DOI: 10.15514/ISPRAS–2020–32(6)–3

Формальная верификация модели мандатного контроля целостности в операционной системе KasperskyOS

В.С. Буренков, ORCID: 0000-0002-0232-774X <Vladimir.Burenkov@kaspersky.com>
АО «Лаборатория Касперского»,
125212, Россия, Москва, Ленинградское шоссе, д. 39А, стр. 3

Аннотация. Модели мандатного контроля целостности в операционных системах, как правило, накладывают ограничения на доступы активных компонент системы к пассивным и представляют эти доступы непосредственно. Такое представление можно использовать в случае операционных систем с монолитной архитектурой, где части системы, обеспечивающие доступ к ресурсам, входят в доверенную вычислительную базу. Однако доказательство отсутствия ошибок в таких компонентах и, следовательно, соответствия такой модели реальной системе является чрезвычайно трудной задачей. KasperskyOS – операционная система, основанная на микроядре и позволяющая организовать доступ к ресурсам с помощью компонентов, не обязательно входящих в доверенную вычислительную базу. Для KasperskyOS была разработана и реализована модель мандатного контроля целостности, учитывающая наличие таких компонентов и возможность их некорректного функционирования. В данной статье представлен процесс формализации этой модели и автоматизированного доказательства свойств, обеспечиваемых моделью. Описана разработка модели на языке Event-B и отражен опыт ее верификации с использованием платформы Rodin.

Ключевые слова: мандатный контроль целостности; Event-B; операционная система; KasperskyOS

Для цитирования: Буренков В.С. Формальная верификация модели мандатного контроля целостности в операционной системе KasperskyOS. Труды ИСП РАН, том 32, вып. 6, 2020 г., стр. 31-48. DOI: 10.15514/ISPRAS–2020–32(6)–3

Formal Verification of a Mandatory Integrity Control Model for the KasperskyOS Operating System

V.S. Burenkov, ORCID: 0000-0002-0232-774X <Vladimir.Burenkov@kaspersky.com>
Kaspersky Lab
125212, Russia, Moscow, Leningradskoe shosse, 39A, 3

Abstract. Models of mandatory integrity control in operating systems usually restrict accesses of active components of a system to passive ones and represent the accesses directly. This is suitable in case of monolithic operating systems whose components that provide access to resources are part of the trusted computing base. However, due to the sheer complexity of such components, it is extremely nontrivial to prove such a model to be adequate to the real system. KasperskyOS is a microkernel operating system that organizes access to resources via components that are not necessarily part of the trusted computing base. KasperskyOS implements a specifically developed mandatory integrity control model that takes such components into account. This paper formalizes the model and describes the process of automated proof of the model's properties. For formalization, we use the Event-B language. We clarify parts specific to Event-B to make our presentation accessible to professionals familiar with discrete mathematics but not necessarily with Event-B. We reflect on the proof experience obtained in the Rodin platform.

Keywords: mandatory integrity control; Event-B; operating system; KasperskyOS

For citation: Burenkov V.S. Formal Verification of a Mandatory Integrity Control Model for the KasperskyOS Operating System. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 6, 2020, pp. 31-48 (in Russian). DOI: 10.15514/ISPRAS–2020–32(6)–3

1. Введение

Контроль целостности компьютерных систем является одним из фундаментальных вопросов, рассматриваемых при разработке безопасных систем. Для обеспечения целостности в операционных системах реализуются механизмы мандатного контроля целостности [1]. Классический путь разработки механизмов безопасности начинается с создания модели системы [2]. Существуют и официальные требования наличия формальной модели политики безопасности. Их предъявляет, например, ФСТЭК России [3].

В данной работе рассматривается формализация и верификация модели мандатного контроля целостности, разработанной для и реализованной в новой микроядерной операционной системе KasperskyOS. Модель изначально была описана на математическом и естественном языках [4].

В работе кратко отмечены архитектурные особенности операционной системы KasperskyOS, нашедшие свое отражение в модели мандатного контроля целостности. Детально представлена формализация модели на языке Event-B [5]. Модель сопровождена пояснениями, облегчающими ее понимание для широкого круга специалистов, знакомых с основами дискретной математики, но не обязательно имеющим опыт работы с языком Event-B. Описаны решения по моделированию различных компонентов и конструкций модели. Сформулированы свойства безопасности, обеспечиваемые моделью, и описан опыт их доказательства с использованием платформы Rodin [6].

2. Выбор метода и инструмента верификации

Для верификации моделей управления доступом используют как метод проверки модели [7–10], так и методы дедуктивной верификации [11–15]. Метод проверки моделей подвержен проблеме взрыва числа состояний. Поэтому его использование приведет либо к необходимости ограничения размерности множеств субъектов и объектов моделей (то есть к неполной верификации), либо к необходимости разработки модификаций метода (например, с целью получения абстрактных моделей) и специализированных инструментов.

Методы дедуктивной верификации не накладывают ограничений на размерность элементов модели, однако заведомо требуют ручного вмешательства. Тем не менее, например, язык Event-B, одной из основ которого является теория множеств, позволяет достаточно удобно описывать модели, состояние которых представлено множествами и отношениями, а изменения состояний – посредством бинарного отношения на множестве состояний, или, другими словами, событий или правил преобразования состояний. Именно в таких терминах описана модель мандатного контроля целостности для KasperskyOS в работе [4]. Использование же платформы Rodin, представляющей инфраструктуру для верификации моделей на языке Event-B, позволяет зачастую достичь высокой степени автоматизации процесса доказательства [16]. Ввиду этих причин в данной работе выбран язык Event-B для формализации модели мандатного контроля целостности.

3. Архитектура операционной системы KasperskyOS

Модель, формализуемая в данной работе, была разработана и реализована в микроядерной операционной системе KasperskyOS и отражает ряд архитектурных свойств этой системы. Основными компонентами операционной системы являются микроядро, монитор безопасности и множество драйверов, реализованных в виде непривилегированных

приложений. Главными функциями микроядра являются разделение ресурсов между процессами (которые в рамках операционной системы именуются *сущностями*), предоставление механизма межпроцессного взаимодействия IPC (interprocess communication) и запуск монитора безопасности для проверки этих взаимодействий. Операционная система разрабатывалась с учетом принципа разделения ресурсов и приложений посредством минимальной доверенной вычислительной базы, направленного на поддержку политик безопасности. Современным воплощением такого подхода является архитектура MILS [17].

Важным компонентом архитектуры MILS является *ядро разделения*, реализация которого гарантирует изоляцию приложений друг от друга. Изоляция приложений очень важна с точки зрения безопасности, так как она позволяет ограничить нежелательное воздействие на систему со стороны отдельных приложений.

Однако, во многих случаях одного только ядра разделения недостаточно: необходимо также убедиться, что взаимодействия в системе удовлетворяют требованиям безопасности. В KasperskyOS для решения этой проблемы существует Kaspersky Security System – набор инструментов для спецификации различных свойств безопасности и синтеза монитора обращений, проверяющего любые взаимодействия внутри системы на предмет соответствия политике. Для спецификации свойств (политики) используется декларативный язык Policy Specification Language (PSL). Мандатный контроль целостности представлен в языке множеством правил, соответствующих правилам модели мандатного контроля целостности. Описанная на PSL политика транслируется в запускаемый образ монитора безопасности, которому ядро и другие сервисы могут направлять свои запросы.

Каждая сущность или ресурс является *доменом безопасности*, с которым ассоциирован *контекст безопасности*, определяемый *идентификатором*. Таким образом, с точки зрения политики безопасности, программно-аппаратная система, управляемая KasperskyOS, представляет собой множество доменов безопасности и информационных потоков между ними.

3.1 Взаимодействие между процессами

В KasperskyOS сущности обмениваются друг с другом сообщениями посредством механизма синхронного межпроцессного взаимодействия. Выделяются две роли: клиент (сущность, инициирующая взаимодействие) и сервер (сущность, обрабатывающая обращение). Клиент направляет серверу сообщение-*запрос*, при этом поток исполнения, из которого производится обращение, блокируется до получения ответа от сервера или ядра (в случае, например, ошибки). Серверный поток исполнения находится в ожидании сообщений. При получении запроса, этот поток получает управление, обрабатывает запрос и отсылает сообщение-*ответ*. При получении сообщения-ответа клиентский поток может продолжать исполнение.

Монитор безопасности имеет возможность контролировать любые IPC-сообщения. С этой целью ядро передает монитору на проверку как сообщение-запрос, так и ответ. Передача сообщения происходит только в случае, если монитор дал разрешение. Кроме операций по контролю запросов и ответов, монитор также поддерживает две специальных операции – execute и security.

Операция *execute* используется ядром для сообщения монитору о создании новой сущности. Операция *security* позволяет сущностям отправлять сообщения монитору безопасности напрямую. Данная операция является основой для реализации сервисов (драйверов), которые предоставляют контролируемый доступ к ресурсам.

3.2 Управление доступом к ресурсам

В KasperskyOS большинство сервисов реализуется как непривилегированные приложения. Такие сервисы часто предоставляют доступ к ресурсам определенного вида. Смысл понятия

«ресурс» определяется самим сервисом. Например, для файловой системы ресурсами могут быть файлы и каталоги. Поскольку данные сервисы не входят в ядро операционной системы, то контролировать доступ к ним только возможностями ядра невозможно. В связи с этим для разграничения доступа к таким ресурсам в KasperskyOS выработан следующий поход.

Ключевым понятием является *драйвер* – это процесс-сервис, который вводит в систему новый тип ресурсов. Доступ к ресурсам определенного типа возможен только путем обращения к соответствующему драйверу. С каждым ресурсом драйвер ассоциирует системный дескриптор, с которым подсистема безопасности связывает информацию, необходимую для контроля доступа (например, метку целостности).

Во время каждого доступа к ресурсам драйвер ресурсов должен предоставлять монитору безопасности необходимый контекст. Монитор безопасности затем разрешает или запрещает доступ. При доступе к ресурсу, драйвер использует операцию security для обращения к подсистеме безопасности. В этом обращении драйвер передает дескриптор ресурса и любую дополнительную информацию, которую посчитает необходимой для осуществления контроля. Например, при чтении файла передается информация о том, кто (дескриптор клиента) хочет выполнить операцию (чтение файла), над чем (дескриптор файла). Монитор безопасности применяет ассоциированные с этим обращением правила и возвращает результат проверки. Ответственность драйвера – правильно проинтерпретировать ответ монитора: заблокировать операцию, если она не была разрешена. Таким образом, для реализации контроля над ресурсами, драйвер должен предоставить механизм, а политика определяется монитором безопасности. Такая архитектура позволяет подсистеме безопасности единообразно трактовать как системные ресурсы (в первую очередь, сущности и IPC-каналы), так и прочие ресурсы, введенные в систему драйверами. Ядро операционной системы при этом занимается только управлением (выдачей и квотированием) системных дескрипторов.

Ядро операционной системы гарантирует разделение адресных пространств сущностей, поэтому драйвер может управлять только теми ресурсами, системные дескрипторы на которые были выданы именно этому драйверу.

Безусловно, уровень доверия (в самом общем смысле) некоторого ресурса не может быть выше, чем уровень доверия драйвера этого ресурса.

4. Разработка и верификация формальной модели мандатного контроля целостности

4.1 Контекст модели

Контекст модели определяет ее неизменное состояние. Далее представлены множества, константы и аксиомы контекста. Имена аксиом предварены символом `@`.

4.1.1 Сущности и объекты

Сущности (субъекты) системы представляют активные компоненты операционной системы.

Объекты системы представляют пассивные компоненты операционной системы. Все сущности и объекты модели принадлежат множеству `Entities_and_objects_universe`:

```
@Entities_and_objects_universe_finite
    finite(Entities_and_objects_universe) // множество конечно
```

Среди сущностей выделяется сущность `core`, представляющая ядро операционной системы KasperskyOS:

```
@core_type
    core ∈ Entities_and_objects_universe
```

4.1.2 Доступы и информационные потоки

Множество *видов доступа* `Access_types` состоит из двух констант `read_a` и `write_a`, где `read_a` обозначает доступ на чтение, `write_a` обозначает доступ на запись:

```
@Access_types
    partition(Access_types, {read_a}, {write_a})
```

Множество *видов информационных потоков* `Flow_types` включает в себя константу `write_m`, обозначающую информационный поток по памяти на запись в сущность или объект:

```
@Flow_types
    Flow_types = {write_m}
```

4.1.3 Уровни целостности

Уровни целостности представлены в модели конечным множеством `Integrity_levels`, на котором определено отношение частичного порядка `le_il` («≤»):

```
@Integrity_levels_finite
    finite(Integrity_levels)
@le_il_type
    le_il ⊆ Integrity_levels × Integrity_levels
@le_il_reflexive // le_il является рефлексивным
    ∀x · x ∈ Integrity_levels ⇒ x ↦ x ∈ le_il
@le_il_transitive // le_il является транзитивным
    ∀x, y, z · x ∈ Integrity_levels ∧ y ∈ Integrity_levels ∧
                z ∈ Integrity_levels ∧
      x ↦ y ∈ le_il ∧ y ↦ z ∈ le_il ⇒ x ↦ z ∈ le_il
@le_il_antisymmetric // le_il является антисимметричным
    ∀x, y · x ∈ Integrity_levels ∧ y ∈ Integrity_levels ∧
      x ↦ y ∈ le_il ∧ y ↦ x ∈ le_il ⇒ x = y
```

На основе отношения `le_il` также определено отношение `l_il` («<»):

```
@l_il_type
    l_il ⊆ Integrity_levels × Integrity_levels
@l_il_def
    ∀x, y · x ∈ Integrity_levels ∧ y ∈ Integrity_levels ∧
      x ↦ y ∈ l_il ⇔ x ↦ y ∈ le_il ∧ x ≠ y
```

Заметим, что в качестве множества уровней целостности в моделях мандатного контроля целостности обычно используют решетку. Превратить множество `Integrity_levels` в решетку можно добавлением аксиом существования наибольшей нижней грани (инфимума) `inf_il` и наименьшей верхней грани (супремума) `sup_il`. Например:

```
@inf_exists
    inf_il ∈ Integrity_levels ∧
      (∀x · x ∈ Integrity_levels ⇒ inf_il ↦ x ∈ le_il) ∧
      (∀lb · lb ∈ Integrity_levels ∧
       (∀y · y ∈ Integrity_levels ⇒ lb ↦ y ∈ le_il) ⇒ lb ↦ inf_il ∈ le_il)
```

4.2 Машина модели

Машина модели содержит ее динамическую часть. Далее представлены переменные, инварианты и события, определяющие машину. Имена инвариантов (свойств, которые всегда должны выполняться), предварены символом `@`.

4.2.1 Сущности и объекты

`Entities` – множество сущностей системы, среди которых выделяется особая сущность – ядро операционной системы:

```
@Entities_type
    Entities ⊆ Entities_and_objects_universe
@core_type
    core ∈ Entities
```

`Objects` – множество объектов системы:

```
@Objects_type
    Objects ⊆ Entities_and_objects_universe
```

Сущности и объекты представлены разными множествами, поскольку они имеют различную природу. Однако в определении некоторых элементов модели (например, множества информационных потоков) используются как сущности, так и объекты. Чтобы в таких случаях иметь возможность использовать объединение множеств сущностей и объектов, эти множества должны быть подмножествами одного универсума, определенного в контексте модели. Заметим, что множества сущностей и объектов не пересекаются:

```
@Entities_and_objects_differ
    Entities ∩ Objects = ∅
```

Для целей моделирования иерархии объектов введена *функция иерархии объектов* `Container_contents`, которая ставит в соответствие каждому объекту множество объектов:

```
@Container_contents_type
    Container_contents ∈ Objects → ℙ(Objects)
```

Будем говорить, что объекты из `Container_contents(c)` непосредственно находятся в контейнере `c`.

Выделяется множество корневых объектов `Root_objects`, не находящихся ни в каком контейнере:

```
@Root_objects_type
    Root_objects ⊆ Objects
```

Для целей моделирования управления работой с ресурсами системы посредством драйверов ресурсов, в модели введено понятия *драйвера объектов*. Драйвер объектов – сущность, посредством которой выполняются операции с объектами. Будем говорить, что драйвер объектов управляет подмножеством объектов.

Функция `Resource_driver` ставит в соответствие любому объекту системы его драйвер:

```
@Resource_driver_type
    Resource_driver ∈ Objects → Entities
```

Несмотря на то что драйверы ресурсов и другие прикладные сущности не обязательно входят в доверенную вычислительную базу, их корректное функционирование предполагает удовлетворение определенных требований [4]. Однако по каким-либо причинам (например, ошибка в исходном коде сущности) эти требования могут быть не выполнены. Описание таких причин может быть составлено на основе деталей реализации системы, что находится за рамками модели. В то же время модель позволяет выполнить анализ последствий нарушения требований и возникновения непредвиденных событий в системе. Для этого в модели введено понятие захвата контроля над сущностью или объектом. Если над сущностью захвачен контроль, то предположения о ее поведении более не выполняются. Захват контроля над объектом означает появление в нем данных, которые он не должен содержать при корректном функционировании системы. Формально последствия захвата контроля над сущностями и объектами описаны далее в правилах преобразования состояний.

Множества `Entities_compromised` и `Objects_compromised` определяют сущности и объекты, над которыми захвачен контроль:

```
@Entities_compromised_type
    Entities_compromised ⊆ Entities
@Objects_compromised_type
    Objects_compromised ⊆ Objects
```

Как будет доказано в подразделе 4.3, модель исключает неограниченный захват контроля частей системы: если захвачен контроль над частью системы, максимальный уровень целостности компонент которой ниже максимально возможного уровня целостности, то это не приводит к захвату контроля над более целостными компонентами системы и, как следствие, всей системы в целом.

4.2.2 Доступы и информационные потоки

`Accesses` – множество *доступов* сущностей к объектам:

```
@Accesses_type
    Accesses ⊆ Entities × Objects × Access_types
```

`Flows` – множество *информационных потоков*:

```
@Flows_type
    Flows ⊆ (Entities ∪ Objects) × (Entities ∪ Objects) × Flow_types
```

4.2.3 Уровни целостности

Уровень целостности сущностей и объектов определен с помощью функции `integrity_level`:

```
@integrity_level_type
    integrity_level ∈ (Entities ∪ Objects) → Integrity_levels
```

При работе операционной системы возникают ситуации, в которых возможно участие сущности в «опасных» взаимодействиях, то есть обращениях на чтение к объектам с более низким либо несравнимым уровнем целостности. Для разрешения таких взаимодействий в модели введена функция `integrity_level_r`, определяющая для каждой сущности минимальный уровень целостности объектов, к которым эта сущность может обращаться на чтение:

```
@integrity_level_r_type
    integrity_level_r ∈ Entities → Integrity_levels
```

В модели также введена *функция привилегий* `upgrade_privilege`, служащая для обозначения сущностей, которым разрешено повышать уровень целостности объектов:

```
@upgrade_privilege_type
    upgrade_privilege ∈ Entities → BOOL
```

Предполагается, что в качестве таких сущностей могут выступать доверенные сущности, для которых значение данной функции устанавливается равным `TRUE`. Заметим, что в рамках модели мы не рассматриваем процесс определения функции привилегий. Предполагается, что ее значение всегда определено корректно. Если ее значение равно `TRUE` для некоторой сущности `s`, то эта сущность действительно обладает привилегией повышения уровня целостности объектов. Если же сущность `s` не обладает такой привилегией, то значение функции привилегий для нее равно `FALSE`.

4.2.4 События модели

События разработаны в соответствии с правилами преобразования состояний, определенными в работе [4]. Все события, кроме события инициализации, включают защиты

– логические условия, определяющие, в каких случаях событие может быть исполнено, – и действия, описывающие результат исполнения события. Событие инициализации включает только действия. Имена защит в представленной модели начинаются с `@grd`, а имена действий – с `@act`.

Инициализация начального состояния модели

Событие инициализации описывает начальное состояние модели. Модель разрабатывалась таким образом, чтобы к ее начальному состоянию предъявлялось как можно меньше требований. Предполагается, что в начальном состоянии существует сущность `core`, и определены значения ее уровней целостности с учетом того, что ядро операционной системы является наиболее доверенной сущностью.

```
event INITIALISATION
    then
      @act1 Entities ≔ {core}
      @act2 Objects ≔ ∅
      @act3 Entities_compromised ≔ ∅
      @act4 Objects_compromised ≔ ∅
      @act5 Accesses ≔ ∅
      @act6 Flows ≔ ∅
      @act7 integrity_level ≔ {core ↦ sup_il}
      @act8 integrity_level_r ≔ {core ↦ sup_il}
      @act9 upgrade_privilege ≔ {core ↦ FALSE}
      @act10 Root_objects ≔ ∅
      @act11 Container_contents ≔ ∅ × {∅}
      @act12 Resource_driver ≔ ∅
end
```

Получение доступа на чтение к объектам

Событие `access_read` определяет получение сущностью `entity` доступа на чтение к объекту `object` посредством драйвера `driver`. Возникающие в результате применения правила информационные потоки зависят от того, является ли драйвер скомпрометированным. Множество информационных потоков формируется в действии `act_basic2` события. Для определения потоков в зависимости от факта компрометации драйвера используются конструкции *set comprehension*, частью условия которых является принадлежность драйвера множеству скомпрометированных сущностей. Такой подход позволяет моделировать конструкции *if-then-else*, используемые в модели на математическом и естественном языках.

```
event access_read
  any
    entity
    driver
    object
  where
    @grd_basic1 entity ∈ Entities
    @grd_basic2 driver ∈ Entities
    @grd_basic3 object ∈ Objects
    @grd_basic4 Resource_driver(object) = driver
    @grd_mic1 (integrity_level(entity) ↦ integrity_level(driver) ∈ le_il)∨
        (¬(integrity_level(entity) ↦ integrity_level(driver) ∈ le_il)
```

```
        ∧ integrity_level_r(entity) ↦ integrity_level(driver) ∈ le_il)
    @grd_mic2 driver ∉ Entities_compromised ⇒
        ((integrity_level(entity) ↦ integrity_level(object) ∈ le_il) ∨
          (¬(integrity_level(entity) ↦ integrity_level(object) ∈ le_il)
         ∧ integrity_level_r(entity) ↦ integrity_level(object) ∈ le_il))
    @grd_mic3 integrity_level(object) ↦ integrity_level(driver) ∈ le_il
  then
    @act_basic1 Accesses ≔ Accesses ∪ {entity ↦ object ↦ read_a}
    @act_basic2 Flows ≔ Flows ∪
        {o ↦ e ↦ w | o = object ∧ e = entity ∧ w = write_m ∧
                     driver ∉ Entities_compromised ∧
         ((integrity_level(entity) ↦ integrity_level(object) ∈ le_il ∧
           integrity_level(entity) ↦ integrity_level(driver) ∈ le_il) ∨
                        entity ∈ Entities_compromised)} ∪
        {o ↦ e ↦ w | o = object ∧ e = entity ∧ w = write_m ∧
                  driver ∈ Entities_compromised ∧
         (integrity_level(entity) ↦ integrity_level(driver) ∈ le_il ∨
                        entity ∈ Entities_compromised)} ∪
        {d ↦ e ↦ w | d = driver ∧ e = entity ∧ w = write_m ∧
                  driver ∈ Entities_compromised ∧
         (integrity_level(entity) ↦ integrity_level(driver) ∈ le_il ∨
                           entity ∈ Entities_compromised)}
end
```

Получение доступа на запись к объектам

Событие `access_write` определяет получение сущностью `entity` доступа на запись к объекту `object` посредством драйвера `driver`. Возникающие информационные потоки для этого события описываются проще, чем в случае получения доступа на чтение, поэтому в действии act_basic2 вместо оператора присваивания и конструкций set comprehension использована формула, устанавливающая связь между переменными после (со штрихом) и до (без штриха) выполнения события. Такой подход еще более близок к модели на математическом и естественном языках.

```
event access_write
  any
    entity
    driver
    object
  where
    @grd_basic1 entity ∈ Entities
    @grd_basic2 driver ∈ Entities
    @grd_basic3 object ∈ Objects
    @grd_basic4 Resource_driver(object) = driver
    @grd_mic1 driver ∉ Entities_compromised ⇒
                integrity_level(object) ↦ integrity_level(entity) ∈ le_il
    @grd_mic2 integrity_level(object) ↦ integrity_level(driver) ∈ le_il
  then
    @act_basic1 Accesses ≔ Accesses ∪ {entity ↦ object ↦ write_a}
    @act_basic2 Flows :| (Flows' = Flows ∪
```



```
        {entity ↦ object ↦ write_m} ∧ driver ∉ Entities_compromised) ∨
                        (Flows' = Flows ∪ {entity ↦ object ↦ write_m,
entity ↦ driver ↦ write_m} ∧ driver ∈ Entities_compromised)
end
```

Создание сущностей и объектов, удаление, перемещение, повышение уровня целостности объектов

Событие `execute` определяет, как сущность `creator` создает (запускает) сущность `new_entity` из объекта `image` с установлением уровней целостности сущности `new_entity`.

```
event execute
  any
    creator
    image
    new_entity
    il
    ilr
  where
    @grd_basic1 creator ∈ Entities
    @grd_basic2 image ∈ Objects
    @grd_basic3 new_entity ∈ Entities_and_objects_universe
    @grd_basic4 new_entity ∉ Entities ∪ Objects
    @grd_mic1 il ∈ Integrity_levels
    @grd_mic2 ilr ∈ Integrity_levels
    @grd_mic3 il ↦ integrity_level(image) ∈ le_il
    @grd_mic4 ilr ↦ il ∈ le_il
  then
    @act_basic1 Entities ≔ Entities ∪ {new_entity}
    @act_basic2 Flows :| (Flows' = Flows ∪
          {image ↦ new_entity ↦ write_m} ∧ image ∈ Objects_compromised) ∨
                        (Flows' = Flows ∧ image ∉ Objects_compromised)
    @act_basic3 upgrade_privilege(new_entity) ≔ FALSE
    @act_mic1 integrity_level(new_entity) ≔ il
    @act_mic2 integrity_level_r(new_entity) ≔ ilr
end
```

Событие `create_object` определяет, как сущность `creator` создает объект `new_object` посредством драйвера `driver`, включает `new_object` в состав контейнера `container` и устанавливает уровень целостности `new_object` в `il`. Для описания изменений в функции иерархии объектов: модификации ее значения для объекта `container` и добавления в область определения нового объекта `new_entity` с установлением значения функции для него в ∅ использован оператор relational overriding ∃.

```
event create_object
  any
    creator
    new_object
    container
    driver
    il
  where
    @grd_basic1 creator ∈ Entities
```

```
    @grd_basic2 new_object ∈ Entities_and_objects_universe
    @grd_basic3 new_object ∉ Entities ∪ Objects
    @grd_basic4 container ∈ Objects
    @grd_basic5 driver ∈ Entities
    @grd_basic6 creator ↦ container ↦ write_a ∈ Accesses
    @grd_basic7 driver ↦ container ↦ write_a ∈ Accesses
    @grd_mic1 il ∈ Integrity_levels
    @grd_mic2 il ↦ integrity_level(creator) ∈ le_il
    @grd_mic3 il ↦ integrity_level(container) ∈ le_il
    @grd_mic4 il ↦ integrity_level(driver) ∈ le_il
  then
    @act_basic1 Objects ≔ Objects ∪ {new_object}
    @act_basic2 Container_contents ≔ Container_contents ∃
      {container ↦ (Container_contents(container) ∪{new_object}),
       new_object ↦ ∅}
    @act_basic_drr Resource_driver(new_object) ≔ driver
    @act_basic3 Flows :| (Flows' = Flows ∪
      {creator ↦ new_object ↦ write_m, creator ↦ driver ↦ write_m}
                                  ∧ driver ∈ Entities_compromised) ∨
                        (Flows' = Flows ∧ driver ∉ Entities_compromised)
    @act_basic4 Objects_compromised :| (driver ∈ Entities_compromised ∧
       Objects_compromised' = Objects_compromised ∪ {new_object} ) ∨
                                     (driver ∉ Entities_compromised ∧
       Objects_compromised' = Objects_compromised)
    @act_mic1 integrity_level(new_object) ≔ il
end
```

Событие `create_root` по созданию корневых объектов определено аналогично событию `create_object`.

Событие `move_object` определяет, как сущность `entity` перемещает объект `object` из контейнера `from` в контейнер `to` посредством драйвера `driver`.

```
event move_object
  any
    entity
    object
    driver
    from
    to
  where
    @grd_basic1 entity ∈ Entities
    @grd_basic2 object ∈ Objects
    @grd_basic3 driver ∈ Entities
    @grd_basic4 Resource_driver(object) = driver
    @grd_basic5 from ∈ Objects
    @grd_basic6 to ∈ Objects
    @grd_basic61 from ≠ to
    @grd_basic7 object ∈ Container_contents(from)
    @grd_basic8 entity ↦ from ↦ write_a ∈ Accesses
```



```
    @grd_basic9 entity ↦ to ↦ write_a ∈ Accesses
    @grd_basic10 driver ↦ from ↦ write_a ∈ Accesses
    @grd_basic11 driver ↦ to ↦ write_a ∈ Accesses
    @grd_basic12 object ≠ from
    @grd_basic13 object ≠ to
    @grd_mic1 driver ∉ Entities_compromised ⇒
         integrity_level(object) ↦ integrity_level(entity) ∈ le_il
    @grd_mic2 integrity_level(object) ↦ integrity_level(driver) ∈ le_il
    @grd_mic3 integrity_level(object) ↦ integrity_level(to) ∈ le_il
  then
    @act_basic1 Container_contents ≔ Container_contents ∃
           {from ↦ (Container_contents(from) \ {object}),
            to ↦ (Container_contents(to) ∪ {object})}
    @act_basic2 Flows :| (driver ∈ Entities_compromised ∧ Flows' = Flows ∪
       {entity ↦ object ↦ write_m, entity ↦ driver ↦ write_m}) ∨
                          (driver ∉ Entities_compromised ∧ Flows' = Flows)
end
```

Событие `delete_object` определяет, как сущность `entity` удаляет объект `object` из контейнера `container` посредством драйвера `driver`. Для удаления объекта `object` из области определения функций `Container_contents`, `integrity_level` и `Resource_driver` используется оператор domain subtraction ⩤.

```
event delete_object
  any
    entity
    object
    driver
    container
  where
    @grd_basic1 entity ∈ Entities
    @grd_basic2 object ∈ Objects
    @grd_basic21 driver ∈ Entities
    @grd_basic3 container ∈ Objects
    @grd_basic4 object ∈ Container_contents(container)
    @grd_basic41 Resource_driver(object) = driver
    @grd_basic5 entity ↦ container ↦ write_a ∈ Accesses
    @grd_basic51 driver ↦ container ↦ write_a ∈ Accesses
    @grd_basic6 Container_contents(object) = ∅
    @grd_mic1 integrity_level(object) ↦ integrity_level(entity) ∈ le_il
    @grd_mic2 integrity_level(object) ↦ integrity_level(driver) ∈ le_il
  then
      @act_basic1 Objects ≔ Objects \ {object}
      @act_basic2 Objects_compromised :| (object ∈ Objects_compromised ∧
             Objects_compromised' = Objects_compromised \ {object}) ∨
                                          (object ∉ Objects_compromised ∧
             Objects_compromised' = Objects_compromised)
      @act_basic3 Accesses ≔ Accesses \
              {s ↦ o ↦ r | s ∈ Entities ∧ o = object ∧ r ∈ Access_types}
```

```
      @act_basic4 Flows ≔ (Flows ∪
              {e ↦ d ↦ w ∣ e = entity ∧ d = driver ∧ d ∈ Entities_compromised
∧ w = write_m}) ∖ ({s ↦ o ↦ w ∣ s ∈ Entities ∪ Objects ∧ o = object ∧ w =
write_m} ∪ {o ↦ s ↦ w ∣ s ∈ Entities ∪ Objects ∧ o = object ∧ w = write_m})
      @act_basic5 Container_contents ≔ ({object} ⩤ Container_contents) ∃
           {container ↦ (Container_contents(container)∖{object})}
      @act_basic6 integrity_level ≔ {object} ⩤ integrity_level
      @act_basic7 Resource_driver ≔ {object} ⩤ Resource_driver
end
```

Событие `upgrade` определяет, как сущность `entity` изменяет уровень целостности объекта `object`, находящегося в контейнере `container`, посредством драйвера `driver`, на новое значение `il`. Операция `upgrade` должна выполняться в реальной системе только доверенными сущностями и подразумевает отсутствие информационных потоков к объекту, уровень целостности которого предполагается повысить. В противном случае источник такого информационного потока может стать менее целостным, чем приемник. В реальной системе для гарантии отсутствия таких потоков используются специальные средства (например, криптографические). Такие средства не рассматриваются в рамках модели. Поэтому при анализе модели на предмет свойств безопасности, которые она обеспечивает, предполагается, что доверенные сущности не выполняют операции, которые могут нарушить целостность системы, в частности, операцию `upgrade`. В модели это предположение отражено добавлением условия ⊥ в качестве защиты события `upgrade`. В отличие от полного исключения данного события из модели, такой подход позволяет выполнить анализ корректности определения защит и действий событий.

```
event upgrade
  any
    entity
    object
    container
    driver
    il
  where
    @grd_basic1 entity ∈ Entities
    @grd_basic2 object ∈ Objects
    @grd_basic3 container ∈ Objects
    @grd_basic4 Resource_driver(object) = driver
    @grd_basic5 object ∈ Container_contents(container)
    @grd_basic6 upgrade_privilege(entity) = TRUE
    @grd_mic1 integrity_level(object) ↦ integrity_level(entity) ∈ le_il
    @grd_mic2 il ↦ integrity_level(entity) ∈ le_il
    @grd_mic3 il ↦ integrity_level(container) ∈ le_il
    @grd_mic4 integrity_level(object) ↦ il ∈ l_il
    @grd ⊥
  then
    @act_mic1 integrity_level(object) ≔ il
end
```

Взаимодействие сущностей

Событие `call` определяет вызов сущностью `caller` метода сущности `callee` с целью получения данных от сущности `callee`.


```
event call
  any
    caller
    callee
  where
    @grd_basic1 caller ∈ Entities
    @grd_basic2 callee ∈ Entities
    @grd_mic1 (integrity_level(caller) ↦ integrity_level(callee) ∈ le_il)∨
          (¬(integrity_level(caller) ↦ integrity_level(callee) ∈ le_il) ∧
           integrity_level_r(caller) ↦ integrity_level(callee) ∈ le_il)
  then
    @act_basic1 Flows :∣
((integrity_level(caller) ↦ integrity_level(callee) ∈ le_il ∨ caller ∈
Entities_compromised) ∧ ((Flows' = Flows ∪ {callee ↦ caller ↦ write_m}
                          ∧ callee ∉ Entities_compromised) ∨
 (Flows' = Flows ∪ {callee ↦ caller ↦ write_m, caller ↦ callee ↦ write_m}
                          ∧ callee ∈ Entities_compromised))) ∨
(¬(integrity_level(caller) ↦ integrity_level(callee) ∈ le_il ∨ caller ∈
Entities_compromised) ∧ Flows' = Flows)
end
```

Событие `invoke` определяет вызов сущностью `invoker` метода сущности `invokee` с целью передачи данных сущности `invokee`.

```
event invoke
  any
    invoker
    invokee
  where
    @grd_basic1 invoker ∈ Entities
    @grd_basic2 invokee ∈ Entities
    @grd_mic1 integrity_level(invokee) ↦ integrity_level(invoker) ∈ le_il
  then
    @act_basic1 Flows :∣ (Flows' = Flows ∪ {invoker ↦ invokee ↦ write_m}
                          ∧ invoker ∉ Entities_compromised) ∨
                         (Flows' = Flows ∪ {invoker ↦ invokee ↦ write_m,
invokee ↦ invoker ↦ write_m} ∧ invoker ∈ Entities_compromised)
end
```

Возникновение неявных информационных потоков

Событие `pass` определяет передачу данных сущностью `z` из объекта `x` в сущность или объект `y`.

```
event pass
  any
    x
    z
    y
  where
    @grd_basic1 z ∈ Entities
    @grd_basic2 x ∈ Objects
    @grd_basic3 y ∈ Entities ∪ Objects
```

```
    @grd_basic4 x ≠ y
    @grd_basic5 (z ↦ y ↦ write_m) ∈ Flows
    @grd_basic6 (z ↦ x ↦ read_a) ∈ Accesses
    @grd_mic1 ¬((¬(integrity_level(z) ↦ integrity_level(x) ∈ le_il) ∧
                  integrity_level_r(z) ↦ integrity_level(x) ∈ le_il) ∨
   (¬(integrity_level(z) ↦ integrity_level(Resource_driver(x)) ∈ le_il) ∧
    integrity_level_r(z) ↦ integrity_level(Resource_driver(x)) ∈ le_il))
    ∨ z ∈ Entities_compromised
  then
    @act_basic1 Flows ≔ Flows ∪ {x ↦ y ↦ write_m}
end
```

Аналогично определены еще два события: `post`, описывающее неявную передачу данных сущностью `x` сущности `y` через объект `z`, и `find`, описывающее появление данных от сущности `x` в объекте `y` (или их прием сущностью `y`) посредством сущности `z`.

Распространение зоны захвата контроля

События `control_e` и `control_o` описывают захват контроля над сущностями и объектами, соответственно. В случае, если захват контроля осуществляется над драйвером объектов, контроль также захватывается и над всеми объектами, которыми управляет этот драйвер. Множество таких объектов представлено с использованием отношения, обратного функции `Resource_driver`.

```
event control_e
  any
    x
    y
  where
    @grd_basic1 x ∈ Entities \ Entities_compromised
    @grd_basic2 y ∈ Entities_compromised ∪ Objects_compromised
    @grd_basic3 y ↦ x ↦ write_m ∈ Flows
  then
    @act1 Entities_compromised ≔ Entities_compromised ∪ {x}
  @act2 Objects_compromised ≔ Objects_compromised ∪ Resource_driver~[{x}]
end
event control_o
  any
    x
  where
    @grd_basic1 x ∈ Objects \ Objects_compromised
    @grd_basic2 ∃y · y ∈Entities_compromised ∪Objects_compromised ∧
                y ↦x ↦write_m ∈Flows
  then
    @act1 Objects_compromised ≔ Objects_compromised ∪ {x}
end
```

4.3 Свойства безопасности, обеспечиваемые моделью

Основное свойство безопасности, гарантируемое моделью, утверждает, что либо возникающие информационные потоки направлены от не менее целостных сущностей или объектов, либо расширение зоны захвата контроля носит строго ограниченный характер. Ограничение заключается в том, что в этом случае в множестве захваченных компонентов

всегда уже имеется сущность с уровнем целостности большим либо равным уровню целостности компонента, к которому реализован информационный поток:

```
@main_safety_prop
    ∀u, v · u ∈ Entities ∪ Objects ∧ v ∈ Entities ∪ Objects ∧
          v ↦ u ↦ write_m ∈ Flows ⇒
      integrity_level(u) ↦ integrity_level(v) ∈ le_il ∨
        (∃w · w ∈ Entities_compromised ∧
              integrity_level(u) ↦ integrity_level(w) ∈ le_il)
```

Помимо основного свойства безопасности и всех инвариантов, представленных ранее, был сформулирован и доказан также ряд других свойств модели. Ниже приведены некоторые примеры.

Уровень целостности объекта всегда меньше либо равен уровню целостности драйвера этого объекта:

```
@driver_and_object_levels
    ∀x · x ∈ Objects ⇒
      integrity_level(x) ↦ integrity_level(Resource_driver(x)) ∈ le_il
```

Минимальный уровень целостности объектов, к которым сущность может обращаться на чтение, всегда меньше либо равен обычному уровню целостности этой сущности:

```
@integrity_levels_prop
    ∀x · x ∈ Entities ⇒ integrity_level_r(x) ↦ integrity_level(x) ∈ le_il
```

Также был определен ряд свойств касаемо иерархии объектов и так далее. Возможность автоматизированно доказывать эти свойства позволила выполнить проверку выполнимости любого интересующего свойства.

На основе сформулированных инвариантов было сгенерировано около 300 теорем (proof obligations). При этом доказательство 54% из них потребовало ручного вмешательства. Однако в большинстве случаев достаточным оказалось проведение несложных манипуляций и вызов программных решателей и SMT-солверов. Однако иногда процесс доказательства был трудоемким, состоящим из множества шагов (упрощений, применений правил логического вывода, разбиений на случаи, введения дополнительных лемм). Особенно можно отметить случаи, когда доказываемая теорема неверна. В этих случаях процесс ее доказательства позволял прийти к утверждениям, достаточно ясно проясняющим проблему, что, в частности, привело к более глубокому пониманию модели. Был исправлен ряд неточностей начальных версий модели на естественном и математическом языках.

Таким образом, возможность автоматизированной проверки выполнения свойств модели позволила доказать множество теорем и провести большое количество экспериментов по модификации модели и определению ее свойств, что при ручном подходе было бы чрезмерно трудоемким. С другой стороны, доказательство в Rodin представляется в виде большого дерева, отражающего, в том числе, множество механизированных шагов доказательства. Видится, что такое представление не может быть полноценной заменой ручного доказательства в том смысле, что последнее может рассматриваться как ясное и лаконичное объяснение, почему выполняются основные свойства модели.

5. Заключение

В работе формализована модель мандатного контроля целостности, реализованная в микроядерной операционной системе KasperskyOS. Выбранный язык Event-B позволил получить формальное описание модели, достаточно близкое к исходному описанию на математическом и естественном языках, и при этом свободное от ошибок и неточностей, которые возникали при использовании естественного языка. Сформулированы и доказаны свойства безопасности, обеспечиваемые моделью. Опыт доказательства этих свойств

является положительным. Достаточно высокая степень автоматизации при доказательстве позволила найти ответы на множество вопросов касаемо свойств модели, что было бы чрезмерно трудоемко при ручном подходе. В то же время, для детального объяснения, почему выполняются ключевые свойства модели, ручное доказательство видится более подходящим.

Список литературы / References

[1]. Jaeger T. Operating System Security. Morgan and Claypool Publishers, 2008, 220 p.

[2]. Landwehr C. Formal Models for Computer Security. ACM Computing Surveys, vol. 13, issue 3, 1981, pp. 247-278.

[3]. Федеральная служба по техническому и экспортному контролю. Информационное сообщение о требованиях по безопасности информации, устанавливающих уровни доверия к средствам технической защиты информации и средствам обеспечения безопасности информационных технологий от 29 марта 2019 г. / Federal Service for Technical and Export Control. Announcement on Information Security Requirements Establishing Levels of Confidence in Information Protection Means and Information Technology Security Means. URL: https://fstec.ru/normotvorcheskaya/informatsionnye-i-analiticheskie-materialy/1812-informatsionnoe-soobshchenie-fstek-rossii-ot-29-marta-2019-g-n-240-24-1525, accessed 2020-04-16 (in Russian).

[4]. Буренков В.С., Кулагин Д.А. Модель мандатного контроля целостности в операционной системе KasperskyOS. Труды ИСПРАН, том 32, вып. 1, 2020 г., стр. 27-56 / Burenkov V.S., Kulagin D.A. A Mandatory Integrity Control Model for the KasperskyOS Operating System. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 1, 2020, pp. 27-56 (in Russian). DOI: 10.15514/ISPRAS–2020–32(1)–2.

[5]. Abrial, J.-R. Modeling in Event-B: System and Software Engineering. Cambridge University Press. 2010, 612 p.

[6]. Event-B and the Rodin Platform. URL: http://www.event-b.org/, accessed 2020-04-16)

[7]. Kozachok A.V. TLA+ based access control model specification. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 5, 2018, pp. 147-162. DOI: 10.15514/ISPRAS-2018-30(5)-9.

[8]. Kim I., Kang M., Choi J., Zegzhda P. Formal Verification of Security Model Using SPR Tool. Computing and Informatics, vol. 25, no. 5, 2006, pp. 353-368.

[9]. Hu V., Kuhn D., Xie T., Hwang J. Model Checking for Verificaton of Mandatory Access Control Models and Properties. International Journal of Software Engineering and Knowledge Engineering, vol. 21, no. 1, 2011, pp. 103-127.

[10]. Zhang N., Ryan M., Guelev D. Evaluating Access Control Policies Through Model Checking. In Proc. of the International Conference on Information Security, 2005, pp. 446-460.

[11]. Devyanin P., Khoroshilov A., Kuliamin V., Petrenko A., Shchepetkov I. Using Refinement in Formal Development of OS Security Model. Lecture Notes in Computer Science, vol. 9609, 2015, pp. 107-115.

[12]. Devyanin P., Khoroshilov A., Kuliamin V., Petrenko A., Shchepetkov I. Formal Verification of OS Security Model with Alloy and Event-B. Lecture Notes in Computer Science, vol. 8477, 2014, pp. 209-313.

[13]. Девянин П.Н., Ефремов Д.В., Кулямин В.В., Петренко А.К., Хорошилов А.В., Щепетков И.В.. Моделирование и верификация политик безопасности управления доступом в операционных системах. М., Горячая линия – Телеком, 2019, 214 стр. / Devyanin P.N., Efremov D.V., Kulyamin V.V., Petrenko A.K., Khoroshilov A.V., Shchepetkov I.V. Modeling and verification of access control security policies in operating systems. M., Hotline – Telecom, 2019, 214 p. (in Russian),

[14]. Девянин П.Н., Кулямин В.В., Петренко А.К., Хорошилов А.В., Щепетков И.В. Интеграция мандатного и ролевого управления доступом и мандатного контроля целостности в верифицированной иерархической модели безопасности операционной системы. Труды Института системного программирования РАН, том 32, вып. 1, 2020, стр. 7-26 / Devyanin P.N., Kuliamin V.V., Petrenko A.K., Khoroshilov A.V., Shchepetkov I.V. Integrating RBAC, MIC, and MLS in Verified Hierarchical Security Model for Operating System. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 1, 2020. pp. 7-26 (in Russian). DOI: 10.15514/ISPRAS–2020–32(1)–1.

[15]. Hussain S., Farid S., Alam M., Iqbal S., Ahmad S. Modeling of Access Control System in Event-B. The Nucleus, vol. 55, no. 2, 2018, pp. 74-84.

[16]. Romanovsky A., Thomas M. (Editors). Industrial Deployment of System Engineering Methods. Springer-Verlag, 2013. 274 pp.

[17]. Alves-Foss J., Oman P., Taylor C. The MILS Architecture for High-Assurance Embedded Systems. International Journal of Embedded Systems, vol. 2, no. 3/4, 2006, pp. 239-247.

Информация об авторе / Information about the author

Владимир Сергеевич БУРЕНКОВ – кандидат технических наук, разработчик-исследователь. Сфера научных интересов: модели программно-аппаратных систем, формальные методы верификации.

Vladimir Sergeevich BURENKOV – PhD, research developer. Research interests: models of computer systems, formal verification methods.