МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Пятигорский институт (филиал) СКФУ
Колледж Пятигорского института (филиал) СКФУ

УПРАВЛЕНИЕ ПРОЕКТАМИ
МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ПРАКТИЧЕСКИХ (СЕМИНАРСКИХ) ЗАНЯТИЙ

Специальности СПО
09.02.07 Информационные системы и программирование
Квалификация специалист по информационным системам

Пятигорск 2022

Методические указания для практических занятий по дисциплине «Управление проектами» составлены в соответствии с ФГОС СПО. Предназначены для студентов, обучающихся по специальности 09.02.07 Информационные системы и программирование

Пояснительная записка

Для многих студентов значительную трудность представляет решение задач. Поэтому в данных методических указаниях главное внимание уделено решению типовых примеров и задач, поясняющих теоретический материал. Однако прежде чем начать решать эти примеры надо добиться полной ясности в понимании соответствующих понятий.

В начале каждой темы кратко излагаются основные теоретические сведения (определения, формулы), необходимые для решения последующих задач. Приводятся решения типовых примеров и задач. Даются упражнения на закрепления темы.

Выполнение студентами практических работ направленно на:

- обобщение, систематизацию, углубление, закрепление полученных теоретических знаний по конкретным темам;
- формирование умений применять полученные знания на практике и реализацию единства интеллектуальной и практической деятельности;
- развитие интеллектуальных умений у будущих специалистов: аналитических, проектировочных, конструктивных и др.;
- выработку при решении поставленных задач таких профессиональных качеств, как самостоятельность, ответственность, точность, творческая инициатива.

В результате изучения учебной дисциплины «Управление проектами» обучающийся должен **уметь**:

- работать с проектной документацией;
- разработанной с использованием графических языков спецификаций;
- выполнять оптимизацию программного кода с использованием специализированных программных средств;
- использовать методы и технологии тестирования и ревьюирования кода и проектной документации;
- применять стандартные метрики по прогнозированию затрат, сроков и качества.

В результате освоения учебной дисциплины обучающийся должен **знать:**

- задачи планирования и контроля развития проекта;
- принципы построения системы деятельностей программного проекта;
- современные стандарты качества программного продукта и процессов его обеспечения.

Практическая работа № 1

Тема 1: Измерительные методы оценки программ: назначение, условия применения.

Цель: Изучить методы оценки программ, их назначения и условия применения.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Система измерения включает метрики и модели измерений, которые используются для количественной оценки качества ПО.

При определении требований к ПО задаются соответствующие им внешние характеристики и их атрибуты (подхарактеристики), определяющие разные стороны управления продуктом в заданной среде. Для набора характеристик качества ПО, приведенных в требованиях, определяются соответствующие метрики, модели их оценки и диапазон значений мер для измерения отдельных атрибутов качества.

Согласно стандарту метрики определяются по модели измерения атрибутов ПО на всех этапах ЖЦ (промежуточная, внутренняя метрика) и особенно на этапе тестирования или функционирования (внешние метрики) продукта.

Остановимся на классификации метрик ПО, правилах для проведения метрического анализа и процесса их измерения.

Типы метрик.

Существует три типа метрик:

- метрики программного продукта, которые используются при измерении его характеристик - свойств;
- метрики процесса, которые используются при измерении свойства процесса ЖЦ создания продукта.
- метрики использования.

Метрики программного продукта включают:

- внешние метрики, обозначающие свойства продукта, видимые пользователю;
- внутренние метрики, обозначающие свойства, видимые только команде разработчиков.

Внешние метрики продукта - это метрики:

- надежности продукта, которые служат для определения числа дефектов;
- функциональности, с помощью которых устанавливаются наличие и правильность реализации функций в продукте;

- сопровождения, с помощью которых измеряются ресурсы продукта (скорость, память, среда);применимости продукта, которые способствуют определению степени доступности для изучения и использования;
- стоимости, которыми определяется стоимость созданного продукта.

Внутренние метрики продукта включают:

- метрики размера, необходимые для измерения продукта с помощью его внутренних характеристик;
- метрики сложности, необходимые для определения сложности продукта;
- метрики стиля, которые служат для определения подходов и технологий создания отдельных компонентов продукта и его документов.

Внутренние метрики позволяют определить производительность продукта и являются релевантными по отношению к внешним метрикам.

Внешние и внутренние метрики задаются на этапе формирования требований к ПО и являются предметом планирования и управления достижением качества конечного программного продукта.

Метрики продукта часто описываются комплексом моделей для установки различных свойств, значений модели качества или прогнозирования. Измерения проводятся, как правило, после калибровки метрик на ранних этапах проекта. Общая мера - степень трассируемости, которая определяется числом трасс, прослеживаемых с помощью моделей сценариев типа UML и оценкой количества:

- требований;
- сценариев и действующих лиц;
- объектов, включенных в сценарий, и локализация требований к каждому сценарию;
- параметров и операций объекта и др.

Стандарт ISO/IEC 9126-2 определяет следующие типы мер:

- мера размера ПО в разных единицах измерения (число функций, строк в программе, размер дисковой памяти и др.);
- мера времени (функционирования системы, выполнения компонента и др.);
- мера усилий (производительность труда, трудоемкость и др.);
- мера учета (количество ошибок, число отказов, ответов системы и др.).

Специальной мерой может служить уровень использования повторных компонентов и измеряется как отношение размера продукта, изготовленного из готовых компонентов, к размеру системы в целом. Данная мера используется также при определении стоимости и качества ПО. Примеры метрик:

- общее число объектов и число повторно используемых;
- общее число операций, повторно используемых и новых операций;
- число классов, наследующих специфические операции;
- число классов, от которых зависит данный класс;
- число пользователей класса или операций и др.

При оценке общего количества некоторых величин часто используются среднестатистические метрики (среднее число операций в классе, наследников класса или операций класса и др.).

Как правило, меры в значительной степени являются субъективными и зависят от знаний экспертов, производящих количественные оценки атрибутов компонентов программного продукта.

Примером широко используемых внешних метрик программ являются метрики Холстеда - это характеристики программ, выявляемые на основе статической структуры программы на конкретном языке программирования: число вхождений наиболее часто встречающихся операндов и операторов; длина описания программы как сумма числа вхождений всех операндов и операторов и др.

На основе этих атрибутов можно вычислить время программирования, уровень программы (структурированность и качество) и языка программирования (абстракции средств языка и ориентация на проблему) и др.

В качестве метрик процесса могут быть время разработки, число ошибок, найденных на этапе тестирования и др. Практически используются следующие метрики процесса:

- общее время разработки и отдельно время для каждой стадии;
- время модификации моделей;
- время выполнения работ на процессе;
- число найденных ошибок при инспектировании;
- стоимость проверки качества;
- стоимость процесса разработки.

Метрики использования служат для измерения степени удовлетворения потребностей пользователя при решении его задач. Они помогают оценить не свойства самой программы, а результаты ее эксплуатации - эксплуатационное качество. Примером может служить - точность и полнота реализации задач пользователя, а такжезатраченные ресурсы (трудозатраты, производительность и др.) на эффективное решение задач пользователя. *Оценка требований* пользователя проводится с помощью внешних метрик.

Практическое занятие № 2

Тема 2: Корректность программ. Эталоны и методы проверки корректности

Цель: Изучить подходы к контролю корректности программного обеспечения.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

При разработке информационных систем очень важно контролировать качество программного обеспечения. Понятие качества программного обеспечения сформулировано стандарте ISO 9126. На сегодняшний день, действует редакция, принятая в 2001 году. По этому стандарту качество можно рассматривать как совокупность различных аспектов. Одним из самых важных аспектов качества, представленных на диаграмме, является функциональность. Функциональность определяет способность ПО решать поставленные задачи. При этом очевидно, что важнейшей характеристикой функциональности является корректность — соответствие создаваемого ПО заданной спецификации.

Если ПО не соответствует своей спецификации, то таким продуктом нельзя пользоваться, даже если он надежен, производителен и удобен в использовании. Контроль корректности ПО может осуществляться двумя способами — вручную и автоматизировано. При ручном контроле корректности огромную роль при получении результата играет человеческий фактор. Чтобы избежать ошибок, возникающих в процессе ручного контроля корректности, используют полную или частичную автоматизацию процесса.

Поэтому важнейшей задачей в IT индустрии является обеспечение автоматизированного контроля корректности ПО. Все подходы к контролю корректности программного обеспечения можно разделить на две большие группы: контроль качества программного обеспечения на этапе создания; контроль качества программного обеспечения на этапе выполнения. Подходы из первой группы позволяют проводить анализ корректности работы программного обеспечения еще до его запуска и использования. Такие подходы делятся на три большие группы: – статический анализ; – тестирование; – верификация. Подходы из второй группы проявляют себя уже после компиляции продукта и старта его использования. Их основной задачей является завершение или приостановка работы программы в случае обнаружения какой-либо критической ошибки. Считается, что гораздо лучше прекратить (или приостановить) выполнение программы, нежели продолжать некорректное выполнение. Также динамический контроль позволяет быстрее находить и устранять ошибки, которые не были выявлены на более ранних этапах. Критическими ошибками в таких случаях могут служить: – Нулевое значение знаменателя при выполнении операции целочисленного деления. – Ошибка при попытке коммуникации с недоступным внешним устройством. – Исчерпание доступной памяти или переполнение стека. – Появление сигнала аварийного отключения электропитания системы. – Нарушение бизнес-требований, заложенных при создании программы.

Например, выдача чека с отрицательной суммой заказа. В рамках представленной работы данный пункт является наиболее интересным и будет исследован более подробно. Такие подходы можно также разделить на три группы, в порядке появления и развития подходов: – коды ошибок; – исключения; – контракты. Далее перечисленные выше подходы рассмотрены более подробно, а также произведено их сравнение. Статический анализ используется для поиска часто встречающихся дефектов по некоторым шаблонам.

Такой анализ хорошо автоматизируется. Однако он способен обнаружить лишь малое число из всех возможных ошибок. Большинство техник статической проверки корректности программ, доказавших свою эффективность, рано или поздно становятся частью средств разработки, компиляторов, а иногда преобразуются в семантические правила языков программирования. Тестирование является динамическим методом проверки программ. Для его проведения необходима работающая система, какой-то ее компонент, либо прототип. Тестирование позволяет обнаруживать в системе ошибки, проявляющиеся во время работы. Однако, для проведения тестирование необходимо подготовить набор тестов, а иногда разработать саму тестовую систему. И, если тестовую систему еще можно повторно использовать при разработке очередного программного продукта, то набор тестов разработчикам приходится писать каждый раз заново, учитывая особенности и требования конкретной системы.

На практике существует множество различных средств, позволяющих частично автоматизировать тестирование. Например, достаточно часто используется вероятностное тестирование, позволяющее быстро проверить корректность работы системы, при подаче ей на вход каких-либо случайных данных. Существенным недостатком тестирования является то, что оно, как правило, помогает выявить значительное число ошибок, но гарантировать, что система будет работать, корректно не может. Верификация является статическим методом проверки программы. Верификация относится к формальным методам и применяется только к тем свойствам системы, которые можно выразить с помощью формальной математической модели. Проблему при данном подходе представляет собой необходимость построения математической модели, а также формальное описание требований, которым эта математическая модель, а, следовательно, и проектируемая система, должны удовлетворять. Достоинством данного метода может служить тот факт, что верификация гарантирует корректность работы системы по всем проверяемым свойствам, если они успешно прошли проверку на модели. Каждый подход имеет свои достоинства и недостатки.

Однако если рассматривать все эти подходы с точки зрения улучшения корректности ПО, наиболее интересным является подход на основе верификации. Только верификация позволяет гарантировать корректность работы программы. Именно поэтому верификация программ представляет наибольший интерес на сегодняшний день. Код ошибки — это номер (или сочетания буквы и номера), который соответствует конкретной проблеме в работе программы. Коды ошибок используются для идентификации неправильной работы аппаратного и программного обеспечения, неверного ввода данных пользователем без обработки, возникающей при этом исключительной ситуации в коде программы. Хотя иногда коды ошибок используются в сочетании с обработкой исключений. Также коды ошибок часто используются вместе с кодами возврата. Во время выполнения программы могут возникать ситуации, когда состояние данных, устройств ввода-вывода или компьютерной системы в целом делает дальнейшие вычисления в соответствии с базовым алгоритмом невозможными или бессмысленными. В этом случае системе генерирует исключение, уведомляя пользователей о возникновении такой ситуации. В отсутствие собственного механизма обработки исключений для прикладных программ наиболее общей реакцией на любую исключительную ситуацию является немедленное прекращение выполнения с выдачей пользователю сообщения о характере исключения. Можно сказать, что в подобных случаях единственным и универсальным обработчиком исключений становится операционная система. Возможно игнорирование

исключительной ситуации и продолжение работы, но такая тактика опасна, так как приводит к ошибочным результатам работы программ или возникновению ошибок впоследствии. Например, проигнорировав ошибку чтения из файла блока данных, программа получит в своё распоряжение не те данные, которые она должна была считать, а какие-то другие. Результаты их использования предугадать невозможно. Обработка исключительных ситуаций самой программой заключается в том, что при возникновении исключительной ситуации управление передаётся некоторому заранее определённому обработчику — блоку кода, процедуре, функции, которые выполняют необходимые действия. Исключения позволяют повысить безопасность и качество ПО, предотвращая выполнение программы в случае возникновения каких-либо ошибок. В смысле контроля качества, исключения гораздо надежнее кодов ошибок, так как неверный код ошибки гораздо легче пропустить или намеренно игнорировать. Основная идея контрактного программирования — это модель взаимодействия элементов программной системы, основывающаяся на идее взаимных обязательств и преимуществ. Контракт некоторого метода или функции может включать в себя: конкретные обязательства, которые любой клиентский модуль должен выполнить перед вызовом метода — предусловия; конкретные свойства, которые должны присутствовать после выполнения метода — постусловия, которые входят в обязательства поставщика; обязательства по выполнению конкретных свойств — инвариантов, которые должны выполняться при получении поставщиком сообщения, а также при выходе из метода.

В объектно-ориентированном программировании контракт метода обычно включает следующую информацию: – возможные типы входных данных и их значение; – типы возвращаемых данных и их значение; – условия возникновения исключений, их типы и значения. При использовании контрактов сам код не обязан проверять их выполнение. Обычно при нарушении контракта выбрасывается исключение, сообщающее о возникшем несоответствии. Таким образом, контракты представляю собой модель контроля логических ошибок, или, другими словами, модель проверки бизнес-правил. Каждый подход имеет свои достоинства и недостатки. Однако если рассматривать все эти подходы с точки зрения улучшения корректности ПО, наиболее интересным является подход на основе контрактов. Только контрактное программирование позволяет описывать и контролировать выполнение различных бизнес-требований не затрачивая при этом значительных усилий.

Практическое занятие № 3

Тема 3: Метрики, направления применения метрик. Метрики сложности. Метрики стилистики

1. Метрики, направления применения метрик.

Цель: Изучить понятие метрики, их применение.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Качество ПО - это совокупность свойств, определяющих полезность изделия (программы) для пользователей в соответствии с функциональным назначением и предъявлёнными требованиями.

Характеристика качества программы - понятие, отражающее отдельные факторы, влияющие на качество программ и поддающиеся измерению.

Критерий качества - численный показатель, характеризующий степень, в которой программе присуще оцениваемое свойство.

Критерии качества включают следующие характеристики : экономичность, документированность, гибкость, модульность, надёжность, обоснованность, тестируемость, ясность, точность, модифицируемость, эффективность, легкость сопровождения и т.д.

Критерий должен :

* численно характеризовать основную целевую функцию программы;

* обеспечивать возможность определения затрат, необходимых для достижения требуемого уровня качества, а также степени влияния на показатель качества различных внешних факторов;

* быть по возможности простым, хорошо измеримым и иметь малую дисперсию.

Для измерения характеристик и критериев качества используют метрики.

Метрика качества программ - система измерений качества программ. Эти измерения могут проводиться на уровне критериев качества программ или на уровне отдельных характеристик качества. В первом случае система измерений позволяет непосредственно сравнивать программы по качеству. При этом сами измерения не могут быть проведены без субъективных оценок свойств программ. Во втором случае измерения характеристик можно выполнить объективно и достоверно, но оценка качества ПО в целом будет связана с субъективной интерпретацией получаемых оценок.

В исследовании метрик ПО различают два основных направления :

* поиск метрик, характеризующих наиболее специфические свойства программ, т.е. метрик оценки самого ПО;

* использование метрик для оценки технических характеристик и факторов разработки программ, т.е. метрик оценки условий разработки программ.

По виду информации, получаемой при оценке качества ПО метрики можно разбить на три группы :

* метрики, оценивающие отклонение от нормы характеристик исходных проектных материалов. Они устанавливают полноту заданных технических характеристик исходного кода.

* метрики, позволяющие прогнозировать качество разрабатываемого ПО. Они заданы на множестве возможных вариантов решений поставленной задачи и их реализации и определяют качество ПО, которое будет достигнуто в итоге.

* метрики, по которым принимается решение о соответствии конечного ПО заданным требованиям. Они позволяют оценить соответствие разработки заданным требованиям.

ОСНОВНЫЕ НАПРАВЛЕНИЯ ПРИМЕНЕНИЯ МЕТРИК.

В настоящее время в мировой практике используется несколько сотен метрик программ. Существующие качественные оценки программ можно сгруппировать по шести направлениям :

* оценки топологической и информационной сложности программ;

* оценки надежности программных систем, позволяющие прогнозировать отказовые ситуации;

* оценки производительности ПО и повышения его эффективности путем выявления ошибок проектирования;

* оценки уровня языковых средств и их применения;

* оценки трудности восприятия и понимания программных текстов, ориентированные на психологические факторы, существенные для сопровождения и модификации программ;

* оценки производительности труда программистов для прогнозирования сроков разработки программ и планирования работ по созданию программных комплексов.

МЕТРИЧЕСКИЕ ШКАЛЫ

В зависимости от характеристик и особенностей применяемых метрик им ставятся в соответствие различные измерительные шкалы.

Номинальной шкале соответствуют метрики, классифицирующие программы на типы по признаку наличия или отсутствия некоторой характеристики без учета градаций.

Порядковой шкале соответствуют метрики, позволяющие ранжировать некоторое характеристики путем сравнения с опорными значениями, т.е. измерение по этой шкале фактически определяет взаимное положение конкретных программ.

Интервальной шкале соответствуют метрики, которые показывают не только относительное положение программ, но и то, как далеко они отстоят друг от друга.

Относительной шкале соответствуют метрики, позволяющие не только расположить программы определенным образом и оценить их положение относительно друг друга, но и определить, как далеко оценки отстоят от границы, начиная с которой характеристика может быть измерена.

МЕТРИКИ СЛОЖНОСТИ ПРОГРАММ

При оценке сложности программ, как правило, выделяют три основные группы метрик:

* метрики размера программ

* метрики сложности потока управления программ

* и метрики сложности потока данных программ

.

МЕТРИКИ РАЗМЕРА ПРОГРАММ.

Оценки первой группы наиболее просты и, очевидно, поэтому получили широкое распространение. Традиционной характеристикой размера программ является количество строк исходного текста. Под строкой понимается любой оператор программы, поскольку именно оператор, а не отдельно взятая строка является тем интеллектуальным "квантом" программы, опираясь на который можно строить метрики сложности ее создания.

Непосредственное измерение размера программы, несмотря на свою простоту, дает хорошие результаты. Конечно, оценка размера программы недостаточна для принятия решения о ее сложности, но вполне применима для классификации программ, существенно различающихся объемами. При уменьшении различий в объеме программ на первый план выдвигаются оценки других факторов, оказывающих влияние на сложность. Таким образом, оценка размера программы есть оценка по номинальной шкале, на основе которой определяются только категории программ без уточнения оценки для каждой категории.

К группе оценок размера программ можно отнести также и метрику Холстеда.

МЕТРИКА ХОЛСТЕДА.

Основу метрики Холстеда составляют четыре измеряемых характеристики программы:

n1 - число уникальных операторов программы, включая символы-
разделители, имена процедур и знаки операций (словарь операторов);

n2 - число уникальных операндов программы (словарь операндов);

N1 - общее число операторов в программе;

N2 - общее число операндов в программе.

Опираясь на эти характеристики, получаемые непосредственно при анализе исходных текстов программ, М. Холстед вводит следующие оценки:

словарь программы
n1=n1+n2,
длину программы
N=N1+N2, (1)
объем программы

V=N*log2(n) (бит). (2)

Под битом подразумевается логическая единица информации - символ, оператор, операнд.

Далее М. Холстед вводит n* - теоретический словарь программы, т.е. словарный запас, необходимый для написания программы, с учетом того, что необходимая функция уже реализована в данном языке и, следовательно, программа сводится к вызову этой функции. Например, согласно М. Холстеду, возможное осуществление процедуры выделения простого числа могло бы выглядеть так:

CALL SIMPLE (X,Y),
где Y - массив численных значений, содержащий искомое число X.
Теоретический словарь в этом случае будет состоять из
n1* : {CALL, SIMPLE (...)},
n1*=2; n2* : {X, Y},
n2*=2,
а его длина, определяемая как
n* = n1* + n2*,
будет равняться 4.
Используя n*, Холстед вводит оценку V*:
V* = n* * log2 n*, (3)
с помощью которой описывается потенциальный объем программы, соответствующий максимально компактному тексту программы, реализующей данный алгоритм.

Практическое занятие № 4

Тема 3: Метрики, направления применения метрик. Метрики сложности. Метрики стилистики

2. Метрики сложности. Метрики стилистики

Цель: Изучить метрики стилистики.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Наиболее простой метрикой стилистики и понятности программ является оценка уровня комментированности программы F:

F=Nком/Nстр, (5)

где Nком - количество комментариев в программе; Nстр - количество строк или операторов исходного текста.

Таким образом, метрика F отражает насыщенность программы комментариями.

Исходя из практического опыта принято считать, что F>=0.1, т. е. на каждые десять строк программы должен приходиться минимум один комментарий. Как показывают исследования, комментарии распределяются по тексту программы неравномерно: в начале программы их избыток, а в середине или в конце - недостаток. Это объясняется тем, что в начале программы, как правило, расположены операторы описания идентификаторов, требующие более "плотного" комментирования. Кроме того, в начале программы также расположены "шапки", содержащие общие сведения об исполнителе, характере, функциональном назначении программы и т. п. Такая насыщенность компенсирует недостаток комментариев в теле программы, и поэтому формула (5) недостаточно точно отражает комментированность функциональной части текста программы.

Более удачен вариант, когда вся программа разбивается на n равных сегментов и для каждого из них определяется Fi:

Fi = sign (Nком/Nстр - 0.1),

при этом

n

F=Сумма(Fi).

i=1

Уровень комментированности программы считается нормальным, если выполняется условие: F=n. В противном случае какой-либо фрагмент программы дополняется комментариями до номинального уровня.

МЕТРИКИ ХОЛСТЕДА

Следующие пять характеристик являются продолжением метрики Холстеда.

1. Для измерения теоретической длины программы N^ М. Холстед вводит аппроксимирующую формулу:

N^ = n1*log2(n1) + n2*log2(n2), (6)

где n1 - словарь операторов; n2 - словарь операндов программы.

Вводя эту оценку, Холстед исходит из основных концепций теории информации, по аналогии с которыми частота использования операторов и операндов в программе пропорциональна двоичному логарифму количества их типов. Таким образом, выражение (6) представляет собой идеализированную аппроксимацию (1), т. е. справедливо для потенциально корректных программ, свободных от избыточности или несовершенств (стилистических ошибок). Несовершенствами можно считать следующие ситуации:

а) последующая операция уничтожает результаты предыдущей без их использования;

б) присутствуют тождественные выражения, решающие совершенно одинаковые задачи;

в) одной и той же переменной назначаются различные имена и т. п.

Подобные ситуации приводят к изменению N без изменения n.

М. Холстед утверждает, что для стилистически корректных программ отклонение в оценке теоретической длины N^ от реальной N не превышает 10%.

Мы предлагаем использовать N^ как эталонное значение длины программы со словарем n. Длина корректно составленной программы N, т. е. программы, свободной от избыточности и имеющей словарь n, не должна отклоняться от теоретической длины программы N^ более чем на 10%. Таким образом, измеряя n1, n2, N1 и N2 и сопоставляя значения N и N^ для некоторой программы, при более чем 10%-ном отклонении можно говорить о наличии в программе стилистических ошибок, т. е. несовершенств.

На практике N и N^ часто существенно различаются.

2. Другой характеристикой, принадлежащей к метрикам корректности программ, по М. Холстеду, является уровень качества программирования L (уровень программы):

L=V*/V, (7)

где V и V* определяется соответственно выражениями (2) и (3).

Исходным для введения этой характеристики является предположение о том, что при снижении стилистического качества программирования уменьшается содержательная нагрузка на каждый компонент программы и, как следствие, расширяется объем реализации исходного алгоритма. Учитывая это, можно оценить качество программирования на основании степени расширения текста относительно потенциального объема V*. Очевидно, для идеальной программы L=1, а для реальной - всегда L<1.

3. Нередко целесообразно определить уровень программы, не прибегая к оценке ее теоретического объема, поскольку список параметров программы часто зависит от реализации и может быть искусственно расширен. Это приводит к увеличению метрической характеристики качества программирования. М. Холстед предлагает аппроксимировать эту оценку выражением, включающим только фактические параметры, т. е. параметры реальной программы:

L^ = 2*n2 / (n1*N2).

4. Располагая характеристикой L^, Холстед вводит характеристику I, которую рассматривает как интеллектуальное содержание конкретного алгоритма, инвариантное по отношению к используемым языкам реализации: I = L^ * V. (8)

На наш взгляд, да и по мнению самого автора, термин интеллектуальность не совсем удачен. Преобразуя выражение (8) с учетом (7), получаем

I = L^V = LV = V*V/V = V*.

Эквивалентность I и V* свидетельствует о том, что мы имеем дело с характеристикой информативности программы.

Введение характеристики I позволяет определить умственные затраты на создание программы. Процесс создания программы условно можно представить как ряд операций:

1) осмысление предложения известного алгоритма;

2) запись предложения алгоритма в терминах используемого языка программирования, т. е. поиск в словаре языка соответствующей инструкции, ее смысловое наполнение и запись.

Используя эту формализацию в методике Холстеда, можно сказать, что написание программы по заранее известному алгоритму есть N^-кратная выборка операторов и операндов из словаря программы n, причем число сравнений (по аналогии с алгоритмами сортировки) составит log2(n).

Если учесть, что каждая выборка-сравнение содержит, в свою очередь, ряд мысленных элементарных решений, то можно поставить в соответствие содержательной нагрузке каждой конструкции программы сложность и число этих элементарных решений. Количественно это можно характеризовать с помощью характеристики L, поскольку 1/L имеет смысл рассматривать как средний коэффициент сложности, влияющий на скорость выборки для данной программы. Тогда оценка необходимых интеллектуальных усилий по написанию программы может быть измерена как

E = N^ * log2(n/L). (9)

Таким образом, E характеризует число требуемых элементарных решений при написании программы.

Однако следует заметить, что E адекватно характеризует лишь начальные усилия по написанию программ, поскольку при построении E не учитываются отладочные работы, которые требуют интеллектуальных затрат иного характера.

Суть интерпретации этой характеристики состоит в оценке не затрат на разработку программы, а затрат на восприятие готовой программы. При этом вместо теоретической длины программы N^ используется ее реальная длина:

E' = N * log2(n/L).

Характеристика E' введена исходя мз предположения, что интеллектуальные усилия на написание и восприятие программы очень близки по своей природе. Однако если при написании программы стилистические погрешности в тексте практически не должны отражаться на интеллектуальной трудоемкости процесса, то при попытке понять такую программу их присутствие может привести к серьезным осложнениям. Эта посылка достаточно хорошо согласуется с нашими выводами относительно взаимосвязи N и N^, изложенными выше.

Преобразуя формулу (9) с учетом выражений (2) и (7), получаем

E = V * V / V*.

Такое представление E', а соответственно и E, так как E=E', наглядно иллюстрирует целесообразность разбиения программ на отдельные модули, поскольку интеллектуальные затраты оказываются пропорциональными квадрату объема программы, который всегда больше суммы квадратов объемов отдельных модулей.

МЕТРИКА ИЗМЕНЕНИЯ ДЛИНЫ ПРОГРАММНОЙ ДОКУМЕНТАЦИИ.

Рассмотрим еще одну метрику, по своему характеру несколько отличающуюся от предыдущих. Она опирается на принцип оценки, при котором используется измерение флуктуации длин программной документации.

Исходным является предположение о том, что чем меньше изменений и корректировок вносится в программную документацию, тем более четко были сформулированы решаемые задачи на всех этапах работ. По мнению автора метрики, неточности и неясности при создании ПО служат причиной увеличения количества корректировок и изменений в документации. И, напротив, демпфированный переходный процесс с немногочисленными изменениями длин документов -естественное следствие хорошо обдуманной идеи, хорошо проведенного анализа, проектирования и ясной структуры программ. Эти взаимосвязи и являются основными для данного метода оценки, суть которого состоит в следующем.

Предположим, что документация изменяется в дискретные моменты времени t(i), i=1,2,...,n. Тогда в любой момент времени t(i) текущая длина документа l(i) может быть определена как

$$l(i) = l(i-1) + a(i) - b(i); \quad l(0) = 0,$$

где l(i-1) - длина документа в предыдущий момент времени; a(i) - добавляемая часть документа; b(i) - исключаемая часть документа.

Далее вводится d(i), представляющая собой отклонение текущей длины документа l(i) от конечного значения l(n):

$$d(i) = l(n) - l(i).$$

Затем рассчитывается интеграл по модулю этого отклонения на интервале от t(i) до t(n), представленный в виде суммы:

$$H(n) = \sum_{i=1}^{n-1} |d(i)| * (t(i+1) - t(i)). \qquad (10)$$

Значение H(n) представляет собой оценку переходного процесса для интервала времени от t(1) до t(n). Однако H(n) не учитывает изменений типа a(i)=b(i), хотя они, бесспорно, влияют на ход дальнейшего процесса.

Чтобы отразить влияние изменений такого рода, называемых в дальнейшем импульсными, вводится экспоненциальная функция, отражающая функцию отклика. Заштрихованная область на рис.5 представляет собой дополнение к оценке H, отражающее влияние импульсного изменения длины документов и вычисляемое как

$$\int_{t(i)} a(i)*e^{(-L^{(-1*(t-t(i)))})}dt = L*L(i) = L*b(i), \quad L>0. \qquad (11)$$

Таким образом, оценка длины документа пропорциональна значению импульсного изменения длины a(i)=b(i) с коэффициентом пропорциональности L.

В принципе импульсное изменение длины документа присутствует и при a(i)<>b(i). Поэтому с учетом (11) автор метрики преобразует выражение (10) к виду

$$H'(n) = \sum_{i=1}^{n-1} [\ |d(i)| * (t(i+1)-t(i)) + L*c(i)\ ], \qquad (12)$$

причем $c(i) = \min \{a(i), b(i)\}$.

Если в процессе работы значения a(i) и b(i) неконтролируемы, импульсное изменение длины учесть нельзя. Тогда c(i)=0, и выражение (12) вырождается в (10). Используя конечное значение длины документа, можно записать

$$H(n)'' = H(n)' / l(n).$$

Практическое занятие № 5

Тема 4: Исследование программного кода на предмет ошибок и отклонения

от алгоритма

1. Исследование программного кода на предмет ошибок

Цель: Научиться исследовать программный код на предмет ошибок и отклонения от алгоритма

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

При проверке правильности программ и систем рассматриваются процессы верификации, валидации и тестирования ПС, которые регламентированы в стандарте ISO/IEC 12207 жизненного цикла ПО. В некоторой зарубежной литературе процессы верификации и тестирования отождествляются.

Тестирование - это процесс обнаружения ошибок в ПО путем исполнения выходного кода ПС на тестовых данных.

Тестирование можно рассматривать, как процесс семантической отладки (проверки) программы, заключающийся в исполнении последовательности различных наборов контрольных тестов, для которых заранее известен результат. Т.е. тестирование предполагает выполнение программы и получение конкретных результатов выполнения тестов.

Цель тестирования - проверка работы реализованных функций в соответствии с их спецификацией. Методы функционального тестирования подразделяются на статические и динамические.

Статические методы тестирования

Статические методы используются при проведении инспекций и рассмотрении спецификаций компонентов без их выполнения. Техника статического анализа заключается в методическом просмотре (или обзоре) и анализе структуры программ, а также в доказательстве их правильности. Статический анализ направлен на анализ документов, разработанных на всех этапах ЖЦ, и заключается в инспекции исходного кода и сквозного контроля программы.

Инспекция ПО - это статическая проверка соответствия программы заданным спецификациям, проводится путем анализа различных представлений результатов проектирования (документации, требований, спецификаций, схем или исходного кода программ) на процессах ЖЦ..

Кроме того, разрабатывается множество новых способов автоматизации символьного выполнения программ. Например, автоматизированное средство статического контроля для языков ориентированной разработки, инструменты автоматизации доказательства корректности и автоматизированный аппарат сетей Петри.

Динамические методы тестирования

Динамические методы тестирования используются в процессе выполнения программ. Они базируются на графе, связывающем причины ошибок с ожидаемыми реакциями на эти ошибки. В процессе тестирования накапливается информация об ошибках, которая используется при оценке надежности и качества ПС.

Цель динамического тестирования программ по принципу "черного ящика" - выявление одним тестом максимального числа ошибок с использованием небольшого подмножества возможных входных данных.

Методы "черного ящика" обеспечивают:

· эквивалентное разбиение;

· анализ граничных значений;

· применение функциональных диаграмм, которые в соединении с реверсивным анализом дают достаточно полную информацию о функционировании тестируемой программы.

"Белый ящик" базируется на структуре программы, в случае "черного ящика", о структуре программы ничего неизвестно. Для выполнения тестирования с помощью этих

"ящиков" известными считаются выполняемые функции, входы (входные данные) и выходы (выходные данные), а также логика обработки, представленные в документации.

Функциональное тестирование

Цель функционального тестирования - обнаружение несоответствий между реальным поведением реализованных функций и ожидаемым поведением в соответствии со спецификацией и исходными требованиями.

Функциональные тесты создаются по внешним спецификациям функций, проектной информации и по тексту на ЯП, относятся к функциональным его характеристикам и применяются на этапе комплексного тестирования и испытаний для определения полноты реализации функциональных задач и их соответствия исходным требованиям.

В задачи функционального тестирования входят:

· идентификация множества функциональных требований;

· идентификация внешних функций и построение последовательностей функций в соответствии с их использованием в ПС;

· идентификация множества входных данных каждой функции и определение областей их изменения;

· построение тестовых наборов и сценариев тестирования функций;

· выявление и представление всех функциональных требований с помощью тестовых наборов и проведение тестирования ошибок в программе и при взаимодействии со средой.

Тесты, создаваемые по проектной информации, связаны со структурами данных, алгоритмами, интерфейсами между отдельными компонентами и применяются для тестирования компонентов и их интерфейсов. Основная цель - обеспечение полноты и согласованности реализованных функций и интерфейсов между ними.

Инфраструктура процесса тестирования ПС

Под инфраструктурой процесса тестирования понимается:

· выделение объектов тестирования;

· проведение классификации ошибок для рассматриваемого класса тестируемых программ;

· подготовка тестов, их выполнение и поиск разного рода ошибок и отказов в компонентах и в системе в целом;

· служба проведения и управление процессом тестирования;

· анализ результатов тестирования.

Объекты тестирования - компоненты, группы компонентов, подсистемы и система.

Практическое занятие № 6

Тема 4: Исследование программного кода на предмет ошибок и отклонения от алгоритма

2. Исследование программного кода на предмет отклонения от алгоритма

Цель: Научиться исследовать программный код на предмет ошибок и отклонения от алгоритма

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Методы поиска ошибок в программах

Международный стандарт ANSI/IEEE-729-83 разделяет все ошибки в разработке программ на следующие типы.

Ошибка (error) - состояние программы, при котором выдаются неправильные результаты, причиной которых являются изъяны (flaw) в операторах программы или в

технологическом процессе ее разработки, что приводит к неправильной интерпретации исходной информации, следовательно, и к неверному решению.

Дефект (fault) в программе - следствие ошибок разработчика на любом из этапов разработки, которая может содержаться в исходных или проектных спецификациях, текстах кодов программ, эксплуатационной документация и т.п. В процессе выполнения программы может быть обнаружен дефект или сбой.

Отказ (failure) - это отклонение программы от функционирования или невозможность программы выполнять функции, определенные требованиями и ограничениями, что рассматривается как событие, способствующее переходу программы в неработоспособное состояние из-за ошибок, скрытых в ней дефектов или сбоев в среде функционирования. Отказ может быть результатом следующих причин:

· ошибочная спецификация или пропущенное требование, означающее, что спецификация точно не отражает того, что предполагал пользователь;

· спецификация может содержать требование, которое невозможно выполнить на данной аппаратуре и программном обеспечении;

· проект программы может содержать ошибки (например, база данных спроектирована без средств защиты от несанкционированного доступа пользователя, а требуется защита);

· программа может быть неправильной, т.е. она выполняет несвойственный алгоритм или он реализован не полностью.

Таким образом, отказы, как правило, являются результатами одной или более ошибок в программе, а также наличия разного рода дефектов.

Классификация ошибок и тестов

Каждая организация, разрабатывающая ПО общесистемного назначения, сталкивается с проблемами нахождения ошибок. Поэтому приходится классифицировать типы обнаруживаемых ошибок и определять свое отношение к устранению этих ошибок.

Таблица 1. Ортогональная классификация дефектов IBM

Контекст ошибки	Классификация дефектов
Функция	Ошибки интерфейсов конечных пользователей ПО, вызванные аппаратурой или связаны с глобальными структурами данных
Интерфейс	Ошибки во взаимодействии с другими компонентами, в вызовах, макросах, управляющих блоках или в списке параметров
Логика	Ошибки в программной логике, неохваченной валидацией, а также в использовании значений переменных
Присваивание	Ошибки в структуре данных или в инициализации переменных отдельных частей программы
Зацикливание	Ошибки, вызванные ресурсом времени, реальным временем или разделением времени
Среда	Ошибки в репозитории, в управлении изменениями или в контролируемых версиях проекта
Алгоритм	Ошибки, связанные с обеспечением эффективности, корректности алгоритмов или структур данных системы
Документация	Ошибки в записях документов сопровождения или в публикациях

Исследования фирм IBM показали, чем позже обнаруживается ошибка в программе, тем дороже обходится ее исправление, эта зависимость близка к экспоненциальной. Так военно-воздушные силы США оценили стоимость разработки одной инструкции в 75 долларов, а ее стоимость сопровождения составляет около 4000 долларов.

Создаются тесты, проверяющие:

· полноту функций;

· согласованность интерфейсов;

· корректность выполнения функций и правильность функционирования системы в заданных условиях;

· надежность выполнения системы;

· защиту от сбоев аппаратуры и не выявленных ошибок и др.

Служба тестирования ПС

Для этих целей, как правило, создается служба проверяющих ПС - команда тестировщиков, которая не зависит от штата разработчиков ПС. Некоторые члены этой команды - опытные или даже профессионалы в этой области. К ним относятся аналитики, программисты, инженеры-тестировщики, которые посвящают в проблемы тестирования систем с начала разработки. Они имеют дело не только со спецификациями, но и с методами и средствами тестирования, организуют создание и выполнение тестов. С самого начала создания проекта тестировщики составляют планы тестирования, тестовые данные и сценарии, а также графики выполнения тестов.

Управление процессом тестирования

Все способы тестирования ПС объединяются базой данных, где помещаются результаты тестирования системы. В ней содержатся все компоненты, тестовые контрольные данные, результаты тестирования и информация о документировании процесса тестирования.

База данных проекта поддерживается специальными инструментальными средствами типа CASE, которые обеспечивают ведение анализа ПрО, сборку данных об их объектах, потоках данных и тому подобное. База данных проекта хранит также начальные и эталонные данные, которые используются для сопоставления данных, накопленных в базе, с данными, которые получены в процессе тестирования системы.

Контрольные вопросы и задания

1. Назовите формальные методы проверки правильности программ.
2. Какие процессы проверки зафиксированы в стандарте?
3. Какие функции у процесса верификации программ?
4. Назовите основные задачи процесса валидации программ.
5. Сравните задачи процессов верификации и валидации программ.
6. В чем отличие верификации и валидации?
7. Определите процесс тестирования.
8. Назовите методы тестирования.
9. Объясните значения терминов "черный ящик", "белый ящик".
10. Назовите объекты тестирования и подходы к их тестированию.
11. Какая существует классификация типов ошибок в программах?
12. Определите основные процессы ЖЦ тестирования ПО.
13. Наведите классификацию тестов для проверки ПО.
14. Какие задачи выполняет группа текстовиков?
15. Какая организация работ в проведении тестирования?

Практическое занятие № 7

Тема 5: Программные измерительные мониторы

1. Программные измерительные мониторы

Цель: Изучение процесса управления программным кодом и документацией модифицируемых программных систем. Понимать значение управления конфигурацией ПО.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Измерения являются источником наиболее достоверных данных о функционировании вычислительных систем и проводятся в следующих целях: 1) для учета выполненных работ: 2) для оценки функционирования; 3) для идентификации вычислительной системы – построения моделей. Измерения могут быть направлены на исследование как системы в целом, так и отдельных подсистем.Схема измерений представлена на рис. 1. Объектом измерений является вычислительная система, функционирующая, как правило, в рабочем режиме. К системе подключаются измерительные средства – мониторы, реагирующие на изменение состояний системы и измеряющие параметры состояний (моменты изменения состояний, продолжительность пребывания в них и др.). Измерительные данные поступают от мониторов в архив на протяжении заданного промежутка времени, накапливаются и затем обрабатываются.

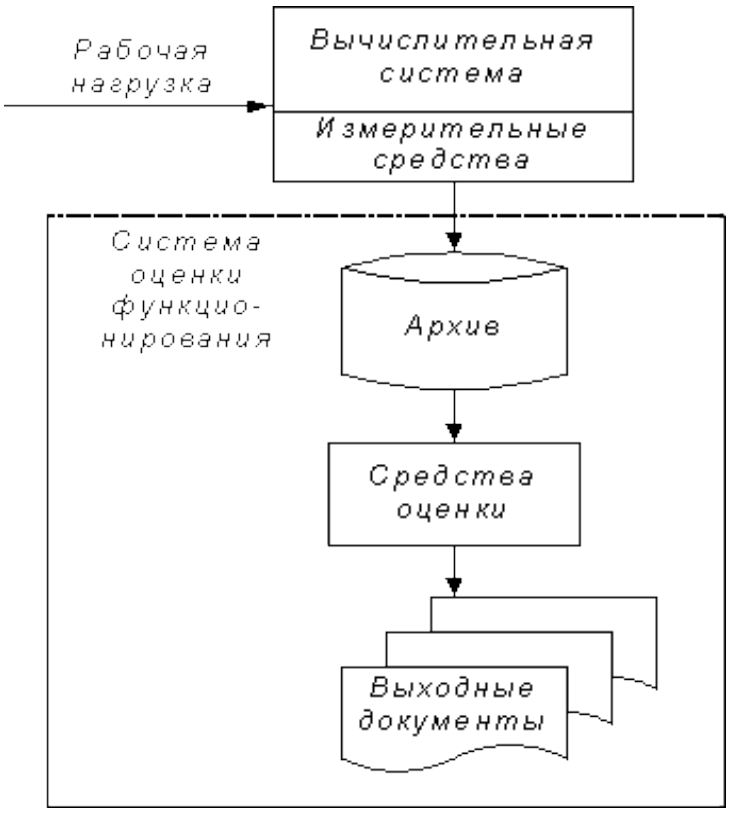

Рис.1. Организация измерений и оценки функционирования

Функционирование вычислительной системы проявляется в изменении состояний процессов и ресурсов. Состояния отображаются в управляющих таблицах, которые

формируются управляющими программами операционной системы, и в первую очередь супервизором. Состояния процессов и ресурсов изменяются в моменты выполнения специальных команд – обращения к супервизору и привилегированных, с помощью которых супервизор управляет процессами и ресурсами, а также при поступлении сигналов прерывания, извещающих супервизор о моментах окончания операций ввода – вывода, особых ситуациях в системе и сигналах на ее входах. Наряду с этим состояния устройств отображаются соответствующими электрическими сигналами.

Мониторы строятся с использованием различных методов измерений и средств и классифицируются в зависимости от этого (рис 2).

Трассировочный и выборочный методы измерений. *Трассировочный метод* измерений основан на регистрации событий, соответствующих моментам изменения состояний вычислительной системы. К таким событиям, в частности, относятся начало и конец ввода задания, шага задания, этапа процессорной обработки, обращения к внешней памяти и т. д. События регистрируются монитором в виде событийного набора данных *T* (рис. 3), состоящего из последовательности записей s_1, s_2, ..., соответствующих последовательности событий. В записи регистрируется момент возникновения события, имена процесса и ресурса, с которыми оно связано, и параметры события, – например емкость занимаемого или освобождаемого блока памяти, число передаваемых байтов данных и т. д. Событийный набор данных, создаваемый монитором, содержит информацию о процессах $J_1, J_2, \cdots$ и одновременно о ресурсах. На рисунке изображена диаграмма использования устройства R_i, представляющая его состояния (0 – свободно и 1 – занято), и диаграмма использования памяти M_j, характеризующая суммарную емкость, занятую процессами. Мониторы, измеряющие процесс функционирования системы трассировочным методом, называются *трассировочными.*

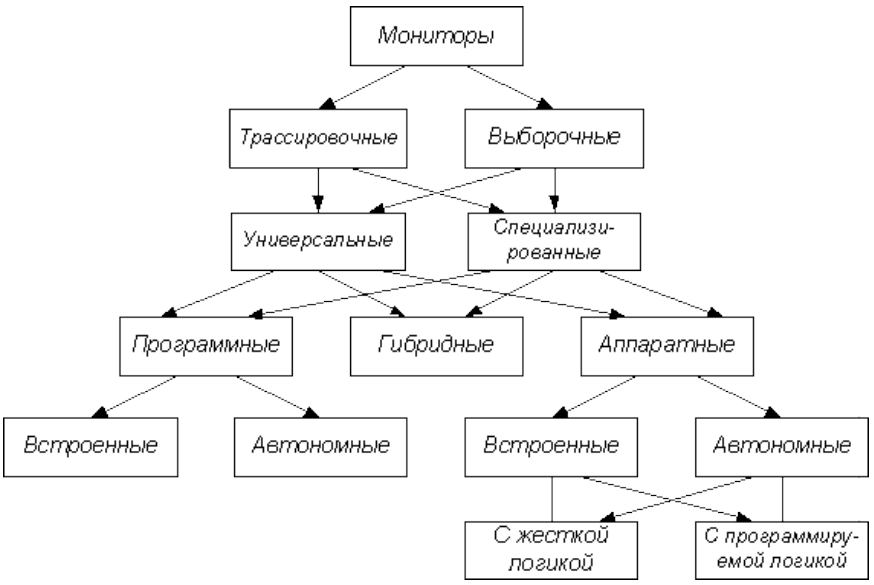

Рис. 2. Классификация мониторов

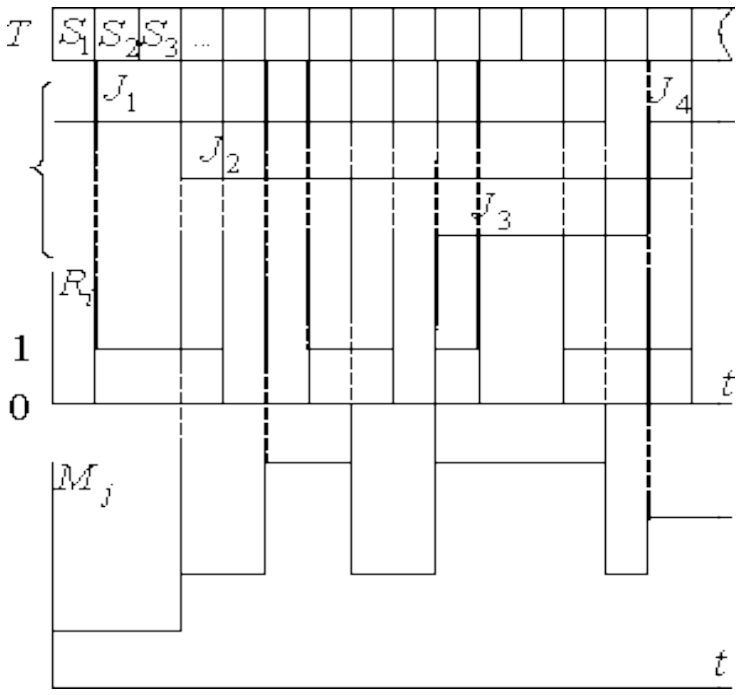

Рис. 3. Состав измерительных данных

Выборочный метод измерений основан на регистрации состояний вычислительной системы в заданные моменты времени, как правило, через промежутки длительностью δ. В моменты $t = n\delta, n = 0, 1, 2...$, выборочный монитор регистрирует состояние системы, фиксируя в соответствующих записях данные из управляющих таблиц, или значения электрических сигналов, характеризующих состояния устройств системы. Полученные данные позволяют с погрешностью не более 26 оценивать продолжительность пребывания процессов и ресурсов в различных состояниях и вероятности состояний. Последние определяются значениями $p_i = n_i / n$, где n_i – число выборок, при которых было зарегистрировано состояние *i*, и *n* – длительность процесса измерений, определяемая числом выборок.

Трассировочные мониторы измеряют отдельные процессы, например обработку одного задания, более точно, чем выборочные. Однако, если функционирование системы оценивается статистическими методами, выборочный монитор обеспечивает такую же точность, как и трассировочный, правда при большей продолжительности измерений. Основное достоинство выборочных мониторов – возможность измерений сколь угодно быстрых процессов при ограниченном быстродействии.

Практическое занятие № 8

Тема 5: Программные измерительные мониторы

2. Универсальные и специализированные мониторы

Цель: Изучение процесса управления программным кодом и документацией модифицируемых программных систем. Понимать значение управления конфигурацией

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Универсальные и специализированные мониторы. В зависимости от регистрируемого состава событий (состояний) мониторы подразделяются на универсальные и специализированные.

Универсальный монитор регистрирует все события (состояния) или подавляющее большинство их, благодаря чему событийный набор данных достаточен для построения трасс процессов и использования ресурсов. Объем измерительных данных чрезвычайно велик и составляет 10^5–10^7 байт на один процесс. Поэтому в мониторы встраиваются средства настройки, позволяющие регистрировать часть событий, соответствующих целым исследований. Универсальные мониторы используются в основном периодически для оценки, например, конкретных системных или прикладных процессов.

Специализированный монитор регистрирует определенную часть событий (состояний), соответствующих конкретной цели измерений, что приводит к умеренному объему измерительных данных и снижает сложность их обработки. Специализированные мониторы широко применяются для учета выполненных работ и оценки загрузки ресурсов. Благодаря умеренному потоку данных специализированные мониторы используются как постоянно действующие измерительные средства для оценки функционирования систем в течение всего рабочего периода.

Программные мониторы. Мониторы, реализованные в виде программы, выполняемой вычислительной системой, называются *программными*.

Программные мониторы трассировочного типа реагируют на определенный класс событий за счет «перехвата» обращений к супервизору, сигналов пребывания и других сигналов, что приводит к передаче управления программным блокам монитора. Приняв управление, монитор выбирает из управляющих таблиц супервизора, программ управления заданиями и данными необходимую информацию, формирует запись, соответствующую событию, и помещает ее в заданную область памяти. Затем передает управление в точку, перед обращением к которой включается в работу в момент возникновения событий, соответствующих смене состояний.

Программные мониторы выборочного типа включаются в работу по таймеру, отмечающему интервалы времени δ, через которые производится опрос состояний вычислительной системы. Как и мониторы трассировочного типа, они получают необходимые данные о состоянии системы из управляющих таблиц и заносят сформированные данные в заданную область памяти.

В зависимости от местоположения и от статуса программные мониторы подразделяются на встроенные и автономные (см. рис. 4).

Встроенный программный монитор – совокупность программных блоков, входящих в состав управляющих программ операционной системы. Встроенный монитор создается совместно с операционной системой и является ее частью. За счет этого обращение к блокам монитора реализуется короткими цепочками команд и минимизируются затраты процессорного времени на выполнение измерительных процедур. Встроенные программные мониторы, как правило, имеют статус управляющих программ операционной системы. В операционные системы встраиваются специализированные измерительные средства для учета выполненных работ, контроля использования ресурсов и получения данных о сбоях и отказах системы. Такого рода измерительные средства дают минимальные сведения о функционировании вычислительной системы.

Для расширения измерительных возможностей используются *автономные программные мониторы* – измерительные программы, выполняемые системой в основном как прикладной процесс. Автономные мониторы загружаются в оперативную память как прикладные программы. Монитор программно связывается с супервизором, за счет чего в момент возникновения событий монитору передается управление. Монитор выбирает необходимые данные из управляющих таблиц, обрабатывает их, формирует запись в наборе измерительных данных и возвращает управление супервизору. Автономные мониторы, как правило, универсальны и позволяют регистрировать широкую номенклатуру событий при трассировочном и состояний и состояний при выборочном методе измерений. Специализированные автономные мониторы используются для

контроля за функционированием отдельных подсистем вычислительной системы, например процессов ввода – вывода, работы внешних запоминающих устройств и др. В отличие от встроенных автономные мониторы используются для оценки функционирования системы лишь периодически.

Основное достоинство программного способа построения мониторов – возможность получения сколь угодно детальной информации. Недостатки – зависимость программных мониторов от типа ЭВМ и операционной системы, а также влияние монитора на временные аспекты функционирования системы. Программные мониторы создаются для ЭВМ и операционных систем конкретных типов. Поэтому постановка на ЭВМ новой операционной системы или расширение операционной системы требует модификации измерительных средств. Поскольку программы монитора реализуются совместно с прикладными и системными программами, обработка заданий при измерениях растягивается во времени. Встроенные программные мониторы характеризуются незначительной ресурсоемкостью и порождаемая ими нагрузка на процессор невелика. Автономные программные мониторы имеют значительную ресурсоемкость, которая может составлять 10-15% процессорного времени. Выборочные мониторы позволяют снижать нагрузку на ресурсы за счет увеличения периода регистрации, однако при этом для получения представительного объема данных приходится увеличивать продолжительность измерений.

Аппаратные мониторы. *Аппаратный монитор* – комплекс технических и, возможно, программных средств, предназначенных для измерения процессов функционирования вычислительных систем. Принцип измерений с помощью аппаратного монитора иллюстрируется рис. 5. Монитор получает информацию о состоянии системы посредством электрических сигналов, характеризующих состояние отдельных устройств и блоков. Для измерений выявляются точки подключения монитора к системе, в которых присутствуют сигналы, представляющие состояние устройств. В качестве точек подключения наиболее часто используются выходы триггеров и линии интерфейсов. Точки подключения связываются с монитором при помощи зондов. Зонд состоит из усилителя, обеспечивающего передачу сигнала по длинной линии и имеющего высокое входное сопротивление, и линии, соединяющей усилитель с входом монитора. Сигналы с зондов обрабатываются селектором – схемой, формирующей на основе входных сигналов сигналы состояний (событий) $S_1,\ldots,S_M$, которые должны обрабатываться монитором.

Сигналы $S_1,\ldots,S_M$ с селектора поступают в измерительный блок, в котором выполняются типичные измерительные процедуры: определяется время поступления сигнала, длительность промежутка между двумя событиями, отмечаемыми соответствующими сигналами, и число событий. Результаты измерений вводятся в микро-ЭВМ по сигналам прерывания или по таймеру. Микро-ЭВМ обрабатывает поступающие измерительные данные, записывает их на некоторый носитель (например, на магнитную ленту) иоперативные оценки функционирования отображает на терминале. Оперативные оценки используются для контроля за ходом процесса изменений и функционированием вычислительной системы. Зарегистрированные на носителе измерительные данные в дальнейшем подвергаются обработке с помощью ЭВМ монитора или другой ЭВМ, в том числе исследуемой.

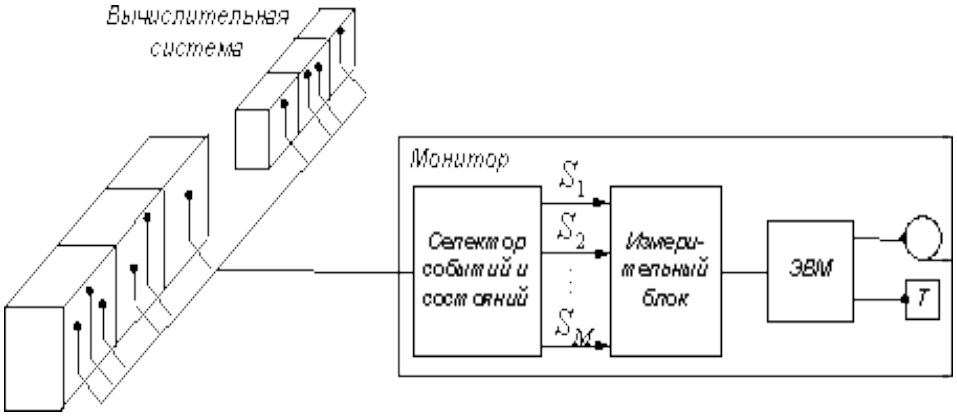

Рис. 4. Измерения с помощью аппаратного монитора

В зависимости от назначения и способа построения аппаратные мониторы подразделяются на автономные, встроенные, с жесткой и программируемой логикой (см. рис. 7.13). *Автономный аппаратный монитор* – переносное устройство для измерений различных вычислительных систем. *Встроенный аппаратный монитор* – устройство или блок, конструктивно встроенный в вычислительную систему. *Аппаратные мониторы с жесткой логикой* предназначены для получения фиксированного набора оценок функционирования, который вычисляется с помощью схемных средств или программ, хранимых в постоянной памяти. Мониторы с жесткой логикой наиболее широко используются для контроля за использованием ресурсов системы, оцениваемым с помощью коэффициентов загрузки. *Аппаратные мониторы с программируемой логикой* имеют в своем составе ЭВМ, путем программирования которой можно получать различные данные о функционировании исследуемой системы. В таких мониторах возможна перестройка функций селектора событий и состояний и функций измерительного блока в зависимости от состава входных сигналов и цели измерений.

Пропускная способность аппаратного монитора, определяемая затратами времени на измерение и регистрацию состояний, ограничивается быстродействием используемых в мониторе средств обработки и хранения измерительных данных. Если частота изменения состояний в системе не превосходит пропускной способности монитора, используется трассировочный метод измерений. В противном случае измерения проводятся по выборочному методу, который не налагает ограничений на скорость процессов в измеряемой системе.

Основные достоинства аппаратных мониторов – системная независимость и независимость процесса функционирования системы от процедуры измерений. Системная независимость обусловлена тем, что аппаратные мониторы прямо не связаны с программными средствами, а поэтому могут использоваться для измерений функционирования любых ЭВМ, работающих в различных режимах под управлением любых операционных систем. Аппаратные мониторы не используют ресурсов исследуемой системы, и поэтому процесс измерений не влияет на функционирование системы. Недостаток аппаратного способа измерений – существенные ограничения на состав информации о функционировании системы, доступный для монитора. Монитор может получать только ту информацию, которая отображается в устройствах в виде сигналов, и ему недоступна информация, формируемая программами и отображаемая в памяти системы. Поэтому аппаратные мониторы не могут регистрировать атрибуты заданий (имена пользователей и программ), состояние очередей и другую информацию. Наиболее доступна для регистрации информация, связанная с использованием ресурсов: загрузка устройств, интенсивность обращения к устройствам, частота различных операций, интенсивность потоков данных, передаваемых через интерфейсы, и др.

Гибридные мониторы. Для использования преимуществ программного и аппаратного способа измерений создаются гибридные мониторы, в которых используются

программные средства для получения данных о состояниях системы и аппаратные средства для регистрации измерительных данных, поступающих от программных средств.

В структурном отношении гибридный монитор состоит из программной и аппаратной части. Программная часть – совокупность программных блоков, фиксирующих изменение состояний прикладных и системных процессов. Программные блоки формируют данные для аппаратной части монитора, которые выводятся через соответствующий канал (интерфейс) ввода – вывода. Аппаратный монитор подключается к каналу и, получая данные от измерительных программ, обрабатывает их собственными средствами. За счет такой организации измерительных средств обеспечивается доступ к информации, формируемой на программном уровне, и существенно снижаются затраты ресурсов системы на измерения, поскольку измерительные программы занимают небольшую область памяти и выполняются с незначительными затратами процессорного времени. Аппаратный монитор имеет статус периферийного устройства и работает в основном автономно, используя собственную память, процессор и средства ввода – вывода.

Организация оценки функционирования. Оценка функционирования вычислительных систем сводится к обработке измерительных данных, зарегистрированных программными и аппаратными мониторами, с целью определения системных характеристик (производительность, время ответа и надежность), показателей использования ресурсов, характеристик рабочей нагрузки, а также с целью идентификации системы. Наиболее широко используется двухэтапный способ оценки функционирования (см. рис.2), На первом этапе собираются и накапливаются в архиве измерительные данные. На втором этапе данные обрабатываются. Программные средства хранения, доступа к данным и оценки функционирования совместно с положенными в их основу концептуальными и математическими моделями функционирования и методами измерений образуют *систему оценки функционирования* (рис. 5).

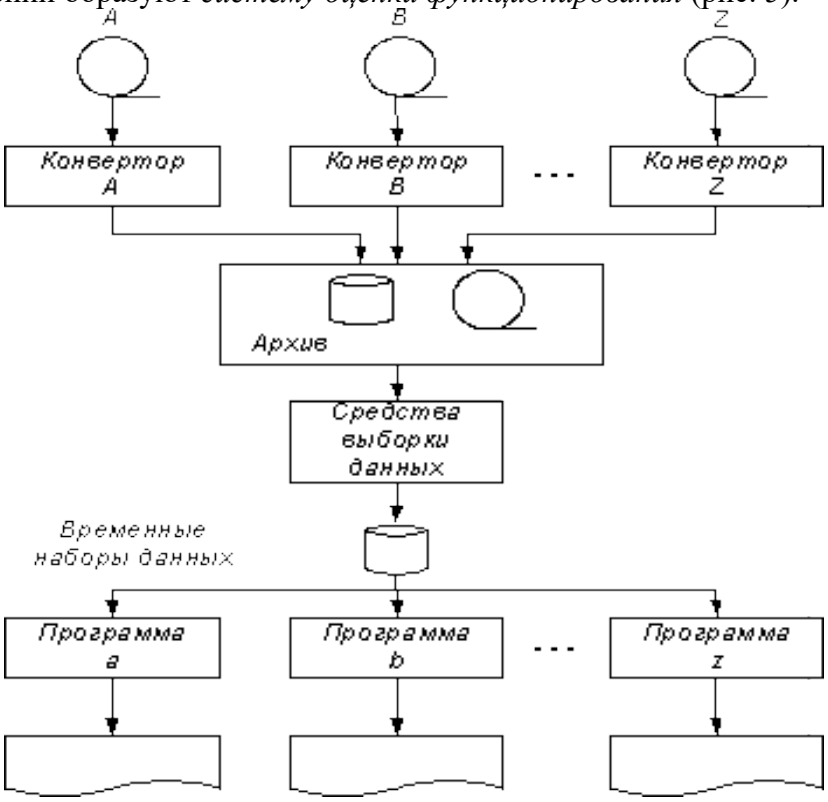

Рис. 5. Состав системы оценки функционирования

Средства хранения создают наборы измерительных данных. Как правило, измерительные данные, формируемые монитором, представляют собой событийные наборы, в которых записи соответствуют отдельным событиям. С целью экономии памяти, используемой для хранения данных, событийные наборы перед записью в архив преобразуются в наборы с объектной структурой, в которых записи соответствуют таким объектам, как задание. Объектные наборы, как и событийные, имеют последовательную организацию с упорядоченными во времени записями об объектах. Преобразование событийных наборов в объектные выполняется программами конвертирования, каждая из которых соответствует монитору определенного типа.

Оценка функционирования системы в большинстве случаев производится на основе некоторого подмножества хранимых в архиве измерительных данных, относящихся к определенным периодам работы системы или к определенным классам заданий. Селективная выборка данных из архива выполняется программными средствами выборки, которые затем формируют наборы данных об определенных классах объектов (временные интервалы, группы пользователей, продолжительность пребываний заданий в системе и др.). Сформированные наборы данных обрабатываются программами оценки системных характеристик, загрузки ресурсов, рабочей нагрузки и т. д. Результаты оценки представляются в виде выходных документов. Для оценки характеристик используются как специально разрабатываемые программы, так и пакеты программ общего применения, например программ статистического анализа.

Практическое занятие № 9

Тема 6: Применение отладчиков и дизассемблера (напримерOllyDbg, WinDbg, IdaPro)

1. Применение отладчиков

Цель: Изучение отладчиков и дизассемблера.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Знание ассемблера позволяет «вскрыть» любую программу дизассемблером и изучить механизм её работы. Ассемблер может помочь при отладке при возникновении ошибок в компиляторах.

Язык Ассемблера - это язык программирования низкого уровня. Для начала вы должны ознакомиться с общей структурой процессора, чтобы в дальнейшем понимать, о чем идет речь.

На рис. 1 представлена структурная схема микропроцессора 8086, в состав которого входят: устройство управления (УУ), арифметико-логическое устройство (АЛУ), блок преобразования адресов и регистры.

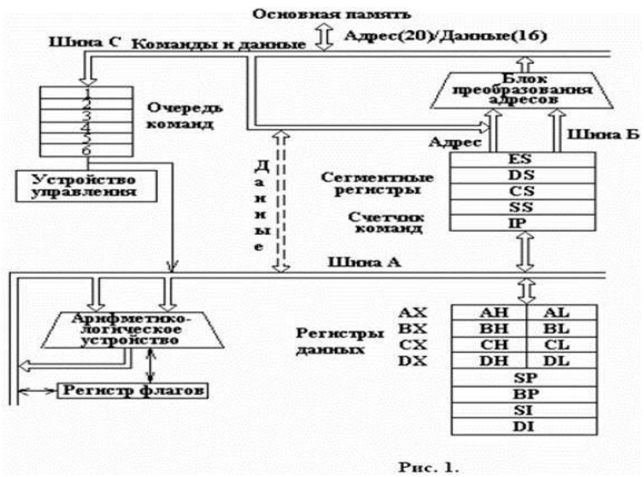

Рис. 1.

Арифметико-логическое устройство (АЛУ) производит операции над двумя величинами для получения результата и выработки ряда признаков (результат меньше нуля, равен нулю, или больше нуля и т. д.).
Аккумулятор – регистр, используемый для размещения подлежащих обработке данных или результатов выполнения операции.
Регистр команд – регистр, служащий для размещения текущей команды.
Регистр адреса – регистр, содержащий адрес ячейки памяти, из которой будет считана команда (операнд) или в которую требуется записать результат выполнения команды.
Регистр(ы) данных – используются в качестве буфера (промежуточного хранилища информации) между памятью и остальными регистрами процессора, через регистр данных пересылаются в процессор команды (операнды) и передаются в память результаты обработки.
Счетчик команд – регистр, указывающий на адрес следующей команды, которая должна быть выполнена после завершения выполнения текущей команды; содержимое счетчика команд увеличивается на единицу в момент выборки из памяти текущей исполняемой команды.
Регистр состояния (флаговый регистр) – регистр, хранящий признаки результата выполнения последней операции; эти признаки используются для организации работы команд перехода.
Устройство управления дешифрирует коды команд и формирует необходимые управляющие сигналы. Арифметико-логическое устройство осуществляет необходимые арифметические и логические преобразования данных. В блоке преобразования адресов формируются физические адреса данных, расположенных в основной памяти. Наконец, регистры используются для хранения управляющей информации: адресов и данных.
Необходимо знать, какие регистры процессора существуют и как их можно использовать. Все процессоры архитектуры x86 (даже многоядерные, большие и сложные) являются дальними потомками Intel 8086 и совместимы с его архитектурой. Это значит, что программы на ассемблере 8086 будут работать и на старших версиях процессоров.
Все внутренние регистры процессора Intel 8086 являются 16-битными:

Всего процессор содержит 12 программно-доступных регистров, а также регистр флагов (FLAGS) и указатель команд (IP).
Регистры общего назначения (РОН) AX, BX, CX и DX используются для хранения данных и выполнения различных арифметических и логических операций. Кроме того, каждый из этих регистров поделён на 2 части по 8-бит, с которыми можно работать как с 8-битными регистрами (AH, AL, BH, BL, CH, CL, DH, DL). Младшие части регистров имеют в названии букву L (от слова *Low*), а старшие H (от

слова *High*). Некоторые команды неявно используют определённый регистр, например, CX может выполнять роль счетчика цикла.

Индексные регистры предназначены для хранения индексов при работе с массивами. SI (*Source Index*) содержит индекс источника, а DI (*Destination Index*) — индекс приёмника, хотя их можно использовать и как регистры общего назначения.

Регистры-указатели BP и SP используются для работы со стеком. BP (*Base Pointer*) позволяет работать с переменными в стеке. Его также можно использовать в других целях. SP (*Stack Pointer*) указывает на вершину стека. Он используется командами, которые работают со стеком. (Про стек я подробно расскажу в отдельной части учебного курса)

Сегментные регистры CS (*Code Segment*), DS (*Data Segment*), SS (*Stack Segment*) и ES (*Enhanced Segment*) предназначены для обеспечения сегментной адресации. Код находится в сегменте кода, данные — в сегменте данных, стек — в сегменте стека и есть еще дополнительный сегмент данных. Реальный физический адрес получается путём сдвига содержимого сегментного регистра на 4 бита влево и прибавления к нему смещения (относительного адреса внутри сегмента).

COM-программа всегда находится в одном сегменте, который является одновременно сегментом кода, данных и стека. При запуске COM-программы сегментные регистры будут содержать одинаковые значения.

Указатель команд IP (*Instruction Pointer*) содержит адрес команды (в сегменте кода). Напрямую изменять его содержимое нельзя, но процессор делает это сам. При выполнении обычных команд значение IP увеличивается на размер выполненной команды. Существуют также команды передачи управления, которые изменяют значение IP для осуществления переходов внутри программы.

Регистр флагов FLAGS содержит отдельные биты: флаги управления и признаки результата. Флаги управления меняют режим работы процессора:

· D (*Direction*) — флаг направления. Управляет направлением обработки строк данных: DF=0 — от младших адресов к старшим, DF=1 — от старших адресов к младшим (для специальных строковых команд).

· I (*Interrupt*) — флаг прерывания. Если значение этого бита равно 1, то прерывания разрешены, иначе — запрещены.

· T (*Trap*) — флаг трассировки. Используется отладчиком для выполнения программы по шагам.

Признаки результата устанавливаются после выполнения арифметических и логических команд:

· S (*Sign*) — знак результата, равен знаковому биту результата операции. Если равен 1, то результат — отрицательный.

· Z (*Zero*) — флаг нулевого результата. ZF=1, если результат равен нулю.

· P (*Parity*) — признак чётности результата.

· C (*Carry*) — флаг переноса. CF=1, если при сложении/вычитании возникает перенос/заём из старшего разряда. При сдвигах хранит значение выдвигаемого бита.

· A (*Auxiliary*) — флаг дополнительного переноса. Используется в операциях с упакованными двоично-десятичными числами.

· O (*Overflow*) — флаг переполнения. CF=1, если получен результат за пределами допустимого диапазона значений.

Практическое занятие № 10

Тема 6: Применение отладчиков и дизассемблера (напримерOllyDbg, WinDbg, IdaPro)

2. Применение дизассемблера.

Цель: Изучение отладчиков и дизассемблера.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Декомпилятор (или *дизассемблер)* – программа, которая преобразует двоичный код программ в исходный текст, написанный на одном из языков программирования, чаще всего – ассемблере. Некоторые дизассемблеры могут представить исходный текст на простом языке С. В процессе трансляции большая часть информации об исходном тексте программы теряется, например имена переменных, поэтому декомпилятор пытается восстановить исходный текст программы настолько, насколько это возможно. Если при декомпиляции таблица соответствия имен была не найдена, то зачастую декомпилятор присваивает переменным имена, составленные из плохо воспринимаемой последовательности цифр и букв.

Проблема несколько упрощается, если исследователь в состоянии разобраться с ассемблерным кодом, генерируемым декомпилятором. В этом случае декомпилятор особенно полезен. Рассмотрим пример результатов работы декомпилятора.

Среди коммерческих декомпиляторов для Windows хорошая репутация у IDA Pro компании DataRescue. IDA Pro может декомпилировать программный код многих процессоров, включая виртуальную машину Java.

На рисунке показан пример применения декомпилятора IDA Pro для дизассемблирования программы pbrush.exe (Paintbrush). IDA Pro нашел секцию внешних функций, используемых программой pbrush.exe. Если программа выполняется под управлением операционной системы, которая поддерживает разделяемые библиотеки (например, под управлением операционных систем Windows или UNIX), то она содержит список необходимых ей библиотек. Обычно этот список представлен в удобочитаемом виде, который легко обнаружить при экспертизе выполняемого кода. Для выполнения программ операционной системе также требуется этот список, поэтому она загружает его в память. В большинстве случаев это позволяет декомпилятору вставить список в двоичный код программы, сделав его более понятным.

Чаще всего таблица соответствия имен pbrush.exe недоступна, поэтому в большей части сгенерированного декомпилятором ассемблерного кода отсутствуют имена.

Оценочную версию IDA Pro, пригодную для первоначального знакомства с программой, можно загрузить с www.datarescue.com/idabase/ida.htm. SoftICE компании Numega – другой популярный отладчик. Дополнительные сведения о нем можно найти по адресу www.compuware.com/products/numega/drivercentral/.

Для сравнения была написана небольшая программа на языке С (классическая небольшая программа, выводящая строку «Hello World»). Для отладки использовался отладчик GNU (GDB). Код программы представлен ниже:

```
#include <stdio.h>
int main ()
{
printf (“Hello World\n”);
return (0);
}
```

Программа была скомпилирована с включением отладочной информации (был включен переключатель *—g):*

```
[elliptic@]$ gcc -g hello.c -o hello
[elliptic@ellipse]$ ./hello
Hello World
```

Пример протокола отладки под управлением GDB показан ниже:

```
[elliptic@ellipse]$ gdb hello
GNU gdb 19991004
Copyright 1998 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or
distribute copies of it under certain conditions.
Type “show copying” to see the conditions.
There is absolutely no warranty for GDB. Type “show
warranty” for details.
This GDB was configured as “i386-redhat-linux”...
(gdb) break main
```

Была установлена точка прерывания при входе в функцию *main.* При ее достижении выполнение программы было приостановлено для выполнения программистом действий по отладке программы. Контрольная точка была установлена до выдачи команды **run.**

```
Breakpoint 1 at 0x80483d3: file hello.c, line 5.
(gdb) run
```

Команда **run** начинает выполнение программы под управлением отладчика.

```
Starting program: /home/ryan/hello
Breakpoint 1, main () at hello.c:5
5 printf (“Hello World\n”);
(gdb) disassemble
```

После того достижения программой точки прерывания и начала сессии отладки была выдана команда **disassemble,** позволяющая вывести дополнительную отладочную информацию о программе.

```
Dump of assembler code for function main:
0x80483d0 <main>: push %ebp
0x80483d1 <main+1>: mov %esp,%ebp
0x80483d3 <main+3>: push $0x8048440
0x80483d8 <main+8>: call 0x8048308 <printf>
0x80483dd <main+13>: add $0x4,%esp
0x80483e0 <main+16>: xor %eax,%eax
0x80483e2 <main+18>: jmp 0x80483e4 <main+20>
0x80483e4 <main+20>: leave
0x80483e5 <main+21>: ret
End of assembler dump.
```

Протокол отображает ассемблерный код программы «hello world», соответствующий ассемблеру x86 Linux. Исследование собственных программ в отладчике – хороший способ изучения листингов ассемблерного кода.

```
(gdb) s
printf (format=0x8048440 “Hello World\n”) at printf.c:30
printf.c: No such file or directory.
```

После задания командой **s** («step») режима пошагового выполнения программы в отладчике выполняется переход к вызову функции *printf.*GDB сообщает о невозможности дальнейшей детализации функции *printf* из-за отсутствия в распоряжении отладчика исходного кода функции *printf.*

```
(gdb) s
31 in printf.c
(gdb) s
Hello World
35 in printf.c
(gdb) c
Continuing.
```

Далее выполняется несколько шагов реализации программы внутри функции *printf* и программа завершается. По команде **c** («continue») программа будет выполнена до следующей точки прерывания или будет завершена.

Program exited normally.
(gdb)

В число аналогичных программ входят программы *nm* и *objdump* пакета GNU. Программа *objdump* позволяет управлять объектными файлами. Она применяется для отображения символов объектного файла, его заголовка и даже дизассемблирования. *Nm* выполняет аналогичные действия, что и *objdump,* позволяя просматривать символьные ссылки на объектные файлы.

Инструментарий и ловушки

Инструментарий никогда не заменит знаний

Некоторые из средств дизассемблирования и отладки предлагают фантастические возможности. Но они, как и любой другой инструментарий, несовершенны. Особенно это проявляется при анализе злонамеренного кода (вирусов, червей, Троянских коней) или специально разработанных программ. Обычно авторы подобных поделок делают все возможное для затруднения их анализа и снижения эффекта от применения отладчиков, дизассемблеров и других подобных инструментальных средств. Например, вирус RST для Linux в случае обнаружения, что он работает под управлением отладчика, завершает свою работу. Этот же вирус при инфицировании исполняемых и компонуемых файлов ELF (Executable and Linkable Format – формат исполняемых и компонуемых модулей) модифицирует их заголовки таким образом, что некоторые дизассемблеры не могут дизассемблировать двоичный код вируса (обычно это достигается путем размещения кода вируса в необъявленном программном сегменте). Часто злоумышленный код шифруется или сжимается, что защищает его от экспертизы. Черви Code Red распространялись половинками своих программ, маскируясь под строки переполнения, и, таким образом, формат их двоичных данных не подходил ни под один из стандартных заголовков файла.

Поэтому при анализе двоичных данных нужно знать не только о том, какие инструментальные средства и как применять, но и как обойтись без них. Анализируя заголовок файла, возможно, потребуется определить, какие данные в файле модифицированы, как и с какой целью. Вероятно, потребуется проанализировать зашифрованные данные и процедуру их расшифровки, восстановить алгоритм работы программы и выяснить результаты ее работы.

Необходимо знать ассемблер не только в объеме, достаточном для чтения ассемблерных программ, но и для написания программ расшифровки или восстановления данных. Писать на ассемблере намного сложнее, чем читать программы, написанные на этом языке.

Из этого не следует, что инструментальные средства бесполезны. Далеко не так. Нужно только выявить часть программы, оказавшуюся не по зубам инструментальным средствам, и проанализировать ее самостоятельно, а остальную часть программы – при помощи инструментальных средств. Кроме того, иногда для лучшего понимания логики работы программы следует воспользоваться инструментарием.

Практическое занятие № 11

Тема 7: Исследование кода вредоносных программ

1. Технология поиска вредоносного кода.

Цель: Научиться определять коды вредоносных программ.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Для начала рассмотрим, как работают технологии поиска вредоносного кода. Для этого я предлагаю использовать следующую модель. В любой защитной технологии можно выделить два компонента: технический и аналитический. Эти компоненты не обязаны быть четко разграничены на уровне модулей или алгоритмов, но на функциональном уровне они различимы. Технический компонент — это совокупность программных функций и алгоритмов, обеспечивающих аналитический компонент данными для анализа. В качестве таковых могут выступать, к примеру, байтовый код файла, текстовые строчки внутри файла, единичное действие программы в рамках операционной системы или целая цепочка таких действий. Аналитический компонент — это система принятия решения. Это алгоритм, который анализирует имеющиеся в его распоряжении данные и выносит о них некое суждение. В соответствии с этим суждением антивирус (либо другое защитное ПО) предпринимает установленные его политикой безопасности действия: оповещает пользователя, запрашивает у него дальнейшие указания, помещает файл в карантин, блокирует несанкционированное действие программы и т.д. Для примера рассмотрим классическую защиту от вредоносных программ, основанную на сигнатурном детектировании. В ней в качестве технического компонента выступает система получения информации о файловой системе, файлах и их цифровом содержимом, а в качестве аналитического — простая операция сравнения байтовых последовательностей. То есть, если говорить упрощенно, на входе у аналитического компонента — код файла, а на выходе — решение, является ли данный файл вредоносным. В рамках описанной модели любая система защиты может быть представлена как «комплексное число» — как связка двух независимых объектов: технического и аналитического компонентов определенного типа. Анализируя технологии таким образом, легко увидеть их соотношение, их принципиальные плюсы и минусы. И, в частности, с помощью этой модели удобно разрешить путаницу в определениях технологий. Например, ниже будет показано, что «эвристика» как способ принятия решений — лишь разновидность аналитического компонента, а не самостоятельная технология. А HIPS (Host Intrusion Prevention System) — лишь разновидность технического компонента, способ сбора данных. Как следствие, данные термины, во-первых, формально не противоречат друг другу, а во-вторых, не полностью характеризуют технологию, в описании которой встречаются: говоря об эвристике, мы не уточняем, по каким именно данным производится эвристический анализ, а говоря о HIPS-системе — ничего не знаем о том, по какому принципу в ней выносится вердикт. Более подробно эти технологии будут обсуждаться в соответствующих разделах, а пока рассмотрим сами принципы, на которых основана любая технология поиска вредоносного кода: технические — как способы сбора информации, и аналитические — как способы ее обработки. Диаграмма. Суммарная схема технологий обнаружения вредоносного кода

Технический компонент

Техническая составляющая системы обнаружения вредоносных программ обеспечивает сбор данных, которые будут использоваться для анализа ситуации. Вредоносная программа — это, с одной стороны, файл с определенным содержимым, с другой — совокупность действий, производимых в операционной системе, с третьей — совокупность конечных эффектов в операционной системе. Поэтому и идентификация программы может быть произведена на разных уровнях: по цепочкам байт, по действиям, по влиянию на операционную систему и т.д. Обобщая, можно выделить следующие способы сбора данных для выявления вредоносных программ: Работа с файлом как с

массивом байтов. Эмуляция1 кода программы. Запуск программы в «песочнице» (sandbox2) (а также использование близких по смыслу технологий виртуализации). Мониторинг системных событий. Поиск системных аномалий. Способы перечислены в соответствии с повышением уровня абстракции при работе с кодом. Под уровнем абстракции в данном случае подразумевается то, под каким углом зрения рассматривается исполняемая программа: как первичный цифровой объект (набор байт), как поведение (более абстрактное следствие из набора байт) или как совокупность эффектов в операционной системе (более абстрактное следствие из поведения). Примерно по этому же вектору шло и развитие антивирусных технологий: работа с файлами, работа с событиями через файл, работа с файлом через события, работа с самой средой, — поэтому приведенный список естественным образом оказался выстроен и по хронологии. Подчеркнем, что приведенные способы — не столько обособленные технологии, сколько условные этапы непрерывного процесса развития технологий сбора данных для обнаружения вредоносных программ. Технологии развиваются и переходят друг в друга более или менее постепенно; например, эмуляция может оказаться ближе к простой работе с файлом, если данная ее реализация лишь частично преобразует файл как набор байт, или же к «песочнице», если речь идет о полной виртуализации системных функций. Рассмотрим эти способы подробно.

Считывание файлов

Самые первые из появившихся антивирусов основывались на анализе кода файлов как наборов байт. Впрочем, анализом это назвать сложно — речь идет о простом сравнении байтовой последовательности с известной сигнатурой. Но нас сейчас интересует техническая сторона описанной технологии, а именно: в процессе поиска вредоносных программ данные, передаваемые в компонент принятия решений, извлекаются из файлов и представляют собой определенным образом упорядоченные массивы байт. Характерная особенность этого способа в том, что антивирус работает только с исходным байтовым кодом программы, не затрагивая ее поведение. Несмотря на то, что способ «архаический», он совершенно не устарел и так или иначе используется во всех современных антивирусах — но уже не как единственный и даже не как основной, а лишь как один из нескольких.

Эмуляция Технология

Эмуляции является промежуточной ступенью между обработкой программы как набора байт и обработкой программы как определенной последовательности действий. Эмулятор разбирает байтовый код программы на команды и каждую команду запускает в виртуальной копии компьютера. Это позволяет средству защиты наблюдать за поведением программы, не ставя под угрозу операционную систему и данные пользователя, что неизбежно произошло бы при исполнении программы в реальной среде. Эмулятор является промежуточной ступенью абстрагирования при работе с программой. Поэтому характерную особенность эмулятора можно грубо сформулировать примерно так: эмулятор все еще работает с файлом, но анализируются уже, фактически, события. Эмуляторы используются во многих (возможно, во всех) крупных антивирусах, главным образом как дополнение к основному, более низкоуровневому файловому движку, либо как «страховка» для более высокоуровневых движков (таких как «песочница», системный мониторинг).

Виртуализация: «песочница»

Виртуализация в том ее виде, в котором она используется в «песочницах», представляет собой логическое продолжение эмуляции. А именно: «песочница» уже работает с исполняющейся в реальной среде программой, но все еще ее контролирует. Суть такой виртуализации хорошо отражена в названии технологии — «песочница». В обычной жизни песочница — это некое огороженное пространство, в пределах которого ребенок может спокойно играть. Если провести аналогию и под реальным миром понимать операционную систему, а под шаловливым ребенком — вредоносную программу, то в роли ограждения будет выступать некий набор правил взаимодействия с операционной системой. Такими правилами может быть запрет на модификацию реестра ОС, ограничение работы с файловой системой посредством ее частичной эмуляции. Например, программе, запущенной в «песочнице», может быть «подсунута» виртуальная копия системного реестра — для того, чтобы изменения, вносимые программой в реестр, не могли повлиять на работу операционной системы. Таким образом могут виртуализироваться любые точки соприкосновения программы со средой: файловая система, реестр. Грань между эмуляцией и виртуализацией тонка, но ощутима. Первая технология предоставляет среду для исполнения программы (и, таким образом, в процессе работы «содержит» программу и полностью управляет ею). Во втором случае в качестве среды уже выступает сама операционная система, а технология лишь контролирует взаимодействие между операционной системой и программой, в отличие от предыдущего случая находясь с последней на равных. Таким образом, средство защиты, основанное на виртуализации описанного типа, работает уже не с файлом, но с поведением программы — однако все еще не с системой. Механизм типа «песочница», так же как и эмулятор, не особенно активно используется в антивирусах — главным образом потому, что в программной реализации он требует значительного объема ресурсов. Антивирусы, имеющие в своем составе «песочницу», легко идентифицируются по существенной временной задержке между запуском программы и началом ее исполнения (либо — в случае успешной идентификации вредоносной программы — между ее запуском и уведомлением, полученным от антивируса, об ее успешном детектировании). Учитывая, что в настоящий момент проводятся активные исследования в области аппаратной виртуализации, ситуация вскоре может поменяться. Пока что движок типа «песочница» используется лишь в нескольких антивирусах.

Мониторинг системных событий

Мониторинг системных событий является более «абстрактным» способом сбора информации для выявления вредоносных программ. Если эмулятор или «песочница» наблюдают за каждой программой в отдельности, то монитор наблюдает за всеми программами сразу посредством регистрации всех событий, происходящих в операционной системе и порожденных работающими программами. Технически такой способ сбора информации реализуется посредством перехватов функций операционной системы. Таким образом, перехватив вызов некой системной функции, механизм-перехватчик получает информацию о том, что определенная программа совершает определенное действие в системе. На протяжении своей работы монитор собирает статистику таких действий и передает ее в аналитический компонент для обработки. Этот технологический принцип наиболее активно развивается в настоящее время. Он используется в качестве одного из компонентов в нескольких крупных антивирусах, и в качестве основы — в отдельных утилитах, специализирующихся на мониторинге системы (их называют «HIPS-утилитами», «HIPS’ами» — это Prevx, CyberHawk и ряд других).

Однако в свете того, что любую защиту можно обойти, данный способ поиска вредоносных программ представляется не самым перспективным, поскольку при запуске программы в реальной среде риск существенно снижает эффективность защиты.

Поиск системных аномалий

Это наиболее абстрактный способ сбора информации о предположительно зараженной системе. Я упоминаю его здесь в первую очередь как логическое продолжение и предел абстракции в приведенном списке технологий. Данный метод основан на следующих положениях: операционная среда вместе со всеми выполняющимися в ней программами — это интегральная система; ей присуще некое «системное состояние»; если в среде исполняется вредоносный код, то состояние системы является «нездоровым» и отличается от состояния «здоровой» системы, в которой вредоносного кода нет. Исходя из этих положений мы можем судить о состоянии системы (и, следовательно, о возможном присутствии в ней вредоносных программ), сравнивая его с эталоном (за эталон принимается «здоровое» состояние системы) или анализируя совокупность отдельных ее параметров. Для эффективного обнаружения вредоносного кода методом анализа аномалий необходима достаточно сложная аналитическая система — наподобие экспертной системы или нейронной сети. Возникает много вопросов: как определить «здоровое состояние», чем оно отличается от «нездорового», какие дискретные параметры можно отслеживать и как их анализировать? По причине такой сложности в настоящее время этот способ разработан мало. Зачатки его можно обнаружить в некоторых антируткит-утилитах, где он реализован на уровне сравнения с определенным срезом системы, взятым за эталон (устаревшие утилиты PatchFinder, Kaspersky Inspector), либо отдельных ее параметров (GMER, Rootkit Unhooker).

Забавная метафора Аналогию с ребенком, приведенную в разделе «Песочница», можно продолжить таким образом: эмулятор похож на няньку, которая непрерывно следит за ребенком, чтобы он не натворил лишнего, мониторинг системных событий — на воспитателя, надзирающего за целой группой детишек, поиск системных аномалий — на предоставление детям полной свободы, ограниченной лишь проверкой оценок в их дневнике. В таком случае байтовый анализ файла — это только планирование ребенка, точнее, поиск признаков шаловливости в характере предполагаемого родителя. Технологии растут и развиваются.

Практическое занятие № 12

Тема 7: Исследование кода вредоносных программ

2. Способы обнаружения вредоносного кода.

Цель: Научиться определять коды вредоносных программ.

Средства обучения: тетради для выполнения практических занятий, Интернет-ресурсы.

Аналитический компонент

Сложность алгоритма принятия решений может быть совершенно любой. Очень условно можно разделить аналитические системы антивирусов на три категории, между которыми может быть множество промежуточных вариантов.

Простое сравнение. Вердикт выносится по результатам сравнения единственного объекта с имеющимся образцом. Результат сравнения бинарный («да» или «нет»). Пример: идентификация вредоносного кода по строго определенной последовательности байт. Другой пример, более высокоуровневый: идентификация подозрительного поведения программы по единственному совершенному ею действию (такому как запись в критичный раздел реестра или в папку автозагрузки).

Сложное сравнение. Вердикт выносится по результатам сравнения одного или нескольких объектов с соответствующими образцами. Шаблоны для сравнения могут быть гибкими, а результат сравнения — вероятностным. Пример: идентификация вредоносного кода по одной из нескольких байтовых сигнатур, каждая из которых задана нежестко (например, так, что отдельные байты не определены). Другой пример, более высокоуровневый: идентификация вредоносного кода по нескольким используемым им и вызываемым непоследовательно API-функциям с определенными параметрами.

Экспертная система. Вердикт выносится в результате тонкого анализа данных. Это может быть система, содержащая в себе зачатки искусственного интеллекта. Пример: идентификация вредоносного кода не по жестко заданному набору параметров, но по результатам многосторонней оценки всей совокупности параметров в целом, с присвоением каждому из событий веса «потенциальной вредоносности» и расчетом общего результата. 3. Реальные названия технологий — что к чему Рассмотрим теперь, какие именно алгоритмы лежат в основе конкретных технологий поиска вредоносных программ. Обычно производитель, разработав новую технологию, дает ей совершенно новое, уникальное имя (примеры: «Проактивная защита» в Kaspersky Antivirus, TruPrevent от Panda, DeepGuard от F-Secure). Это очень правильный подход, поскольку он позволяет избежать автоматического восприятия технологии в узких рамках некого термина-штампа. Тем не менее, использование штампов — таких как «эвристика», «эмуляция», «песочница», «поведенческий блокиратор» — неизбежно при любых попытках доступно и не вдаваясь в технические подробности охарактеризовать технологию. Тут-то и начинается терминологическая путаница. За терминами жестко не закреплены значения (а в идеале термин должен быть однозначен, на тот он и термин). Один человек так понимает этот термин, другой — иначе. Вдобавок, значения, которые вкладывают в термины авторы так называемых «доступных описаний», часто очень существенно отличаются от значений, принятых в среде профессионалов. Иначе чем объяснить тот факт, что описание технологии на сайте производителя может изобиловать терминами, но при этом ничего не сообщать о сути технологии или сообщать нечто, не соответствующее ей. Например, некоторые производители антивирусных систем характеризуют свои продукты как оснащенные HIPS, «проактивной технологией» или «несигнатурной технологией». Пользователь, который понимает термин HIPS как мониторинг системных событий и их анализ на наличие вредоносного кода, может оказаться обманутым. В действительности под этими характеристиками может скрываться все что угодно — например, движок типа «эмулятор», оснащенный аналитической системой типа «эвристика» (определение см. ниже). Еще чаще встречается ситуация, когда защита характеризуется как «эвристическая» без каких-либо уточнений. Подчеркнем, что речь не идет об умышленном обмане пользователя производителем — вероятнее всего, составитель описания просто сам запутался в терминах. Речь идет лишь о том, что описание технологии, составленное для конечного пользователя, может не отражать ее суть, и что опираться на него при ответственном подходе к выбору защиты следует с осторожностью. Рассмотрим наиболее распространенные термины в области антивирусных технологий (см. диаграмму). Меньше всего разночтений у термина «сигнатурное детектирование»: так или иначе, с технической стороны оно подразумевает работу с байтовым кодом файлов, а с аналитической — примитивный способ обработки данных, обычно — простое сравнение. Это самая старая технология, но она же и наиболее надежная — поэтому, несмотря на большие производственные издержки, связанные с пополнением базы, активно используется по сей день во всех антивирусах. Если в характеристике используется название технического компонента из приведенного выше списка — «эмулятор» или «песочница» — это также вызывает минимум толкований. При этом аналитический компонент такой технологии может быть представлен алгоритмом любой степени сложности, от простого сравнения до экспертной системы. Термин «эвристика» уже немного туманен. По определению словаря Ожегова-Шведовой, «эвристика — совокупность исследовательских методов, способствующих обнаружению ранее неизвестного». Эвристика — это в первую очередь тип аналитического компонента защиты, а не определенная технология. Вне конкретной темы, в контексте решения задач, он приблизительно соответствует «нечеткому» способу решения нечетко поставленной задачи. На заре антивирусных технологий, когда и был впервые задействован термин «эвристика», он подразумевал вполне определенную технологию — идентификацию вируса по нескольким гибко заданным байтовым шаблонам, т.е. систему из технического

компонента типа «работа с файлами» и аналитического — «сложное сравнение». Сейчас термин «эвристика» обычно используется в более общем значении «технологии поиска неизвестных вредоносных программ». Иными словами, говоря об «эвристическом детектировании», производитель подразумевает некую систему защиты, аналитический компонент которой работает по принципу нечеткого поиска решения (что может соответствовать типу аналитического компонента «сложный анализ» или «экспертная система», см. диаграмму). При этом технологическая основа защиты, способ сбора информации для последующего анализа, может быть какой угодно — от работы с файлами до работы с событиями или состоянием операционной системы. Еще меньше определенности с такими названиями, как «поведенческое детектирование», «проактивное детектирование». Они могут подразумевать широкий спектр технологий — от эвристики до мониторинга системных событий. Термин HIPS используется в описаниях антивирусных технологий очень часто, и не всегда оправданно. Несмотря на то что расшифровка аббревиатуры (Host Intrusion Prevention System) никак не отражает суть технологии, применительно к антивирусной защите технология четко определена: HIPS — это защита, технически основанная на мониторинге системных событий. При этом аналитический компонент защиты может быть каким угодно — от пресечения единичных подозрительных событий до сложного анализа цепочек программных действий. Таким образом, под определением «HIPS» в описании антивируса может скрываться, например, примитивная защита нескольких ключей реестра, либо система уведомлений о попытках доступа к определенным директориям, либо более сложная система анализа поведения программ, либо какая-то другая технология, в основе которой лежит мониторинг системных событий.

Плюсы и минусы способов обнаружения вредоносного кода. Если рассматривать технологии защиты от вредоносных программ не по отдельности, а обобщенно, с точки зрения представленной модели, то вырисовывается следующая картина. Технический компонент технологии отвечает в основном за такие ее характеристики, как нагрузка на систему (и как следствие — ее быстродействие), безопасность и защищенность. Нагрузка на систему — это доля процессорного времени и оперативной памяти, непрерывно или периодически задействованных в обеспечении защиты и ограничивающих быстродействие системы. Эмуляция выполняется медленно, вне зависимости от реализации: на каждую проэмулированную инструкцию приходится несколько инструкций искусственной среды. То же можно сказать и про виртуализацию. Мониторинг системных событий также безусловно равномерно тормозит всю систему, но степень этой нагрузки зависит от реализации. В случае с файловым детектированием и поиском системных аномалий степень нагрузки всецело зависит от реализации. Под «безопасностью» подразумевается степень риска, которому подвергается операционная система и данные пользователя в процессе идентификации потенциально вредоносного кода. Такой риск существует всегда, когда вредоносный код исполняется реально, в операционной системе. Для систем мониторинга событий такое реальное исполнение кода архитектурно обусловлено, в то время как эмуляция и файловое сканирование могут обнаружить вредоносный код еще до того, как он начал исполняться. Защищенность. Этот параметр отражает уязвимость технологии, то, насколько вредоносный код может затруднить процесс идентификации себя. Противостоять файловому детектированию очень легко: достаточно хорошо упаковать файл, либо сделать его полиморфным, либо воспользоваться руткит-технологией для сокрытия файла. Противостоять эмуляции немного сложнее, но также возможно — для этого используются многочисленные трюки3, встроенные в код вредоносной программы. Но скрыться от системного мониторинга программе уже сложно — по той причине, что практически невозможно скрыть поведение. Подводя итог: в среднем, чем менее абстрактна защита, тем она безопаснее, но и тем проще ее обойти. Аналитический аспект технологии отвечает за такие характеристики, как проактивность (и зависящую от нее необходимую частоту обновления антивируса), процент ложных срабатываний и нагрузка на пользователя. Под проактивностью подразумевается способность технологии обнаруживать новые, еще не попадавшие в руки специалистов вредоносные программы. К примеру, простейший тип анализа («простое сравнение») соответствует наиболее далеким от проактивности технологиям, таким как сигнатурное детектирование: при помощи таких технологий могут быть обнаружены лишь известные вредоносные программы. По мере возрастания сложности аналитической системы, возрастает и ее проактивность. С проактивностью непосредственно связана и такая характеристика защитной системы, как необходимая частота обновлений. Например, базы сигнатур нужно часто обновлять, в то время как более сложные эвристические системы остаются адекватными текущей ситуации более длительный срок, а экспертные аналитические системы могут успешно функционировать без обновлений месяцами. Процент ложных срабатываний так же непосредственно связан со сложностью технологии анализа. Если вредоносный код идентифицируется жестко заданной сигнатурой или последовательностью действий — при условии достаточной длины сигнатуры (байтовой, поведенческой или какой-то еще), такая идентификация однозначна: сигнатура идентифицирует только определенную вредоносную программу и не подходит для других. Но чем больше «жертв» старается захватить идентификационный алгоритм, тем более нечетким он становится, вследствие чего способен захватить больше безвредных программ. Под нагрузкой на пользователя подразумевается степень его участия в формировании политики защиты — правил, исключений, белых и черных списков — и участия в процессе вынесения вердикта — подтверждение или опровержение «подозрений» аналитической системы. Нагрузка на пользователя зависит от реализации, но общее правило таково, что чем дальше анализ от примитивного сравнения, тем больше

случается ложных срабатываний — которые нужно как-то корректировать. Для этого и необходимо участие пользователя. Подводя итог: чем сложнее аналитическая система, тем она могущественнее, но и тем выше процент ложных срабатываний. Последние компенсируются взаимодействием с пользователем. Теперь, если рассмотреть какую-либо технологию через призму этой модели, легко теоретически оценить ее преимущества и недостатки. Возьмем, например, эмулятор со сложным аналитическим компонентом. Это защита очень безопасная (поскольку не требуется запуск проверяемого файла), но «пропускающая» некоторый процент вредоносных программ вследствие использования в них антиэмуляционных трюков, либо вследствие неизбежных недоработок в реализации самого эмулятора. Такая защита обладает высоким потенциалом и будет при грамотной реализации качественно идентифицировать большой процент неизвестных вредоносных программ, но неизбежно медленно.

Как выбрать систему несигнатурной защиты?

В настоящий момент большинство решений в области компьютерной безопасности реализуются как комплекс нескольких технологий. В классических антивирусах сигнатурное детектирование обычно используется в связке с той или иной реализацией мониторинга системных событий, эмулятора, песочницы. Как сориентироваться в спецификациях и выбрать систему защиты от вредоносного кода, оптимально отвечающую потребностям конкретного пользователя? Прежде всего, следует помнить, что не существует ни универсального, ни «самого лучшего» решения. У каждой технологии есть свои плюсы и минусы. Например, мониторинг событий в системе постоянно занимает процессорное время, но его труднее всего обмануть; процессу эмуляции можно помешать использованием в коде определенных команд, зато при ее использовании обнаружение вредоносного кода выполняется в упреждающем режиме, система остается незатронута. Другой пример: простые правила принятия решений требуют от пользователя слишком активного участия в процессе, порождая много вопросов к пользователю, а сложные и «тихие» чреваты ложными срабатываниями. Выбор технологии — это выбор золотой середины с учетом конкретных потребностей и обстоятельств. Например, тому, кто работает в уязвимых условиях («непропатченная» система, отсутствие запретов на использование расширений браузера, скриптов и т.п.), сильно беспокоится за свою безопасность и имеет достаточно ресурсов, больше всего подойдет система типа «песочница», с качественным аналитическим компонентом. Такая система обеспечивает максимум безопасности, но в сегодняшней реализации задействует много оперативной памяти и процессорного времени — что может проявляться в «торможении» ОС. Специалисту, желающему контролировать критичные системные события, и заодно оградить себя от неизвестных вредоносных программ, подойдет система мониторинга реального времени. Такая система равномерно, но не существенно загружает операционную систему и требует специального участия в создании правил и исключений. А пользователю, который ограничен в ресурсах либо не хочет загружать свою машину постоянным мониторингом, а разум — созданием правил, подойдет более простая эвристика. В конце концов, за качество детектирования неизвестных вредоносных программ отвечает не какая-то одна составляющая защитной системы, а вся система в целом; более простой технологический способ можно компенсировать более умелым принятием решений. Системы несигнатурного обнаружения ранее неизвестного кода распадаются на две категории. Первая — это самостоятельные HIPS-системы, такие как уже приведенные в качестве примера Prevx или Cyberhawk. Вторая — крупные антивирусы, эволюционировавшие в поиске большей эффективности до несигнатурных технологий. Преимущества той или иной категории очевидны: узкая специализация, в рамках которой можно неограниченно совершенствовать качество, — в первом случае, и значительный опыт разносторонней борьбы с вредоносными программами — во втором. В выборе того или иного продукта рекомендуется руководствоваться в первую очередь результатами независимых тестов, а также отзывами пользователей, которым вы доверяете.

Основная литература:

1. Алдан А. Введение в генерацию программного кода [Электронный ресурс]/ Алдан А.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 184 с.— Режим доступа: http://www.iprbookshop.ru/57370.html.— ЭБС «IPRbooks»
2. Основы проектирования компонентов автоматизированных систем: учебное пособие/ Т. В. Волкова Оренбург: ОГУ, 2016, 226 с. То же [Электронный ресурс]. - URL: http://biblioclub.ru/index.php?page=book&id=471129
3. Трубилин, А. И. Управление проектами : учебное пособие / А. И. Трубилин, В. И. Гайдук, А. В. Кондрашова. — Саратов : Ай Пи Ар Медиа, 2019. — 163 с. — ISBN 978-5-4497-0069-8. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/86340.html. — Режим доступа: для авторизир. пользователей

Дополнительная литература

1. Рудаков А. Технология разработки программных продуктов: учебник. / Рудаков А. - Изд.Academia. Среднее профессиональное образование. 2013 г. 208 стр.

Интернет-ресурсы:

1. Северо-Западный Заочный Государственный Технический Университет [Электронный ресурс] - Режим доступа: http:// www.nwpi.ru.
2. Интернет Университет Информационных технологий [Электронный ресурс]