С. С. Агафьин, П. В. Смирнов

МЕТОДЫ ОБНАРУЖЕНИЯ НЕДЕКЛАРИРОВАННЫХ ВОЗМОЖНОСТЕЙ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ ВНЕШНИХ АППАРАТНЫХ МОДУЛЕЙ

Существует множество методов обнаружения недекларированных возможностей (НДВ) программного обеспечения, которые используются для обеспечения безопасности приложений для различных архитектур. Однако лишь часть из них применима к программному обеспечению внешних аппаратных модулей в силу множества ограничений, которые они накладывают.

В данной статье рассматриваются существующие подходы к выявлению НДВ и оценивается их применимость к приложениям внешних аппаратных модулей.

Методы статического анализа

Статический анализ представляет собой проверку свойств программы по ее коду, без выполнения. В данном разделе проанализированы основные методы статического анализа, которые могут быть применены к программам, компилируемым в промежуточное представление. Их специфика по сравнению с остальными методами заключается в том, что статическому анализу можно подвергнуть не только к исходному коду, но и к скомпилированным в промежуточное представление модулям.

Поскольку в общем случае исполнение приложений внешних аппаратных модулей отдельно от аппаратной платформы невозможно, методы статического анализа являются основными для поиска НДВ в программном обеспечении внешних аппаратных модулей. Однако они не могут обеспечить полноты исследования из-за сложности с построением графа потока управления программы, связанной с тем, что мощность входного множества для программного обеспечения, согласно ГОСТ Р ИСО/МЭК 7816-4-2004 [1], равна 2^{2048}.

Платформа JavaCard [2] подвержена ряду атак, которые используют некорректную реализацию разделяемых интерфейсов либо подмену типов, что может заставить виртуальную машину JavaCard интерпретировать, например, некоторый объект как массив и полностью вывести конфиденциальную информацию через известную нарушителю APDU-команду.

Проверка кода на наличие сигнатур позволяет находить потенциально недопустимые последовательности байт-кодов и оповещать аналитика о том, что данное место программы может быть использовано для нарушения безопасности.

Сигнатурный анализатор кода можно представить кортежем [3]:

CSA = < SDB, HDB, MLA, MSA, MSY, MLO, MRP >,

где SDB – база данных сигнатур (паттернов) ПОК, HDB – база данных структурной информации о коде (иерархического представления программы), MLA – программный

модуль лексического анализа, MSA — программный модуль синтаксического анализа, MSY — программный модуль сигнатурного анализа, MLO — программный модуль логического вывода, MRP — программный модуль построения отчетов.

Сигнатурный анализатор осуществляет сквозной поиск в исходных текстах программы шаблонов, используя базу сигнатур, которая представляет собой набор следующих кортежей:

SDB = <PE, DL,TP>,

где PE — описание фрагмента (например, с использованием регулярных выражений), DL — оценка степени их опасности (например, числовая шкала 1—10), TP — тип (класс, категория) найденного дефекта или потенциальной уязвимости.

Преимуществом данного подхода является полная детерминированность, так как ряд последовательностей байт-кода не может присутствовать в корректных приложениях. Также он является крайне простым в реализации, так как полная проверка промежуточного представления на наличие сигнатур осуществляется элементарными алгоритмами, а также может быть в обязательном порядке включена в инструментальные средства разработчика, что не позволит внедрять подобные уязвимости на этапе разработки.

Недостатком этого этапа является отсутствие у него эвристической составляющей, что негативно сказывается на времени реагирования на новые угрозы. Более того, поскольку средства сигнатурного анализа для внешних аппаратных модулей мало распространены и редко обновляются, проходит значительное время между обнаружением потенциально небезопасной последовательности байт-кодов и внедрением сведений о ней в средства статического анализа.

Также данные средства не позволяют обнаружить возможность нарушения конфиденциальности, путем использования корректных байт-код-инструкций.

Примерами средств, реализующих данный метод статического анализа, являются Oracle JavaCard Off-Card Verifier, включенный в инструментарий разработчика JavaCard SDK, SAB JavaCard Verifier и PolySpace JavaCard Verifier.

Проверка формальных моделей

Проверка формальных моделей [4] (model checking) заключается в преобразовании программного кода в некоторую хорошо изученную формальную модель. Свойства, выполнимость которых требуется установить, формулируются в терминах этой модели и проверяются с помощью разработанных для нее алгоритмов. Преимуществом такого подхода является возможность использовать для решения задачи существующий математический аппарат и эффективно реализующие его инструментальные средства. SAT-системы (satisfability) позволяют установить истинность логической формулы или найти аргументы, при которых она становится ложной (так называемый контрпример).

SMT-системы (satisfability modulo theory), такие как STP и Yices, позволяют устанавливать истинность утверждений в отношении поддерживаемых конкретной SMT-системой теорий, таких как арифметика Пресбургера. Многие SMT-задачи могут быть выражены через SAT, например, целые числа можно кодировать битовыми векторами, а арифметические операции над числами выражать через логические операции над составляющими их битами. SMT-системы, специализирующиеся на целых числах, решают соответствующие задачи быстрее, чем обобщенные SAT-системы, но главным преимуществом SMT-систем являются принципиально меньшие ограничения на размер решаемых задач.

Чаще всего платформа JavaCard моделируется с помощью абстрактной машины состояний. Множеством состояний абстрактной машины является совокупность всех объектов, которые хранятся в ее памяти, входами являются стимулы, поступающие в виде APDU-запросов, а выходами реакции в виде APDU-ответов. Множество состояний обозначим S, множество стимулов — C, а множество реакций — R. Функцию выхода обозначим $f : S \times C \to R$, а функцию переходов — $\delta : S \times C \to R$.

Авторами ряда работ вводится понятие эквивалентности состояний абстрактной машины JavaCard, заключающееся в изоморфизме между наборами данных, соответствующих этим состоя-ниям. Каждое состояние представляется кортежем вида $s = <a_1, a_2, ..., a_n>$, где a_i— объект, хранимый в памяти виртуальной машины при выполнении инструкции. Два состояния называются эквивалентными, если существует биекция ϕ, для которой выполняется равенство вида

$\phi(s) = < \phi(a_1), \phi(a_2), ..., \phi(a_n)>$.

Тогда проверка свойства конфиденциальности программы заключается в следующем. Для любой пары эквивалентных состояний s^1, $s^2 \in S$ и любого стимула $c \in C$ справедливо, что $f(s^1, c) = f(s^2, c)$, а состояния $\delta(s^1, c)$ и $\delta(s^2, c)$ также будут эквивалентными.

Доказательство данного свойства с помощью Coq является довольно простым, однако его недостаточно для подтверждения отсутствия недокументированных возможностей.

Другим представлением JavaCard часто являются автоматные модели. Они описывают поведение системы (или компонента), задавая набор ее (его) состояний, стимулы, с помощью которых можно воздействовать на нее, возможные реакции и набор переходов между состояниями, каждый из которых может вызываться определенным стимулом, быть связан с выдачей некоторой реакции или быть внутренним, то есть происходить без видимых извне событий. Привязка стимулов и реакций к переходам может быть различной — в простейшем случае каждый переход соответствует паре «стимул — реакция», часто каждый переход может быть внутренним либо связан только с одним стимулом или только с одной реакцией. В практически используемых автоматных моделях состояния, стимулы и реакции могут содержать достаточно сложные структуры данных.

Анализ потоков данных

Анализ потоков данных работает с представлением программы в виде графа потока управления. С каждой вершиной $v \in V$ может связываться состояние $s \in S$ программы (например, возможные значения всех переменных) в ней, а с каждым ребром $e \in E \subseteq V \times V$ — условие $ce(s)$, при котором для перехода будет выбрано именно это ребро, а также передаточная функция $te \in S \times S$. Строго говоря, анализ потоков данных можно рассматривать как частный случай проверки формальных моделей, однако традиционно он считается отдельным направлением. Задача потоков данных сводится к решению системы уравнений, вытекающих из правил перехода по ребрам и начальных значений в определенных вершинах. В такой постановке ее можно решать с использованием абстрактной интерпретации. Граф потока управления можно считать системой переходов; действительно, множеством состояний такой системы будет $Ss = S \times V$, а отношение перехода будет задано как $ts = \{((s1, v1), (s2, v2)) | \exists e = (v1, v2) \in E : ce(s1) \wedge s2 = te(s1)\}$. Операции абстракции и конкретизации обычно строятся так, чтобы не изменять структуру графа потока управления, а абстрагировать лишь состояния программы. Таким образом, для упрощения имеет смысл рассматривать абстрактные состояния программы, а не тривиально вытекающие из них абстрактные состояния системы переходов.

Абстрактная интерпретация

Абстрактная интерпретация (abstract interpretation) [5] представляет собой метод анализа систем переходов (transition systems) с потенциально бесконечным количеством возможных состояний.

Для задания системы переходов S необходимо отношение перехода $t \subseteq S \times S$, множества начальных $I \subseteq S$ и конечных $F \subseteq S$ состояний. Анализ системы переходов будет заключаться в проверке какого-либо правила. Например, если задается правило безопасности, то система должна обладать таким свойством, что из каждого безопасного состояния за любое число переходов она может перейти только в другое безопасное состояние. Если обозначить множество состояний, достижимых из некоторого состояния $t \in T$ при заданном свойстве безопасности P как $post_t[P] = \{s' | \exists s: s \in P \wedge (s, s') \in t\}$, то проверка будет заключаться в том, что должно выполняться $post_t[I] \subseteq P$.

Однако прямая проверка этого свойства для всех начальных состояний является крайне сложной задачей, поскольку множество состояний может быть бесконечным. И даже в случае конечности количество требуемых ресурсов может превысить допустимый предел.

Можно показать, что если абстрактное свойство безопасности выполняется для абстрактной системы T', то для любой соответствующей конкретной системы $\in C(T')$ будет выполняться любое соответствующее конкретное свойство безопасности. Таким образом, абстрактная интерпретация позволяет уменьшить размер анализируемой системы за счет отбрасывания несущественных с точки зрения решаемой задачи деталей. Универсальных операций абстракции и конкретизации не существует; они должны подбираться отдельно для каждой решаемой задачи.

Абстрактной интерпретацией конкретной системы переходов T называется абстрактная система переходов T', для множеств состояний S и L которых существует включение Галуа, задаваемое парой отображений (A, C).

Отображение $A : S \to L$ называется операцией интерпретации, а $C : L \to S^*$ — операцией конкретизации. С помощью первого отображения конкретная система переходов преобразуется в абстрактную систему, а с помощью второго — абстрактная система уточняется до конкретной. Множество L называется абстрактным доменом.

Можно доказать, что если некоторое свойство безопасности $post'_{i^*}[I'] \subseteq P'$ выполняется для абстрактной системы T', то для любой конкретной системы T = C (T') соответствующее свойство безопасности также будет выполнено.

Методы динамического анализа

Динамический анализ представляет собой проверку свойств программы с помощью наблюдения за ее выполнением. В данном разделе представлены основные группы методов динамического анализа. Динамические методы, за исключением тривиальных случаев, могут показать наличие, но не могут доказать отсутствия НДВ. С другой стороны, динамические методы менее подвержены ложным срабатываниям, чем статические.

Методы контроля поведения заключаются в запуске программы и отслеживании с помощью специального инструментария ее взаимодействия с внешней средой: потребления ресурсов процессора и оперативной памяти, дисковой и сетевой активности, запуска внешних программ, вызова определенных системных функций и т. д. Наиболее часто используемыми методами контроля поведения являются модульное, функциональное и системное тестирование, направленные на проверку того, что поведение модуля или системы в целом в заданных условиях соответствует спецификации. При таком тестировании обычно осуществляется проверка основных результатов работы программы и стабильности выполняющей ее системы.

Несмотря на все преимущества, которые предоставляет контроль поведения, его практически невозможно реализовать применительно к внешним аппаратным модулям, так как они в основном являются однопоточными, а платформы не предоставляют базовой поддержки для отслеживания процесса исполнения программного обеспечения.

Инструментальное оснащение (instrumentation) — это модификация программного кода или выполнение его в специальной среде таким образом, что становится возможной проверка его внутренних свойств. Одними из наиболее часто встречающихся видов инструментального оснащения являются профилировка и анализ покрытия, в ходе которых строится список выполненных операторов (или машинных инструкций), а также измеряется время их выполнения. Оснащение можно выполнять как вручную, расставляя в коде операции работы с интересующими свойствами, так и автоматически. Инструментальному оснащению могут быть подвергнуты не все, а только важные с точки зрения решаемой задачи участки, такие как обращения к массивам или входы и выходы из подпрограмм. С другой стороны, иногда требуется оснащать всю программу,

как это происходит в случае с полной эмуляцией. Полная эмуляция может быть полезна для динамического продвижения меток, в ходе которого выявляются все данные, зависящие от данных, указанных пользователем. В отдельных случаях оснащение может приводить к значительной (на несколько порядков) потере производительности. Однако если потери незначительны, то оснастку можно использовать для проверок не только в ходе разработки, но и в ходе эксплуатации.

В связи с тем, что инструментальное оснащение предполагает необходимость взаимодействия между несколькими процессами, его использование для анализа однопоточных внешних аппаратных модулей также не представляется возможным. Эмуляция же программного обеспечения внешних аппаратных модулей в общем случае невозможна, так как оно может использовать программные интерфейсы приложений, которые реализованы на аппаратной платформе.

Экспертная оценка

Обычно в качестве видов экспертиз выделяют организационные экспертизы (management review), технические экспертизы (technical review), сквозной контроль (walkthrough), инспекции (inspection) и аудиты (audit).

Экспертизу отличает возможность выполнять ее, используя только само программное обеспечение, а не его модели или результаты работы.Экспертиза применима к любым свойствам ПО, хотя для разных целей могут использоваться разные ее виды. Она позволяет выявлять практически любые виды ошибок, причем делать это на самом раннем этапе, минимизируя таким образом время существования дефекта и его последствия для приложения. В то же время экспертиза не может быть автоматизирована и требует активного участия людей. Эмпирические наблюдения показывают, что эффективность экспертиз в терминах отношения количества обнаруживаемых дефектов к затрачиваемым на это ресурсам несколько выше, чем для других методов обнаружения проблем безопасности ПО. Так, различные отчеты показывают, что от 50 до 90 % всех зафиксированных в жизненном цикле ПО ошибок могут быть обнаружены с помощью экспертиз. За счет их раннего обнаружения может быть достигнута существенная экономия ресурсов — затраты на обнаружение ошибки составляют от 5 до 80 % от таких же затрат при использовании тестирования. Кроме того, регулярное участие в экспертизах является важным фактором в обучении сотрудников и способствует повышению качества результатов их работы. В то же время эффективность экспертизы существенно зависит от опыта и мотивации ее участников, организации процесса, а также от обеспечения корректного взаимодействия между различными участниками. Это накладывает дополнительные ограничения на распределение ресурсов в проекте и может приводить к конфликтам между разработчиками, если руководство проекта обращает мало внимания на коммуникативные аспекты проведения экспертиз.

Несмотря на целый ряд преимуществ данного метода, он обладает двумя критическими недостатками: высокой вероятностью ошибки второго рода и ограничением объема программного кода, который может авторитетно изучить эксперт. Учитывая, что сложность приложения для внешних аппаратных модулей постоянно увеличивается, для ряда программ данный метод не применим.

Генерация кода по спецификациям

Помимо проверки уже существующей реализации имеется методика, позволяющая генерировать код из высокоуровневых спецификаций, которые можно легко верифицировать различными средствами.

Наибольшее распространение для генерации JavaCard-кода получил Java Modelling Language (JML) — язык спецификации Java-модулей (интерфейсов и классов), описывающий как поведение, так и сигнатуру. JML базируется на идеях проектирования по контракту и спецификациях, основанных на моделях.

Ключевыми элементами любой спецификации являются предусловия (requires), постусловия (ensures) и инварианты (invariant). В языке JML существует заимствованное из Eiffel (языка

программирования, разработанного для поддержки технологии проектирования по контракту [TODO]) выражение \old(E), которое эквивалентно значению выражения E в момент входа в тело метода. Изменение значения переменной можно ограничить с помощью ключевого слова constraint, выражения \ old() и логических конструкций, таких как импликация или эквивалентность. Спецификации записываются между символами /*@ @*/ или после //@. Таким образом, стандартный Java-компилятор считает их комментариями и просто игнорирует. Они обрабатываются специально разработанными средствами.

Для поддержки JML было разработано большое число инструментов, обладающих различной функциональностью [TODO]. Наиболее простым способом проверки соответствия кода спецификации является runtime-проверка: компилятор JML (jmlc) запускает аннотированный Java-код и динамически проверяет соответствие типов и нарушение спецификации. Средства статической проверки (ESC/Java2) и формальной верификации (KeY, Loop, JACK) не требуют запуска приложения и способны решать значительно более сложные задачи, но с помощью пользователя и ценой некоторых допущений.

JML может применяться для аннотирования произвольных Java-приложений, но на данный момент основной сферой его применения считается JavaCard [TODO].

Несмотря на все преимущества, данный подход обладает и рядом недостатков. Прежде всего, они заключаются в том, что генерация кода по формально безопасным спецификациям не позволяет исключить из рассмотрения возможность модификации скомпилированного кода в редакторе шестнадцатеричного представления, что делает ее не применимой для ряда моделей нарушителя.

Выводы

На настоящий момент проблема обнаружения недокументированных возможностей в программном обеспечении JavaCard не решена окончательно. Несмотря на то что ее отличия от платформы Java минимальны, специфика реализации виртуальной машины JavaCard не позволяет использовать существующие методики, разработанные для Java.

Более того, существующие методики поиска недекларированных возможностей для JavaCard не проработаны в достаточном объеме, а их применимость для анализа защищенности при наличии нарушителя, способного разрабатывать или модифицировать программное обеспечение, мало изучена.

СПИСОК ЛИТЕРАТУРЫ:

1. ГОСТ Р ИСО/МЭК 7816-4-2004. Информационная технология. Карты идентификационные. Карты на интегральных схемах с контактами. Часть 4. Межотраслевые команды для обмена. Введ. 2005-01-01. М.: Изд-во стандартов, 2005. — 78 с.
2. Java Card Classic Platform Specification 3.0.4. URL: http://www.oracle.com/technetwork/java/ (дата обращения: 01.12.2013).
3. *Марков А. С., Фадин А. А.* Проблемы автоматизации выявления программных закладок и дефектов безопасности кода методом статического сигнатурного анализа. URL: http://www.npo-echelon.ru/doc/ (дата обращения: 01.12.2013).
4. *Кулямин В. В.* Методы верификации программного обеспечения. URL: http://www.ict.edu.ru/ft/005645/ (дата обращения: 01.12.2013).
5. *Леошкевич И. О.* Система выявления недекларированных возможностей программного обеспечения, влекущих нарушение конфиденциальности информации. Дисс. ... канд. техн. наук. М., 2011.

REFERENCES:

1. GOST R ICO/MEK 7816-4-2004. Informatsionnay tekhnologiay. Karty identifikatsionnye. Karty na integral`nykh skhemakh s kontaktami/ chast` 4. Mezhotraslevye komandy dlay obmena. Vved. 2005-01-01. M.: Izd-vo standartov, 2005. — 78 s.
2. *Markov A. S., Fadin A. A.* Problemy avtomatizatsii vyayvleniay programmnikh zakladok I defektov bezopasnosti koda metodom statisticheskogo signaturnogo analiza. URL: http://www.npo-echelon.ru/doc/ (data obrasheniay 01.12.2013).
3. *Kulaymin V. V.* Metody verifikatsii programmnogo obespecheniay URL: http:// www.ict.edu.ru/ft/005645/ (data obrasheniay 01.12.2013).
4. *Leoshkevich I. O.* Sistema viayvleniay nedeklarirovannikh vozmozhnosteiy programmnogo obespecheniay, vlekushikh narushenie konfidentsial`nosti informatsii: dissertatsiay kandidata tekhnicheskikh nauk Nats.issled.ayderniy un-t «MIPhI». Moskva, 2011.