MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 31, NO. 2, 2024
journal homepage: www.mais-journal.ru

COMPUTER SYSTEM ORGANIZATION

On the Study of One Way to Detect Anomalous Program Execution

Y. V. Kosolapov[1], T. A. Pavlova[1]

DOI: 10.18255/1818-1015-2024-2-152-163

[1]Southern Federal University, Rostov-on-Don, Russia

MSC2020: 93B11
Research article
Full text in Russian

Received May 18, 2024
Revised May 27, 2024
Accepted May 29, 2024

Developing more accurate and adaptive methods for detecting malicious code is a critical challenge in the context of constantly evolving cybersecurity threats. This requires constant attention to new vulnerabilities and attack methods, as well as the search for innovative approaches to detecting and preventing cyber threats. The paper examines an algorithm for detecting the execution of malicious code in the process of a protected program. This algorithm is based on a previously proposed approach, when the legitimate execution of a protected program is described by a profile of differences in the return addresses of called functions, also called a distance profile. A concept has been introduced called positional distance, which is determined by the difference between the call numbers in the program trace. The main change was the ability to add to the profile the distances between the return addresses of not only neighboring functions, but also several previous ones with a given positional distance. In addition to modifying the detection algorithm, the work developed a tool for automating the construction of a distance profile and experimentally studied the dependence of the probability of false detection of an atypical distance on the training duration for four well-known browsers. Experiments confirm that with a slight increase in verification time, the number of atypical distances detected by the proposed algorithm can be significantly less than the number of atypical distances detected by the basic algorithm. However, it should be noted that the effect of the transition from the basic algorithm to the proposed one, as the results showed, depends on the characteristics of the specific program being protected. The study highlights the importance of continually improving malware detection techniques to adapt them to changing threats and software operating conditions. As a result, this will ensure more reliable protection of information and systems from cyber attacks and other cyber threats.

Keywords: exploits; program protection; abnormal program execution

INFORMATION ABOUT THE AUTHORS

Kosolapov, Yury V. (corresponding author)	ORCID iD: 0000-0002-1491-524X. E-mail: itaim@mail.ru Associated professor, PhD
Pavlova, Tatjyana A.	ORCID iD: 0009-0007-4565-6950. E-mail: tapavlova@sfedu.ru Student

For citation: Y. V. Kosolapov and T. A. Pavlova, "On the study of one way to detect anomalous program execution", *Modeling and Analysis of Information Systems*, vol. 31, no. 2, pp. 152–163, 2024. DOI: 10.18255/1818-1015-2024-2-152-163.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 31, № 2, 2024
сайт журнала: www.mais-journal.ru

COMPUTER SYSTEM ORGANIZATION

Об исследовании одного способа выявления аномального выполнения программы

Ю. В. Косолапов [1], Т. А. Павлова[1]

DOI: 10.18255/1818-1015-2024-2-152-163

[1]Южный федеральный университет, Ростов-на-Дону, Россия

УДК 004.056.5
Научная статья
Полный текст на русском языке

Получена 18 мая 2024 г.
После доработки 27 мая 2024 г.
Принята к публикации 29 мая 2024 г.

Разработка более точных и адаптивных методов обнаружения вредоносного кода является критической задачей в контексте постоянно эволюционирующих угроз кибербезопасности. Это требует постоянного внимания к новым уязвимостям и методам атак, а также поиска инновационных подходов к обнаружению и предотвращению киберугроз. В работе исследуется алгоритм обнаружения исполнения вредоносного кода в процессе защищаемой программы. Этот алгоритм основан на ранее предложенном подходе, когда легитимное исполнение защищаемой программы описывается профилем разностей адресов возврата вызываемых функций, называемым также профилем расстояний. Введено такое понятие, как позиционное расстояние, которое определяется разницей между номерами вызовов в трассе программы. Основным изменением стала возможность добавления в профиль расстояний между адресами возврата не только соседних функций, а также нескольких предыдущих с заданным позиционным расстоянием. Кроме модификации алгоритма обнаружения, в работе разработано средство автоматизации построения профиля расстояний и экспериментально исследуется зависимость вероятности ложного обнаружения нетипичного расстояния от длительности обучения для четырех известных браузеров. Эксперименты подтверждают, что при незначительном увеличении времени проверки число нетипичных расстояний, обнаруживаемых предложенным алгоритмом, может быть существенно меньше числа нетипичных расстояний, выявляемых базовым алгоритмом. Однако следует отметить, что при этом эффект перехода от базового алгоритма к предложенному, как показали результаты, зависит от характеристик конкретной защищаемой программы. Исследование подчеркивает важность постоянного совершенствования методов обнаружения вредоносного кода, чтобы адаптировать их к изменяющимся угрозам и условиям эксплуатации программного обеспечения. В итоге это позволит обеспечить более надежную защиту информации и систем от кибератак и других киберугроз.

Ключевые слова: эксплойты; защита программ; аномальное выполнение программы

ИНФОРМАЦИЯ ОБ АВТОРАХ

Косолапов, Юрий Владимирович (автор для корреспонденции) | ORCID iD: 0000-0002-1491-524X. E-mail: itaim@mail.ru
Доцент, канд. техн. наук

Павлова, Татьяна Александровна | ORCID iD: 0009-0007-4565-6950. E-mail: tapavlova@sfedu.ru
Студент

Для цитирования: Y. V. Kosolapov and T. A. Pavlova, “On the study of one way to detect anomalous program execution”, *Modeling and Analysis of Information Systems*, vol. 31, no. 2, pp. 152–163, 2024. DOI: 10.18255/1818-1015-2024-2-152-163.

Введение

Актуальным направлением защиты программного обеспечения (ПО) является защита от эксплойтов — исполнимого кода, использующего уязвимости в ПО с целью нарушения его штатной работы. Помимо установки обновлений, устраняющих уязвимости в ПО, и поиска различными техниками возможных уязвимостей, можно выделить два основных способа борьбы с эксплойтами [1]: сигнатурный способ и способ на основе выявления аномалий. Первый предполагает, что разработчикам средств защиты уже известны особенности эксплойта (сигнатура), и поэтому защита программ выполняется путем сравнения входных данных с сигнатурой. Недостатком такого подхода является невозможность обнаружения новых эксплойтов («эксплойтов нулевого дня»). Во втором способе для защищаемой программы в период обучения строится профиль «нормального» состояния, а защита программы заключается в выявлении отклонений от этого состояния (аномалий).

Одним из известных подходов построения профиля является подход на основе цепочек системных вызовов [2]. При таком подходе возможно обнаружение новых эксплойтов, но также имеется ненулевая вероятность принять легитимное исполнение программы за аномалию. Однако более существенной проблемой этого подхода является невозможность обнаружения мимикрирующих эксплойтов [3].

В работах [4, 5] предложен способ обнаружения эксплойтов, в котором на этапе обучения строятся профили расстояний между системными вызовами, а на этапе тестирования (эксплуатации) выполняется проверка вызовов на типичность — принадлежность профилям. В [4] предлагается использовать взвешенный показатель нетипичности вызова, значение которого зависит от числа профилей, по которым выполняется проверка, и количества учитываемых предыдущих вызовов. В этом подходе расстояния до всех учитываемых предыдущих вызовов сравниваются со всеми профилями, построенными при обучении. В [5] используется пороговое значение на количество предыдущих типичных вызовов, при превышении которого вызов считается типичным. При этом расстояния до предыдущих вызовов проверяются не по всем профилям: если разность номеров позиций вызовов в тестируемой трассе вызовов равна i, то и проверка выполняется по профилю, в котором расстояния вычисляются между вызовами в обучающей трассе, разность номеров позиций которых также равна i.

В [4, 5] с помощью средства перехвата вызовов API Monitor [6] получены предварительные экспериментальные результаты для защищаемой программы Firefox. При этом в [4] целью экспериментов было исследование зависимости вероятности ложного обнаружения от числа предыдущих вызовов и числа профилей, а в [5] — зависимость этой вероятности от длительности обучения. Формат хранения данных API Monitor не позволяет автоматизировать накопление данных для построения профилей. Поэтому в [7] подход, лежащий в основе алгоритмов [4, 5], реализован для 32-битных операционных систем Windows: реализован механизм отслеживания вызовов в защищаемой программе и разработаны утилиты для построения и обновления профиля по обучающим трассам, а также утилита для проверки типичности вызовов в тестируемой трассе. Отметим, что в [7] строится только один профиль расстояний — для соседних вызовов; проверка типичности вызовов выполняется также только по соседним вызовам. Результаты экспериментального исследования, проведенного в [7] для защищаемой программы Firefox, показали нулевую частоту ложного пропуска смоделированного поведения эксплойта, при этом частота ложного обнаружения эксплойта имеет тенденцию к снижению в зависимости от длительности обучения, что подтвердило предварительные результаты работы [5] для программы Firefox. Одним из препятствий более полного исследования зависимости вероятности ложного обнаружения от длительности обучения является ручной ввод входных данных для защищаемой программы: в экспериментах [4, 5, 7] все действия на сайтах, загружаемых Firefox, выполнялись вручную.

В настоящей работе ставятся следующие задачи: 1) дальнейшее исследование характеристик способа, лежащего в основе алгоритмов [4, 5], с целью изучения возможностей уменьшения вероятности ложного обнаружения эксплойтов; 2) автоматизация построения профиля расстояний для защищаемых программ с графическим пользовательским интерфейсом; 3) экспериментальное исследование зависимости вероятности ложного обнаружения от длительности обучения на большем количестве входных данных.

1. Базовый алгоритм из [4, 5]

Сначала в удобном виде опишем базовый способ обнаружения, лежащий в основе алгоритмов из [4, 5]. Пусть P — защищаемая программа, L — множество имен отслеживаемых функций, используемых программой. Во множество отслеживаемых функций могут входить, например функции, которые обычно используются эксплойтами [8]. Вызов функции определим в виде пары $c = (n, r)$, где $n \in L$ — имя вызываемой функции, r — адрес возврата из этой функции. Отметим, что модули программы P в силу технологии рандомизации размещения адресного пространства могут не иметь постоянного адреса загрузки. Однако, так как информация о размещении модулей известна после запуска программы, то без потери общности можно считать, что каждый модуль всегда имеет фиксированный адрес загрузки [7]. Тогда программа P может быть представлена в виде ориентированного графа $G(P) = (V, E \subseteq V \times V)$, где множество узлов V — это множество разных вызовов функций с именами функций из множества L, а пара вызовов $(c = (n, r), \tilde{c} = (\tilde{n}, \tilde{r}))$ принадлежит множеству ребер E, если существуют такие входные данные, при которых после вызова $c = (n, r)$ следует вызов $\tilde{c} = (\tilde{n}, \tilde{r})$. Все вызовы из V, а также все возможные пути в графе $G(P)$ будем называть легитимными. Ребру $e = (c, \tilde{c}) \in E$, где $c = (n, r)$, $\tilde{c} = (\tilde{n}, \tilde{r})$, поставим в соответствие число $d(c, \tilde{c}) = d(e) = \tilde{r} - r$, которое в [4, 5] для удобства названо расстоянием между вызовами $\tilde{c}$ и c. (Отметим, что в математическом смысле, $d(c, \tilde{c})$ не является расстоянием, так как $d(c, \tilde{c}) \neq d(\tilde{c}, c)$.)

Граф $G(P)$, для каждого ребра v которого известно число $d(e)$, может следующим образом использоваться для выявления исполнения эксплойта: если при выполнении программы найден вызов $\tilde{c} = (\tilde{n}, \tilde{r})$, расстояние от которого до предыдущего вызова $c = (n, r)$ не равно $d(e)$ для всех $e = ((n, r'), (\tilde{n}, r'')) \in E$, то вызов $\tilde{c}$ считается нелегитимным или нетипичным. Однако построение графа $G(P)$ для больших программ представляется сложной задачей. Информация о расстояниях между вызовами (не вся) может быть получена из трасс исполнения программы P путем многократного запуска этой программы на разных входных данных. Трассой вызовов T назовем путь в графе $G(P)$ при заданных входных данных. Трасса T может быть представлена в виде последовательности вызовов $(c_i)_{i=1}^{N}$. Для двух последовательных вызовов $c_i = (n_i, r_i)$ и $c_{i+1} = (n_{i+1}, r_{i+1})$ расстоянием между ними в [4, 5] названо число $d(c_i, c_{i+1}) = r_{i+1} - r_i$. В общем случае расстояние между вызовами c_i и c_j, $i \leqslant j$, естественно определить так: $d(c_i, c_j) = r_j - r_i$. Здесь предполагается, что расстояния могут вычисляться только для вызовов, совершенных из одного потока [4, 5]. Без потери общности считается, что трасса T состоит из вызовов одного потока. Трассу вызовов, используемую для построения профиля на этапе обучения или дообучения, обозначим T_{learn}, а трассу, вызовы которой проверяются на типичность, обозначим T_{test}. На основе обучающей трассы вызовов $T_{learn} = (c_i)_{i=1}^{N}$, полученной путем запуска программы P с разными входными данными в период обучения, может быть построен *профиль расстояний* $D(P)$, представляющий собой набор множеств вида

$$D_{(n,m)} = \{d(c_i, c_{i+1}) : c_i, c_{i+1} \in T_{learn}, i \in [1, N-1], \text{name}(c_i) = n, \text{name}(c_{i+1}) = m\}, \tag{1}$$

где $\text{name}(c) = n$ для $c = (n, r)$. Таким образом, профиль $D(P)$ состоит из $2 \cdot \binom{|L|}{2}$ множеств вида (1): $D(P) = \left(D_{(n,m)} : n, m \in L\right)$. Некоторые из этих множеств могут быть пустыми, что означает, что в трассе T_{learn} программы P не встречалось пары соседних вызовов с соответствующими именами. Профиль $D(P)$ может строиться по разным легитимным запускам программы P и представляет

собой описание её нормального поведения. Способ выявления эксплойтов через аномалии в поведении сводится к обнаружению нетипичных вызовов в трассе T_{test}, то есть таких вызовов, расстояния от которых до предыдущих вызовов не входят в профиль $D(P)$.

Отсутствие расстояния в профиле $D(P)$ может свидетельствовать о том, что был произведен запуск эксплойта или о том, что при легитимном исполнении программы была задействована ветвь исполнения кода, не встречавшаяся при обучении (ложное обнаружение). В работах [5, 7] на примере программы Firefox экспериментально продемонстрировано, что, с одной стороны, выполнение эксплойта (на примере одного реального эксплойта и одной модели эксплойта) всегда обнаруживается, а с другой стороны, при увеличении длительности обучения вероятность ложного обнаружения снижается. Для практического использования вероятность ложного обнаружения должна быть как можно меньше, в идеале она должна быть нулевой. В следующих двух разделах строится и экспериментально исследуется одна модификация подхода из работы [4].

2. Модификация алгоритма из [4]

Пусть $T_{learn} = (c_i)_{i=1}^{N}$ — обучающая трасса защищаемой программы P. Для вызовов c_i и c_j, $j > i$, из трассы вызовов T *позиционным расстоянием* между этими вызовами назовем число $pd(c_i, c_j) = j - i$. Введем три целочисленных параметра: глубину построения профиля при обучении (LMD — learning model depth), глубину проверки трассы (TTD — test trace depth) и глубину проверки профилей (TMD — test model depth), при этом $TTD \leqslant LMD$ и $TMD \leqslant LMD$. Профилем расстояний $D_{learn}(P)$ программы P назовем семейство множеств

$$D_{learn}(P) = \left(D^{j}_{(n,m)} | n, m \in L, j \in [1, LMD]\right), \tag{2}$$

где множество $D^{j}_{(n,m)}$ имеет вид

$$D^{j}_{(n,m)} = \{d(c, \tilde{c}) | c, \tilde{c} \in T_{learn}, \text{name}(c) = n, \text{name}(\tilde{c}) = m, pd(c, \tilde{c}) = j\}.$$

Таким образом, $D^{j}_{(n,m)}$ состоит из возможных расстояний между вызовами c и $\tilde{c}$, находящимися на позиционном расстоянии j, причем в вызове c имя функции равно n, а в вызове $\tilde{c}$ — m. Построение профиля описано в алгоритме 1. Алгоритм Train принимает на вход начальное значение профиля, обучающую трассу вызовов и параметр LMD. Начальное значение профиля может быть пустым, когда профиль создается впервые, или непустым, когда требуется пополнить (дообучить) профиль на основе новых трасс. Отметим, что такой подход построения профиля используется в [4, 5].

Способ выявления эксплойтов через аномалии в поведении можно свести к построению профиля расстояний тестового запуска и обнаружению расстояний из этого профиля, не входящих в профиль $D_{learn}(P)$, а значит ни в одно множество $D^{j}_{(n,m)}$, где параметр j меняется от 1 до $TMD(\leqslant LMD)$. Профиль расстояний тестового запуска имеет вид

$$D_{test}(P) = \{\tilde{D}^{j}_{(n,m)} | n, m \in L, j \in [1, TTD]\}, \tag{3}$$

где переменная TTD отвечает за глубину профиля (т. е. какое количество предыдущих вызовов для текущего участвуют в профиле), аналогично переменной LMD при обучении трассы, а

$$\tilde{D}^{j}_{(n,m)} = \{d(c, \tilde{c}) | c, \tilde{c} \in T_{test}, \text{name}(c) = n, \text{name}(\tilde{c}) = m, pd(c, \tilde{c}) = j\}.$$

Отметим, что при таком подходе выявления аномалий фиксируется число нетипичных расстояний, а не число нетипичных вызовов, однако очевидно, что этот подход может быть модифицирован для подсчета числа нетипичных вызовов.

Algorithm 1. Train — the algorithm for constructing (updating) a distance profile

Алгоритм 1. Train — алгоритм построения (обновления) профиля расстояний

Data: $D_{learn}(P)$, $T_{train} = (c_i)_{i=1}^{N}$, $LMD \in \mathbb{N}$
Result: $D_{learn}(P)$ — обновленный профиль расстояний
/* Цикл по всем вызовам трассы T_{train}, начиная со второго */
for $i \in [2, N]$ **do**
  /* Получить имя функции в c_i */
  $m = \text{name}(c_i)$
  /* Цикл по вызовам, находящимся на позиционном расстоянии не более LMD */
  for $k \in [1, \min\{LMD, i-1\}]$ **do**
    /* Получить имя функции в c_{i-k}, находящемся на позиционном расстоянии k от c_i */
    $n = \text{name}(c_{i-k})$
    /* Если в $D_{learn}(P)$ еще нет элемента $D^k_{(n,m)}$, то добавить в список $D_{learn}(P)$ элемент с номером $D^k_{(n,m)}$, и инициализировать этот элемент пустым значением */
    if $D^k_{(n,m)} \notin D_{learn}(P)$ **then**
      $D^k_{(n,m)} = \varnothing$
      $D_{learn}(P) = D_{learn}(P) \parallel D^k_{(n,m)}$
    /* Во множество $D^k_{(n,m)}$ добавить расстояние $d(c_{i-k}, c_i)$ */
    $D^k_{(n,m)} = D^j_{(n,m)} \cup \{d(c_{i-k}, c_i)\}$
return $D_{learn}(P)$

Алгоритм CheckTraceBack проверки трассы на наличие нетипичных расстояний представлен в виде псевдокода 2. Кроме профиля расстояний, тестовой трассы, параметров *TMD* и *TTD*, этот алгоритм принимает на вход булев аргумент *learn*, при истинности которого выполняется дообучение профилей в момент проверки трассы. Под дообучением здесь понимается добавление во множество $D^i_{(n,m)}$ таких расстояний из $\tilde{D}^i_{(n,m)}$, которые были определены как типичные, то есть были найдены в $D^j_{(n,m)}$, где $j \in [1, TMD]$. Отличие предлагаемого способа от способа из [4] заключается в том, что 1) введен дополнительный параметр *TMD*, который регулирует глубину проверки расстояний, 2) при обучении обновляется не профиль $D^1_{(n,m)}$, а соответствующий $D^i_{(n,m)}$, что, как представляется, в дальнейшем позволит более тонко проводить анализ типичности; 3) нетипичность определяется не на основе взвешенного значения типичности, а на основе простого правила: вызов считается нетипичным, если ни одно из *TTD* расстояний до этого вызова не принадлежит ни одному из *TMD* профилей.

Для каждого вызова c_i, где $i \geqslant 1$ — номер вызова в трассе T программы, определим список из r предыдущих вызовов $\text{prev}_r(c_i, T) = (pc^i_1, pc^i_2, \ldots, pc^i_k)$, где $k = r$, если $r < i$, в противном случае $k = i$. Таким образом, $pc^i_j = c_{i-j}$, то есть вызов c_i находится на позиционном расстоянии j от вызова pc^i_j. Из построения алгоритма CheckTraceBack вытекают следующие утверждения.

Утверждение 1. *Пусть* $LMD, TMD, TTD \in \mathbb{N}$ *— такие, что* $TMD, TTD \leqslant LMD$. *Легитимный вызов* $c \in T_{test}$ *считается типичным в* CheckTraceBack, *если найдется* $p \in \text{prev}_{TTD}(c, T_{test}) \cap \text{prev}_{TMD}(c, T_{learn})$.

Утверждение 2. *Пусть* $LMD, TMD, TTD \in \mathbb{N}$ *— такие, что* $TMD \leqslant LMD$, $TTD \leqslant LMD$. *Вызов* $c \in T_{test}$ *в* CheckTraceBack *считается типичным, если найдется* $p \in \text{prev}_{TTD}(c, T_{test})$, *для которого в* T_{learn} *найдется* $\tilde{c}$ *и такой* $\tilde{p} \in \text{prev}_{TMD}(\tilde{c}, T_{learn})$, *что* $\text{name}(\tilde{p}) = \text{name}(p)$, $\text{name}(\tilde{c}) = \text{name}(c)$ *и* $d(\tilde{p}, \tilde{c}) = d(p, c)$.

Algorithm 2. CheckTraceBack

Data: $D_{learn}(P)$, $T_{test} = (c_i)_{i=1}^{M}$, $TMD \in \mathbb{N}$, $TTD \in \mathbb{N}$, $learn \in \{\text{true}, \text{false}\}$
Result: *atypical* — число нетипичных расстояний
/* Инициализировать число нетипичных расстояний */
1 $atypical = 0$
/* Построить профиль расстояний для тестового запуска */
2 $D_{test}(P) = \text{Train}(\varnothing, T_{test}, TTD)$
/* Цикл по всем упорядоченным парам возможных имен функций */
3 **for** $(n, m) \in L \times L$ **do**
  /* Построить общее множество D возможных расстояний для вызовов, находящихся в обучающей трассе на позиционном расстоянии не более TMD */
4 $D = \varnothing$
  /* Расстояния из $\tilde{D}^i_{(n,m)}$, полученного на тестовой трассе, не входящие в D, считаются нетипичными */
5 **for** $i \in [1, TTD]$ **do**
    /* Получить множество $\tilde{D}^i_{(n,m)}$ из $D_{test}(P)$ */
6 $\tilde{D}^i_{(n,m)} \leftarrow D_{test}(P)$
7 $D = \tilde{D}^i_{(n,m)}$
8 **for** $j \in [1, TMD]$ **do**
      /* Получить множество $D^j_{(n,m)}$ из $D_{learn}(P)$ */
9 $D^j_{(n,m)} \leftarrow D_{learn}(P)$
10 **if** *learn* **then**
        /* Произвести дообучение */
11 $D^i_{(n,m)} = D^i_{(n,m)} \cup \left(\tilde{D}^i_{(n,m)} \cap D^j_{(n,m)}\right)$
12 $D = D \setminus D^j_{(n,m)}$
    /* Обновить число нетипичных расстояний */
13 $atypical = atypical + |D|$
14 **return** *atypical*

Из утверждения 1 вытекает, что часть легитимных путей из $G(P)$, которые не встретились в трассе T_{learn}, будут признаны легитимными в алгоритме CheckTraceBack. На рис. 1 изображена часть графа $G(P)$ с тремя легитимными путями исполнения защищаемой программы P: сплошной линией показаны пути $a-b-c-d$ и $a-e-f-g$, которые выполнялись при обучении, а пунктирной линией показан путь $a-b-c-g$, который при обучении не встретился. Расстояние между соседними вызовами c и g, скорее всего, будет нетипичным, однако при $LMD \geqslant 3$ расстояние между вызовами a и g будет во множестве $D^3_{(n,m)}$, где $n = \text{name}(a)$ и $m = \text{name}(g)$. Поэтому исполнение пути $a-b-c-g$ при $TTD, TMD \geqslant 3$ будет признано легитимным.

Из утверждения 2 вытекает, что возможны случаи, когда $c \neq \tilde{c}$, но при этом c считается типичным. К таким случаям относятся: 1) $c \notin T_{learn}$; 2) $c \in T_{learn}$, причем $\text{prev}_{TMD}(c, T_{learn}) \cap \text{prev}_{TTD}(c, T_{test}) = \varnothing$. Типичным вызов $c \in T_{test}$ в таких ситуация может быть признан только в случае, когда найдется $\tilde{c} \in T_{learn}$, что $\text{name}(\tilde{c}) = \text{name}(c)$, причем в $\text{prev}_{TMD}(\tilde{c}, T_{learn})$ имеется хотя бы один такой вызов $\tilde{p}$, что $\text{name}(\tilde{p}) = \text{name}(p)$, $p \in \text{prev}_{TTD}(c, T_{test})$, и $d(p, c) = d(\tilde{p}, \tilde{c})$. Отметим, что случаи 1) и 2) создают потенциальную лазейку для обхода защиты на основе профиля расстояний.

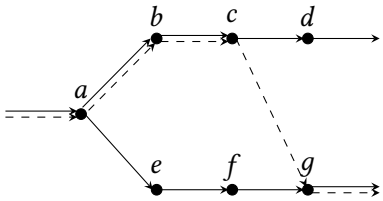

Fig. 1. An example of a legitimate path (shown as a dotted line) in $G(P)$ that would be recognized as legitimate by the CheckTraceBack algorithm for $LMD \geqslant TTD \geqslant 3$

Рис. 1. Пример легитимного пути (показан пунктирной линией) в $G(P)$, который будет распознан как легитимный в алгоритме CheckTraceBack при $LMD \geqslant TTD \geqslant 3$

Таким образом, алгоритм CheckTraceBack может как обнаруживать неизвестные на этапе обучения легитимные пути графа $G(P)$, так и принимать нелегитимные вызовы за легитимные. Похожая особенность отмечена и в [4] для соответствующего алгоритма проверки. Следующий раздел посвящен исследованию зависимости вероятности ложного обнаружения алгоритмом CheckTraceBack от длительности обучения.

3. Экспериментальное исследование

В работах [4, 5] предварительные эксперименты для защищаемой программы Firefox проводились на небольшом числе сайтов. Разработанное в [7] средство позволило автоматизировать процесс накопления профиля для Windows-программ платформы x86 и, в частности, увеличить число сайтов. Однако в [7] также, как и в [4, 5], при экспериментальном исследовании действия на сайтах выполнялись вручную. В настоящей работе доработано средство из [7] с целью автоматизации построения профилей в соответствии с алгоритмом 1 и автоматизации проверки трассы в соответствии с алгоритмом 2. Также разработано средство эмуляции действий пользователя при работе с программами, имеющими графический интерфейс.

В работе в качестве защищаемых программ выбраны четыре Интернет-браузера для операционной системы Windows на платформе x86: Mozilla Firefox 52.9 ESR, MiniBrowser 1.0.0.127, Maxton 5.3.8.2000, Vivaldi 1.0.435.46. В качестве входных данных выбран набор из 2000 сайтов наиболее популярных сайтов по данным www.similarweb.com. Выбранные защищаемые программы обладают графическим пользовательским интерфейсом, поэтому для автоматизации построения профиля разработано средство эмуляции действий пользователя на сайте. В основе разработанного средства лежит библиотека PyAutoGUI [9] для Python, которая предоставляет возможности эмуляции действий пользователя-человека. Она позволяет программно управлять мышью и клавиатурой, выполнять другие действия в графическом пользовательском интерфейсе и получать информацию об экране. При помощи библиотеки PyAutoGUI были разработаны функции, позволяющие перемещать указатель мыши на случайно сгенерированные координаты на экране, кликать мышью, прокручивать колесико мыши, менять масштаб открытой вкладки, а также выделять часть объектов на экране и копировать их в буфер обмена. Для каждого из выбранных браузеров с помощью разработанного средства автоматизации получены трассы вызовов отслеживаемых функций, где в качестве набора отслеживаемых функций L выбраны функции, которые обычно вызываются из эксплойтов [8]. Для каждого браузера было выполнено не менее 100 запусков, где в каждом запуске открывалось 20 сайтов. На каждом сайте выполнялась эмуляция действий пользователя: нажатия мыши в случайных координатах, прокрутка колеса мыши на случайную величину, выделение и копирование случайной области страницы, увеличение и уменьшение масштаба. В результате этих запусков были получены файлы с трассами вызовов. При экспериментальном исследовании трассы вызовов размера меньше 10 МБ не учитывались, а трассы более 400МБ разбивались на части

Table 1. Call trace statistics **Таблица 1.** Статистика трасс вызовов

P	K	S
Mozilla Firefox 52.9 ESR	100	28551930
Maxton 5.3.8.2000	95	57725599
MiniBrowser 1.0.0.127	109	123871995
Vivaldi 1.0.435.46	117	56680562

Table 2. Statistics of called functions **Таблица 2.** Статистика вызываемых функций

name(c) $\in L$	**Firefox**	**MiniBrowser**	**Maxthon**	**Vivaldi**
CreateFileMappingA	0,000742332	3,50239E-08	0	0
CreateFileMappingW	0,002046657	0,004444233	0,013014573	0,006753462
CreateFileMappingNumaW	0	0	0	0
DecodePointer	0,000240964	0,004816537	0,000683787	0,000362417
GetDC	0,003635026	0,001073011	7,81802E-05	9,91522E-06
GetModuleHandleA	6,48292E-05	0,001963777	9,97824E-06	3,55148E-05
GetModuleHandleW	0,002033978	0,684743868	0,000834482	0,000681945
HeapCreate	0	2,29269E-06	1,47248E-06	1,23499E-07
IsDebuggerPresent	0,003687351	6,64476E-05	7,26541E-05	4,56947E-06
LoadResource	0	0	6,65216E-06	8,82137E-08
MapViewOfFile	0,000466729	0,002017736	0,009120979	0,004527637
MapViewOfFileEx	0,001987606	0	0	0
MapViewOfFileFromApp	0	0	0	0
OpenFile	0	0	0	0
ReadFile	0,235920444	0,125970006	0,449466986	0,438129724
SizeofResource	0	0	6,65216E-06	8,82137E-08
VirtualAlloc	0,156430721	2,36615E-05	0,000145343	3,52855E-07
VirtualAllocEx	0,000354932	0,00010016	0,000165438	2,8987E-05
VirtualProtect	0,06968289	2,77948E-05	0,000121956	4,80941E-05
VirtualProtectEx	0,003072472	0	0	0,000163213
WinExec	0	0	0	0
WriteFileEx	0	0	0	0
WriteFile	0,519633069	0,174750467	0,526270866	0,549253869

по 200 МБ. В итоге для каждого из браузеров было построено K файлов с трассами вызовов, где значение K указано в таблице 1. Также в этой таблице приводится общее число вызовов функций S.

Статистика вызываемых функций, приведенная в таблице 2, показывает, что для некоторых защищаемых программ появление функции в трассе может уже свидетельствовать о нетипичном выполнении кода. Для рассмотренных программ таким функциями являются CreateFileMappingNumaW, MapViewOfFileFromApp, OpenFile, WinExec и WriteFileEx.

В [5, 7] зависимость вероятности ложного обнаружения от длительности обучения проводилась путем построения профиля на K трассах и тестирования на X новых трассах, где K растет, а X фиксировано. В настоящей работе исследование зависимости вероятности ложного обнаружения от длительности обучения выполнено по следующей схеме: 1) набор из K файлов трасс случайным образом перемешивается с помощью перестановки π; 2) для каждого $F \in \{1, \ldots, K-1\}$ строится профиль расстояний $D_{learn}(P)$ вида (2) по первым F файлам трасс при $LMD = 20$; 3) на основе файла с номером $F+1$ строится профиль расстояний $D_{test}(P)$ вида (3) для заданного параметра $TTD \in \{1, 2, 5, 10, 20\}$;

Table 3. Average time to check a trace by profile (in milliseconds)

Таблица 3. Среднее время проверки трассы по профилю (в миллисекундах)

	(TTD, TMD)						
	$(1, 1)$	$(2, 5)$	$(5, 2)$	$(5, 10)$	$(10, 5)$	$(10, 20)$	$(20, 10)$
Firefox	8919,39	8998,10	8957,56	9236,05	9237,89	9183,709	9766,80
MiniBrowser	36117,21	35659,88	35856,69	37209,40	37179,05	36238,13	40028,02
Maxthon	15172,39	15642,39	15636,33	16140,23	16165,13	15944,82	17170,20
Vivaldi	17768,15	17899,98	17925,43	18433,87	18450,35	18135,84	17758,96

4) для заданного параметра $TMD \in \{1, 2, 5, 10, 20\}$ с помощью алгоритма CheckTraceBack находится количество нетипичных E_π расстояний, то есть расстояний, не содержащихся в профиле $D_{learn}(P)$; 5) выполняется дообучение профиля $D_{learn}(P)$ путем объединения с профилем $D_{test}(P)$ (шаг 11 алгоритма CheckTraceBack). Шаги 1)-5) выполняются 10 раз для 10 разных случайных перестановок π. Таким образом было получено 10 наборов по $K - 1$ измерению числа нетипичных расстояний. По этим наборам строится набор усредненных значений E числа нетипичных расстояний из $K - 1$ измерения. Использование случайных перестановок позволяет частично нивелировать резкие всплески числа нетипичных расстояний. Такие всплески соответствуют ситуациям, когда трассы для тестируемого профиля $D_{test}(P)$ получены при открытии сайтов, задействующих функционал браузера, не используемый на этапе предыдущего обучения. Отметим, что проверка принадлежности расстояний к профилю может выполняться как в ходе исполнения программы, так и по сгенерированной программой трассе. В первом случае имеется возможность предотвратить выполнение эксплойта, но, как представляется, может замедлиться производительность самой программы. Во втором обнаружение эксплойта выполняется постфактум. В настоящей работе нетипичность расстояний оценивается по сгенерированной трассе. В ходе эксперимента замеряется время проверки наличия нетипичных расстояний в профиле (3) и усредняется по 10 перестановкам. Результаты экспериментов показали, что среднее время проверки сгенерированной трассы имеет тенденцию к росту с увеличением параметров TTD и TMD (см. таблицу 3).

Результаты оценки зависимости вероятности ложного обнаружения от длительности обучения путем нахождения среднего числа нетипичных расстояний в зависимости от длительности обучения показаны на рис. 2. На первых этапах обучения для всех браузеров фиксировалось большое число нетипичных расстояний, поэтому на графиках показаны этапы при $F \geqslant 10$. На рисунке для наглядности изображены только графики при ($TTD = 1$, $TMD = 1$) и ($TTD = 20$, $TMD = 10$). Графики для других сочетаний параметров (TTD, TMD) не приводятся, так как, учитывая незначительность в приросте времени проверки, для оценки эффекта предлагаемого алгоритма целесообразно сравнивать базовый алгоритм ($TTD = 1$, $TMD = 1$) с алгоритмом при больших значениях этих параметров.

Анализ графиков показывает, что 1) большие значения параметров TTD и TMD снижают среднее число нетипичных расстояний, по сравнению с базовым алгоритмом из [7]; 2) эффект от использования больших значений TTD и TMD зависит от защищаемой программы: например, для Firefox, MiniBrowser и Maxthon число нетипичных расстояний, особенно на ранних этапах обучения, заметно ниже для больших значений TTD и TMD, в то время как для Vivaldi этот эффект проявляется в меньшей степени; 3) число нетипичных расстояний имеет тенденцию к снижению в зависимости от длительности обучения, особенно эта тенденция заметна на ранних этапах обучения; 4) обучение на трассах со всех запусков не исключает появления нетипичных расстояний. Последнее наблюдение говорит о том, что без дополнительной информации о вызываемых функциях выявление нетипичного выполнения на основе расстояний между функциями может приводить к нежелательным ложным обнаружениям. В то же время как источник дополнительной информации о выполнении защищаемой программы в системах типа SeismoMeter[10] или в гибридных системах,

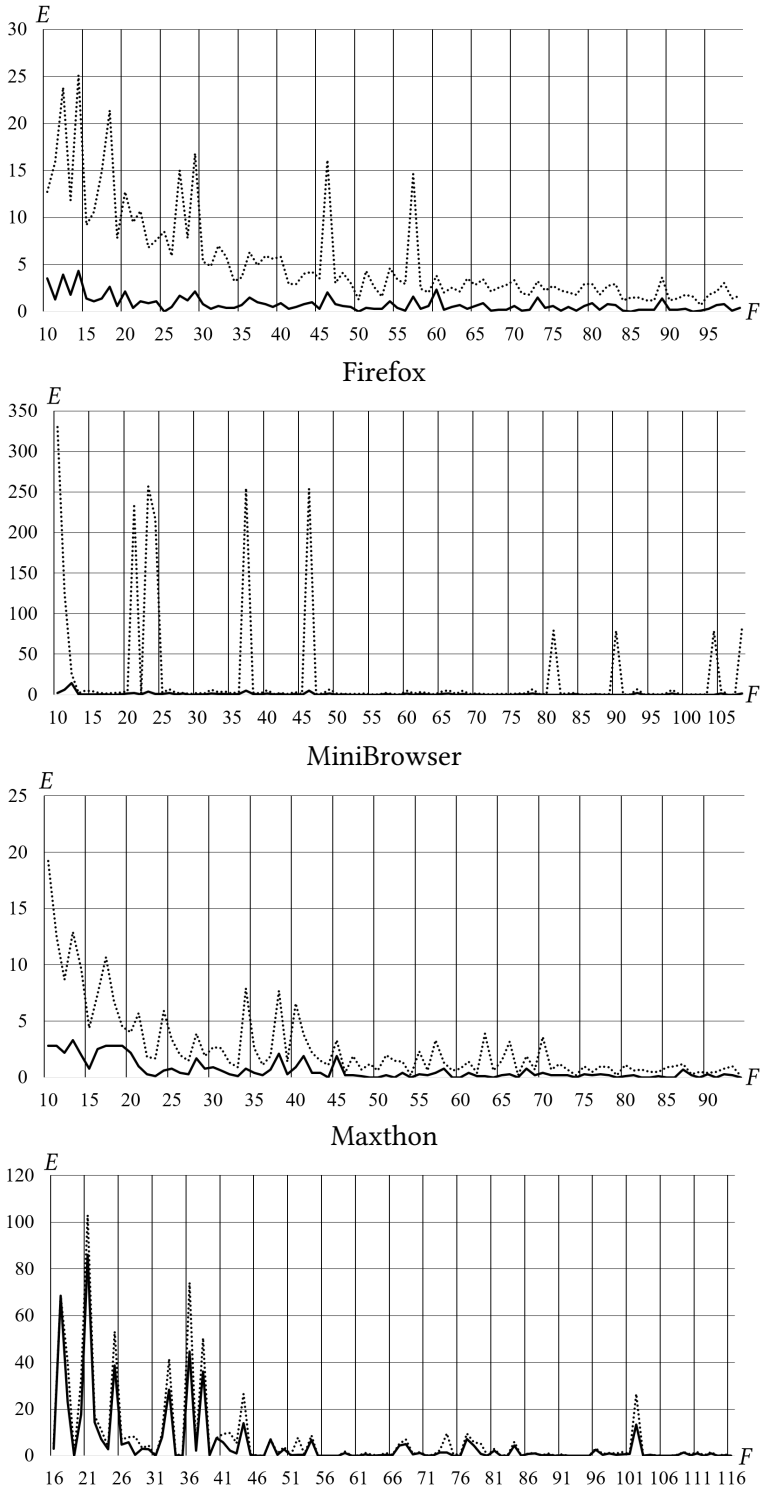

Fig. 2. Average number of atypical distances, where the dotted line shows the graph for the parameters $TTD = 1$ and $TMD = 1$, which corresponds to the basic algorithm implemented in [7], and the solid line shows the graph at $TTD = 20$ and $TMD = 10$

Рис. 2. Среднее число нетипичных расстояний, где пунктирной линией изображен график для $TTD = 1$ и $TMD = 1$, что соответствует алгоритму, реализованному в [7], а сплошной линией показан график при $TTD = 20$ и $TMD = 10$

использующих сигнатурный подход и подход на основе аномалий, такой способ использоваться может.

Заключение

Результаты исследования алгоритмов обнаружения нетипичного выполнения кода на основе построения профиля расстояний между системными функциями, полученные в [4, 5, 7], и результаты настоящей работе, свидетельствуют о возможном применении этого способа как дополнительном источнике информации о легитимном исполнении кода. Особенностью этого подхода является отсутствие необходимости доступа к исходному коду программы, так как широко известны разные способы получения информации о вызываемых в программе функциях. Длительность обучения естественно влияет на уменьшение вероятности ложного обнаружения нетипичного исполнения. Как продемонстрировано в настоящей работе, средства эмуляции действий пользователя позволяют автоматизировать построение такого профиля и сократить время его построения. Отметим, что в [5] приводятся положительные результаты выявления запуска реального эксплойта, а в [7] положительные результаты получены для смоделированного поведения эксплойта, когда вызов из «эксплойта» выполнялся по случайному адресу. Однако наибольший интерес представляет вопрос: может ли атакующий, зная систему защиты, адаптировать эксплойт так, чтобы его запуск остался незамеченным. Таким образом, дальнейшим направлением исследования является анализ возможных путей обхода предлагаемой защиты программы и пути совершенствования защиты.

References

[1] K. Lee, J. Lee, and K. Yim, “Classification and analysis of malicious code detection techniques based on the APT attack”, *Applied Sciences*, vol. 13, no. 5, p. 2894, 2023.

[2] A. Hofmeyr, S. Forrest, and A. Somayaji, “Intrusion detection using sequences of system calls”, *Journal of computer security*, vol. 6, no. 3, pp. 151–180, 1998.

[3] D. Wagner and P. Soto, “Mimicry attacks on host-based intrusion detection systems”, in *Proceedings of the 9th ACM conference on Computer and communications security*, 2002, pp. 255–264.

[4] Y. Kosolapov, “On one method for detecting exploitation of vulnerabilities and its parameters”, *Sistemy i Sredstva Informatiki [Systems and Means of Informatics]*, vol. 31, no. 4, pp. 48–60, 2021, in Russian.

[5] Y. Kosolapov, “On the detection of exploitation of vulnerabilities that leads to the execution of a malicious code”, *Automatic Control and Computer Sciences*, vol. 55, pp. 827–837, 2021.

[6] R. Batra. “Api monitor”. (2013), [Online]. Available: http://www.rohitab.com/apimonitor (visited on 04/21/2024).

[7] A. Kechahmadze and Y. Kosolapov, “Method for detecting exploits based on the profile of differences between function call addresses”, *Informatika i sistemy upravleniya*, vol. 73, no. 3, pp. 106–116, 2022, in Russian.

[8] “Exploit protection reference”. (2023), [Online]. Available: https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/exploit-protection-reference?view=o365-worldwide (visited on 04/21/2024).

[9] A. Sweigart. “Pyautogui documentation”. (2021), [Online]. Available: https://readthedocs.org/projects/pyautogui/downloads/pdf/latest/ (visited on 04/21/2024).

[10] Y. Ding, T. Wei, H. Xue, Y. Zhang, C. Zhang, and X. Han, “Accurate and efficient exploit capture and classification”, *Science China. Information Sciences*, vol. 60, 052110:1–052110:17, 2017.