schear N., Albrecht D.R., Borisov N. High-Speed Matching of Vulnerability Signatures [PDF] (http://www.cs.uiuc.edu/homes/nschear2/raid-2008.pdf)
13. Vossen J.P. Snort Intrusion Detection and Prevention Guide [HTML] (http://searchsecurity.techtarget.com/generic/0,295582,sid14_gci1083823,00.html)

14. Jacob N., Brodley C. Offloading IDS computation to the GPU [PDF] (http://www.acsac.org/2006/papers/74.pdf)

15. Vasiliadis G., Antonatos S., Polychronakis M., Markatos E.P., Ioannidis S. Gnort: High Performance Network Intrusion Detection Using Graphics Processors [PDF] (http://www.ics.forth.gr/dcs/Activities/papers/gnort.raid08.pdf)

16. Kazachkin D., Gamayunov D., Network traffic analysis optimization at signature-based intrusion detection systems. // In Proceedings of SYRCoSE 2008, Vol 1, p. 27-32

17. REDSecure: обнаружение и предотвращение атак [HTML] (http://redsecure.ru/)

18. Гамаюнов Д. Ю., Обнаружение компьютерных атак на основе анализа поведения сетевых объектов. // Москва, 2007. Кандидатская диссертация.

19. Eckmann S.T., Giovanni V. G., Kemmerer R.A. STATL: An Attack Language for State-based Intrusion Detection [PDF] (www.cs.ucsb.edu/~vigna/publications/2000_eckmann_vigna_kemmerer_statl.pdf)

20. Lattner C., Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation [PDF] (http://llvm.org/pubs/2003-09-30-LifelongOptimizationTR.pdf)

21. Гамаюнов Д.Ю., Казачкин Д.С., AURA: программная платформа высокоскоростного анализа сетевого трафика для задач информационной безопасности. // Материалы конференции РусКрипто'09, 2009 г. [PDF] (http://ruscrypto.ru/netcat_files/File/ruscrypto.2009.008.zip)

22. Blaise B., POSIX Threads Programming [HTML] (https://computing.llnl.gov/tutorials/pthreads/)

23. Presentation on Multi-Threading on UNIX [WWW] (http://www.geocities.com/dipak123/software/MultiThreading.swf)

24. Threading Building Blocks [HTML] (http://www.threadingbuildingblocks.org/)

25. Колдовский В., Язык Erlang и программирование для мультиядерных процессоров [HTML] (http://itc.ua/node/26721/)

26. Open Source Erlang [HTML] (http://www.erlang.org/)

27. NESL: A Parallel Programming Language [HTML] (http://www.cs.cmu.edu/~scandal/nesl.html)

28. Giovanni V. G., Kemmerer R.A., NetSTAT: A Network-based Intrusion Detection Approach [PDF] (http://www.acsac.org/1998/presentations/wed-a-1030-vigna.pdf)
29. Eckmann S.T., Translating Snort rules to STATL scenarios [PDF] (http://www.raid-symposium.org/raid2001/papers/eckmann_raid2001.pdf)
30. Fekete J.D., Richard M., Dragicevic P., Specification and Verification of Interactors: A Tour of Esterel [PDF] (http://www-ihm.lri.fr/~fekete/ps/esterel.pdf)
31. Berry G., Gonthier G., Incremental development of an HDLC protocol in Esterel [PDF] (ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-1031.pdf)
32. Schear N., David R. Albrecht D.R., Nikita Borisov N., Presentation of High-Speed Matching of Vulnerability Signatures [PDF] (http://www.ll.mit.edu/RAID2008/Files/RAID2008-s3-3-Schear-HighSpeedMatching.pdf)
33. Schear N., David R. Albrecht D.R., Nikita Borisov N., High-Speed Matching of Vulnerability Signatures [PDF] (http://www.hatswitch.org/~nikita/papers/vespa-raid08.pdf)
34. Paxson V., Bro: A System for Detecting Network Intruders in Real-Time [PS] (http://staff.washington.edu/dittrich/papers/bro-usenix98-revised.ps)
35. Bro Intrusion Detection System [HTML] (http://bro-ids.org)
36. Van Roy P., How to say a lot with few words [PDF] (http://www.info.ucl.ac.be/Enseignement/Cours/INGI1131/2007/Scripts/ircamTalk2006.pdf)
37. Haridi S., Nils Franzén N., Tutorial of Oz [HTML] (http://www.mozart-oz.org/documentation/tutorial/index.html)
38. MPI Documents [HTML] (http://www.mpi-forum.org/docs/)
39. Real-Time Message Passing Specification based on MPI [HTML] (http://www.mpirt.org/)
40. The Occam archive [HTML] (http://archive.comlab.ox.ac.uk/occam.html)
41. Карпов В.Э., Объектно-ориентированное программирование, Часть I. Язык Смолток [HTML] (http://avt.miem.edu.ru/Kafedra/Uiits/Chit_kurs/smalltalk11.html)
42. Smalltalk: Getting The Message. The Essentials of Message-Oriented Programming with Smalltalk [HTML] (http://www.smalltalk-resources.com/Smalltalk-Getting-the-Message.html)
43. The OpenMP API specification for parallel programming [HTML] (http://openmp.org/)

44. Summary of OpenMP 3.0 C/C++ Syntax [PDF] (http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf)
45. Беляев А., Петренко С., Системы обнаружения аномалий: новые идеи в защите информации [HTML] (http://www.citforum.ru/security/articles/anomalis/)
46. Система обнаружения вторжений «ФОРПОСТ 1.1» [HTML] (http://www.rnt.ru/to_content/action_desc/id_46/lang_ru/)
47. The Sourcefire 3D System [HTML] (http://www.sourcefire.com/products/3D)
48. NitroGuard Intrusion Prevention System: Next Generation IPS [HTML] (http://nitrosecurity.com/information/products/nitroguard-intrusion-prevention-system/)