Логика первого порядка для задания требований к безопасному программному коду

А.В. Козачок <a.kozachok@academ.msk.rsnet.ru>
Академия Федеральной службы охраны РФ,
302034, Россия, г. Орёл, ул. Приборостроительная, д. 35

Аннотация. В настоящее время вопросу защиты информации при проектировании и эксплуатации объектов критической информационной инфраструктуры уделяется особое внимание. Одним из распространенных подходов к обеспечению безопасности информации, обрабатываемой на объектах, при этом является создание изолированной программной среды. Безопасность среды обуславливается ее неизменностью. Однако эволюционное развитие систем обработки информации порождает необходимость запуска в данной среде новых компонентов и программного обеспечения при условии выполнения требований по безопасности. Наиболее важным при этом является вопрос доверия к новому программному коду. Данная работа посвящена разработке формального логического языка описания функциональных требований к программному коду, который позволит в дальнейшем предъявлять требования на этапе статического анализа и контролировать их выполнение в динамике.

Ключевые слова: формальный логический язык; моделирование; процесс; вредоносная программа; проверка моделей; автомат безопасности

DOI: 10.15514/ISPRAS-2017-29(5)-8

Для цитирования: Козачок А.В. Логика первого порядка для задания требований к безопасному программному коду. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 135-148. DOI: 10.15514/ISPRAS-2017-29(5)-8

1. Введение

В последние несколько лет особое внимание в исследованиях, посвященных защите информации, уделяется вопросу, касающемуся обеспечения информационной безопасности объектов критической информационной инфраструктуры (КИИ). Подтверждением данного факта является разработанный и внесенный на рассмотрение федеральный закон "О безопасности критической информационной инфраструктуры Российской Федерации" 6 декабря 2016 года. Особое внимание при этом уделяется защите объектов КИИ от компьютерных атак и деструктивных программных

воздействий [1]. Предметом рассмотрения данной статьи является защита объектов КИИ от деструктивных программных воздействий.

Зачастую возможность реализации данных угроз обуславливается наличием доступа в глобальную сеть Интернет объектов КИИ. При этом существующие системы и средства защиты информации не обеспечивают гарантированной защиты. Так, например, исследование, проведенное компанией AV-Comparatives, показало, что применяемые в современных антивирусных средствах механизмы обнаружения позволяют достичь уровня эвристического обнаружения 0,974 ("Avast Internet Security"), но при этом остаются необнаруженными 468 образцов вредоносного программного обеспечения [2].

Одним из возможных подходов к защите от деструктивных программных воздействий является использование изолированной программной среды, которая является доверенной и безопасной при условии сохранения ее неизменности. Однако эволюционное развитие систем сбора и обработки информации, а также наличие доступа объектов КИИ в глобальную сеть Интернет порождает необходимость запуска в данной среде новых компонентов и программного обеспечения, которые могут нарушить ее целостность и безопасность. Наиболее важным при этом является вопрос доверия к новому контенту и программному коду.

Одним из возможных вариантов построения безопасной среды с возможностью доверия к приходящему контенту является применение системы безопасного исполнения программного кода [3]. Предлагаемая система является расширенной композицией двух активно развивающихся подходов к обнаружению деструктивных программных воздействий, а именно: применения методов формальной верификации модели исследуемой программы [4–7] и использование автомата безопасности для контроля за функционированием исследуемой программы в реальном времени [8–12].

В основе системы безопасного исполнения программного кода лежит предположение о том, что если безопасность программного кода при априорно известных функциональных требованиях не подтверждена, то ее использование запрещается. Настоящее исследование посвящено разработке формального языка описания функциональных требований к программному коду для применения его в разрабатываемой системе безопасного исполнения программного кода.

2. Краткий обзор исследований в области формальной верификации моделей программ для обнаружения вредоносного кода

Идея применения метода формальной верификации "Model checking" при решении задачи обнаружения деструктивных программных воздействий состоит в том, чтобы построить формальную (математическую) модель вредоносной программы, которая отражает (моделирует) ее возможное

поведение в операционной системе. Разрешенное поведение программы при этом задается в виде спецификации. На основе этой спецификации и модели исполняемого файла с использованием метода "Model checking" принимается решение о возможности использования анализируемой программы.

Впервые данный метод при решении задачи по обнаружению вредоносного кода был применен в 2005 году в работе Киндера [4]. Группа авторов предложила анализировать поведение программ и на основе "Model checking" принимать решение о его схожести с поведением вредоносных программ. Предложенный ими подход заключается в задании спецификаций с помощью формул темпоральной логики для каждого из классов деструктивных программ, представители которого имели схожее поведение, однако существенно отличались по бинарному представлению, вследствие чего не могли быть обнаружены сигнатурным методом. Каждый исследуемый бинарный файл автоматически преобразовывался в модель, заданную на языке верификатора. На основе этой модели верификатор определял соответствует ли она одной из множества заданных спецификаций, соответствующих семействам вредоносных программ. Для сокращения записи спецификаций авторами была введена логика CTPL (Computation Tree Predicate Logic), являющаяся расширением известной логики CTL. Достоинством предложенного подхода является возможность обнаружения семейств вредоносных программ. Ограничением является то, что он не учитывает работу исследуемой программы со стеком, а также необходимость ручного задания спецификации поведения для каждого класса вредоносного кода.

В 2012 году Фу Сонг и Тассир Туили предложили использовать метод "Model checking" для обнаружения вредоносных программ с учетом их поведения и взаимодействия со стеком [5]. Для описания модели поведения вредоносного кода авторами была введена логика SCTPL (Stack Computation Tree Predicate Logic), позволяющая учитывать работу со стеком. Применение данного подхода позволило существенно повысить точность обнаружения вредоносных программ. В результате развития данного подхода авторами была предложена логика SLTPL (Stack Linear Tree Predicate Logic) [6].

3. Описание системы безопасного исполнения программного кода

Отличительная особенность разрабатываемой системы заключается в извлечении информации о поведении исследуемой программы, как на этапе хранения, так и на этапе исполнения (рис. 1). Полученная информация сравнивается со спецификацией поведения в соответствии с заданными функциональными требованиями. В случае их соответствия исследуемая программа признается безопасной.

Для реализации предложенного подхода на практике необходимо решение следующих частных задач:

- анализ (извлечение и преобразование информации из исследуемой программы в вид, пригодный для дальнейшей обработки – спецификации);
- синтез (построение модели безопасного исполнения программного кода в соответствии с заданными функциональными требованиями);
- верификация (алгоритм, получающий на вход спецификацию анализируемой программы и принимающий решение о ее соответствии модели безопасного исполнения программного кода);
- контроль исполнения (реализация монитора исполнения программы, позволяющего перехватывать все сигналы взаимодействия процесса с операционной системой, отслеживать соответствие состояния программы заданной конфигурации и завершить целевую программу в случае необходимости).

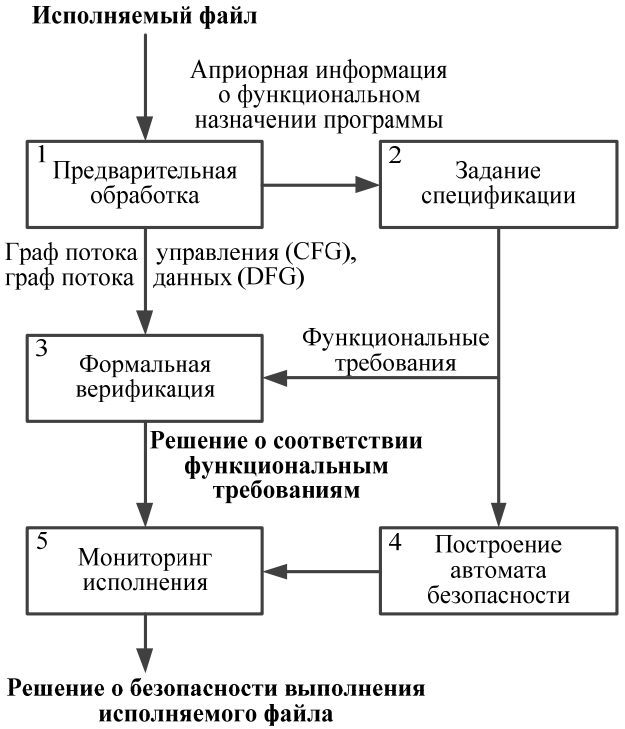

Рис. 1. Модель системы безопасного исполнения программного кода
Fig. 1. Secure code execution system model

На вход блока 1 подается исследуемый исполняемый файл, безопасность которого необходимо установить. В данном блоке происходит проверка наличия в коде программы конструкций самомодификации (включая упаковщики) и механизмов защиты кода от анализа. Выполнение этих требований является необходимым условием для дальнейшего исследования исполняемого файла. В случае их наличия файл признается небезопасным и его дальнейшая проверка прекращается. Далее происходит преобразование исполняемого файла из бинарного представления в последовательность команд языка ассемблера и данных, на их основе строится граф потока управления (Control Flow Graph) и граф потока данных (Data Flow Graph). Также на этом этапе осуществляется сбор и систематизация априорных сведений о функциональном предназначении программы, которые подаются на вход блока 2.

В блоке "Задания спецификации" на основе априорных сведений о функциональном предназначении программы формируется перечень ограничений на ее функциональные возможности, выполнение которых необходимо для безопасного использования программы. Данный перечень включает в себя функциональные требования, выполнение которых может быть обеспечено в рамках работы системы безопасного исполнения программного кода. Они могут быть разделены на группы в зависимости от категории исследуемой программы. Выходом данного блока являются формализованные функциональные ограничения на работу программы в виде формулы темпоральной логики и конфигурации автомата безопасности.

В блоке 3 осуществляется процесс формальной верификации модели исполняемого файла, построенной на основе графов потоков управления и данных, на соответствие функциональным требованиям статического этапа проверки с помощью метода "Model checking". Требования по корректности безопасного поведения при этом описываются в виде спецификации, которая отражает рамки разрешенного поведения программы. Поскольку происходит именно математическая верификация, решение о согласованности, то есть соответствии возможного поведения требуемому, является корректным. Алгоритмы проверки моделей, как правило, базируются на исчерпывающей достижимости всего множества состояний модели [13].

Таким образом, для каждого состояния проверяется, удовлетворяет ли оно заданным в спецификации требованиям. В самой простой форме алгоритмы проверки моделей позволяют дать ответ на вопрос о достижимости заданных состояний. В таком случае необходимо определить все запрещенные состояния, достижение которых является небезопасным, и выяснить, существует ли такая последовательность их смены, которая приводит к одному из запрещенных. Если такая последовательность существует, то принимается решение о запрете использования исследуемой программы. Стоит отметить, что исчерпывающая достижимость множества состояний гарантируется ввиду конечности числа состояний модели [14]. В случае рассогласованности модели программы и

спецификации безопасности исполняемый файл признается небезопасным и его дальнейшее исследование прекращается. Выходом данного блока является решение о безопасности использования данной программы по результатам верификации на статическом этапе проверки.

Для контроля над выполнением функциональных ограничений во время работы программы предлагается использование системы, схожей с системой предотвращения вторжений на уровне компьютера. Монитор исполнения запускается параллельно с исполняемой программой и перехватывает все, совершаемые ею, системные вызовы. Изначально монитор исполнения загружает модель разрешенного поведения и устанавливает начальное состояние. Каждый вызов системной функции сопоставляется с моделью разрешенного поведения и происходит переход в новое состояние или, в случае отсутствия такого перехода, подается команда на завершение процесса. В основу монитора исполнения положен автомат безопасности (Security Automata) [8], который строится, как правило, на основе автомата со стеком. Входными символами автомата является множество событий функционирования процесса, а конфигурация автомата определяет множество разрешенных операций для каждого состояния.

В блоке 4 происходит преобразование функциональных требований к программе в конфигурацию конечного автомата со стеком. В блоке 5 производится непрерывный мониторинг функционирования программы в рамках заданной модели безопасного исполнения. Выходом данного блока является решение о безопасности выполнения исполняемого файла.

4. Формальный логический язык описания функциональных требований

Базовыми понятиями при рассмотрении работы операционной системы (ОС) являются процесс и ресурс. Согласно [15] процесс – это контейнер для набора ресурсов, используемых при выполнении экземпляра программы. К основным видам ресурсов ОС относятся следующие элементы [16]:

- процессорное время;
- оперативная память;
- внешняя память;
- устройства ввода-вывода.

В основе построения предлагаемой системы безопасного исполнения программного кода лежит модель описания функционирования процесса в операционной системе. К субъектам относятся процессы, совершающие действия по отношению к объектам. Объектами являются ресурсы ОС и процессы, в отношении которых совершаются действия другими субъектами:

- "процесс" (p);
- "оперативная память" (m);

- "внешняя память" (*e*);
- "периферийные устройства" (*d*);
- "сетевая подсистема" (*n*).

Для осуществления доступа к ресурсу процесс выполняет соответствующую функцию ОС, то есть подает заявку на выполнение некоторых действий. ОС на основе внутренних механизмов распределения ресурсов, а также на основе политики безопасности, принимает решение о доступе исполняемого кода данного процесса к запрашиваемому ресурсу.

Прикладные программы в процессе работы обладают полным доступом к своему виртуальному адресному пространству для выполнения операций чтения и записи. Для ввода или вывода данных за пределы своего адресного пространства прикладной программе необходимо вызывать соответствующие функции ОС, если она имеет соответствующие привилегии для осуществления таких операций. К таким функциям ОС можно отнести операции чтения, записи, запуска или завершения процесса, выделения дополнительной области памяти, ее освобождения и др.

Анализ исследований в области формальной верификации показал, что существующие подходы к описанию спецификаций при решении задачи обнаружения вредоносных программ не являются универсальными. Так как часть из них ориентирована на команды языка ассемблера, а часть – на API-функции.

Поэтому предлагается следующий формальный логический язык для задания функциональных требований с возможностью однозначного перехода к формулам темпоральной логики для дальнейшей верификации по моделям.

В соответствии с определением логики первого порядка [17] необходимо задать следующие подмножества:

$$FormSpec = Func \bigcup \Pr ed \bigcup Var \bigcup Log \bigcup Aux.$$

При этом множество функциональных символов будет включать в себя следующие операции:

$$Func = \{create, open, delete, read, write\},$$

где *create* – операция создания объекта, *open* – операция открытия объекта, *delete* – операция удаления (завершения) объекта, *read* – операция чтения из объекта, *write* – операция записи в объект.

Множество предикатных символов включает в себя базовые предикаты темпоральной логики CTL* [18] и предикат проверки безопасности:

$$\Pr ed = \{IsSecure, AX, AF, AG, AU,$$
$$AR, EX, EF, EG, EU, ER, EC\},$$

где *IsSecure* – предикат проверки безопасности текущего состояния относительно трассы исполнения в целом, *A* – квантор всеобщности, указывающий на то, что данное свойство выполнено для всех путей, *E* – квантор

существования, указывающий на то, что данное свойство выполняется для некоторого пути, *X* – унарный оператор, указывающий на то, что данное свойство выполняется на следующем состоянии текущего пути, *G* – унарный оператор, указывающий на то, что данное свойство выполняется на каждом состоянии текущего пути, *F* – унарный оператор, указывающий на то, что данное свойство выполняется в некотором состоянии в будущем, *U* – бинарный оператор, указывающий на то, что первое свойство выполняется для всех состояний пути, предшествующих состоянию, где выполняется второе свойство, *R* – бинарный оператор, указывающий на то, что второе свойство выполняется для всех состояний, следующих до состояния, в котором выполняется первое свойство, *C* – унарный оператор, указывающий на то, что данное свойство выполняется в текущем состоянии текущего пути (дополнительно введен авторами).

Множество символов предметных переменных включается в себя следующие элементы:

$$Var = \{p, m, n, e, d, cat\},$$

где *p,m,n,e,d* – объекты над которыми совершаются действия, *cat* – индекс категории объекта (субъекта) (табл. 1).

Табл. 1. Категории объектов и субъектов.
Table 1. Categories of subjects and objects.

Обозначение	Описание
Субъект "Процесс" (*p*)	
1	системный процесс
2	привилегированный процесс
3	пользовательский процесс
Объект "Оперативная память" (*m*)	
1	адресное пространство системного процесса
2	адресное пространство другого процесса
3	собственное адресное пространство процесса
Объект "Внешняя память" (*e*)	
1	исполняемые файлы
2	системные каталоги и конфигурация системы
3	файлы и каталоги других пользователей
4	системные библиотеки
5	собственные файлы и каталоги
Объект «Периферийные устройства» (d)	
1	устройства вывода
2	устройства ввода
Объект "Сетевая подсистема" (*n*)	
1	сервисы узлов глобальной сети
2	сервисы узлов локальной сети
3	локальные сетевые сервисы

Множество логических символов включает в себя следующие элементы:

$$Log = \{\neg, \wedge, \vee, \rightarrow, \exists, \forall\},$$

где $\neg$ – символ логического отрицания, $\wedge$ – символ конъюнкции, $\vee$ – символ дизъюнкции, $\rightarrow$ – символ импликации, $\exists$ – квантор существования, $\forall$ – квантор всеобщности.

Множество вспомогательных символов включает в себя следующие элементы:

$$Aux = \{,()\}.$$

5. Базис функциональных требований, обеспечивающих безопасное исполнение программного кода

На основе предложенного формального логического языка *FormSpec* были сформулированы базовые правила (формулы) безопасного исполнения программного кода для каждого из функциональных символов. Символ * означает объект (субъект) любой категории из числа возможных для данного класса.

Для операции создания объекта:

- $\neg$ *EF create (p,*,p,*)* – запрет на создание дочерних процессов;
- *EF create (p,*,m,3)* – выделение памяти только в своем адресном пространстве процесса;
- *EF create (p,*,e,5)* – разрешено создание новых файлов (каталогов) только в каталоге текущего процесса;
- $\neg$ *EF create (p,*,n,*)* – запрет на создание сетевых соединений;
- $\neg$ *EF create (p,*,d,*)* – запрет на создание устройств (драйверов).

Для операции открытия объекта:

- $\neg$ *EF open (p,*,p,*)* – запрет на открытие процессов;
- *EF open (p,*,e,4)* $\vee$ *EF open (p,*,e,5)* – разрешено открытие системных библиотек и файлов, содержащихся в текущем каталоге процесса;
- $\neg$ *EF open (p,*,d,*)* – запрет на открытие устройств.

Для операции удаления (завершения) объекта:

- *EF delete (p_i,*,p_i,*)* – процесс может завершить свою работу;
- *EC open (p,*,e_j,5)* $\wedge$ *EF delete(p,*,e_j,5)* – процесс может удалять файлы им созданные.

Для операции чтения из объекта:

- $\neg$ *EF read(p,*,p,*)* – запрет на получение информации о процессах;
- *EF read(p,*,m,3)* – разрешено чтение адресного пространства своего процесса;
- *EC open(p,*,e_j,4)* $\wedge$ *EF read(p,*,e_j,4)* – разрешено чтение из системных

библиотек;

- *(EC open(p,*,e_j,5)* $\vee$ *EC create(p,*,e_j,5))* $\wedge$ *EF read(p,*,e_j,5)* – разрешено чтение файлов открытых (созданных) процессом;
- $\neg$ *EF read(p,*,n,*)* – запрет на работу с сетью;
- $\neg$ *EF read(p,*,d,*)* – запрет на работу с устройствами.

Для операции записи в объект:

- *EF write(p,*,m,3)* – разрешена запись в адресное пространство своего процесса;
- *(EC open(p,*,e_j,5)* $\vee$ *EC create(p,*,e_j,5))* $\wedge$ *EF write(p,*,e_j,5)* – разрешено чтение файлов открытых (созданных) процессом;
- $\neg$ *EF write(p,*,n,*)* – запрет на работу с сетью;
- $\neg$ *EF write(p,*,d,*)* – запрет на работу с устройствами.

Следует отметить, что представленный базис можно рассматривать как аксиоматический, так как его выполнение обеспечивает безопасность выполнения программного кода (выполнение предиката *IsSecure*) в аспекте защиты от вредоносного кода. Также ограничения, вводимые им на предмет взаимодействия с сетевой и файловой подсистемами, возможно преодолеть за счет введения ограничений на последовательность выполняемых действий и изоляции возможных информационных контуров.

6. Обсуждение результатов

Одним из вариантов практического приложения предложенного формального логического языка описания функциональных требований к программному коду является формализация угроз из "Банка данных угроз безопасности информации" ФСТЭК [19].

"Угроза изменения системных и глобальных переменных" нарушителем может быть реализована за счет применения вредоносного программного обеспечения, которое может привести к совершению опосредованного деструктивного воздействия на определенные программы или систему в целом. Для нейтрализации данной угрозы необходимо задать следующее правило: "Запрет процессам 3 категории на совершение изменений системных и глобальных переменных", выраженное следующем образом:

$$\neg EF\ (create\ (p,3,e,2) \vee write\ (p,3,e,2)) \quad (1)$$

Выражение (1) формально означает, что процессы 3 категории не могут осуществлять создание или модификацию системных каталогов и файлов конфигурации. Этим же правилом можно предотвратить "угрозу несанкционированного редактирования реестра".

Суть "угрозы несанкционированного копирования защищаемой информации" заключается в получении злоумышленником копии защищаемой информации

другого пользователя и дальнейшем выводе ее за пределы системы. Для ограничения последовательности таких действий необходимо ввести следующее правило:

$$\neg EF(EC\ read\ (p,*,e,3) \wedge (EF\ create\ (p,*,e,5) \vee EF\ write\ (p,*,e,5) \vee \vee EF\ write\ (p,*,d,1) \vee EF\ write\ (p,*,n,1)) \quad (2)$$

Из выражения (2) следует, что процессу любой категории запрещается последовательно сначала считывать некоторую информацию из файлов других пользователей, а затем осуществлять ее запись в файлы в собственном каталоге или отправлять на устройства вывода или в сеть.

Для "угрозы перехвата вводимой и выводимой на периферийные устройства информации" можно задать правило, ограничивающее прямое взаимодействие процессов 3 категории и устройств ввода:

$$\neg EF\ read(p,3,d,2) \quad (3)$$

Выражение (3) запрещает непосредственный доступ к считыванию информации с устройств ввода в обход существующих в операционной системе механизмов.

Ряд приведенных примеров подтверждает состоятельность и полноту возможностей описания существующих угроз на предложенном формальном логическом языке. Их учет при работе системы безопасного исполнения кода позволит исключить возможность реализации заданных угроз.

7. Заключение

Предложенный формальный логический язык описания функциональных требований, позволяющий описать поведение процесса без конкретизации операций или элементарных действий (на высоком уровне абстракции), и в обобщенной математической форме выразить субъектно-объектные отношения процессов с ресурсами ОС различных категорий., задает основу построения системы безопасного исполнения программного кода, которая позволит доверять новому программному коду и не нарушать целостность изолированной программной среды.

Направлением дальнейших исследований является построение полного набора правил безопасного исполнения программного кода с использованием введенного формального логического языка, позволяющего устранить ограничения, введенные аксиоматическим базисом.

Список литературы

[1]. Проект федерального закона от 06.12.2016 № 9198-П10 "О безопасности критической информационной инфраструктуры Российской Федерации". URL: http://asozd2.duma.gov.ru/main.nsf/(SpravkaNew)?OpenAgent&RN=47571-7&02 (дата обращения 14.03.2017).

[2]. Козачок А.В., Кочетков Е.В., Татаринов А.М. Обоснование возможности построения эвристического механизма распознавания вредоносных программ на основе статического анализа исполняемых файлов. Вестник компьютерных и информационных технологий, 2017, №. 3, с. 50-56. DOI: 10.14489/vkit.2017.03. pp.050-056.

[3]. Козачок А.В., Кочетков Е.В. Обоснование возможности применения верификации программ для обнаружения вредоносного кода. Вопросы кибербезопасности. 2016. № 3 (16). С. 25-32.

[4]. Kinder J. et al. Detecting malicious code by model checking. International Conference on Detection of Intrusions and Malware and Vulnerability Assessment. Springer Berlin Heidelberg, 2005, pp. 174-187.

[5]. Song F., Touili T. Efficient malware detection using model-checking. International Symposium on Formal Methods. Springer Berlin Heidelberg, 2012, pp. 418-433.

[6]. Song F., Touili T. PoMMaDe: pushdown model-checking for malware detection. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013, pp. 607-610.

[7]. Jasiul B., Szpyrka M., Śliwa J. Formal Specification of Malware Models in the Form of Colored Petri Nets. Computer Science and its Applications. Springer Berlin Heidelberg, 2015, pp. 475-482.

[8]. Schneider F.B. Enforceable security policies. ACM Transactions on Information and System Security (TISSEC), 2000, vol. 3, no. 1, pp. 30-50.

[9]. Feng H.H. et al. Formalizing sensitivity in static analysis for intrusion detection. Proc. IEEE Symposium on Security and Privacy, 2004, pp. 194-208.

[10]. Basin D. et al. Enforceable security policies revisited. ACM Transactions on Information and System Security (TISSEC), 2013, vol. 16, no. 1, pp. 3-8.

[11]. Feng H.H. et al. Anomaly detection using call stack information. Proc. IEEE Symposium on Security and Privacy, 2003, pp. 62-75.

[12]. Basin D., Klaedtke F., Zălinescu E. Algorithms for monitoring real-time properties. International Conference on Runtime Verification. Springer Berlin Heidelberg, 2011, pp. 260-275.

[13]. Кларк Э., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking. Москва, 2002, изд-во МЦНМО, 416 с.

[14]. Вельдер С.Э., Лукин М.А., Шалыто А.А., Яминов Б.Р. Верификация автоматных программ. Санкт-Петербург, 2011, изд-во Наука, 244 с.

[15]. Руссинович М., Соломон Д. Внутреннее устройство Microsoft Windows. 6-е изд. Санкт-Петербург, 2013, изд-во Питер, 800 с.

[16]. Гордеев А.В. Операционные системы: по направлению подгот. "Информатика и вычисл. техника". Санкт-Петербург, 2009, изд-во Питер, 416 с.

[17]. Коротков М.А., Степанов Е.О Основы формальных логических языков. Санкт-Петербург, 2003, изд-во СПб ГИТМО (ТУ), 84 с.

[18]. Hafer T., Thomas W. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. International Colloquium on Automata, Languages and Programming. Springer Berlin Heidelberg, 1987, pp. 269-279.

[19]. ФСТЭК "Банк данных угроз безопасности информации" URL: http://bdu.fstec.ru/ (дата обращения: 14.03.17).

First order logic to set requirements for secure code execution

A.V. Kozachok < a.kozachok@academ.msk.rsnet.ru >
The Academy of the Federal Guard Service of the Russian Federation,
34, Priborostroitelnaya st., Oryol, 302034, Russia

Abstract. Currently the problem of information security during designing and exploiting the objects of critical information infrastructure is paid special attention to. One of the most common approaches to providing information security, processed on the objects, is creating isolated programming environment. The environment security is determined by its invariability. However, the evolutional development of data processing systems gives rise to the necessity of implementing the new components and software in this environment under condition that security requirements are satisfied. The most important requirement consists in trust in the new programming code. The given paper is devoted to developing formal logical language of description of functional requirements for programming code, allowing to make further demands at the stage of static analysis and to control their implementation in dynamics.

Keywords: formal logical language, modeling, process, malware, model checking, security automata.

DOI: 10.15514/ISPRAS-2017-29(5)-8

For citation: Kozachok A.V. First order logic to set requirements for secure code execution. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 5, 2017. pp. 135-148 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-8

References

[1]. Federal law draft no. 9198-P10 "On the security of the critical information infrastructure of the Russian Federation". URL: http://asozd2.duma.gov.ru/main.nsf/(SpravkaNew)/OpenAgent&RN=47571-7&02 (in Russian).
[2]. Kozachok A.V., Kochetkov E.V, Tatarinov A.M. Construction Heuristic Malware Detection Mechanism Based on Static Executable File Analysis Possibility Proof. Vestnik komp'juternyh i informacionnyh tehnologij [Herald of Computer and Information Technologies], 2017, no. 3, pp. 50-56. DOI: 10.14489/vkit.2017.03. pp. 050-056 (in Russian).
[3]. Kozachok A.V., Kochetkov E.V. Using program verification for detecting malware. Voprosy kiberbezopasnosti [Cybersecurity issues], 2016, no. 3, vol. 16, pp. 25-32 (in Russian).
[4]. Kinder J. et al. Detecting malicious code by model checking. International Conference on Detection of Intrusions and Malware and Vulnerability Assessment. Springer Berlin Heidelberg, 2005, pp. 174-187.
[5]. Song F., Touili T. Efficient malware detection using model-checking. International Symposium on Formal Methods. Springer Berlin Heidelberg, 2012, pp. 418-433.

[6]. Song F., Touili T. PoMMaDe: pushdown model-checking for malware detection. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013, pp. 607-610.
[7]. Jasiul B., Szpyrka M., Śliwa J. Formal Specification of Malware Models in the Form of Colored Petri Nets. Computer Science and its Applications. Springer Berlin Heidelberg, 2015, pp. 475-482.
[8]. Schneider F.B. Enforceable security policies. ACM Transactions on Information and System Security (TISSEC), 2000, vol. 3, no. 1, pp. 30-50.
[9]. Feng H.H. et al. Formalizing sensitivity in static analysis for intrusion detection. Proc. IEEE Symposium on Security and Privacy, 2004, pp. 194-208.
[10]. Basin D. et al. Enforceable security policies revisited. ACM Transactions on Information and System Security (TISSEC), 2013, vol. 16, no. 1, pp. 3-8.
[11]. Feng H.H. et al. Anomaly detection using call stack information. Proc. IEEE Symposium on Security and Privacy, 2003, pp. 62-75.
[12]. Basin D., Klaedtke F., Zălinescu E. Algorithms for monitoring real-time properties. International Conference on Runtime Verification. Springer Berlin Heidelberg, 2011, pp. 260-275.
[13]. Klark Je., Gramberg O., Peled D. Program models verification: Model Checking. Moscow, MCNMO, 2002, 416 p (in Russian).
[14]. Vel'der S.Je., Lukin M.A., Shalyto A.A., Jaminov B.R. Automata program verififcation. St. Petersburg, Nauka, 2011, 244 p (in Russian).
[15]. Russinovich M. E., Solomon D. A., Ionescu A. Windows internals. Pearson Education, 2012, 800 p.
[16]. Gordeev A.V. Operating systems. Izdatel'skij dom “Piter”, 2009, 412 p (in Russian).
[17]. Korotkov M.A., Stepanov E.O Fundamentals of formal logical languages. St. Petersburg, SPb GITMO (TU), 2003, 84 p (in Russian).
[18]. Hafer T., Thomas W. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. International Colloquium on Automata, Languages and Programming. Springer Berlin Heidelberg, 1987, pp. 269-279.
[19]. Federal Service for Technical and Export Control. Information security threats database. URL: http://bdu.fstec.ru (in Russian).