Подход к реализации системы верифицированного исполнения программного кода

А.В. Козачок <a.kazachok@academ.mks.rsnet.ru>
Е.В. Кочетков <e.kochetkov@academ.msk.rsnet.ru>
Академия Федеральной службы охраны Российской федерации,
302034, Россия, г. Орёл, ул. Приборостроительная, д. 35

Аннотация. В настоящей статье представлено описание технической реализации системы верифицированного исполнения программного кода. Функциональным предназначением данной системы является проведение исследования произвольных исполняемых файлов операционной системы в условиях отсутствия исходных кодов с целью обеспечения возможности контроля исполнения программного кода в рамках заданных функциональных требований. Описаны предпосылки создания такой системы, дан порядок действий пользователя по двум типовым сценариям использования. Представлено общее описание архитектуры построения системы и используемые для ее реализации программные средства, а также механизм взаимодействия элементов системы. Рассмотрен модельный пример реализации такой системы, демонстрирующий возможность гибкого задания комплекса функциональных ограничений, применение которых основывается на атрибуте времени совершения операции. В завершении статьи приведено краткое сравнение с наиболее близкими аналогами.

Ключевые слова: формальная верификация, автомат безопасности, контролируемое выполнение, вредоносное программное обеспечение

DOI: 10.15514/ISPRAS-2017-29(6)-1

Для цитирования: Козачок А.В., Кочетков Е.В. Подход к реализации системы верифицированного исполнения программного кода. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 7-24. DOI: 10.15514/ISPRAS-2017-29(6)-1

1. Введение

Развитие общества и государства влечет повсеместное внедрение информационных технологий во все сферы государственного управления и жизнеобеспечения общества, в том числе и при построении автоматизированных систем управления технологическими процессами (АСУ ТП), функционирующими на объектах критической информационной

инфраструктуры (КИИ). Защита таких объектов относится к стратегическим целям государства в области обеспечения информационной безопасности [1].

Достижение этой цели невозможно без рассмотрения вопросов информационной безопасности, связанное с риском появления ранее неизвестных угроз. К ним относится оказание деструктивного программного воздействия на элементы сетевой инфраструктуры объекта КИИ, что может привести к нарушению штатного режима функционирования АСУ ТП.

Вопросы обеспечения информационной безопасности информационных систем органов государственной власти в значительной степени проработаны, так как они непосредственно или косвенно касаются защиты конфиденциальной информации. Внимание к вопросам обеспечения информационной безопасности объектов КИИ, включающих информационные системы, вычислительные сети, АСУ ТП, функционирующих в оборонной, топливной, атомной и других областях, существенно возросло за последние годы, что обусловлено рядом инцидентов, связанных с нарушением штатного режима функционирования таких объектов [2].

Характерной особенностью современных компьютерных атак на объекты КИИ является их долговременность, целенаправленность и комплексность – APT-атаки (Advanced Persistent Thread) [3], суть которых заключается в проведении злоумышленниками ряда подготовительных мероприятий по комплексной оценке защищенности информации, циркулирующей в КИИ, включающих идентификацию используемых программно-аппаратных средств защиты информации, маршрутов ее движения, порядка обработки и хранения, а также проводимых организационных мероприятий по ее защите от несанкционированного доступа. Такой подход злоумышленников к организации атак позволяет обнаруживать в системе защиты информации наиболее уязвимые элементы и использовать их для проникновения в сетевую инфраструктуру объекта КИИ.

Очевидно, что для оказания деструктивных программных воздействий на критические процессы, протекающие в объектах КИИ, без учета внутреннего нарушителя или с его частичным привлечением, применяется специализированное вредоносное программное обеспечение (ВПО) в совокупности с элементами социальной инженерии [4].

Основным средством противодействия ВПО в сетях объектов КИИ, совместно с другими механизмами защиты, являются средства антивирусной защиты. Согласно экспериментальной оценке современных средств антивирусной защиты вероятности ошибок первого и второго рода в применяемых в них механизмов обнаружения ВПО составляют порядка 10^{-3}–10^{-4}. Однако даже при этом существенное количество образцов ВПО остаются необнаруженными [5].

По мнению авторов, наиболее перспективным является разработка механизмов, базирующихся на «запрещающей» стратегии разделения полномочий и исполняемых файлов [6], ввиду того, что реализуемые в ее рамках решения

позволяют существенно повысить уровень доверия к контенту, приходящему из внешней среды.

2. Предшествующие работы

Для решения данной задачи авторами предлагается реализация системы верифицированного исполнения программного кода (СВИПК), базирующиеся на применении метода формальной верификации «Model checking» для проверки соответствия исполняемого файла заданной спецификации безопасности как на статическом, так и на динамическом этапе проверки [6,7]. Ее применение позволяет допускать к использованию только такие программы, уровень доверия к которым подтверждается формальным математическим доказательством и непрерывным контролем за их выполнением. Работа СВИПК представляет собой последовательное выполнение ряда этапов, реализованных в виде отдельных блоков и представленных на рис. 1.

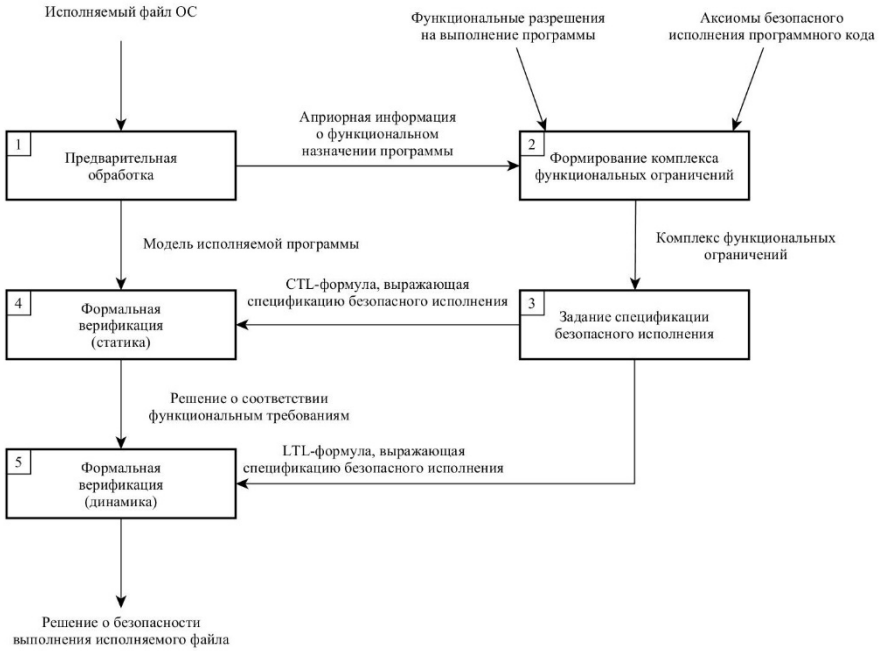

Рис. 1. Функциональная схема системы верифицированного исполнения программного кода
Fig. 1. Functional diagram of the system of verified program code execution

На вход блока 1 подается исследуемый исполняемый файл, безопасность использования которого необходимо установить. В данном блоке происходит проверка на наличие в коде программы конструкций самомодификации

(включая упаковщики) и механизмов защиты кода программы от анализа. Далее происходит преобразование исполняемого файла из бинарного представления в модель Крипке. Кроме того, на этом этапе осуществляется сбор и систематизация априорных сведений о функциональном предназначении программы, которые подаются на вход блока 2. В блоке «Формирование комплекса функциональных ограничений» на основе априорных сведений о функциональном предназначении программы формируется перечень ограничений на ее функциональные возможности, выполнение которых необходимо для ее безопасного использования. На 3 этапе осуществляется процесс преобразования комплекса функциональных ограничений в формулы темпоральной логики, выражающие требуемое свойство безопасного исполнения. В блоке 4 происходит формальная верификации модели исполняемого файла, построенной в блоке 1, на соответствие функциональным требованиям, сформулированным в блоке 2 методом формальной верификации «Model checking». В блоке 5 производится верификация исполнения программы на каждом шаге путем проверки трассы на соответствие спецификации безопасного исполнения, выраженной LTL-формулой. Выходом данного блока является решение о безопасном выполнении исполняемого файла на динамическом этапе проверки.

Для обеспечения масштабируемости при описании поведения программы авторами разработана модель функционирования процесса в операционной системе (ОС), основанная на применении субъектно-объектного подхода [8]. Ее особенностью является высокоуровневая абстракция взаимодействия между процессом и ресурсами ОС, что позволяет применить полученные на ее основе результаты к широкому классу аналогичных систем. Применение данной модели необходимо для совершения перехода от объекта реального мира (процесса) к формальной модели, позволяющей произвести процедуру верификации.

Для задания функциональных ограничений на выполнение программой некоторых действий с возможностью однозначного перехода к формулам темпоральной логики авторами предлагается формальный логический язык описания таких требований на основе модели функционирования процесса в ОС [9].

Основная идея формальной верификации применительно к обнаружению вредоносного кода состоит в построении формальной (математической) модели исследуемой программы, которая отражает (моделирует) ее возможное поведение в операционной системе. Требования корректности безопасного поведения при этом описываются в виде спецификации, которая отражает рамки разрешенного поведения программы. На основе спецификаций и модели исполняемого файла, используя метод «Model checking», осуществляется проверка, согласуется ли возможное поведение с разрешенным. Поскольку происходит именно математическая верификация, решение о согласованности, то есть соответствии возможного поведения требуемому, является корректным.

В данном контексте под возможным поведением программы понимается модель программы, построенная на основе многократного запуска исполняемого файла с использованием интеллектуального фаззера [10]. Требуемое поведение представляет собой систему ограничений на функциональные возможности программы, то есть разрешенное поведение.

3. Архитектура системы верифицированного исполнения программного кода

Система верифицированного исполнения программного кода (СВИПК) представляет собой комплекс программных средств, обеспечивающих контролируемое безопасное исполнение пользовательских приложений в среде эмулятора. Архитектура СВИПК представлена на рис. 2. Она состоит из следующих компонентов:

- базовая ОС;
- программное средство управления СВИПК;
- полносистемный эмулятор ОС.

Выбор базовой ОС определяется удобством конфигурации политик безопасности и стабильностью работы. Она является необходимой платформой для функционирования программного средства управления СВИПК и полносистемного эмулятора. Базовая ОС должна быть сконфигурирована таким образом, чтобы пользователь имел возможность работы только с программным средством управления СВИПК и не имел возможности вносить изменения в конфигурацию базовой ОС, состав входящих в нее программных средств и т. д.

Программное средство управления СВИПК представляет собой оконное кроссплатформенное приложение, которое является связующим звеном между пользователем и эмулятором гостевой ОС. Назначением данного программного средства является выполнение следующих функций:

- запуск полносистемного эмулятора по требованию пользователя;
- создание, изменение, удаление профилей программ;
- построение модели исследуемой программы на основе трасс исполнения, полученных интеллектуальным фаззером;
- задание функциональных ограничений на работу программ;
- поддержание постоянного сетевого соединения динамически подключаемым модулем, находящимся в адресном пространстве отслеживаемых приложений для осуществления непрерывного взаимодействия.

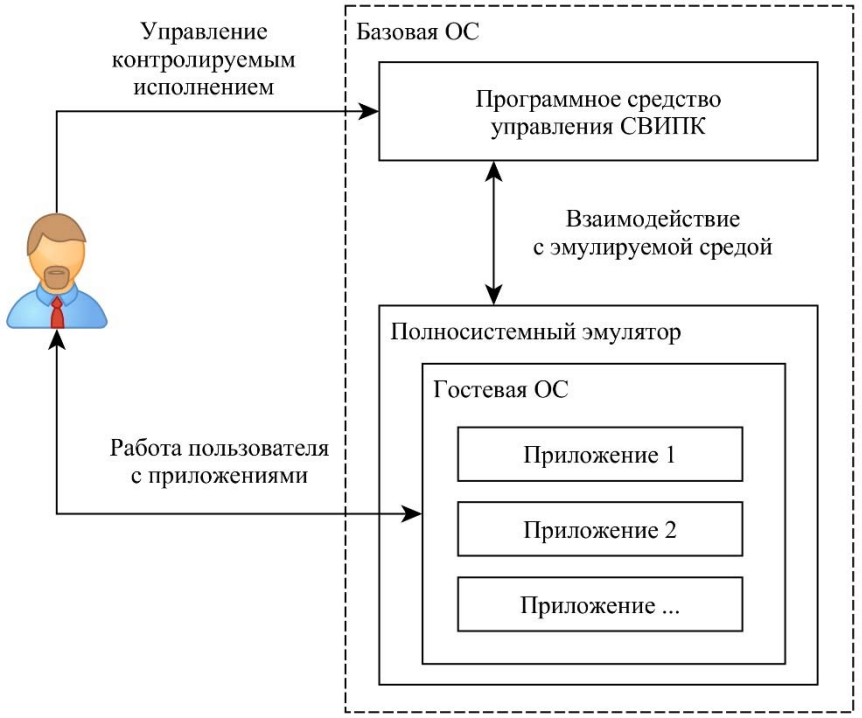

Рис. 2. Архитектура системы верифицированного исполнения программного кода
Fig. 2. Architecture of the system of verified program code execution

Полносистемный эмулятор ОС должен позволять полностью имитировать работу реального оборудования. Он запускает на исполнение гостевую ОС и создает для нее виртуальное окружение. На этом этапе производится эмуляция работы процессора, оперативной памяти и другого аппаратного обеспечения различных архитектур и платформ. Работа исследуемых приложений происходит через исполнение инструкций процессора гостевой ОС. Выбор полносистемного эмулятора для реализации СВИПК определяется следующими факторами:

- широкий диапазон поддерживаемого оборудования;
- возможность детальной конфигурации состава и параметров периферийного оборудования;
- наличие протокола, позволяющего в реальном времени управлять работой полносистемного эмулятора из внешнего приложения;
- открытый исходный код, позволяющий произвести соответствующие проверки на отсутствие недекларированных возможностей.

Ключевым фактором является поддержка внешних плагинов, позволяющих разрабатывать дополнительные модули и интегрировать их динамически в рабочую среду полносистемного эмулятора.

Все исполняемые файлы в эмулируемой ОС условно разделим на две категории:

- априорно безопасные;
- априорно небезопасные.

К первой категории исполняемых файлов относятся такие файлы, которые входят в штатный дистрибутив ОС, то есть заложены разработчиками и априорно не должны нанести вред. Ко второй категории относятся исполняемые файлы, попавшие в файловое пространство ОС из внешней среды, созданные сторонними разработчиками, достоверной информации об использовании которых нет.

Для каждой используемой программы создается «Профиль», включающий следующие сведения:

- контрольная сумма исполняемого файла;
- модель безопасного исполнения.

Профиль программы необходим для исключения этапов предварительной обработки и статической верификации исполнения при повторном использовании программы.

На рис. 3 представлена обобщенная архитектура СВИПК.

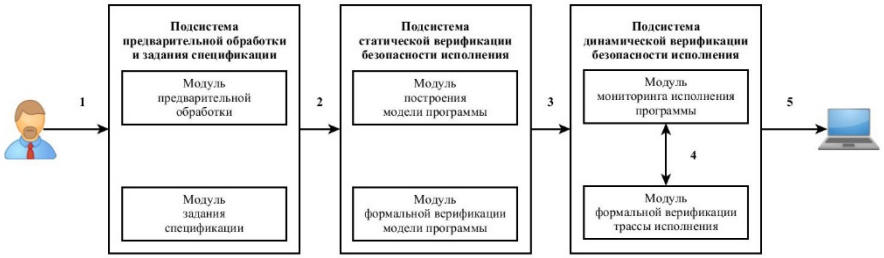

Рис. 3. Обобщенная архитектура системы верифицированного исполнения программного кода

Fig. 3. Generalized architecture of the system of verified program code execution

Использование СВИПК для запуска приложения возможно по следующим двум сценариям в зависимости от того, было ли проведено соответствующее исследование безопасности ранее:

- «первоначальная проверка»;
- «регулярное использование».

Этапы работы в рамках первого сценария производятся в отношении исполняемых файлов программ, впервые запускаемых пользователем. Данный сценарий работы предполагает осуществление следующих действий:

1. Пользователь осуществляет запуск исполняемого файла. Модуль «Предварительной обработки» осуществляет проверку на соответствие минимальным требованиям безопасности. С помощью интерфейса модуля «Задания спецификации» и с учетом требований принятой политики безопасности пользователь формирует спецификацию, ограничивающую функциональные возможности исследуемой программы.
2. В рамках подсистемы «Статической верификации безопасности исполнения» производится построение модели программы и ее формальная верификации методом «Model checking» на предмет соответствия заданной спецификации.
3. «Подсистемой динамической верификации безопасности исполнения» осуществляется перехват системных вызовов, совершаемых исследуемой программой и построение трассы исполнения.
4. Модулем «Формальной верификации трассы исполнения» реализуется процедура верификации трассы исполнения программы на предмет соответствия комплексу функциональных ограничений на динамическом этапе проверки.
5. Пользователь осуществляет непосредственную работу с программой.

Сценарий «регулярного использования» отличается тем, что работа пользователя с заданной программой начинается с этапа 4.

4. Прототип системы верифицированного исполнения программного кода

Для технической реализации прототипа СВИПК были выбраны программные средства, удовлетворяющие обозначенному выше описанию.

В качестве базовой ОС предлагается использование ОС семейства Linux, а именно «Ubuntu 16.04», ее выбор определяется удобством конфигурации политик безопасности. В качестве «гостевой» ОС предполагается использование «Windows XP SP3» (для демонстрации работы прототипа).

В качестве полносистемного эмулятора предполагается использование «Qemu», ключевой особенностью которого является поддержка системы плагинов, разработанная «Институтом системного программирования им. В.П. Иванникова РАН» [11]. Разработчиками создан механизм «signal-subscriber», позволяющий реагировать на события, генерируемые ядром «Qemu». Взаимодействие программного средства управления СВИПК с полносистемным эмулятором производится посредством протокола QMP (Qemu machine protocol) [12]. Данный протокол позволяет внешним приложениям управлять экземпляром виртуальной машины «Qemu» путем

обмена сообщениями в формате JSON. На рис. 4 представлена структурная схема прототипа СВИПК.

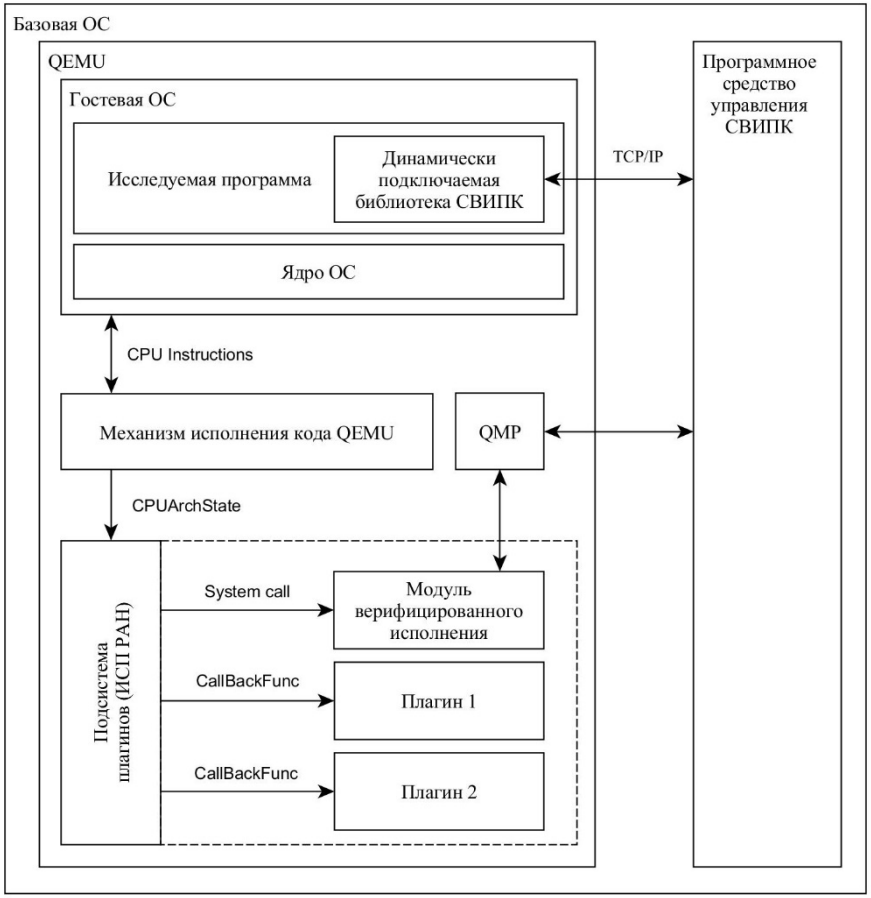

Рис. 4. Структурная схема прототипа системы верифицированного исполнения программного кода
Fig. 4. Structural scheme of the verified program code execution system prototype

При запуске любой исполняемой программы в ее адресное пространство проецируется специальная динамически подключаемая библиотека. В момент вызова функции точки входа происходит сбор информации о параметрах запуска и самом исполняемом файле. Затем собранная информация посредством сетевой подсистемы гостевой ОС передается программе управления СВИПК. Если контрольная сумма исполняемого файла существует в базе профилей множества «Априорно безопасных» программ, то

функционирование гостевой ОС продолжается в штатном режиме. Таким образом, программы, входящие в стандартный дистрибутив ОС, не контролируются СВИПК.

Если контрольная сумма отсутствует в обозначенном выше множестве, то СВИПК переходит в режим контролируемого исполнения. В случае существования профиля исполняемого файла по заданной контрольной сумме, происходит его запуск в контролируемой автоматом безопасности среде. Если такого профиля нет, то происходит работа по описанному выше сценарию «первоначальной проверки».

Автомат безопасного исполнения представляет собой плагин, обрабатывающий каждый системный вызов, сгенерированный приложением. В случае выхода контролируемого приложения за рамки спецификации безопасности, работа эмулятора приостанавливается, а пользователю выводится сообщение, содержащее причину блокировки, при этом приложение пользователя завершается.

База спецификаций хранится в виде набора файлов в файловой системе базовой ОС и доступна для программы управления СВИПК. Для каждого запускаемого в эмуляторе приложения происходит загрузка соответствующей спецификации в автомат безопасности, интегрированный в виде плагина в экземпляр «Qemu», через протокол QMP.

Предложенная СВИПК предоставляет следующие возможности:

- задание функциональных ограничений на выполнение программы в рамках эмулируемой ОС;
- формальная верификация функциональных требований до запуска исполняемого файла в рабочей среде пользователя;
- контролируемое выполнение пользовательских приложений в эмулируемой среде под непрерывным контролем автомата безопасного исполнения.

6. *Модельный пример*

Согласно «Разрешающей» стратегии построения модели исполняемой программой, все совершаемые ею действия изначально запрещены. Для недопущения совершения процессом действий, которые могут привести к потенциально неблагоприятным последствиям каждый шаг исполнения программы должен быть верифицирован на соответствию базису аксиом «Безопасного исполнения программного кода» и функциональных разрешений. Для описания выполнения этого требования вводится предикат:

$$isDynSecure(trace^*, FA) = \begin{cases} \forall s \in trace^* \vDash AX_{sec} \vee FA, & true \\ \text{иначе}, & false' \end{cases}$$

где: $trace^*$ – расширенная трасса исполнения; AX_{sec} – аксиомы безопасного исполнения; FA – множество функциональных разрешений.

Дополненная трасса исполнения описывается следующим выражением:

$$trace^* = trace \parallel \omega_{curr},$$

где: $trace$ – последовательность действий процесса, совершенных на настоящий момент; ω_{curr} – действие процесса, совершаемое на следующем шаге.

Базис аксиом «Безопасного исполнения программного кода» представлен в табл. 1.

Табл. 1. Базис аксиом «Безопасного исполнения программного кода»
Table 1. The basis of axioms for secure execution program code

Формула	Интерпретация
$p^* \xrightarrow{c,r,w,d} m^3$	Выполнение операций над памяти только в своем адресном пространстве
$p^* \xrightarrow{c,r,w,d} e^5$	Выполнение операций над файловыми объектами только в своем домашнем каталоге
$p^* \xrightarrow{r} e^2$	Чтение системных файловых объектов и конфигурации
$p^* \xrightarrow{r,w} e^3$	Чтение и запись параметров пользовательского окружения
$p_i^* \xrightarrow{d} p_i^*$	Процесс может завершать только свою работу

С целью предотвращения передачи конфиденциальной информации через сетевые соединения и сохранения при этом возможности выхода программы в сеть, например для получения обновления, вводится следующе функциональное разрешение: «Разрешается чтения каталога или файловых объектов пользователя отличного от данного пользовательским процессом, только если в будущем не будут созданы любые сетевые соединения». Данное функциональное разрешение выражается следующим образом:

$$p^3 \xrightarrow{r} e^4 \wedge \neg F(p^3 \xrightarrow{c} n^1) \qquad (1)$$

Табл. 2. Таблица истинности функционального разрешения
Table 2. Truth table of the functional authorization

$p^3 \xrightarrow{r} e^4$	$F(p^3 \xrightarrow{c} n^1)$	$\neg F(p^3 \xrightarrow{c} n^1)$	$p^3 \xrightarrow{r} e^4 \wedge \neg F(p^3 \xrightarrow{c} n^1)$
0	0	1	**0**
0	1	0	**0**
1	0	1	**1**
1	1	0	**0**

Формула функционального разрешения принимает истинное значение только в том случае, если левая и правая ее части будут истинны одновременно. В соответствии с таблицей истинности (табл. 2), это достижимо только если

процесс совершит действие $p^3 \xrightarrow{r} e^4$ и при этом относительно этого действия в будущем не совершится действие $F(p^3 \xrightarrow{c} n^1)$, во всех остальных случаях результат вычисления функционального разрешения является ложным.

Таким образом итоговая формула для проверки безопасности исполнения программы в динамике примет следующий вид:

$$G(AX_{sec} \vee [p^3 \xrightarrow{r} e^4 \wedge \neg F(p^3 \xrightarrow{c} n^1)]) \qquad \text{а}$$

На рис. 5 представлена модель исполняемой программы, построенная в блоке 1 «Предварительная обработка» с использованием интеллектуального фазера.

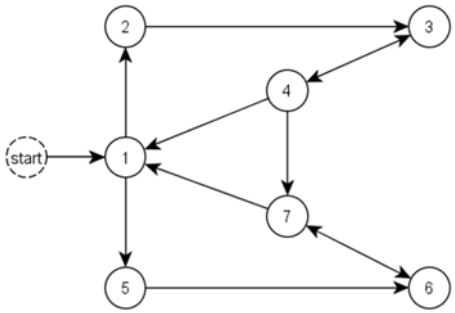

Рис. 5. Модель программы
Fig. 5. The program model

Вершины представляют собой действие процесса, ребра отображают возможность программы совершить действие процесса из соответствующего состояния. Соответствие номера вершины и действия процесса представлены в табл. 3.

Табл. 3. Соответствие номеров вершин и действий процесса
Table 3. States and process actions

№	Действие процесса	№	Действие процесса
1	$p^* \xrightarrow{c} m^3$	5	$p^* \xrightarrow{c} n^*$
2	$p^* \xrightarrow{c} e^*$	6	$p^* \xrightarrow{w} n^*$
3	$p^* \xrightarrow{r} e^*$	7	$p^* \xrightarrow{r} n^*$
4	$p^* \xrightarrow{w} e^*$		

Верифицированное исполнение программы, не нарушающее комплекс функциональных ограничений, пошагово представлено в табл. 3.

Сценарий работы программы: после запуска программа выделяет в своем адресном пространстве некоторую область памяти, далее создает некоторый файловый объект в своем домашнем каталоге, производит чтение информации

из каталога другого пользователя, производит запись в своих файловые объекты, после этого завершает работу.

Табл. 4. Верифицируемое исполнение (Легитимно)
Table 4. Verified execution (Legitimate)

№	ω_{curr}	$trace^*$	Шаг трассы выполнения	AX_{sec}	FA	$isDynSecure$
1	$p^3 \xrightarrow{c} m^3$	(1)	1	**1**	**0**	**1**
2	$p^3 \xrightarrow{c} e^5$	1-(2)	1	**1**	**0**	**1**
			2	**1**	**0**	
3	$p^3 \xrightarrow{r} e^4$	1-2-(3)	1	**1**	**0**	**1**
			2	**1**	**0**	
			3	**0**	**1**	
4	$p^3 \xrightarrow{w} e^5$	1-2-3-(4)	1	**1**	**0**	**1**
			2	**1**	**0**	
			3	**0**	**1**	
			4	**1**	**0**	

Верификатор «Model checking» проверят для каждого шага выполнимость логической формулы стоящей в предикате G, то есть выполнение свойства на каждом шаге. Интерпретация, данного выражения с точки зрения безопасности выполнения программного кода заключается в том, что на каждом шаге выполнения программы должно быть правило, которое данную операцию разрешает, если такого правила нет, то весь предикат принимает отрицательное значение, что свидетельствует о попытке совершения действия, явным образом не разрешенного, а значит запрещенного.

Первым выполняемым действием процесса является $p^3 \xrightarrow{c} m^3$ – выделение памяти в своем адресном пространстве для дальнейшего использования. На данном шаге на вход верификатору поступает трасса исполнения, состоящая из одного действия, поэтому она проводится только один раз для одного состояния. Базис аксиом «Безопасного исполнения программного кода» при этом не нарушается, так как действия, совершаемое процессом, входит в него явно, что отражено в табл. 1, при этом функциональные разрешения FA не выполняются, так как ω_{curr} не входит в формулу FA. В результате верификации первого действия программы проверена его безопасность, так как для заданной трассы исполнения выполняется предикат $isDynSecure$.

Шаг два проверяется аналогично первому. На третьем шаге работы программы при верификации 3 шага исполнения, действие процесса $p^3 \xrightarrow{r} e^4$ не входит в базис аксиом «Безопасного исполнения программного кода» ввиду этого базис не выполняется, но в дополнение к нему существует функциональное разрешение (выр. 1). Функциональное разрешение в данном случае принимает истинное значение в соответствии с таблицей истинности (табл. 2). Так как для

вычисления предиката $isDynSecure$ необходимо выполнение базиса аксиом или функциональных разрешений, то итоговый результат для проверки данного шага принимает истинное значение. По итогам проверки всех этапов работы принимается решение для всей трассы целиком. Проверка на шаге четыре проводится аналогична предыдущим шагам.

Пошаговое выполнение программы, нарушающей в процессе работы, комплекс функциональных ограничений представлено в табл. 5.

Сценарий работы программы: после запуска программа выделяет в своем адресном пространстве некоторую область памяти, далее создает некоторый файловый объект в своем домашнем каталоге, производит чтение информации из каталога другого пользователя, создает несанкционированное подключение к узлу глобальной сети.

Табл. 5. Верифицируемое исполнение (Нарушение)
Table 5. Verified execution (Violation)

№	ω_{curr}	$trace^*$	Шаг трассы выполнения	AX_{sec}	FA	$isDynSecure$
1	$p^3 \xrightarrow{c} m^3$	(1)	1	**1**	**0**	**1**
2	$p^3 \xrightarrow{c} e^5$	1-(2)	1	**1**	**0**	**1**
			2	**1**	**0**	
3	$p^3 \xrightarrow{r} e^4$	1-2-(3)	1	**1**	**0**	**1**
			2	**1**	**0**	
			3	**0**	**1**	
4	$p^3 \xrightarrow{c} n^1$	1-2-3-(4)	1	**1**	**0**	**0**
			2	**1**	**0**	
			3	**0**	**0**	
			4	**1**	**0**	

Первые три шага выполнения программы происходят как и в случае с легитимной программой. Выполнение четвертом шаге при верификации третьего состояния не выполняется базис аксиом «Безопасного исполнения программного кода» и функциональное разрешение, так как выражение $\neg F(p^3 \xrightarrow{c} n^1)$ принимает ложное значение. В итоге проверки по четвертому шагу предикат $isDynSecure$ принимает ложное значение, что свидетельствует о нарушении модели безопасного использования программного кода. В этом случае выполнение программы прерывается.

Рассмотренный модельный пример демонстрирует принципиальную возможности применения методов формальной верификации для обеспечения верифицированного исполнения программного кода в соответствии с функциональными разрешениями и базисом аксиом «Безопасного исполнения программного кода.

7. Сравнение с аналогами

Для предложенного решения, реализующего СВИПК найти прямые аналоги не удалось. Проведенный анализ существующих средств виртуализации исполнения программного кода позволил разделить программные средства создания изолированной программной среды на два класса: «Песочницы, предназначенные выявления признаков вредоносного функционала путем анализа их поведения в изолированной программной среде» и «Песочницы, предназначенные для рядового использования» с целью предотвращения нежелательных последствий от запуска программ из недоверенных источников.

Предложенное в настоящей статье решение по предназначению относится ко второму классу, но по архитектуре реализации оно является полносистемным эмулятором.

Наиболее близкие к предлагаемому решения аналоги представлены в таблице 6.

Табл. 6. Программные средства для создания изолированной программной среды
Table 6. The sandbox software

Аналог	Платформа	Задание ограничений пользователем	Применение ограничений имеет временной атрибут
Sandboxie	Windows	из перечня	–
Time Freeze	Windows	–	–
Shade Sandbox	Windows	из перечня	–
Mbox	Linux	–	–
FireJail	Linux	+	–
Sandbox	Linux (Fedora)	–	–

Ключевым отличием предлагаемого решения от аналогов, представленных в табл. 6, является возможность гибкого разграничения доступа к ресурсам ОС на основе атрибута времени совершения процессом действия.

Применение гибкой системы функциональных ограничений с использованием атрибута времени совершения операции позволяет ограничить лишь потенциально опасные трассы исполнения программы, существенно меньше при этом ограничивая функциональные возможности исследуемой программы в целом.

Первые эксперименты по верифицируемому исполнению программного кода показали линейный рост времени, требуемого для проведения верификации трассы исполнения программы. Также для снижение временных затрат на проведения верификации трассы исполнения комплекс функциональных ограничений может быть разделен на два подмножества формул по признаку наличия операторов темпоральной логики.

8. Заключение

Предложенная СВИПК существенно повысит уровень доверия к исполняемым файлам, так как позволяет выявить потенциально опасный функционал исполняемой программы еще до ее запуска. В случае нарушения заданных функциональных ограничений пользователь может отказаться от ее допуска в сетевую инфраструктуру объекта КИИ или использовать верифицируемое исполнение предварительно задав функциональные разрешения, которое предотвратит действия, потенциально ведущие к нежелательным последствиям. Использование СВИПК возможно в рамках сетевой инфраструктуры объекта КИИ как в качестве активного контроля исполнения программ, так и в качестве средства сертификация программ перед допуском их исполнения. Ее использование позволяет повысить защищенность информации, циркулирующей в сетях КИИ, за счет использования программ, уровень доверия к которым достаточен для их безопасного использования.

Список литературы

[1]. Указ Президента Российской Федерации от 05.12.2016 № 646 «Об утверждении Доктрины информационной безопасности Российской Федерации».

[2]. Гарбук С. В., Комаров А. А., Салов Е. И. Обзор инцидентов информационной безопасности АСУТП зарубежных государств: Аналитический отчет. ЗАЩИТА ИНФОРМАЦИИ. ИНСАЙД, вып. 6, 2010 г., стр. 50-58.

[3]. Яремчук С. APT: реальность или паранойя? Системный администратор, вып. 7-8, 2012 г., стр. 52-56.

[4]. Довголенко А.А. Социальная инженерия в сети интернет. Информационная безопасность и вопросы профилактики киберэкстремизма среди молодежи Материалы внутривузовской конференции. Под редакцией Г.Н. Чусавитиной, Е.В. Черновой, О.Л. Колобовой, Сборник материалов, 2015 г., стр. 183-191, Магнитогорский государственный технический университет им. Г.И. Носова (Магнитогорск)

[5]. Козачок А.В. Распознавание вредоносного программного обеспечения на основе скрытых Марковских моделей. Диссертация на соискание ученой степени кандидата технических наук. Воронежский государственный технический университет. Орел, 2012, 209 стр.

[6]. Козачок А.В., Кочетков Е.В. Обоснование возможности применения верификации программ для обнаружения вредоносного кода. Вопросы кибербезопасности, вып. 3, 2016 г., стр. 25-32.

[7]. Cimitile A. et al. Model checking for mobile Android malware evolution. Formal Methods in Software Engineering (FormaliSE), 2017 IEEE. ACM 5th International FME Workshop on., 2017, pp. 24-30, IEEE

[8]. Козачок А.В., Кочетков Е.В. Формальная модель функционирования процесса в операционной системе. Труды СПИИРАН, вып. 2, 2017 г., стр. 78-96.

[9]. Kozachok A., Bochkov M., Lai Minh T., Kochetkov E. First order logic for program code functional requirements description. Вопросы кибербезопасности, вып. 3, 2017 г., стр. 2-7.

[10]. Jesse Hertz. Project Triforce: Run AFL on Everything! (online) Доступно по ссылке: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/projecttriforce-run-afl-on-everything/, 01.11.17
[11]. Институт системного программирования им. В.П. Иванникова РАН. Репозиторий исходного кода эмулятора «Qemu». (online). Доступно по ссылке: https://github.com/ispras/qemu, 09.10.2017
[12]. Documentation/QMP – QEMU. QEMU Machine Protocol. Доступно по ссылке: https://wiki.qemu.org/Documentation/QMP, 01.11.17

Verified program code execution system prototype

A.V. Kozachok <a.kozachok@academ.msk.rsnet.ru>
E.V. Kochetkov <e.kochetkov@academ.msk.rsnet.ru >
Academy of Federal Guard Service,
35, Priborostroitelnaya st., Oryol, 302034, Russia

Abstract. The article represented the technical implementation of the system of verified program code execution. The functional purpose of this system is to investigate arbitrary executable files of the operating system in the absence of source codes in order to provide the ability to control the execution of the program code within the specified functional requirements. The prerequisites for the creation of such a system are described, the user's operating procedure is given according to two typical usage scenarios. A general description of the architecture of the system and the software used for its implementation, the mechanism of interaction of the elements of the system are presented. The model example of implementation this system is presented. Demonstrating the flexible set of functional constraints, based on temporal attribute process action. At the end of the article given a brief comparison with the closest analogues.

Keywords: formal verification, security automata, controlled execution, malware

DOI: 10.15514/ISPRAS-2017-29(6)-1

For citation: Kozachok A.V., Kochetkov E.V. Verified program code execution system prototype. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 7-24 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-1

References

[1]. Ukaz Prezidenta Rossijskoj Federatsii ot 05.12.2016 № 646 «Ob utverzhdenii Doktriny informatsionnoj bezopasnosti Rossijskoj Federatsii» [Decree of the President of the Russian Federation of 05.12.2016 No. 646 «On Approving the Doctrine of Information Security of the Russian Federation»] (in Russian).
[2]. Garbuk S.V., Komarov A.A., Salov E.I. Overview of incidents of information security of SCADA of foreign countries: Analytical report. Zashhita informatsii. Insajd [Data protection. Inside], no. 6, 2010, pp. 50-58 (in Russian).

[3]. Yaremchuk S. APT: Reality or Paranoia. Sistemnyj administrator [System Administrator], no. 7-8, pp. 52-56 (in Russian).
[4]. Dovgolenko A.A. Social engineering in the Internet. Informatsionnaya bezopasnost' i voprosy profilaktiki kiberehkstremizma sredi molodezhi Materialy vnutrivuzovskoj konferentsii. Pod redaktsiej G.N. CHusavitinoj, E.V. CHernovoj, O.L. Kolobovoj [Proc. of Information security and issues of the prevention of cyber extremism among young people Materials of the intra-university conference]. 2015. pp. 183-191 (in Russian)
[5]. Kozachok A.V. Detection of malicious software based on hidden Markov models: PhD thesis. Oryol, 2012, 209 p. (in Russian)
[6]. Kozachok A.V., Kochetkov E.V. Using Program Verification for Detecting Malware. Cybersecurity issues, vol. 16, no. 3, 2016, pp. 25–32 (in Russian)
[7]. Cimitile A. et al. Model checking for mobile Android malware evolution. Formal Methods in Software Engineering (FormaliSE), 2017 IEEE. ACM 5th International FME Workshop on., 2017, pp. 24-30, IEEE
[8]. Kozachok A.V., Kochetkov E.V. Formal model of functioning process in the operating system. SPIIRAS Proceedings, vol. 51, issue 2, 2017, pp. 78-96 (in Russian).
[9]. Kozachok A., Bochkov M., Lai Minh T., Kochetkov E. First order logic for program code functional requirements description. Cybersecurity issues, vol. 3, issue 21, 2017, pp 2-7.
[10]. Jesse Hertz. Project Triforce: Run AFL on Everything! (online) Available at: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/projecttriforce-run-afl-on-everything/, accessed 01.11.17
[11]. Ivannikov Institute for System Programming of the RAS. Source Repository of the «Qemu». (online) Available at: https://github.com/ispras/qemu, accessed 05.09.17
[12]. Documentation/QMP – QEMU. QEMU Machine Protocol. (online) Available at: https://wiki.qemu.org/Documentation/QMP, accessed 15.10.17