УДК 004.05, 004.4[2+9], 004.94, 519.7 https://doi.org/10.15407/pp2020.01.047

В.М. Яковлев

АЛГЕБРАЇЧНІ ШАБЛОНИ ВРАЗЛИВОСТЕЙ БІНАРНОГО КОДУ

Пошук вразливостей у програмному забезпеченні є на поточний час актуальним завданням та джерелом наукових викликів. Описаний у статті алгебраїчний підхід покликаний збільшити ефективність та достовірність алгоритмів пошуку. Запропоновано засоби формального опису поведінки бінарного коду та вразливостей в термінах алгебри поведінок, а методику створення шаблонів вразливостей бінарного коду.

Ключові слова: вразливості програмного забезпечення, символьне моделювання, алгебраїчне зіставлення, алгебра поведінок.

Вступ

Однією з найважливіших сучасних проблем в галузі інформаційних технологій є виявлення вразливостей в бінарному коді. Проблема настільки серйозна, що дослідження в цьому напрямку організовуються й фінансуються на рівні державних структур. Зокрема, в США в 2018 році на базі Національного Директорату з Захисту та Програмах (National Protection and Programs Directorate, NPPD) створено Агентство з Кібербезпеки та Інфраструктури (Cybersecurity and Infrastructure Security Agency, CISA), наділене широкими правами щодо захисту програмних систем та комп'ютерних мереж. У 2016 році створено Європейську Організацію з Кібербезпеки (European Cyber Security Organisation, ECSO), а в квітні 2019 року Рада Європи схвалила Акт з Кібербезпеки (Cybersecurity Act), який, зокрема, встановлює загальноєвропейські правила сертифікації комп'ютерних систем та створює загальноєвропейське Агентство з Кібербезпеки на базі існуючого Загальноєвропейського Агентства з Мережевої та Інформаційної безпеки (European Union Agency for Network and Information Security, ENISA).

Американське агентство DARPA в 2016 році започаткувало спеціальний конкурс, Cyber Grand Challenge [1], що заохочує дослідницькі організації до створення систем для автоматизованого, масштабованого та швидкісного програмного забезпечення для виявлення вразливостей та кіберінфекцій. Всі троє переможців конкурсу 2016 року в своїх продуктах застосовували алгебраїчний підхід, яких дозволяє більш ефективно реалізувати алгоритми виявлення вразливостей.

Переможець конкурсу Mayhem [2], розроблений командою ForAllSecure з Пітсбургу, включає інструменти, які використовують символічне виконання, зокрема, комбінований (concolic) підхід, що підвищує ефективність обходу програмних шляхів. Крім того, індексована пам'ять та увід користувача розглядаються як символьні значення, що забезпечило більш повне покриття програмного середовища.

Команда Xandra TECHx [3] з Ітаки, штат Нью-Йорк, та Шарлоттсвіля посіла друге місце у загальному заліку в фінальній події. Тим не менш, ця команда була першою у вирішенні проблеми виявлення та запобігання визискам (exploit) за рахунок використання символьного інструменту пошуку.

Система Mechanical Phish [4], розроблена командою «Шелфи» з Санта-Барбари, Каліфорнія, також використовує символьне виконання з розмиванням вхідних даних.

Також існує багато інших подібних інструментів, які працюють з бінарним кодом, використовують символьні техніки моделювання. До них належать S2E [5] та інші системи з обмеженим символьним виконанням, такі як CUTE [6] та Клі [7].

В даній статті пропонується застосування алгебраїчного підходу до виявлення вразливостей за допомогою алгебраїчних моделей та виявлення поведінок, які

б відповідали певним зразкам у бінарному коді. Ця методика базується на алгебраїчній системі програмування (APS), і покладена в основу прототипу алгебраїчної системи виявлення вразливих місць у бінарному коді з використанням алгебраїчного методу відповідності.

1. Алгебра поведінок

Вразливість розглядається як поведінка системи, яка потенційно дозволяє зловмиснику виконувати такі дії, як, наприклад, пошкодження або крадіжка даних. Для конкретизації такої поведінки використовується певний алгебраїчний формалізм. Прикладом такого формалізму є алгебра поведінок, представлена Девідом Гілбертом та Олександром Летичевським [8].

Розглянемо множини поведінок та дій. Кожна поведінка складається з певних дій та іншої поведінки. Алгебра поведінок – це двосортна алгебра над множинами поведінок та дії з наступними операціями:

- операція префіксингу `a.B` означає, що перед поведінкою `B` виконується дія `a`;
- операція недетермінованого вибору поведінок `u + v` встановлює альтернативну поведінку.

Алгебра включає три термінальні константи: успішне припинення Δ, тупик 0 та невідома поведінка ⊥. Алгебра поведінки також збагачена двома операціями: паралельні (||) та послідовні (;) композиції поведінок. Терм, створений з операцій алгебри поведінки над поведінками та діями, формує вираз алгебри поведінок. Будь-яка поведінка може бути представлена як набір рівнянь, які містять ім'я поведінки в лівій частині та вираз алгебри поведінок у правій. Наприклад, наведена далі поведінка визначає поведінку програми, яка друкує щось у циклі.

```
Program = initCycle.Cycle; Δ
Cycle = print.Cycle + endCycle
```

`initCycle`, `endCycle` та `print` – це дії, а `Program` і `Cycle` – поведінки.

Всі дії об'єкта, для якого описується поведінка, передбачає зміну його стану, що визначається як множина атрибутів, представлена у вигляді типізованих змінних або функцій. Кожна дія також визначається парою формальних виразів – передумовою та постумовою дії. Семантика цих виразів визначається складністю об'єкта, що формалізується. Наприклад, наведена далі дія передбачає можливість зміни стану об'єкта, визначеного атрибутами `A` і `B`:

```
Action(A, B) =
      (A> B) && !(A == 0) ->
            B = (B + 1) / A.
```

Семантика дії, яка представлена в С-подібному синтаксисі, означає, що якщо передумова `(A > B) && !(A == 0)` справедлива для конкретного значення `A` і `B` або підходить для символічних (довільних) значень з `A` і `B`, тоді ми можемо змінити атрибут `B`:

```
B = (B + 1) / A.
```

Дію можна параметризувати за атрибутами, що згадуються в її передумові дії. Для множини поведінок ми можемо визначити іншу поведінку високого рівня, яка міститиме усі інші поведінки.

Підстановка високорівневої поведінки для дій дає набір послідовностей дій, тобто різні сценарії зазначеної поведінки високого рівня. Ця процедура називається розгортанням поведінки. Набір формул над атрибутами визначає об'єкт на кожному кроці розгортання. Формули містять символьні атрибути, а процес розгортання називається символьним моделюванням.

2. Семантика бінарного коду

Розглянемо виконавчий модуль, який складається з набору інструкцій процесора, представлених у вигляді бінарного коду. Дизасемблювавши бінарний код, отримаємо набір асемблерний текст. У подальшому розглядатимемо асемблер Intel x86 [9]. Поведінка програми визначається послідовністю інструкцій, а різні сценарії виконання генеруються за рахунок зміни вхідних даних. У формальній моделі кожна інструкція відповідає певній дії.

Потік управління програмою може бути представлений виразами алгебри по-

ведінки. У поведінковому виразі `A = a.B`, `a` – позначає певну дію, яка відповідає поточній інструкції, а `B` – представляє поведінку решти програми, що виникає після дії `a`. В інструкціях, що містять альтернативи в контрольному потоці, потрібна операція "+". Приклад поведінкових виразів і відповідний код показані далі:

```
B8049865 = sub(1,eax,0x81d01e0).B804986a,
B804986a = sar(1,eax,0x2).B804986d,
B804986d = mov(1,edx,eax,mov).B8049876,
B8049876 = jmp(1,jne).B8049879
```

```
8049865:  2d e0 01 1d 08          sub eax,0x81d01e0
804986a:  c1 f8 02                sar eax,0x2
804986d:  89 c2                   mov edx,eax
8049876:  75 01                   jne 8049879
```

Параметризовані дії та імена поведінок позначаються відповідно інструкціям та адресам операторів в асемблерному лістингу. Кожна дія моделі змінює її стан, який визначається певною формулою в атрибутному середовищі. Розглянемо структуру такого середовища.

З урахуванням структури новітніх процесорів Intel, середовище складається з наступних компонентів:

- набору регістрів загального та спеціального призначення. Деякі атрибути ідентифікуються за іменами регістрів: `ax, al, bx, bl, .., eax, ebx,…, rax, rbx,…, ebp, esp, rbp, rsp, rip`;
- фізичної пам'яті, яку можна розглядати як функцію `Memory(addr)`, де `addr` визначає адресу наявної пам'яті.

Семантика інструкції визначається передумовою і відповідною зміною середовища. Наприклад, якщо розглянути семантику інструкції `cjne A B z`, то передумова визначається значеннями операндів інструкції, а поведінка описується диз'юнкцією:

```
Bx1 = cjne.Bz +!cjne.Bx2
Bx2 =…
```

Дії:

```
cjne(n,A,B) = !(A == B) ->
    PI = PI+z+3; FLAG_C = (B > A)
!cjne(n,A,B) = (A == B) ->
    PI = PI + 3;
```

Тобто, якщо два операнди рівні, переходимо до поведінки Bz, змінюємо вказівник на значення z+3 та признаємо булеве значення атрибуту FLAG_C. В іншому випадку змінюємо вказівник на 3 та переходимо до наступної інструкції. Після відповідної трансляції набори поведінок та дій можуть бути отримані з асемблерного лістинга автоматично.

3. Алгебраїчні шаблони вразливостей

Шаблон вразливості визначає поведінку програми, що призводить до стану вразливості. Треба розрізняти стан помилки програми та стан вразливості, оскільки останній, як правило, містить також дії, що відповідають введенню в програму певних даних, обробка яких призводить до «спрацювання» вразливості. Шаблон вразливості складається з виразів алгебри поведінки та відповідних дій. Загальна форма моделі вразливості – це поведінка:

```
VulnerabilityPattern = Input;
                       ProgramBehavior;
                       VulnerabilityPoint
```

Наведений шаблон складається з трьох узагальнених поведінок.

`Input` – це поведінка програми, яка представляє інструкції, що діють від зовнішнього середовища. Вона може включати введення даних з командного рядку, зчитування файлів, отримання даних з мережі тощо. Переважно, введення реалізується як системний виклик основного коду операційної системи.

`ProgramBehavior` – це частина програми, яка виконується між точками введення даних та вразливості. В описі поведінки може бути більше однієї такої поведінки.

`VulnerabilityPoint` представляє дії, які тягнуть за собою «спрацювання» вразливості. Узагальнена поведінка охоплює всі можливі сценарії або послідовності інструкцій, що призводять до вразливості. Це є найскладніша частина формалізації

структури вразливості з огляду на необхідність зібрати всі можливі сценарії. Наприклад, копіювання пам'яті може бути представлено декількома способами та, відповідно, різними послідовностями інструкцій. Одним із таких способів є копіювання областей пам'яті в циклі.

```
writeMemory =
    mov(1,
        GeneralRegister,
        MemoryOperand).
    mov(2,
        MemoryOperand,
        GeneralRegister).
    add(1,GeneralRegister,1).
    add(2,GeneralRegister,1).
    X3;
    (writeMemory + Delta)
```

Така поведінка означає, що запис у пам'ять виконується в циклі, і кінець поведінки відповідає його завершенню. Параметр `GeneralRegister` посилається будь-який регістр загального призначення, а `MemoryOperand` позначає вміст фізичної пам'яті. `X3` є довільною поведінкою.

Відповідні дії виглядають наступним чином:

```
mov(1,GeneralRegister,
      MemoryOperand) = 1 ->
         GenPurpose(x) &&
         x = Memory(y),
mov(2 MemoryOperand,
      GeneralRegister) = 1 ->
         Memory(z) = x,
add(1, r, 1) = 1-> y = y + 1
add(2, r, 1) = 1-> z = z + 1
```

Цей простий приклад поведінки має на увазі, що вміст фізичної пам'яті за адресою y передається в регістр x і предикат `GenPurpose (x)` дорівнює `true` або `x == ax || x == bx || x == cx || x == dx` для дії `mov (1,…)`. Дія `mov (2,…)` передає вміст регістру `x` у фізичну пам'ять за адресою `z`. У діях `add (1,…)` та `add (2,…)` ми збільшуємо адреси, за якими ми звертаємося до пам'яті. Змінна `r` означає, що не має значення, де саме знаходяться адреси, за якими ми звертаємося до областей пам'яті – в регістрі загального призначення або в пам'яті. При розгортанні поведінки враховуються лише необхідні нам об'єкти, зазначені у виразах алгебри поведінки.

Розглянемо декілька узагальнених шаблонів вразливостей, пов'язаних з переповненням буфера пам'яті, що є найбільш розповсюдженою причиною вразливостей програм.

Поведінка `Input`, що відповідає введенню даних у програму, в загальній формі виглядає як

```
Input=mov(i,
          GeneralRegister,
          MemoryAddr).
      X1.
      mov(j,
          Memory(x),
          GeneralRegister).
      Input,
mov(i,
    GeneralRegister,
    MemoryAddr)=
      1-> Dirty(GeneralRegister),
mov(j,Memory(x),GeneralRegister)=
      Dirty(GeneralRegister) ->
          Dirty(x) &
          !Dirty(GeneralRegister)
```

В сучасних операційних системах введення даних у програмний буфер здійснюється шляхом копіювання в цей буфер даних з системного буфера, представленого прямою адресою `MemoryAddr`. Оскільки в одноадресній архітектурі Intel будь-яке копіювання має відбуватися через регістр загального призначення, маємо дві дії, перша з яких копіює байт з системного буфера в регістр, а друга – з регістра в локальний програмний буфер за адресою `x`.

При цьому ми відмічаємо вміст регістра, і далі вміст локального буфера як `Dirty`.

Треба зазначити, що наведена поведінка `Input` відповідає найпростішому варіанту реалізації введення даних, оскільки в розширеній множині інструкцій Intel існують інструкції копіювання пам'ять-пам'ять, наприклад, `movs`. Таким чином, поведінка `Input` має бути доповнена ін-

шими існуючими реалізаціями алгоритму введення даних.

Тепер розглянемо розповсюджений сценарій вразливості, пов'язаний з переповненням буфера, розташованого в області стека. «Спрацювання» такої вразливості може призвести до порушення роботи програми, або, при певних умовах, до виконання зловмисного коду.

```
StackVulnerability =
    (mov(l,
         Memory(s),
         GeneralRegister).
     Delta +
     !mov(l,
         Memory(s),
         GeneralRegister)).
    StackVulnerability,
```

Вказана поведінка означає, що з регістрів пам'ять у циклі пишеться у пам'ять, яка може вміщувати експлойт.

```
mov(l, Memory(s),
    GeneralRegister) =
       Dirty(GeneralRegister) &
      (s <= BP)     -> 1,
mov(m, Memory(s),
    GeneralRegister) =
       !(m = l) &
       Dirty(GeneralRegister)
                  -> Dirty(s),
mov(n, Memory(s),
    GeneralRegister) =
       !(n = l)  & (Dirty(s) &
       !Dirty(GeneralRegister))
                  -> !Dirty(s)
mov(m, GeneralRegister,
    Memory(s)) =
       !(m = l) & Dirty(s) ->
          Dirty(GeneralRegister),
mov(n, GeneralRegister,
    Memory(s))=
       !(n = l) & (!Dirty(s) &
       Dirty(GeneralRegister)) ->
          !Dirty(GeneralRegister)
```

Наведена поведінка відповідає одній з реалізацій процедури копіювання буфера вводу в буфер, розташований у стеку. «Спрацювання» вразливості відбувається, коли для дії `mov(l, Memory(s), GeneralRegister)` спрацьовує передумова `Dirty(GeneralRegister) & (s<=BP)`, що є переповненням стеку.

Інший вірогідний сценарій вразливості пов'язаний із зверненням до пам'яті, розташованої поза межами буфера, розташованого в області динамічної пам'яті, або «купи».

```
HeapVulnerability = AllocHeap;V2;
                    AccessHeap,
```

Дві важливі частини поведінки описують виділення пам'яті в «купі» та звернення до пам'яті в «купі» з метою читання або запису.

Шаблон, що відповідає поведінці виділення пам'яті, виглядає наступним чином:

```
AllocHeap = mov(I,AX,y),
            V3;
            mov(j,AX,z),
mov(I,AX,y)=1->N=AX,
mov(j,AX,z) = 1->addr=AX;
          Forall( a+N > j >= a,
                  Allocated(j) ),
```

Наведений шаблон описує операцію виділення `N` байт пам'яті. Функція виділення пам'яті приймає значення `N` як аргумент, передаючи його через регістр `AX`. Після звернення до менеджера пам'яті адреса виділеної області повертається через згаданий регістр. Одночасно, модель маркує виділені комірки пам'яті як `Allocated`. Нарешті,

```
AccessHeap=ReadHeap+WriteHeap,
ReadHeap=(mov(k,
              GeneralRegister,
              h).
          Delta +
          !mov(k,
               GeneralRegister,
               h)).
          ReadHeap,
mov(k, GeneralRegister, h) =
   Allocated(h) & (h >=N | h < 0)
                              -> 1
WriteHeap=(mov(l,
               Memory(h),
               GeneralRegister).
           Delta +
           !mov(l,
```

```
                Memory(h),
                GeneralRegister)
          ).WriteHeap,
mov(l,
    Memory(h),
    GeneralRegister) =
         Allocated(h) &
         (h >= N | h < 0) -> 1,
```

Таким чином, наведені поведінки «спрацьовують», коли операції читання або запису звертаються до пам'яті поза межами виділеного блоку.

Ще один варіант – поведінка, що описує звернення до статичної пам'яті (зовнішні змінні), дещо відрізняється від наведеної для динамічної пам'яті. Нагадаємо, такі блоки пам'яті існують в програмі впродовж всього часу її роботи.

```
AccessStatic=ReadStatic+
             WriteStatic,
ReadStatic=lea(x,
               GeneralRegister,
               offset).
           V4;DoRead,
DoRead=(mov(y,GeneralRegister,s).
        Delta +
        !mov(y,GeneralRegister,s)
       ).DoRead,
WriteStatic=lea(x,
                GeneralRegister,
                offset).
            V5;
            mov(y,
                GeneralRegister,
                s),
lea(x,GeneralRegister,offset)=
               1->Size(s)=offset,
mov(y,GeneralRegister,s)=
         s>=Size(s) | s < 0 -> 1,
```

Як правило, передумовою звернення до блоку статичної пам'яті є налаштування певного регістру на адресу цього блоку, що здійснюється через інструкцію `lea(…)`, а подальші дії аналогічні тим, що відбувається при зверненні до пам'яті інших типів.

Очевидно, наведені приклади шаблонів вразливостей описують дані вразливості лише частково, тому що існують інші сценарії подібних дій. Більш того, існує багато інших видів вразливостей, і створення відповідних шаблонів є предметом подальших досліджень.

4. Алгебраїчне зіставлення

Процедура алгебраїчного зіставлення до певної міри схожа на традиційний пошук відповідностей, що використовується в антивірусних програмах. Традиційні шаблони програмного коду містять конкретні значення, які співставляються з кодом, що перевіряється, тоді як алгебраїчне зіставлення передбачає вирішення або доведення кожного кроку зіставлення. Ця властивість алгебраїчного підходу дозволяє охопити більше випадків вразливостей та значно підвищує точність виявлення, зменшуючи кількість випадків хибного виявлення.

Алгебраїчне зіставлення виконується під час символьного моделювання бінарного коду. При зіставленні на рівні поведінок на кожному кроці виявляється істинність виразу `Env && Prec`, де `Env` – символьне середовище, а `Prec-` передумова відповідної дії. У такому випадку виконується формула постумови дії у середовищі шаблона та в середовищі моделі програми. Символьне моделювання виконується до виявлення вразливостей або вичерпання шляхів у просторі пошуку.

Висновки

Основна проблема алгебраїчного підходу полягає у тому, що досяжність у загальному випадку не визначається. Типовими проблемами також є експоненційний вибух простору станів або можливих програмних сценаріїв. Такі проблеми можуть вирішуватися за допомогою альтернативних символьних методів, таких як генерація інваріантів, апроксимація або зворотне символьне моделювання. За допомогою налаштувань пошуку можливо зменшити простір станів, наприклад, визначити певне обмежене покриття рядків коду. З іншого боку, таке звуження простору пошуку може спричинити до того, що деякі вразливості буде пропущено.

Іншою проблемою є узагальнення алгебраїчних шаблонів. Власне узагаль-

нення засновується на аналізі великої кількості прикладів та визначенні загальних рис можливих сценаріїв. Оскільки наразі універсальна методика побудови та узагальнення шаблонів відсутня, немає гарантій того, що побудований шаблон охоплює всі можливі сценарії.

Тим не менш, важлива перевага алгебраїчного підходу полягає у тому, що вразливості можуть бути виявлені набагато точніше порівняно з традиційними підходами. Крім того, описи вразливостей можуть охоплювати цілі сімейства можливих сценаріїв виконання коду.

Ефективність виявлення вразливостей може бути підвищена за рахунок реалізації зіставлення на декількох рівнях, наприклад, на рівні тільки поведінок, що може виконуватися досить швидко, і лише по тому – на рівні моделювання поведінок з доведенням досяжності на сильно звуженому просторі пошуку.

З огляду на проведені численні експерименти з робочим прототипом системи алгебраїчного зіставлення на основі алгебраїчної системи APS [10] згаданий підхід довів свою перспективність, що обумовлює необхідність подальших досліджень.

Література

1. DARPA, "Cyber Grand Challenge." [Online]. Available: https://www.cybergrandchallenge.com/.
2. Cha S.K., Avgerinos T., Rebert A., Brumley D., "Unleashing Mayhem on binary code," Proceedings. IEEE Symposium on Security and Privacy. 2012. P. 380–394.
3. Nguyen-Tuong A., Melski D., Davidson J.W., Co M., Hawkins W., Hiser J.D., Morris D., Nguen D., and Rizzi E. "Xandra: An autonomous cyber battle system for the Cyber Grand Challenge," IEEE Security & Privacy. 2008. Vol. 16. N. 2. P. 42–53.
4. Mechaphish, "Github repository." [Online]. Available from:https://github. com/ mechaphish/mecha-docs.
5. Chipounov V., Kuznetsov V., Candea G. "S2E: A platform for invivo multi-path analysis of software systems." Asplos. 2011. Vol. 46. P. 1–14.
6. Sen K., Marinov D., Agha G., Sen K., Marinov D., Agha G. "CUTE: A concolic unit testing engine for C." 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05). 2005. Vol. 30. N 5. P. 263.
7. Cadar C., Dunbar D., Engler D.R. "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. 2008. P. 209–224.
8. Gilbert D., Letichevsky A. "Interaction of agents and environments," Recent trends in algebraic development technique, LNCS 1827 (D. Bert and C. Choppy, eds.), Springer-Verlag, 1999.
9. Intel 64 and IA-32. "Architectures software developer's manual." Intel Corporation. 1997–2016.
10. Algebraic Programming System, APS, [Online]. www.apsystem.org.ua

References

1. DARPA, "Cyber Grand Challenge." [Online]. Available: https://www.cybergrandchallenge.com/.
2. Cha S.K., Avgerinos T., Rebert A., Brumley D., "Unleashing Mayhem on binary code," Proceedings. IEEE Symposium on Security and Privacy. 2012. P. 380–394.
3. Nguyen-Tuong A., Melski D., Davidson J.W., Co M., Hawkins W., Hiser J.D., Morris D., Nguen D., and Rizzi E. "Xandra: An autonomous cyber battle system for the Cyber Grand Challenge," IEEE Security & Privacy. 2008. Vol. 16. N. 2. P. 42–53.
4. Mechaphish, "Github repository." [Online].
5. Available from:https://github. com/ mechaphish/mecha-docs.
6. Chipounov V., Kuznetsov V., Candea G. "S2E: A platform for invivo multi-path analysis of software systems." Asplos. 2011. Vol. 46. P. 1–14.
7. Sen K., Marinov D., Agha G., Sen K., Marinov D., Agha G. "CUTE: A concolic unit testing engine for C." 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on

Foundations of Software Engineering (ESEC/FSE'05). 2005. Vol. 30. N. 5. P. 263.
8. Cadar C., Dunbar D., Engler D.R. "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. 2008. P. 209–224.
9. Gilbert D., Letichevsky A. "Interaction of agents and environments," Recent trends in algebraic development technique, LNCS 1827 (D. Bert and C. Choppy, eds.), Springer-Verlag, 1999.
10. Intel 64 and IA-32. "Architectures software developer's manual." Intel Corporation. 1997–2016.
11. Algebraic Programming System, APS, [Online]. www.apsystem.org.ua

Одержано 30.01.2020

Про автора:

Яковлев Віктор Михайлович,
провідний математик.
Кількість наукових публікацій в українських виданнях – 7.
Кількість наукових публікацій в зарубіжних виданнях – 1.
http://orcid.org/ 0000-0001-6000-5215.

Місце роботи автора:

Інститут кібернетики імені В.М. Глушкова НАН України,
03187, м. Київ-187,
проспект Академіка Глушкова, 40.

E-mail: victoryakovlev@ukr.net