Инструментация — это процесс модификации исследуемой программы с целью ее дальнейшего анализа.

**Динамический vs Статический анализ**

При анализе программ есть два возможных сценария: когда у нас есть исходный код приложения и когда у нас его нет. В большинстве случаев исходный код исследуемого приложения отсутствует, что при статическом анализе исполняемых файлов приводит к появлению целого ряда проблем. Например, невозможности отличить данные от кода, что, в свою очередь, ведет не только к неполному анализу, но и к получению ложных данных. А если исследуемая программа использует динамически генерируемый код, то даже наличие исходников не даст полную картину ее работы. Современные приложения все чаще собираются и определяются во время своего выполнения, используя разделяемые библиотеки, виртуальные функции, подключаемые плагины, динамически генерируемый код и прочие динамические механизмы. Поэтому количество получаемой с помощью статического анализа информации об исследуемой программе со временем становится все меньше и меньше. Статические инструменты уже не способны дать даже полного покрытия кода, что раньше было их неоспоримым преимуществом, не говоря уже о невозможности точно отличать данные от кода. В связи с этим появляется четкая необходимость в мощном динамическом подходе, сочетающем в себе простоту реализации и хорошую производительность.

В качестве более эффективной альтернативы динамическому анализу часто рассматривают статический анализ. В случае статического анализа поиск возможных ошибок осуществляется без запуска исследуемой программы, например, по исходному коду приложения. Обычно при этом строится абстрактная модель программы, которая, собственно, и является объектом анализа.   
При этом следует подчеркнуть следующие характерные особенности статического анализа:

* Возможен раздельный анализ отдельно взятых фрагментов программы (обычно отдельных функций или процедур), что дает достаточно эффективный способ борьбы с нелинейным ростом сложности анализа.
* Возможны ложные срабатывания, обусловленные либо тем, что при построении абстрактной модели некоторые детали игнорируются, либо тем, что анализ модели не является исчерпывающим.
* При обнаружении дефекта возникают, во-первых, проблема проверки истинности обнаруженного дефекта ([false positive](http://ru.wikipedia.org/wiki/False_positive)) и, во-вторых, проблема воспроизведения найденного дефекта при запуске программы на определенных входных данных.

В отличие от статического анализа, динамический анализ осуществляется во время работы программы. При этом:

* Для запуска программы требуются некоторые входные данные.
* Динамический анализ обнаруживает дефекты только на трассе, определяемой конкретными входными данными; дефекты, находящиеся в других частях программы, не будут обнаружены.
* В большинстве реализаций появление ложных срабатываний исключено, так как обнаружение ошибки происходит в момент ее возникновения в программе; таким образом, обнаруженная ошибка является не предсказанием, сделанным на основе анализа модели программы, а констатацией факта ее возникновения.

Динамический анализ программ можно разделить на три основных этапа:

* инструментация приложения;
* профилирование приложения;
* анализ полученной информации.

На первом этапе идет подготовка приложения к профилированию — инструментация. Затем запуск, и уже при выполнении накапливается информация о работе приложения. Собранная информация представляет собой трассу приложения (список выполненных инструкций программы), которая затем обрабатывается. Как уже говорилось, инструментацией называют процесс модификации исследуемой программы. За вставку дополнительного кода обычно отвечают инструментирующие процедуры. Они вызываются только раз при возникновении необходимого события и модифицируют целевую программу, добавляя в нее анализирующие процедуры, которые отвечают за необходимый анализ, изменение и мониторинг исследуемой программы и вызываются каждый раз при достижении определенного участка кода. Инструментация бинарного приложения может быть выполнена на разных уровнях гранулярности программы:

* инструкции;
* базового блока;
* трассы;
* процедуры;
* секции бинарного файла;
* бинарного образа.

Анализирующие процедуры обычно реализованы в виде процедур обратного вызова, которые срабатывают при достижении определенного участка кода или когда в программе происходит заданное событие. Для создания таких инструментов, работающих во время выполнения программы, были разработаны специальные наборы библиотек динамической бинарной инструментации. Инструменты же, которые с их помощью можно создать, называют динамическими бинарными анализаторами (DBA, Dynamic Binary Analysis). Ими уже достаточно давно пользуются программисты и тестировщики для решения целого ряда повседневных задач: моделирования, эмуляции, анализа производительности, проверок правильности работы, отладки памяти, параллельной оптимизации, построения графа вызовов, сбора метрик кода, автоматической отладки. Эти наборы библиотек также хорошо подходят и для задач, решаемых специалистами по информационной безопасности, администраторами, людьми, ищущими уязвимости, и антивирусными аналитиками. Так как обычно данные специалисты не имеют исходного кода программ, а влиять на их выполнение для сбора информации и последующего анализа как-то надо.

# Виды инструментации

Для составления полной картины об инструментации кода рассмотрим все ее виды. В этом нам поможет следующая классификация:

1. Исходные данные: исходный код, байт-код, исполняемый файл.
2. Объект изменения: исходный код, компилятор, исполняемый файл, процесс, интерпретатор / виртуальная машина, среда выполнения, архитектурные возможности аппаратного обеспечения.
3. Технология изменения: инструментация исходного кода, инструментация кода компилятором, статическая инструментация кода, динамическая инструментация кода, отладчики, воспроизведение работы среды, возможности по отладке, предоставленные аппаратным обеспечением.
4. Способ достижения технологии: ручная/автоматизированная вставка анализирующих функций, настройка компилятора, статическая бинарная инструментация, модификация загрузчика, эмуляция, виртуализация, паравиртуализация, патчинг, статическая бинарная инструментация, динамическая бинарная инструментация, отладочный API OC, эмуляция работы оборудования, отладочные регистры процессора, аппаратные отладчики.

Из приведенной классификации видно, что исходные данные (то есть то, что мы начинаем исследовать) могут быть представлены в одном из трех возможных видов:

* исходный код;
* байт-код;
* исполняемый бинарный файл.

Ситуация, когда доступен исходный код, возможна, только если ты являешься автором проекта или исследуемый проект опенсорсный. При этом возможно проанализировать только ту часть приложения, исходный код которой присутствует, — проанализировать сторонние (например, системные или чужие) библиотеки невозможно, так как их исходники почти всегда отсутствуют. Значительный плюс наличия исходников исследуемого приложения — полное знание о его высокоуровневом устройстве (названия переменных, типы переменных и так далее). Обратной стороной этого является полное отсутствие низкоуровневой информации (значение регистров, флагов, представления, как данные хранятся в памяти, и так далее). Байт-код — машинно-независимый код низкого уровня, генерируемый транслятором и исполняемый интерпретатором / виртуальной машиной. Большинство инструкций байт-кода эквивалентны одной или нескольким командам ассемблера. Трансляция в байт-код занимает промежуточное положение между компиляцией в машинный код и интерпретацией. C ним можно встретиться при анализе программ, написанных на Java/C#. Наличие только исполняемого бинарного файла — наиболее часто встречаемая ситуация, тем более при работе исследователя информационной безопасности. В этом случае уже хорошо, если для исследуемой программы доступны хотя бы отладочные символы. К данной категории в основном относятся приложения, написанные на компилируемых языках программирования, то есть, когда компилятор переводит исходный код непосредственно в машинные команды.

# Инструментация байт-кода

При наличии только лишь байт-кода инструментация производится другими способами, так как имеющееся промежуточное представление, перед тем как выполниться, загружается в виртуальную машину (ВМ) и уже там переводится в машинные инструкции и выполняется в операционной системе. В связи с такой схемой выполнения программы возникают и соответствующие проблемы:

* отладка приложений, работающих через интерпретатор / виртуальную машину, крайне тяжела по сравнению с отладкой обычных приложений;
* вся логика инкапсулирована внутри виртуальной машины;
* уязвимостями обычно являются ошибки в сгенерированном JIT-коде;
* ручная отладка и трассировка неправильного сгенерированного JIT-кода непрактична;

Что же касается инструментации байт-кода, то она может происходить на следующих стадиях:

* До запуска: o статическая инструментация: файл с байт-кодом инструментируется до его загрузки в ВМ. Например, создается копия файла, которая затем изменяется добавлением инструментации.
* Во время загрузки байт-кода в ВМ: o инструментация на этапе загрузки: возможно написание собственного загрузчика байт-кода в ВМ, который в процессе загрузки и будет производить инструментацию кода.
* Во время выполнения: o динамическая инструментация: байт-код, который уже загружен и, возможно, даже запущен, модифицируется.

## Программы для инструментации байт-кода

1. ReFrameworker — <http://bit.ly/160ro14> (.NET)
2. Javassist — <http://bit.ly/2xnqYs> (Java)
3. ObjectWeb ASM — <http://asm.ow2.org> (Java)
4. Byte Code Engineering Library (BCEL) — <http://bit.ly/13BqJY9> (Java)
5. The Microsoft Bytecode Engineering Library (MBEL) — <http://bit.ly/YWoN42> (.NET)
6. JavaSnoop — <http://bit.ly/aCYA82> (Java)
7. reJ — <http://bit.ly/b6kspj> (Java)
8. Serp — <http://bit.ly/gdxQrR> (Java)
9. Runtime Assembly Instrumentation Library (RAIL) — <http://rail.dei.uc.pt> (.NET)
10. Cecil — <http://bit.ly/4ibv7E> (.NET)
11. JOIE — <http://bit.ly/13BrCQt> (Java)
12. JMangler — <http://bit.ly/YJnmwu> (Java)

# Инструментация бинарного кода

Вариант, когда на руках у нас только сам исполняемый файл. Для его анализа мы можем влиять на него непосредственно (вставить в файл инструментирующий код, а затем запустить) либо в процессе работы (на процесс), на программное окружение, в котором он выполняется. Также мы можем использовать архитектурные возможности аппаратного обеспечения, на котором анализируется приложение. Все варианты можно представить в виде следующего списка:

* Исполняемый файл
  + Статическая инструментация кода
    - Статическая бинарная инструментация
* Процесс
  + Динамическая инструментация кода
    - Динамическая бинарная инструментация
  + Перехват таблицы вызовов
    - IAT
  + Отладчики
    - Отладочный API OC
* Среда выполнения
  + Перехват таблицы вызовов
    - IDT, CPU MSRs, GDT, SSDT, IRP-таблица
  + Воспроизведение работы среды
    - Эмуляция
    - Виртуализация
* Архитектурные возможности
  + Возможности по отладке, предоставленные аппаратным обеспечением
    - Отладочные регистры процессора
    - Аппаратные отладчики

Здесь нас интересуют два варианта — это статическая и динамическая бинарная инструментация.

# Статическая бинарная инструментация

Статическая бинарная инструментация кода заключается во вставке, изменении или удалении определенной функциональности в бинарном исполняемом файле, находящемся на диске, и сохранении получившегося файла в виде отдельной копии. После чего можно запускать модифицированный файл. При данном подходе получается, что инструментация проводится всего лишь один раз, а использовать ее можно многократно. Накладные расходы при таком подходе во время выполнения программы связанны только с выполнением вставленного кода. При статической бинарной инструментации инструментирующая программа, в зависимости от ее реализации, может влиять:

* на заголовок исполняемого файла и сегмент кода и/или данных;
* сегмент кода и/или данных.

Данный подход также иногда еще именуют физической интеграцией кода или статической бинарной перезаписью.

Статическую бинарную инструментацию также можно классифицировать по использованию так называемого перевыделения:

* с перевыделением: заключается в создании нового сегмента кода, куда переносится инструментируемая функция с уже вставленными участками анализирующего кода. Обычно перевыделение происходит на уровне функций, из-за наибольшей простоты и практичности. Для этого в начале оригинальной функции вставляется переход на проинструментированную;
* без перевыделения: при таком подходе анализирующий код интегрируется в оригинальный код, что в большинстве случаев влечет за собой почти полную пересборку бинарного файла.

Правда, у статического метода остается проблема отличия данных от кода, так как инструментация выполняется до выполнения программы. Кроме этого, недостатком еще является то, что разделяемые библиотеки необходимо инструментировать отдельно, в то время как при динамической бинарной инструментации инструментируется весь выполняемый код. Статические анализаторы также обеспечивают меньшую гибкость для разработчиков инструментов, поскольку любой инструментирующий код сохраняется в течение всей работы приложения, в то время как динамический инструментарий предоставляет средства для удаления или изменения такого кода по мере необходимости. Можно сказать, что статическая инструментация похожа на статическую трансляцию кода (когда код из одного представления транслируется в другое).

## Программы для статической бинарной инструментации

1. Dyninst — <http://www.dyninst.org>
2. EEL — <http://bit.ly/WZvNAs>
3. ATOM — <http://bit.ly/1714s59>
4. PEBIL — <http://bit.ly/16kOhyk>
5. ERESI — <http://bit.ly/yP5Dx>
6. TAU — <http://bit.ly/TFNUvU>
7. Vulcan — <http://bit.ly/11Slja5>
8. BIRD — <http://bit.ly/YMocmh>
9. Aslan (4514N) — <http://bit.ly/g5ZMlP>

# Динамическая бинарная инструментация

Динамическая бинарная инструментация выполняется во время работы программы и состоит из нескольких базовых состояний: 1) получение управления от приложения; 2) сохранение состояния программы; 3) выполнение задач; 4) восстановление состояния программы; 5) возвращение управления приложению.

Получение управления от приложения обычно происходит с помощью сплайсинга/trampoline/hot-patching’а, реализующих перезапись текущих инструкций, с их сохранением в определенном месте, на:

* на инструкцию передачи управления (jmp);
* функциональный кусок кода (snippets).

Вставка кода в запущенную программу

При перезаписи инструкций играет важную роль, на какой платформе это происходит, так как все платформы по типу инструкций делятся на две категории:

* с фиксированной длиной инструкций: в этом случае при перезаписи инструкции на инструкцию передачи управления всегда переписывается только одна инструкция. Примеры таких платформ — ARM и SPARC;
* с переменной длиной: в данном случае при перезаписи инструкции на инструкцию передачи управления может переписаться несколько инструкций (при этом последняя не полностью), что вносит определенные трудности. Примерами таких платформ являются x86 и x86–64.

Данный подход также называют физической интеграцией кода или динамической бинарной перезаписью. Преимущества и недостатки динамической бинарной инструментации перед статической:

* Нет необходимости в перелинковке и перекомпиляции
* Обнаружение кода в процессе выполнения
* Обработка динамически генерируемого кода
* Присоединение к запущенным процессам
* Анализ всего приложения как единого целого  
  – Увеличение накладных расходов  
  – Сложность реализации

Увеличение накладных расходов связано с тем, что во время динамической бинарной инструментации, помимо вставки анализирующих функций, также производятся задачи разбора (парсинга), дизассемблирования, генерации кода и так далее. Данные недостатки со временем становятся все менее заметными, так как снижение производительности устраняют использованием кеша и оптимизирующими алгоритмами. А для реализации динамической инструментации создают специальные программные библиотеки с достаточно простым и удобным API.

## Программы для динамической бинарной инструментации

1. PIN — <http://www.pintool.org>
2. DynamoRIO — <http://dynamorio.org>
3. Dyninst — <http://www.dyninst.org>
4. Valgrind — <http://valgrind.org>
5. BAP — <http://bap.ece.cmu.edu>
6. KEDR — <http://kedr.berlios.de>
7. ERESI — <http://www.eresi-project.org>
8. Detours — <http://bit.ly/lIumJ>
9. Vulcan — <http://bit.ly/11Slja5>
10. SpiderPig — <http://bit.ly/14znwIr>