**Управление проектами**

02.09.22

**Методы и средства измерений характеристик программ.  
Аппаратные измерительные мониторы.**

На ряду с аналитическими методами для исследования и оценки параметров программ активно используются измерительные методы. Главной стороной этих методов является их высокая достоверность. Измерительные методы имеют следующее **назначение:**

1. Измерение параметров потребления программами ресурсов вычислительной системы с целью устранения дефектов производительности.
2. Предварительное измерение параметров системы для имитационных или аналитических моделей программ перед их следующим использованием.
3. Проверка адекватности имитационных или аналитических моделей и методов расчёта характеристик программ по результатам моделирования.

**Необходимые условия применения измерительных методов:**

1. Наличие готовой программы, подлежащей измерительному исследованию;
2. Наличие реально вычислительной системы для прогона программы;
3. Наличие аппаратных или программных средств проведения измерений;
4. Создание условий снижения искажений, вносимых в функционирование системы в процессе проведения измерений;

**Схема измерений включает в себя следующие компоненты:**

1. Исследуемую ВС с установленными программами;
2. Средства регистрации параметров потребляемых ресурсов при выполнении данной рабочей нагрузки;
3. Архив для хранения результатов многочисленных измерений;
4. Результаты измерений обрабатываются некоторой ВС (отдельная ВС или та же, на которой снимались измерения, но после выполнения сеанса измерений);
5. Рабочая нагрузка – одна или несколько программ, или наборов данных для получения статистики проводимых измерений.

**Процесс измерения подготовки и проведения измерений включает следующие три этапа:**

1. Выбор рабочей нагрузки, представленной с точки зрения исследования параметров выполнения программы на исследуемой системе;
2. Выбор (разработка) средств регистрации параметров потребления ресурсов системы;
3. Выбор (разработка) алгоритмов расчётов характеристик программ по результатам измерений.

**Классификация измерительных мониторов**

Измерительные мониторы

Уникальные

Серийные

С фиксированной программой измерений

С изменяемой программой измерений

Специальные

Универсальные

Встроенные

Автономные

Встроенные

Автономные

Аппаратные (АИМ)

Гибридные мониторы

Программные (ПИМ)

**Аппаратные измерительные мониторы (АИМ)**

Встроенные АИМ – включаются в аппаратуру системы заводом изготовителем, как правило, для выполнения тестовых измерений в фиксированном наборе внутренних точек устройств системы.

Основное назначение встроенного АИМ – для проверки, контроля и настройки ВС. Но т.к. в его составе могут находиться триггеры состояний схемы, счётчики и некоторые другие устройства, то они могут использоваться и для измерения параметров программ.

Автономные АИМ подключаются к измерительным точкам системы через специальные разъёмы и могут задавать любые точки доступные для измерений.

Автономные АИМ – наиболее общий случай.

**Статические и динамическое тестирование**

Для полной убедительности в корректности программ одних формализованных правил недостаточно.

1. Программные спецификации – относятся к функциональным эталонам и в основном обеспечивают проверку корректности программ в статике.

2. Тесты.

В зависимости от стадии и характера проверки тесты делятся на статические и динамические.

**Статическое тестирование** – ручное тестирование программы, начиная со стадии формирования требований к программе. На стадии кодирования при статическом тестировании некоторую часть маршрутов исполнения тестируют вручную.

**Динамическое тестирование** подразумевает достаточно полную структурную и функциональную проверку выполнения программы.

**Способы формирования эталонов тестирования:**

1. Использование аналитических выражений. Этот способ особенно подходит при детерминированном тестировании, так как имеется возможность сравнить результаты тестирования с ожидаемыми результатами. Имеются ограничения в использовании этого метода, если неизвестны или отсутствуют аналитические выражения, связывающие входные данные и результаты; иногда требуется использовать много допущений.
2. Использование моделирования на ЭВМ. Способ является универсальным. При этом ряд данных моделируется другим способом и по другим алгоритмам, нежели испытываемая программа и на других ЭВМ. Причём наборы входных данных создаются по случайным законам, что обеспечивает высокую гибкость этого способа.
3. Использование результатов испытаний предшествующих вариантов программ. При этом используется ранее накопленный опыт испытателя или других исследователей, выраженный в экспертных оценках ожидаемых результатов.

Степень достоверности проверки корректности программ при использовании этих методов убывает по номерам способов формирования эталонов.

В 1-ом случае обеспечивается 100% гарантия корректности программ, в третьем случае такой уверенности нет, но мы можем убедиться в том, что программа работает так же или иначе, чем аналогичный вариант. Менее достоверные тесты приходится использовать из-за недостаточности сил и средств.

05.09.2022

**Корректность программ**

**Под корректностью** программы понимают её соответствие некоторому эталону или совокупности формализованных эталонных правил и характеристик.

**Программная спецификация** – наиболее полный эталон корректности программ. Её особенностью является задание требований поведения программы для допустимых наборов входных данных. Поэтому корректная программа может неправильно работать или даже сбиваться на недопустимых наборах входных данных. Свойством устойчивости к недопустимым наборам входных данных обладает надёжная программа – в этом заключается разница между надёжной и корректной программами.

Требования к корректности делятся в зависимости от двух типов критериев качества:

- Для функциональных критериев они определяются предметной областью и функциями выполняемой программы;

- Для конструктивных критериев они определяются общими для всех программ свойствами.

В зависимости от проверяемых компонентов программ различают следующие виды их корректности, показанные на рисунке:

- структурных

- инженерные значений

Корректность данных

Корректность синтаксических программ

- структурных

- функциональных

- структуры межмодульных связей

- функциональные

- детерминированные

Корректность комплексов программ

Корректность программных модулей

Корректность текстовых программ

Корректность программ

1. Корректность текстов программ имеет только конструктивную составляющую; благодаря жёстким правилам языков программирования синтаксическая и семантическая корректность программ проверяется на этапе трансляции программы, и прошедшая трансляцию программа является корректной с этой точки зрения.

2. Корректность программных модулей имеет и конструктивную и функциональную составляющие;

**Конструктивная составляющая** определяется правилами построения структуры программных модулей, задаваемыми в технологии и языке программирования.

**Функциональная составляющая** корректности модулей зависит от предметной области и функциональных спецификаций программы.

Функциональная составляющая корректности может проверяться в различных условиях:

**Детерминированная** – для фиксированных наборов данных должны быть получены конкретные значения результатов.

**Стохастическая** – входные данные задаются случайными величинами с известными законами распределения и результаты также должны быть случайными величинами с требуемыми законами распределения и заданными корреляционными связями между входными и выходными данными.

**Динамическая** – характерная для систем реального времени и определяется согласованием во времени порядка поступления входных данных и порядка выдачи результата выполнения программы.

В общем случае функциональные спецификации программы определяют и функциональные требования к программе, и характеристики, с которыми они должны обеспечиваться, как это показано на рисунке:

Выполняемые функции

Потребление времени и памяти

Результаты

Функциональные спецификации программы

Точность

Исходные данные

3. Корректность данных имеет конструктивную и функциональную составляющие.

Структурная корректность данных относится к конструктивной составляющей и предполагает правильность построение структурированных данных в программе: массивов стеков, очередей и т.п. Функциональная корректность данных определяется диапазонами изменения их значений и соответствием типов полей структур типам значений данных.

4. Корректность комплексов программ также имеет конструктивную и функциональную составляющие: конструктивная составляющая определяется корректностью структуры межмодульных связей по управлению и данным, определяемых в интерфейсных требованиях к программе; функциональной корректность комплекса программ определяется так же, как и функциональная корректность модулей.

**Эталоны и методы проверки корректности**

**Эталоны для проверки корректности программ** могут использоваться в следующих трёх формах:

1. Формализованные правила;
2. Программные спецификации;
3. Тесты.

**Формализованные правила** имеют достаточно неопределённостей, т.к. определяются двумя видами требований:

* Требования стандартов (общероссийских и стандартов предприятий);
* Требования языков и технологий программирования.

Для полной убедительности корректности программ одних формализованных правил недостаточно.

3. Тесты

* Статические
* Динамические
  + Детерминированные
  + Статистические

Статическое и динамическое тестирование

* Контроль полноты спецификаций
* Верификация программы
* На модули
* На данные
* На взаимодействие модулей

2. Программные спецификации

1.

* Синтаксический контроль
* Семантический контроль
* Структурный контроль
* Описание алгоритма;
* Описание данных;
* Структуры модулей;
* Структуры компонентов.

Эталоны

Методы проверки корректности программ

Средства проверки корректности программ

**Программные спецификации** относятся к функциональным эталонам и в основном обеспечивают проверку корректности программ в статике.

08.09.2022

**Аналитическая проверка корректности программ. Верификация программ.**

**Корректность** является статическим свойством программы, т.к. не зависит от времени и отражает специфику ошибок разработки программ.

**Виды проверки корректности:**

1. Валидация - установление соответствия между тем, что делает программа и тем, что нужно заказчику;
2. Верификация - установление соответствия между программой и её спецификацией.

Определение верификации симметрично в виду относительного характера свойства корректности: если программа не соответствует спецификации, то либо программа, либо спецификация, либо оба этих объекта содержат ошибку.

**Понятие метрики. Направления применения метрик. Метрические шкалы. Метрики сложности. Метрики стилистики.**

**Качество ПО** – совокупность свойств, определяющих полезность изделия (ПО) для пользователей в соответствии с функциональным назначением и предъявленными требованиями.

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

**Критерии качества** – численный показатель, характеризующий степень, в которой программе присуще оцениваемое свойство.

Критерии качества включают:

* Экономичность;
* Документированность;
* Гибкость;
* Модульность;
* Обоснованность;
* Надёжность;
* Лёгкость сопровождения;
* Тестируемость;
* Точность;
* Модифицируемость…

Критерий должен:

1. Численно характеризовать основную целевую функцию программы;
2. Обеспечивать определение затрат для достижения необходимого уровня качества и степени влияния на показатель качества внешних факторов;
3. Быть простым, хорошо измеримым и иметь малую дисперсию.

Метрики используются для измерения характеристик и критериев качества.

**Метрика качества программ** – это система измерений качества ПО. Эти измерения проводятся на уровне критериев качества программ или на уровне отдельных характеристик качества.

- В первом случае, система измерений позволяет сравнивать программы по качеству, при этом сами измерения не могут быть проведены без субъективных оценок свойств программы.

- Во втором случае, измерения характеристик можно выполнить объективно и достоверно, но оценка будет связана с субъективной интерпретацией оценок.

**Направления исследования метрик ПО:**

1. Поиск метрик самого ПО;
2. Использование метрик оценки условий разработки ПО.

По виду информации, получаемой при оценке качества, метрики бывают:

1. Оценивающие отклонение от нормы характеристик исходного кода;
2. Позволяющие прогнозировать качество разрабатываемого ПО;
3. Метрики, по которым принимаются решения о соответствии конечного ПО заданным требованиям.

**Основные направления применения метрик:**

1. Оценки топологической и информационной сложности программ;
2. Оценки надёжности Программных Систем для прогнозирования отказовых ситуаций;
3. Оценки производительности ПО и повышение эффективности путём выявления ошибок проектирования;
4. Оценки уровня языковых средств;
5. Оценки трудности восприятия и понимания программных текстов;
6. Оценки производительности труда программистов.

**Метрические шкалы**

**Номинальной шкале** соответствуют метрики, классифицирующие программы на типы по признаку наличия или отсутствия какой-либо характеристики без учёта градаций.

**Порядковой шкале** соответствуют метрики, позволяющие ранжировать некоторые характеристики путём сравнения с опорными значениями.

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

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

09.09.2022

**Метрики сложности программ**

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

* Метрики размера программ;
* Метрики сложности потока управления программ;
* 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 её создания.

Непосредственное измерение размера программы, несмотря на свою простоту, даёт хорошие результаты. Конечно, оценка размера программы недостаточна для принятия решения о её сложности, но вполне применима для классификации программ, существенно различающихся объёмами. 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.

МЕТРИКИ СЛОЖНОСТИ ПОТОКА УПРАВЛЕНИЯ ПРОГРАММ.

Вторая наиболее представительная группа оценок сложности программ - метрики сложности потока управления программ. Как правило, с помощью этих оценок оперируют либо плотностью управляющих переходов внутри программ, либо взаимосвязями этих переходов.

И в том и в другом случае стало традиционным представление программ в виде управляющего ориентированного графа G=(V,E), где V - вершины, соответствующие операторам, а E - дуги, соответствующие переходам.

МЕТРИКА МАККЕЙБА.

Впервые графическое представление программ было предложено Маккейбом. Основной метрикой сложности он предлагает считать цикломатическую сложность графа программы, или, как ее еще называют, цикломатическое число Маккейба, характеризующее трудоемкость тестирования программы.

Для вычисления цикломатического числа Маккейба Z(G) применяется формула

Z(G)=e-v+2p,

где e - число дуг ориентированного графа G;

v - число вершин;

p - число компонентов связности графа.

Число компонентов связности графа можно рассматривать как количество дуг, которые необходимо добавить для преобразования графа в сильно связный. Сильно связным называется граф, любые две вершины которого взаимно достижимы. Для графов корректных программ, т. е. графов, не имеющих недостижимых от точки входа участков и "висячих" точек входа и выхода, сильно связный граф, как правило, получается путем замыкания дугой вершины, обозначающей конец программы, на вершину, обозначающую точку входа в эту программу.

По сути, Z(G) определяет число линейно независимых контуров в Сильно связном графе. Иначе говоря, цикломатическое число Маккейба показывает требуемое количество проходов для покрытия всех контуров сильно связного графа или количество тестовых прогонов программы, необходимых для исчерпывающего тестирования по критерию "работает каждая ветвь".

Для программы, граф которой изображен на рис.1, цикломатическое число при e=10, v=8, p=1 определится как Z(G)=10-8+2=4.

Цикломатическое число зависит только от количества предикатов, сложность которых при этом не учитывается. Например, имеется два оператора условия:

IF X>0

THEN X=A;

ELSE;

и

IF (X>0 & FLAG='1'B)!

(X=0 & FLAG='0'B)

THEN X=A;

ELSE;

15.09.2022

**Защита программных средств от исследования**

Изучение логики работы программы может выполняться в одном из двух режимов: статическом и динамическом.

* **Статический режим** – изучение исходного текста программ.
* **Динамический режим** изучения алгоритма программы предполагает выполнение трассировки программы.

**Трассировка** – выполнение программы на ЭВМ с использованием специальных средств, позволяющих выполнять программу в пошаговом режиме, получать доступ к регистрам, областям памяти, производить остановку программы по определённым адресам и т.д.

Для защиты программ от изучения необходимо иметь средства противодействия как дизассемблированию, так и трассировке.

**Методы противодействия дизассемблированию:**

1. Шифрование;
2. Архивация;
3. Использование самогенерирующих кодов;
4. «Обман» дизассемблера.

**Архивацию** можно рассматривать как простейшее шифрование. При этом она должна быть объединена с шифрованием. Такая комбинация позволяет получать надёжно закрытые компактные программы. Зашифрованную программу невозможно дизассемблировать без расшифрования. Зашифрование (расшифрование) – может осуществляться с помощью аппаратных средств или отдельных программ.

Другой подход к защите от дизассемблирования связан с совмещением процесса расшифрования с процессом выполнения программ. Если расшифрование всей программы осуществляется блоком, получающим управление первым, то такую программу расшифровать довольно просто.

**Сущность метода самогенерируемых кодов** заключается в том, что исполняемые коды программы получаются самой программой в процессе её выполнения. Самогенерируемые коды получаются в результате определённых действий над специально выбранным массивом данных.

**Под «обманом» дизассемблера** понимают такой стиль программирования, который вызывает нарушение правильной работы стандартной дизассемблера за счёт нестандартных приёмов использования отдельных команд, нарушения общепринятых соглашений.

**Способы обмана ассемблера:**

* Нестандартная структура программы;
* Скрытые переходы, вызовы процедур, возвраты из них и из прерываний;
* Переходы и вызовы подпрограмм по динамически изменяемым адресам;
* Модификация исполняемых кодов.

Для дезориентации дизассемблера часто используются скрытые переходы, вызовы и возвраты за счёт применения нестандартных возможностей команд.

Маскировка скрытых действий часто осуществляется с помощью стеков.

Трассировка программы осуществляется с помощью отладчиков. Покомандное выполнение осуществляется процессором при установке пошагового режима работы.

**Контрольная точка** – любое место в программе, на котором обычное выполнение программы приостанавливается и осуществляется переход в особый режим. Например, в режим покомандного выполнения.

При наличии современных средств отладки полностью исключить возможность изучения алгоритма невозможно. Основной задачей противодействия трассировке является увеличение числа и сложности ручных операций, необходимых программисту-аналитику.

**Противодействия трассировке:**

1. Изменением среды функционирования – запрет или переопределение прерываний, изменение режимов работы, состояние управляющих регистров, триггеров и т.п.;
2. Модификация кодов программы – изменяющиеся коды программ приводят к тому, что каждое выполнение процедуры выполняется по различным ветвям алгоритма;
3. «Случайные» переходы – выполняются за счёт вычисления адресов переходов. Исходные данные – характеристики среды функционирования, контрольные суммы процедур.

**Защищаемая программа включает:**

* Инициализатор;
* Зашифрованная секретная часть;
* Деструктор.

16.09.2022

**Инициализатор должен обеспечивать выполнение следующих функций:**

* Сохранение параметров операционной среды функционирования (векторов прерываний, содержимого регистров процессора и т.д.);
* Запрет всех внутренних и внешних прерываний, обработка которых не может быть запротоколирована в защищаемой программе;
* Загрузка в оперативную память и дешифрование кода секретной части программы;
* Передача управления секретной части программы.

**Секретная часть программы** предназначена для выполнения основных целевых функций программы и защищается шифрованием для предупреждения внесения в неё программной закладки.

**Деструктор после выполнения секретной части программы должен выполнить следующие действия:**

* Обнуление секретного кода программы в оперативной памяти;
* Восстановление параметров операционной системы (векторов прерываний, содержимого регистров процессора и т.д.), которые были установлены до запрета неконтролируемых прерываний;
* Выполнение операций, которые невозможно было выполнить при запрете неконтролируемых прерываний;
* Освобождение всех незадействованных ресурсов компьютера и завершение работы программы.

Для большей надёжности инициатор может быть частично зашифрован и по мере выполнения может дешифровать сам себя.

Для повышения эффективности защиты программ от исследования необходимо внесение в программу дополнительных функций безопасности, направленных на защиту от трассировки.

**К ним относятся:**

* Периодический подсчёт контрольной суммы области оперативной памяти, занимаемой защищаемым исходным кодом, сравнение текущей контрольной суммы с предварительно сформированной эталонной и принятие необходимых мер в случае несовпадения;
* Проверку количества занимаемой защищаемой программой оперативной памяти, сравнение с объёмом, к которому программа адаптирована, и принятие необходимых мер в случае несоответствия;
* Контроль времени выполнения отдельный частей программы;
* Блокировку клавиатуры на время отработки особо секретных алгоритмов.

19.09.2022

**Анализ кода вредоносных программ**

**BotNet** – группа компьютеров, заражённых одним и тем же вредоносным ПО (боты), ожидающих получением инструкций от командно-контрольного сервера, контролируемого злоумышленником (может осуществлять DDoS атаки, спам).

**RootKit** – вредоносная программа, предоставляющая злоумышленнику привилегированный доступ к системе. Скрывает своё наличие или наличие другого ПО.

**Dropper** – используется для загрузки другого вредоносного ПО.

**Анализ вредоносного программ** – это изучение или процесс определения функциональности, происхождения и потенциального воздействия образца вредоносного ПО.

**Цель анализа вредоносного ПО** – понять работу вредоносных программ и методы их обнаружения и устранения. Он включает в себя анализ подозрительного двоичного файла в безопасной среде для определения его характеристик и функциональных возможностей, чтобы можно было выстроить лучшую оборонительную стратегию для защиты сети организации.

**Компоненты защитных технологий:**

* Технический;
* Аналитический;

**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 (или 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.

**Способы сбора данных для выявления вредоносных программ:**

* Работа с файлом как с массивом байт;
* Эмуляция кода программы;
* Запуск программы в «песочнице» (sandbox2);
* Мониторинг системных событий;
* Поиск системных аномалий.

**Считывание файлов** – код формируется в упорядоченные массивы байт и анализируется.

**Эмуляция** – моделирование работы программного устройства на другом компьютере без исходного программного кода программы.

**«Песочница»** - запуск объекта в виртуальной среде и анализ его поведения.

**Мониторинг системных событий** – сбор, анализ, передача сигналов о событиях абонентам (процессам операционной системы, операторы-люди, активные правила БД).

**Поиск системных аномалий** – инструмент, помогающий выявлять нарушения, ошибки в конфигурации системы, неправильную работу ПО.

**Этапы изучения вредоносного ПО:**

1. Обратное проектирование;
2. Интерактивный анализ поведения;
3. Анализ статических свойств;
4. Полностью автоматизированный анализ.

**Обратное проектирование** (Reverse Engineering) – исследования устройства и документации с целью понять принцип его работы.

**Интерактивный анализ поведения** – взаимодействие с программами с целью их изучения.

**Анализ статических свойств** – анализ данных, которые можно собрать без необходимости запуска программы (детали заголовка, хеши, встроенные ресурсы, подписи упаковщика, метаданные).

**Полностью автоматизированный анализ** – основные функции и шаги теста (запуск, инициализация, выполнение, анализ, выдача результата) проводятся полностью автоматически.

23.09.2022

**Методы тестирования**

**Чёрный ящик (Black Box)** – тестировщик не знает устройство программы. Он может только воздействовать на входы программы, и получать на выходе некоторый результат.

Чёрный ящик может представлять собой набор классов или модулей с известными внешними интерфейсами, но недоступным исходным кодом.

**Выявляемые проблемы:**

1. Несоответствие поведения системы требованиям (Обычно вызывает изменение кодов);
2. Неадекватное поведение системы в ситуациях, не предусмотренных требованиям (явно вызывают изменения требований).

**Стеклянный (белый) ящик** – тестировщик видит код. Может определить соответствие требованиям участков программного кода, видеть участки кода, для которых не существуют требования (непокрытый требованиями код). Более углубленный анализ.

**Тестирование моделей** – анализируется и тестируется модель системы, а не она сама. На модели можно создать ситуации, которые невозможно создать при лабораторных тестах. На модели тестируются оптимальность алгоритмов, устойчивость системы.

Но т.к. разработать формальное описание для теста моделей сложно, такое тестирование используется редко (например, в системе связи).

30.09.2022

**Средства исследования программ**

**Дизассемблирование** – получение исполняемого кода программы на языке ассемблера.

**Дизассемблер** – программа, осуществляющая дизассемблирование.

**Интерактивный дизассемблер** – программа, тесно взаимодействующая с пользователем в процессе дизассемблирования.

**Отладчик** – программа, предназначенная для анализа поведения другой программы, обеспечивающая остановку в указанных точках и позволяющая просматривать (редактировать) содержимое ячеек памяти, регистров процессора и команды программы.

**Эмулирующий отладчик** – самостоятельно интерпретирует и выполняет команды программы (без использования реального процессора).

**Программы-утилиты** (предназначены для вспомогательных операций по изучению логики работы механизма защиты):

* Шестнадцатеричные просмотрщики – редакторы;
* Редакторы таблиц экспорта/импорта;
* Файловые мониторы, позволяющие отслеживать операции работы с файлами;
* Мониторы реестра, создающие протокол обращений к реестру;
* Другие.

FileMonitor может отследить работу защищённой программы с файлами и обнаружить ключ (пароль), хранящийся в некотором файле.

RegMon (монитор реестра) может обнаружить ключ (пароль), хранящийся в системной базе данных Registry.

**Эффективная защита** – та, на взлом которой необходимы материальные и трудовые затраты, во много раз превышающие затраты на покупку ПО.

**Защита программ от дизассемблирования:**

Шифрование:

* Если код полностью однократно шифруется/дешифруется, это неэффективно;
* Необходимо осуществить выбор эффективного ключа, надёжно хранить ключ;
* Для защиты программ от дизассемблирования не рекомендуется использование симметричных криптографических алгоритмов;
* Рекомендуется использовать шифрование с открытым ключом (алгоритм RSA, шифр Эль-Гамаля…) – не позволит просто так внести изменения в расшифрованный код;
* Защиту усиливает динамическое шифрование и многопроходная расшифровка кода;
* Шифрование с архиваций – неплохой вариант;

Самогенерируемые коды:

* Перемещения участков кода;
* Всевозможные функции от истинного кода.