Комплекс алгоритмов функционирования системы безопасного исполнения программного кода

А.В. Козачок <a.kozachok@academ.msk.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 представлен комплекс алгоритмов, составляющих основу функционирования системы безопасного 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. Представленный в работе комплекс алгоритмов включает в себя: алгоритм функционирования системы безопасного исполнения программного кода и алгоритм построения модели программы, пригодной для проведения верификации и сохраняющей свойства исходной программы.

Ключевые слова: алгоритм; вредоносное программное обеспечение; model checking; security automata

DOI: 10.15514/ISPRAS-2017-29(3)-2

Для цитирования: Козачок А.В., Кочетков Е.В. Комплекс алгоритмов функционирования системы безопасного исполнения программного кода. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 17-30. DOI: 10.15514/ISPRAS-2017-29(3)-2

1. Введение

В настоящее время становятся все более актуальными вопросы формального доказательства свойств безопасности систем. Математическое доказательство свойств безопасности позволяет доверять исследуемым объектам, но его применение возможно только к ограниченному классу объектов, а именно – позволяющих построить формализованную модель с сохранением важных для проведения исследования свойств. Актуальной задачей обеспечения информационной безопасности сетей и объектов критической информационной инфраструктуры (КИИ) является определение степени

доверия к объектам, приходящим из внешних источников [1]. Полное функциональное содержание таких объектов априорно неизвестно.

На текущий момент допуск к использованию во внутреннем контуре сети объектов недоверенного происхождения сводится к принятию решения на основе результатов антивирусного сканирования. Однако результаты тестирования средств антивирусной защиты (САВЗ) показывают, что вероятность пропуска цели достаточно высока [2]. Это связано с постоянным совершенствованием возможностей вредоносного программного обеспечения (ВПО) злоумышленниками и принципиальными ограничениями заложенных в САВЗ методов обнаружения [3]. Авторами предлагается подход к оценке степени безопасности функционирования программного кода в операционной системе (ОС) за счет построения модели программы и оценки ее соответствия заданным функциональным требованиям.

2. Предшествующие работы

В ранней работе авторов [4] предлагалось построение системы безопасного исполнения программного кода (СБИПК), являющейся расширенной композицией таких подходов к защите от ВПО, как применение метода формальной верификации "Model checking" и использование автомата безопасности для контроля над выполнением функциональных ограничений программы как на статическом, так и динамическом этапе исследования. Ключевой особенностью такой системы является построение модели безопасного исполнения программного кода (МБИПК). Под безопасностью исполнения в данном случае подразумевается выполнение ряда ограничений, выраженных на языке формальной логики и представляющих собой спецификацию, описывающую безопасное поведение программы [5].

Формальная верификация – это процесс математического доказательства соответствия определенной модели некоторым заданным свойствам (спецификации), основанный на строгих математических принципах [6]. При этом для описания свойств модели и задания требований к ней применяются специализированные языки, а для проведения верификации – специализированные программные средства – верификаторы. Одним из широко распространенных методов формальной верификации является метод "Model checking". Его суть состоит в построении некоторой модели реактивной системы (системы, способной менять свое состояние под воздействием внешних факторов); задании спецификации работы данной системы, отражающей свойства, выполнение которых требуется доказать; проведении процесса верификации. Результатом верификации является решение о согласованности модели заданной спецификации.

Вопросами обнаружения ВПО с использованием методов формальной верификации, в частности методом "Model checking", в разное время занимались такие ученые как J. Kinder, Song Fu, P. Beaucamps, A. Holzer [7–12].

Общим направлением перечисленных исследований является применение метода "Model checking" для построения модели ВПО, которая позволит в будущем обнаруживать семейства вредоносных программ данного вида. Недостатком данного подхода является принципиальная невозможность построения универсальной модели, которая бы позволила обнаруживать штаммы различных семейств ВПО.

Важным этапом при проведении такого исследования является построение модели, пригодной для проведения формальной верификации. Такой моделью является модель Крипке [6], которая описывается четверкой элементов:

$$(S, s_0, R, L), \quad (1)$$

где S – конечное множество состояний; $s_0 \subseteq S$ – начальное состояние; $R \subseteq S \times S$ – отношение переходов, которое должно быть тотальным, т.е. для каждого состояния $s \in S$ должно существовать такое состояние $s' \in S$, что имеет место $R(s, s')$; $L : S \to 2^{AP}$ – функция оценки состояния $s \in S$.

Структура Крипке представляет собой модель, которая может быть верифицирована на соответствие спецификации, выраженной с использованием операторов темпоральной логики. Для применения метода формальной верификации "Model checking" в СБИПК требуется построить модель, позволяющую произвести такую верификацию.

Состояние в модели Крипке – это мгновенное значение всех рассматриваемых в рамках модели переменных. Попытка применения данного подхода к описанию поведения бинарной программы приводит к «комбинаторному взрыву», который вызван чрезмерно большим количеством различных значений рассматриваемых переменных. Сокращение числа переменных позволит построить модель и применить к ней подход "Model checking". В работе [5] была введена модель функционирования процесса в ОС, которая основывается на принципе разделения всех взаимодействующих элементов ОС на субъекты и объекты. К субъектам относятся процессы (множество P), совершающие действия по отношению к объектам (множество O). Объектами являются ресурсы ОС и процессы, в отношении которых совершаются действия субъектами. Все объекты были разделены на следующие категории: "Процесс", "Оперативная память", "Внешняя память", "Периферийные устройства", "Сетевая подсистема". Действия субъектов по отношению к объектам были выделены следующие: $create$, $open$, $delete$, $read$, $write$. Согласно [5] поведение процесса в ОС можно описать последовательностью действий $\{s_i\}_{i \in \mathbb{N}}$, выполняемых субъектом ${p_i}^{k_p}$, каждое из которых в операторном виде можно представить как:

$${p_i}^{k_p} \xrightarrow{a_o} {o_y}^{k_o}, \quad (2)$$

где ${p_i}^{k_p}$ – субъект категории k_p с идентификатором i ; a_o – некоторое действие субъекта по отношению к объекту; ${o_y}^{k_o}$ – некоторый объект категории k_o с идентификатором y.

3. Комплекс алгоритмов

Комплекс алгоритмов безопасного исполнения программного кода включает в себя следующие алгоритмы:

- алгоритм функционирования системы безопасного исполнения программного кода, отличающийся применением расширенной композиции таких подходов к обнаружению ВПО, как формальная верификация методом "Model checking" и использование автомата безопасности;
- алгоритм преобразования бинарной исполняемой программы в структуру Крипке, позволяющую производить формальную верификацию на соответствие функциональным требованиям, отличающийся применением интеллектуального фаззера для увеличения степени покрытия бинарного кода исследуемой программы.

3.1 Алгоритм функционирования системы безопасного исполнения программного кода

Исходными данными для работы алгоритма функционирования СБИПК являются следующие параметры:

- B – исследуемый бинарный файл;
- FR – функциональные требования, заданные на формально логическом языке [13];
- L – функция оценки;
- $SCTT$ ("System Call Translate Table") – таблица соответствия системных вызовов ОС и действий процесса.

Функциональные требования по выполнению программы строятся на основе аксиом безопасного исполнения программного кода. Под аксиомой безопасного исполнения программного кода AX понимается описание параметров разрешенных действий $\{s_k\}_{k \in \mathbb{N}}$ для процесса ${p_i}^{k_P}$ во время его функционирования, позволяющее исключить потенциальную возможность выполнения программой вредоносных действий:

$$AX_j = \cup_k s_k, k \in \mathbb{Q}_j, \quad (3)$$

где $\mathbb{Q}_j$ – множество индексов безопасных действий для процесса ${p_i}^{k_P}$ при заданных функциональных требованиях FR_j.

Функция оценки $L(x)$ определяет, какое подмножество из множества атомарных предикатов AP является истинным в заданном состоянии.

Таблица соответствия системных вызовов ОС и действий процесса ($SCTT$) формируется на основе экспертных данных путем задания соответствия между системными вызовами и функциями декодирования их параметров. Данная таблица формируется отдельно для каждого семейства ОС, а также ее версии,

если при этом были внесены изменения в параметры вызовов системных функций. Структура $SCTT$ показана в выражении 4. Она представляет собой множество кортежей, первым элементом которого является наименование системного вызова, вторым – соответствующее действие в ОС, а третьим – объект, над которым совершается действие.

$$\left\{ \begin{array}{c} (name_1; action_1; object_1), \\ (name_2; action_2; object_2), \\ \dots \\ (name_N; action_N; object_N) \end{array} \right\} \quad (4)$$

Алгоритм преобразования системного вызова в действие процесса в ОС представлен на рис. 1. Первым этапом работы данной функции является получение категории отслеживаемого процесса путем вызова функции $getProcessCategory$ (шаг 2), которая согласно [5] может принимать следующие значения: "Системный процесс", "Привилегированный процесс", "Пользовательский процесс". Далее в цикле (шаги 3–10) происходит перебор элементов таблицы $SCTT$. В случае совпадения имени системной функции с именем записи в таблице $SCTT$ происходит присвоение переменной a значения соответствующего действия (шаг 5) и переменной o значения соответствующего объекта (шаг 6), а также декодирование параметров системной функции $getObjInfo$ (шаг 7), возвращающей категорию объекта k_o . Результатом работы функции в целом являются сведения, достаточные для формирования действия процесса в рамках модели функционирования процесса в ОС.

```
1  Function TranslateSysCall(sysCall, SCTT)
       Result: (k_p, a, o, k_o)
2      k_p = getProcessCategory()
3      foreach record ∈ SCTT do
4          if record.name = sysCall.name then
5              a ← record.action
6              o ← record.object
7              k_o ← getObjInfo(sysCall)
8              return (k_p, a, o, k_o)
9          end
10     end
11 return ∅
```

Рис. 1. Функция преобразования системного вызова в действие в ОС
Fig. 1. Function that transforms a system call into an action

Обобщенный алгоритм функционирования СБИПК представлен на рис. 2. Первым этапом работы алгоритма функционирования СБИПК является проверка соответствия исполняемого файла ряду требований (функция $checkMinCriterias$):

- программа не должна быть упакована, зашифрована или каким-либо иным образом защищена от анализа;
- в программе должны отсутствовать антиотладочные механизмы;
- бинарный файл программы должен быть предназначен для исполнения в ОС из семейства, для которого задана $SCTT$.

```
1  Function SecureCodeExec(B, FR, L, SCTT)
       Result: Decision
2      // Проверка ограничений
3      if checkMinCriterias(B) ≠ true then
4          return false
5      end
6      // Построение модели
7      M ← buildModel(B, SCTT, L)
8      // Статический этап
9      φ ← buildSpec(FR)
10     if Verify(M, φ) ≠ true then
11         return false
12     end
13     // Динамический этап
14     safeConfig ← buildSecurityAutomata(φ)
15     if SecurityAutomataMonitor(B, safeConfig, M) ≠ true then
16         return false
17     end
18 return true
```

Рис. 2. Алгоритм функционирования СБИПК
Fig. 2. Secure code execution system operation algorithm

Функция $checkMinCriterias$ (шаг 3) принимает на вход исполняемый бинарный файл B и возвращает результат проверки. Если бинарный файл B не удовлетворяет минимальным требованиям для проведения исследования, то он признается потенциально опасным и дальнейший анализ не производится. В этом случае результатом работы алгоритма является решение о недопуске исследуемого исполняемого файла к работе на объектах КИИ.

В случае успешного прохождения проверки на соответствие минимальным требованиям производится построение модели исполняемого файла в функции $buildModel$ (рис. 3). Результатом ее работы является структура Крипке, позволяющая производить формальную верификацию.

Следующим этапом работы алгоритма (шаги 9–12) является статическая верификация модели программы на соответствие требованиям безопасного исполнения в рамках заданных функциональных требований FR. Для реализации этого этапа необходимо преобразовать множество FR в множество

формул φ, задающих спецификацию безопасного исполнения. Данное преобразование производится функцией $buildSpec(\varphi)$ [13].

В функции $Verify$ производится формальная верификация на основе метода "Model checking" с использованием полученных ранее модели программы M и спецификации φ. Результатом работы данной функции является решение о соответствии модели и спецификации. В случае несоответствия – исполняемый файл B признается небезопасным, дальнейший анализ не производится.

Для контроля за исполнением функциональных требований и аксиом безопасности в процессе работы исполняемого файла строится автомат безопасности [14]. Для работы автомата безопасности необходимо задать его конфигурацию. Построение конфигурации автомата безопасности осуществляется функцией $buildSecurityAutomata$, принимающей в качестве аргумента спецификацию φ.

Функция $SecurityAutomataMonitor$ осуществляет подготовку и запуск исследуемого исполняемого файла B с отслеживанием его выполнения в рамках конфигурации безопасного исполнения $safeConfig$. Каждый вызов системной функции, производимый исполняемым файлом, изменяет состояние автомата безопасности. В случае перехода автомата в состояние, отсутствующее в модели, но не противоречащее спецификации, исполнение программы продолжается, а данная трасса добавляется во множество трасс исполнения T, полученных на этапе построения модели M. В случае перехода автомата в запрещенное состояние в рамках спецификации выполнение исполняемого файла приостанавливается.

Текущая трасса исполнения сохраняется для дальнейшего анализа. Таким образом, предлагаемый подход предполагает возможность исключения выполнения потенциально небезопасных действий. Если в течение выполнения функции $SecurityAutomataMonitor$ не возникло исключительных ситуаций, то есть во время работы программы все совершаемые ею последовательности системных вызовов не противоречили заданной конфигурации автомата безопасного исполнения, то функция возвращает значение "$true$", означающее, что анализируемый исполняемый файл соответствует модели безопасного исполнения программного кода.

3.2 Алгоритм преобразования бинарной исполняемой программы в структуру Крипке

Алгоритм преобразования бинарной исполняемой программы в структуру Крипке реализует функцию $buildModel$, функциональным предназначением которой является построение модели исполняемого файла на основе полученных методом фаззинга трасс исполнения. Результатом работы данной функции является четверка элементов, задающая модель Крипке и позволяющая производить формальную верификацию методом "Model checking" (рис. 3).


```
1  Function buildModel(B, SCTT, L)
       Result: {S, R, L}
2      // Начальная инициализация
3      T ← ∅
4      R ← ∅
5      S ← {s_0}
6      // Выделение потока управления
7      D ← Dissasembler(B)
8      // Оценка степени покрытия кода
9      while Cover(D, T) ≤ C_min do
10         repeat
11             data ← Mutation(data, T)
12             trace ← Fuzzer(B, data)
13         until trace ∉ T
14         T ← T ∪ trace
15         C ← split(trace)
16         seq ← {s_0}
17         foreach c ∈ C do
18             sysCall ← ExtractSysCall(c)
19             if sysCall = ∅ then
20                 continue
21             end
22             s ← TranslateSysCall(sysCall, SCTT)
23             seq ← seq‖s
24             if s ∉ S then
25                 S ← S ∪ s
26             end
27         end
28         for i ← 0 to | seq | −2 do
29             s_1 ← seq[i]
30             s_2 ← seq[i + 1]
31             r ← (s_1, s_2)
32             if r ∉ R then
33                 R ← R ∪ r
34             end
35         end
36     end
37 return {S, R, L}
```

Рис. 3. Преобразование множества трасс исполнения программы в модель
Fig. 3. Converting a set of program execution traces into a model

На первом этапе (шаги 3–5) производится инициализация переменных начальными значениями: T – множество трасс исполнения программы; R – множество связей между действиями программы; S – множество состояний модели (по определению модели Крипке включает в себя начальное состояние s_0).

Следующим этапом (шаг 7) является статическое преобразование бинарного исполняемого файла в модель потока управления D, отражающую связи между элементарными блоками операций. D представляет собой направленный граф, вершинами которого являются условные операторы, а с каждым ребром ассоциирован блок элементарных операций.

Цикл построения модели программы повторяется до тех пор, пока истинно условие в выражении 5.

$$Cover(D,T) \le C_{min}, \tag{5}$$

функция $Cover$ – определяет степень покрытия графа потока управления D трассами из множества T; C_{min} – задает минимальный порог уровня точности построения модели.

Каждая итерация данного алгоритма начинается с получения от фаззера [15] трассы исполнения программы $trace$ (шаг 12), представляющей собой последовательность ассемблерных инструкций. Следует отметить, что получаемая на очередном шаге цикла трасса исполнения программы должна отличаться от предыдущих, то есть не входить во множество T (шаг 13). Это обеспечивается за счет работы функции $Mutation$ (шаг 11), в основе работы которой лежит анализ множества уже пройденных трасс T и исходных данных $data$, подаваемых на вход программе B на предыдущей итерации. Затем трасса $trace$ добавляется ко множеству T (шаг 14).

Результатом работы функции $split$ (шаг 15) является множество последовательностей ассемблерных команд C, полученных из $trace$ путем разделения в местах вызова команды $call$. Правило преобразования данной функции представлено в выражении 6.

$$\begin{aligned} &split(\{a_1, a_2, \ldots, call_1, a_{N+1}, a_{N+2}, \ldots, call_2\}) = \\ &= \{\{a_1, a_2, \ldots, call_1\}, \{a_{N+1}, a_{N+2}, \ldots, call_2\}\}, \end{aligned} \tag{6}$$

где a_i – ассемблерная команда (отличная от $call$); $call_j$ – команда вызова подпрограммы.

По определению из [6], структура Крипке имеет начальное состояние s_0, из которого выходят все остальные ветви, поэтому оно добавляется ко множеству seq (шаг 16).

Важным этапом работы данного алгоритма является преобразование множества последовательностей ассемблерных команд C в системные вызовы функцией $ExtractSysCall$ (шаг 18). Результатом ее работы в случае обнаружения

системного вызова является структура, представленная в выражении 7, иначе – функция возвращает пустое множество.

$$\begin{cases} name, \\ params = \{(p_1, v_1), (p_2, v_2), \ldots, (p_N, v_N)\}; \end{cases} \tag{7}$$

где $name$ – имя системной функции; $params$ – параметры вызова системной функции.

После преобразования системного вызова в действие процесса в ОС посредством вызова функции $TranslateSysCall$ (шаг 22) переменная s содержит формальное описание действия в ОС.

Следующим этапом является конкатенация полученного действия процесса s и сформированной последовательности seq (шаг 23). Затем производится проверка наличия действия s во множестве S, и его добавление в случае отсутствия (шаг 24).

После прохождения по всем элементам списка C переменная seq содержит упорядоченную последовательность действий процесса, которую можно представить следующим выражением:

$$seq = \{s_0, s_1, s_2, \ldots, s_{N-1}\}.$$

Согласно определению модели Крипке, множество R содержит информацию о переходах между состояниями в виде двухместных кортежей. В каждом кортеже на первом месте находится текущее состояние, а на втором – состояние, в которое осуществляется переход:

$$R = \{(s_i; s_j), (s_f; s_z), \ldots, (s_x; s_y)\}. \tag{8}$$

Формирование множества R производится в цикле для i равного от 0 до $|\, seq \,| - 2$ (шаги 28–35). Из последовательности действий процесса выделяются два следующих друг за другом элемента (i, $i+1$). На их основе формируется переход r и проверяется, входит ли он во множество R (шаг 31). Если такого перехода во множестве R нет, то он добавляется. Данная последовательность действий повторяется до тех пор, пока не будут пройдены все пары элементов. На этом обработка текущей трассы считается завершенной, и осуществляется проверка условия окончания цикла – оценка степени покрытия кода (шаг 9).

Результатом работы алгоритма является структура Крипке, позволяющая производить верификацию на предмет соответствия заданной спецификации, выраженной в виде темпоральной логики.

4. Заключение

В настоящей статье рассмотрен комплекс алгоритмов, лежащих в основе функционирования СБИПК. Подробно рассмотрен алгоритм функционирования СБИПК в целом, алгоритм преобразования бинарной исполняемой программы в структуру Крипке. Применение предложенного

комплекса алгоритмов для реализации СБИПК позволяет построить формальную модель объектов, в частности исполняемых файлов, приходящих из внешних источников и произвести их формальную верификацию с использованием метода "Model checking" и автомата безопасности. Данное исследование позволит допускать к использованию на объектах КИИ только программы, уровень доверия к которым достаточен для работы с ними.

Список литературы

[1] Козачок А. В., Бочков М. В., Фаткиева Р. Р., Туан Л. М. Аналитическая модель защиты файлов документальных форматов от несанкционированного доступа. Труды СПИИРАН, т. 43, № 6, 2015, стр. 228–252

[2] Anti-Virus Comparative Summary Report. 2017. URL: https://www.avcomparatives.org/wp-content/uploads/2017/02/avc_sum_201612_en.pdf

[3] Козачок А.В. Распознавание вредоносного программного обеспечения на основе скрытых марковских моделей: дис. канд. техн. наук: 05.13.19. Орел, 2012

[4] Козачок А.В., Кочетков Е.В. Обоснование возможности применения верификации программ для обнаружения вредоносного кода. Вопросы кибербезопасности, т. 16, № 3, 2016, стр. 25–32

[5] Козачок А.В., Кочетков Е.В. Формальная модель функционирования процесса в операционной системе. Труды СПИИРАН, т. 51, № 2, 2017, стр. 78–96

[6] Clarke Edmund M, Grumberg Orna, Peled Doron. Model checking. MIT press, 1999

[7] Kinder Johannes, Katzenbeisser Stefan, Schallhart Christian, Veith Helmut. Detecting malicious code by model checking. International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2005, pp. 174–187.

[8] Song Fu, Touili Tayssir. Pushdown model checking for malware detection.International Journal on Software Tools for Technology Transfer, vol. 16, no. 2, 2014, pp. 147–173.

[9] Song Fu, Touili Tayssir. Efficient malware detection using model-checking. International Symposium on Formal Methods. Springer, 2012, pp. 418–433

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

[11] Beaucamps Philippe, Gnaedig Isabelle, Marion Jean-Yves. Abstraction-based malware analysis using rewriting and model checking. European Symposium on Research in Computer Security. Springer, 2012. pp. 806–823

[12] Holzer Andreas, Kinder Johannes, Veith Helmut. Using verification technology to specify and detect malware. Computer Aided Systems Theory–EUROCAST. 2007, pp. 497–504

[13] Kozachok A.V., Bochkov M.V., Lai M.T., Kochetkov E.V. First Order Logic For Program Code Functional Requirements Description. Вопросы кибербезопасности, т. 21, № 3, 2017, стр. 2–7

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

[15] Jesse Hertz Project Triforce: Run AFL on Everything! 2016. URL: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/.

Secure code execution system operation algorithm

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 presented a set of algorithms that form the basis of the safe code execution system. The functional purpose of it 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 set of algorithms presented in this work includes: the algorithm for the functioning of a system for the safe execution of the program code; the algorithm for constructing a program model suitable for verification, which accurately preserves the properties of the source program.

Keywords: algorithm; malware; model checking; security automata

DOI: 10.15514/ISPRAS-2017-29(3)-2

For citation: Kozachok A.V., Kochetkov E. V. Secure code execution system operation algorithm . Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017, pp. 17-30 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-2

References

[1] Kozachok A.V., Bochkov M.V., Fatkieva R.R, Tuan L.M. Analytical Model for Protecting Documentary File Formats from Unauthorized Access. SPIIRAS Proceedings, vol. 43, no. 6, 2015, pp. 228–252 (in Russian)

[2] Anti-Virus Comparative Summary Report. 2017. URL: https://www.avcomparatives.org/wp-content/uploads/2017/02/avc_sum_201612_en.pdf

[3] Kozachok A.V. Detection of malicious software based on hidden Markov models: PhD thesis. Oryol, 2012, 209 p. (in Russian)

[4] Kozachok A.V., Kochetkov E.V. Using Program Verification for Detecting Malware. Cybersecurity issues, vol. 16, no. 3, 2016, pp. 25–32 (in Russian)

[5] Kozachok A.V., Kochetkov E.V. Formal model of functioning process in the operating system. SPIIRAS Proceedings, vol. 51, no. 2, 2017, pp. 78–96 (in Russian)

[6] Clarke Edmund M, Grumberg Orna, Peled Doron. Model checking. MIT press, 1999.

[7] Kinder Johannes, Katzenbeisser Stefan, Schallhart Christian, Veith Helmut. Detecting malicious code by model checking. International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer. 2005, pp. 174–187.

[8] Song Fu, Touili Tayssir. Pushdown model checking for malware detection. International Journal on Software Tools for Technology Transfer, vol. 16, no. 2, 2014, pp. 147–173.

[9] Song Fu, Touili Tayssir. Efficient malware detection using model-checking. International Symposium on Formal Methods. Springer, 2012, pp. 418–433

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

[11] Beaucamps Philippe, Gnaedig Isabelle, Marion Jean-Yves. Abstraction-based malware analysis using rewriting and model checking. European Symposium on Research in Computer Security. Springer. 2012, pp. 806–823.

[12] Holzer Andreas, Kinder Johannes, Veith Helmut. Using verification technology to specify and detect malware. Computer Aided Systems Theory–EUROCAST. 2007, pp. 497–504.

[13] Kozachok A.V., Bochkov M.V., Tuan L.M., Kochetkov E.V. First Order Logic For Program Code Functional Requirements Description. Cybersecurity issues, vol. 21, no. 3, 2017, pp. 2–7.

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

[15] Jesse Hertz. Project Triforce: Run AFL on Everything! 2016. URL: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/