DOI https://doi.org/10.15407/usim.2019.06.005
УДК 004.05, 004.4(2+9), 004.94, 519.7

**О.О. ЛЕТИЧЕВСЬКИЙ**, доктор фіз.-мат. наук, зав. відділом,
Ін-т кібернетики ім. В.М. Глушкова НАН України,
03187, м. Київ, просп. Акад. Глушкова, 40, Україна,
oleksandr.letychevskyi@garuda.ai

**Я.В. ГРИНЮК**, аспірант, Ін-т кібернетики ім. В.М. Глушкова НАН України,
03187, м. Київ, просп. Акад. Глушкова, 40, Україна,
yaroslav.hryniuk@garuda.ai

**В.М. ЯКОВЛЕВ**, провідний математик,
Ін-т кібернетики ім. В.М. Глушкова НАН України,
03187, м. Київ, просп. Акад. Глушкова, 40, Україна,
victoryakovlev@ukr.net

# АЛГЕБРАЇЧНИЙ ПІДХІД У ФОРМАЛІЗАЦІЇ ВРАЗЛИВОСТЕЙ В БІНАРНОМУ КОДІ

*Пошук вразливостей у програмному забезпеченні є на поточний момент 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 у термінах алгебри поведінок, а також двоступеневий загальний алгоритм пошуку, описано прототип відповідної програмної системи.*

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

## Вступ

Останнім часом проблема виявлення вразливостей програмних систем привертає дедалі більшу увагу розробників та дослідників. Було розроблено значну кількість технологій та інструментів для аналізу вразливостей для різноманітних мов програмування та формальних специфікацій. Однак, дедалі більші збитки від дій кіберзловмисників, що сягають разючих цифр у мільярди доларів на рік, доводять недостатню ефективність наявних інструментів, що базуються на традиційних підходах статичного аналізу. Особливо складною проблемою є аналіз бінарного коду, який спрямовується на виявлення поведінки програми на той час, коли зловмисник може виконувати шкідливі дії, такі як пошкодження пам'яті, крадіжку даних або порушення системи шифрування. Такі дії можуть стати можливими завдяки наявності помилок, використання небезпечного коду або небезпечних компонент (бібліотек), отриманих від сторонніх розробників.

Тому в останнє десятиріччя стає популярним алгебраїчний підхід до пошуку вразливостей у програмних системах. Інтерес до алгебраїчного підходу, особливо в частині застосування методів символьного моделювання, відображається в результатах останнього конкурсу *DARPA Cyber Grand Challenge* [1], присвяченого створенню систем кіберзахисту з можливостями автоматизованого, масштабованого, та достатньо швидкого виявлення вразливостей і кіберінфекцій. Усі переможці цього конкурсу

у той чи інший спосіб застосовували методи символьного моделювання або виконання програм.

Переможець конкурсу *Mayhem* [2] використовував алгебраїчну модель пам'яті; друге місце поділили *Xandra* [3], що використовував символьне виконання програм, і *Shellfish*, що застосовував технологію розпорошення (*fuzzing*) і символьне виконання від проблеми до точки можливого вторгнення [4]. Однак є ще низка проблем, які стосуються швидкості та точності виявлення. Особливо складні алгоритми символьного моделювання вимагають використання спеціальних дедуктивних інструментів, які є набагато повільнішими, ніж евристичний пошук або пошук певних шаблонів, що відповідають показникам вразливості. Іншою проблемою є точність, оскільки недостатня формалізація вразливості може спричинити фальшиві виявлення або неможливість виявлення взагалі.

У статті розглядається алгебраїчний підхід, який пропонується застосовувати під час пошуку вразливостей у бінарному коді, поєднуючи техніку правил переписування та символьне моделювання. Також пропонується часткове вирішення проблеми швидкодії алгоритмів пошуку завдяки дворівневому виявленню потенційних вразливостей.

## Алгебра поведінок та алгебраїчне зіставлення

Вразливість визначається як поведінка системи, яка потенційно дає зловмиснику змогу виконувати шкідливі дії, такі як пошкодження або крадіжка даних. Для формалізації поведінок використовується алгебраїчний формалізм, який визначає поведінку об'єкта. Одним із таких прикладів є алгебра поведінки, представлена Девідом Гілбертом та Олександром Летичевським [5].

Розглянемо набір поведінок і набір дій. Кожна поведінка складається з дій та інших поведінок Алгебра поведінки являє собою двосортну алгебру над набором поведінок і дій з наступними операціями.

Операція префіксінга `a.B` означає, що після дії `a` відбувається поведінка `B`. Операція недетермінованого вибору поведінки `u + v` встановлює альтернативні поведінки. Алгебра має три термінальні константи: успішне завершення, глухий кут (*deadlok*) 0 і невідома поведінка.

Алгебра поведінки також збагачена двома операціями: паралельними (||) та послідовними (;) композиціями поведінки. Терм, створений з операцій алгебри поведінки над поведінкою та діями, формує вираз алгебри поведінки.

Будь-яка поведінка може бути представлена у вигляді набору рівнянь, що задають поведінку в лівій частині та вираз алгебри поведінки в правій частині. Наприклад, поведінка, представлена далі, визначає поведінку програми, яка друкує щось у циклі.

```
Program = initCycle.Cycle,
Cycle = print.Cycle + endCycle
```

`initCycle`, `endCycle` і `print` є діями, а `Program` та `Cycle` — поведінки.

Усі дії об'єкта, поведінку яких вказано, передбачають зміну стану об'єктів, визначеного для середовища атрибутів, представленого як типізовані змінні або функції.

Кожна дія також визначається парою, а саме: передумовою та постумовою дії, наведеною як вираз у певній формальній теорії. Семантика теорії визначається складністю формалізованого об'єкта. Наведений далі приклад передбачає можливість зміни стану об'єкта, визначеного атрибутами `A` і `B`.

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

Семантика дії, представлена в *C*-подібному синтаксисі, означає, що якщо передумова

```
(A>B) && !(A == 0)
```

виконується, то ми можемо змінити атрибут `B` присвоюванням

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

Дію можна параметризувати за допомогою атрибутів, що використовуються в умовах дії.

Для набору поведінок можна визначити поведінку високого рівня, яка містить усі інші поведінки. Підстановка поведінки високого рівня для дій дає набір послідовностей дій,

тобто різні сценарії заданої 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.

Застосування символьного моделювання в пошуку та визначенні вразливостей програмного забезпечення передбачає наявність певної процедури порівняння формальної моделі бінарного коду та формальних моделей вразливостей. Така процедура називається алгебраїчним зіставленням.

Алгебраїчне зіставлення базується на недетермінованій системі переписування, яка була реалізована в рамках системи алгебраїчного програмування [6]. Цю систему впроваджено в Інституті кібернетики ім. Глушкова НАН України у 1987 р. Історично *APS* є першою системою, яка почала використовувати технологію переписування термів у поєднанні з визначеними користувачем стратегіями переписування.

Система переписування містить набір правил переписування, в нашому випадку таких, які можна представити як рівність:

`A (x, y,…) = B (x, y, ...),`

де `A(x, y, ...)` і `B(x, y, ...)` є алгебраїчними виразами над змінними `x, y,...`, які є поведінкою та діями.

Алгебраїчна система збігається з двома виразами і переписує `A` відповідно `B`, виконуючи заміну відповідних атрибутів. Правила переписування можуть бути також умовними:

`C (a, b, ...) -> A (x, y, ...) = B (x,y, ...),`

де `a, b, ...` — атрибути середовища. Це означає, що переписування може виконуватися, якщо умова `C(a, b,...)` над атрибутами є істинною.

Стратегія є функцією, що визначає стратегію переписування; наприклад, переписування з лівого або правого боку. Також переписування може бути визначено користувачем, що визначає охоплення станів навколишнього середовища.

При виявленні вразливості, модель можна представити як наступну систему правил переписування:

```
precond (a1) -> a1.y = 1, y,
precond (X1) -> 1, X1.y = 2, y,
precond (a2) -> 1, a2.y = 2, y,
precond (X2) -> 2, X2.y = 3, y,...
...
precond (a7) -> 10, a10.y = Delta
```

Існують умовні правила переписування, які визначаються передумовами відповідних дій і середньої поведінки `(X1, X2, ...)`. Переписування починається в програмі, що підлягає перевірці від першого входження дії `a1`, де її передумова задовольняється. Якщо такий випадок існує, переписування починається з цієї точки і це означає, що перше правило `a1.y` є зіставленим. Змінна у позначає решту програми, яку потрібно перевірити, і `a1.y` перепишеться як `1.y`. Нумерація означає порядок зіставлення поведінки, й переписування буде виконано, якщо відповідний номер також збігається.

Наступний крок полягає в зіставленні у, решти програми, з іншими лівими частинами систем переписування, які містять номер 1. Це може бути довільна поведінка `X1`, яка задовольняє передумову `(X1)` або дію `a2`, що задовольняє передумову `(a2)`. Зіставлення продовжується до моменту, коли буде зіставлене `a10.y`, що буде переписано як `Delta`. Це означає, що вразливість виявлено.

Алгебраїчне зіставлення може продовжуватися завдяки різним стратегіям і вибору різного покриття коду. Наприклад, для знаходження всіх найкоротших поведінок $X_i$ або отримання тільки першої зіставленої послідовності дій, що призводить до вразливості.

Реалізуючи алгебраїчне зіставлення, необхідно пам'ятати, що перевірка передумов є найдорожчою процедурою. Враховуючи наявність параметрів програми, ми маємо алгебраїчний вираз із невідомими змінними, де слід виявити виконуваність із застосуванням спеціальних систем доведення або розв'язання.

Із міркувань ефективності алгебраїчна процедура зіставлення може бути розділена на два етапи. Першим є алгебраїчне зіставлення, коли передумова містить тільки конкретні значення і її легко обчислити. Другий етап полягає у виконанні символьного моделювання отриманого результату.

Таким чином, на першому етапі обмежуються підозрілі регіони, на яких виконується зіставлення, без використання систем розв'язання. Результатом виконання першого етапу є множина поведінок, які обмежують області символьного виконання, яке відбувається на другому етапі. Це може суттєво скоротити час, необхідний для виконання розв'язувальних і доказових процедур.

Другий етап алгебраїчного зіставлення реалізується через порівняння стану середовища програми, що підлягає перевірці, з передумовами кожної дії з передумовою моделі вразливості. Якщо їхній перетин є задовільним, то він зіставляється. Критерієм виявлення вразливості є зіставлення середовища всіх трас на обох рівнях.

Відповідний технологічний ланцюжок реалізовано як науковий прототип для виявлення вразливостей у бінарному коді з використанням алгебраїчного підходу та символьного моделювання. Коротко, прототип виконує таке:

- переклад набору інструкцій у вирази алгебри поведінки;
- створення алгебраїчних моделей відомих вразливостей у термінах алгебри поведінки;
- зіставлення поведінки даних вразливостей із кодом через розв'язання рівнянь алгебри поведінки;
- зіставлення моделі вразливостей через використання символьних методів виконання;
- доказ досяжності виявлених вразливостей через зворотнє символьне моделювання.

## Формальна модель коду

Програмний код розглядається як набір інструкцій процесорів *Intel* 64 і *IA*-32. З іншого боку, програма має розглядатися як взаємодія між процесором і пам'яттю в алгебраїчному середовищі, яке містить атрибути, що є набором регістрів загального призначення (`AH, AL, AX, EAX, RAX` тощо) різних типів (байт, слово, подвійне слово тощо). Крім того, атрибутами є набір спеціальних даних («флагів»), що містяться в реєстрі `EFLAGS/RFLAGS`.

У великій кількості інструкцій треба розрізняти:

- інструкції керування потоком виконання (`JCC, JMP, CALL` тощо);
- інструкції, що змінюють атрибут середовища. Ці інструкції змінюють значення регістрів або пам'яті, можуть забезпечувати обчислення та порівнювати значення в регістрах із налаштуванням відповідних флагів.

Послідовність інструкцій перетворюється на вирази алгебри поведінки з діями з передумовами, що містять предикати та постумови, що визначають, як змінюються атрибути. Наприклад, інструкція переходу

```
60c984: jne 60cb50
```

може бути перетворена на вираз алгебри поведінки, що охоплює можливі результати на основі стану флагу `ZF` в `EFLAGS`:

```
B_60c984 = a_jne1. B_60cb50 +
a_jne2. B_60c98a
```

та відповідні зміни середовища:

```
a_jne1 = (ZF = 0) → 1,
a_jne2 = ~ (ZF = 0) → 1.
```

Ідентифікатори поведінки формуються в такий спосіб, що вони включають шістнадцяткову адресу відповідної інструкції у сегменті програми для відстеження коду асемблера. Наведений вище вираз означає, що інструкція за адресою `60c984` передасть управління команді за адресою `60cb50`, якщо флаг `ZF` дорівнює 0; інакше буде виконана наступна інструкція за адресою `60c98a`. Передумовою дії є рівності `(ZF = 0)` і `~(ZF = 0)`. Постумова відсутня, оскільки стан середовища не змінюється після виконання команди.

Інструкції, що змінюють середовище та його атрибути, можуть бути представлені як дії з постумови, що містить цю зміну. Наприклад, інструкцію

```
60c99d: ADD DWORD PTR [r13 + 0x15c],
r8d
```
буде перетворено на

```
B_60c99d = a_add_3480.B_60c9a4,
```

де

```
a_add_3480 = 1 → Memory (r13 + 348): =
Memory (r13 + 348) + r8d;
ZF: = (Memory (r13 + 348) + r8d =0);
SF: = (Memory (r13 + 348) + r8d <0)
```

Ця інструкція виконує додавання елемента пам'яті, доступного за вказаною адресою в регістрі `r13`, до вмісту регістру подвійного слова `r8d`. Дані флаги буде встановлено в біт 1 або 0, що відповідає істинності заданої рівності або нерівності. Існують інші флаги (наприклад, `CF`, `OF`, `AF`, `PF` тощо), на які впливають наведені дії, але їх не проілюстровано для простоти.

Уся семантика інструкцій визначається безпосередньо в їхній специфікації, і ми бачимо, що формалізація виконуваного коду не виглядає складною для представлення мовою формальної логіки.

## Формальна модель вразливості

Вразливість передбачає небажану поведінку програми, яку зловмисник може використовувати для зловмисних дій. Така поведінка може бути наслідком помилок розробника, поганого дизайну програми тощо. Алгебраїчне зіставлення виконує інтелектуальний пошук вразливості в бінарному коді, тому разом із алгебраїчною моделлю бінарного коду має розроблятися модель вразливості.

Розгляньмо приклад створення моделі, що вивчає відому вразливість, пов'язану з переповненням буфера, описану *Howard et al.* [7]. Беручи до уваги поведінку програми, можна побачити, що доступ до адрес байтів, які є більшими за заявлену довжину буфера, може забезпечити доступ до автоматичної пам'яті (стека), що призводить до можливості виконання певного коду. Зловмисник може скопіювати адресу шкідливого коду до цієї частини стека та запустити його.

Розгляньмо приклад програми (див. рис.1) на мові *C*, яка ілюструє вразливість, пов'язану з переповненням стека. Вочевидь, якщо довжина рядка `argv[1]` перевищить 16 байт, масив `buf` у функції `DontDoThis` буде переповнено, що, власне, і призведе до переповнення стека в процесі наповнення масиву `buf`. Згадане наповнення відбувається за виконання функції `strcpy`, код якої також наведено в даному прикладі. Відповідний до цієї функції асемблерний код виділено прямокутником у правій колонці прикладу (зазначмо, що програму було скомпільовано без оптимізації коду).

У наведеному прикладі опис цього коду в термінах алгебри поведінок виглядає так:

```
B488B45F0= mov(RAX,Memory(RBP-16))
           .movzx(edx,Memory(RAX))
           .mov(RAX, Memory(RBP-8))
           .mov(Memory(RAX),DL)
           .add(Memory(RBP-16),1)
           .add(Memory(RBP-8),1)
           .mov(RAX,Memory(RBP-16))
           .movzx(EAX, Memory(RAX))
           .test(AL,AL)
            .(jne.B488B45F0 + !jne.
B488B45F8)
B488B45F8=…
```

Опис відповідних дій створюється так, як викладено в попередньому розділі.

Таким чином, розбираючи код, його можна перевести у вирази алгебри поведінки та набір дій, що визначають семантику інструкцій через мову дій.

Наступним етапом є використання бази даних подібних моделей, створених завдяки формалізації з описів відомих вразливостей. Однак треба зазначити, що для визначення вразливості програмного коду недостатньо визначити наявність у ньому поведінок, що відповідають заданим моделям. Необхідно також довести, що такі поведінки, по-перше, є досяжними, тобто програма містить способи виконання, що ведуть до відповідного коду, та, по-друге, можуть існувати умови, які роблять цей код реальною вразливістю. Для цього має

```
#include <stdio.h>
char * strcpy( char *dst, char * src )
{
    for( ; 0 != *src; src++, dst++ )
    {
        *dst = *src;
    }
    *dst = 0;
    return dst;
}

void DontDoThis( char * input )
{
    char buf[16];
    strcpy( buf, input );
    printf( "%s\n", buf );
}

int main( int argc, char * argv[] )
{
    DontDoThis( argv[1] );
    return 0;
}
```

```
12 0001 48889E5   mov     rbp, rsp
13           .cfi_def_cfa_register 6
14 0004 48897DF8  mov   QWORD_PTR[rbp-8],rdi
15 0008 488975F0  mov   QWORD_PTR[rbp-16],rsi
16 000c EB17      jmp     .L2
17              .L3:
18 000e 488B45F0 mov rax, QWORD_PTR[rbp-16]
19 0012 0FB610   movzx   edx, BUTE_PTR[rax]
20 0015 488B45F8 mov  rax, QWORD_PTR[rbp-8]
21 0019 8810     mov     BYTE_PTR[rax],dl
22 001b 488345F0 add  QWORD_PTR[rbp-16],1
23 0020 488345F8 add   QWORD_PTR[rbp-8],1
24              .L2:
25 0025 488B45F0 mov rax, QWORD_PTR[rbp-16]
26 0029 0FB600   movzx  eax, BUTE_PTR[rax]
27 002c 84C0     test    al,al
28 002e 75DE     jne     .L3
29 0030 488B45F8 mov   rax, QWORD_PTR[rbp-8]
30 0034 C60000      mov     BYTE_PTR[rax],0
31 0037 488B45F8 mov   rax, QWORD_PTR[rbp-8]
32 003b 5D        pop   rbp
```

***Рис. 1.*** Приклад програми на мові *C*

використовуватися система алгебраїчного зіставлення, вхідними даними для якої мають бути перекладений (дизасембльований) код і база даних вразливостей.

## Трансляція двійкового коду в Алгебру поведінок

Трансляція двійкового коду в Алгебру поведінок відбувається у два кроки. Перший — використання системної утиліти (в операційній системі *Linux* — `objdump` або `readelf`) для переводу двійкового коду в асемблерний текст. Другий крок — парсинг асемблерного тексту і генерація файлів поведінок, дій та опису середовища. Усі згадані файли мають текстовий формат.

Наприклад, для асемблерного фрагмента (див. рис. 2), який відповідає програмному коду на мові *C*

```
void getss(char *str)
{
    unsigned char i = 0;
    while((i = getchar()) != 0xD){
        *str++ = i;
    }
    *str=0;
}
```

транслятор продукує такі фрагменти.

Поведінка:

```
B4009ae = push(476,rbp).B4009af,
B4009af = mov(477,rbp,rsp).B4009b2,
B4009b2 = sub(478,rsp,32).B4009b6,
B4009b6 = mov(479,Memory,rdi).B4009ba,
B4009ba = mov(480,Memory,0).B4009be,
B4009be = jmp(481,4196818).B4009d2,
B4009c0 = mov(482,rax,Memory).B4009c4,
B4009c4 = lea(483,rdx,Memory).B4009c8,
B4009c8 = mov(484,Memory,rdx).B4009cc,
B4009cc = mov(485,edx,Memory).B4009d0,
B4009d0 = mov(486,Memory,dl).B4009d2,
B4009d2 = .call_(487,4265040)
          .call B411450.B4009d7,
B4009d7 = mov(488,Memory,al).B4009da,
B4009da = cmp(489,Memory,13).B4009de,
B4009de = jne(490,4196800).B4009c0 +
         !jne(490).B4009e0,
B4009e0 = mov(491,rax,Memory).B4009e4,
B4009e4 = mov(492,Memory,0).B4009e7,
B4009e9 = ret(495)
```

Дії (див рис. 3).

Середовище (див рис. 4)

```
00000000004009ae <getss>:
  4009ae:    55                  push   rbp
  4009af:    48 89 e5            mov    rbp,rsp
  4009b2:    48 83 ec 20         sub    rsp,0x20
  4009b6:    48 89 7d e8         mov    QWORD PTR [rbp-0x18],rdi
  4009ba:    c6 45 ff 00         mov    BYTE PTR [rbp-0x1],0x0
  4009be:    eb 12               jmp    4009d2 <getss+0x24>
  4009c0:    48 8b 45 e8         mov    rax,QWORD PTR [rbp-0x18]
  4009c4:    48 8d 50 01         lea    rdx,[rax+0x1]
  4009c8:    48 89 55 e8         mov    QWORD PTR [rbp-0x18],rdx
  4009cc:    0f b6 55 ff         movzx  edx,BYTE PTR [rbp-0x1]
  4009d0:    88 10               mov    BYTE PTR [rax],dl
  4009d2:    e8 79 0a 01 00      call   411450 <getchar>
  4009d7:    88 45 ff            mov    BYTE PTR [rbp-0x1],al
  4009da:    80 7d ff 0d         cmp    BYTE PTR [rbp-0x1],0xd
  4009de:    75 e0               jne    4009c0 <getss+0x12>
  4009e0:    48 8b 45 e8         mov    rax,QWORD PTR [rbp-0x18]
  4009e4:    c6 00 00            mov    BYTE PTR [rax],0x0
  4009e9:    c3                  ret
```

***Рис. 2.*** Фрагмент ассемблерного коду

## Створення алгебраїчних моделей вразливостей

Алгебраїчні моделі, або шаблони вразливостей використовуються для позначення поведінки програми, що веде до стану вразливості. Слід наголосити, що необхідно розрізняти стан помилки програми та стан уразливості, оскільки останній може починатися з дій, доступних для зловмисника, які можуть бути використані для того, щоб отримати контроль над програмою.

Схема вразливості складається з виразів поведінки та відповідних дій. Загальна форма шаблону вразливості — це поведінка:

`VulnerabilityPattern=IntruderInput; ProgramBehavior;VulnerabilityPoint`

Існують три основні форми поведінки в загальній формі. `IntruderInput` — це поведінка програми, яка представляє інструкції, що діють від зовнішнього середовища. Така по-ведінка може включати обробку командного рядка, читання файлів, отримання даних з сокету або введення з клавіатури. Здебільшого вона реалізується як системний виклик, тобто звернення до коду ядра операційної системи.

`ProgramBehavior` представляє частину програми, яка з'єднує вхідну точку з точкою вразливості. В описі може бути більше однієї такої поведінки.

`VulnerabilityPoint` представляє дії, що призводять до «спрацювання» вразливості. Узагальнена поведінка демонструє всі можливі сценарії або послідовності інструкцій, які призводять до вразливості. Це — найскладніша частина формалізації шаблону вразливості, тому що має бути зібрано й узагальнено всі можливі сценарії. Наприклад, копіювання пам'яті може бути представлено багатьма способами й різними послідовностями інструкцій.

Далі наводиться простий приклад шаблону вразливості переповнення буфера, що розташований у стеку. Загальна поведінка вразливості може бути сформульована як

`vulnerabilityBufferOverflow=input; X1;allocateStack;X2;writeStack`

Таким чином, розглядається поведінка від входу до розподілу стека та до запису пам'яті, що пошкодить вміст стека. Вона визначається відомими послідовностями інструкцій, `allocateStack i writeStack` та невідомими довільними поведінками `X1` та `X2`.

Як прклад вводу даних input розгляньмо отримання даних (датаграми) з сокета:

```
input=mov(9,eax,0x66).mov(10,ebx,
0x11).lea(11,ecx,MemoryOperand).
call(12,MemoryOperand)
```

Ця вхідна частина може розглядатися як стандартний ланцюжок команд для отримання інформації із сокета. Необхідні параметри

```
push(476,rbp) = 1 -> <> (Mem(ESP-0)=Reg(rbp,0); Mem(ESP-1)=Reg(rbp,1);
    Mem(ESP-2)=Reg(rbp,2); Mem(ESP-3)=Reg(rbp,3); Mem(ESP-4)=Reg(rbp,4);
    Mem(ESP-5)=Reg(rbp,5); Mem(ESP-6)=Reg(rbp,6); Mem(ESP-7)=Reg(rbp,7); ESP = ESP-8;
    rip = 4196783;),
mov(477,rbp,rsp) = 1 -> <> (Reg(rbp,7)=Reg(rsp,7); Reg(rbp,6)=Reg(rsp,6);
    Reg(rbp,5)=Reg(rsp,5); Reg(rbp,4)=Reg(rsp,4); Reg(rbp,3)=Reg(rsp,3);
    Reg(rbp,2)=Reg(rsp,2); Reg(rbp,1)=Reg(rsp,1); Reg(rbp,0)=Reg(rsp,0);
    rip = 4196786),
sub(478,rsp,Numeric) = 1 -> <> (rsp=sub(rsp,32); ZF=(sub(rsp,32)=0);
    SF=(sub(rsp,32)<0); rip = 4196790),
mov(479,Memory,rdi) = 1 -> <> (Mem(rbp-17)=Reg(rdi,7); Mem(rbp-18)=Reg(rdi,6);
    Mem(rbp-19)=Reg(rdi,5); Mem(rbp-20)=Reg(rdi,4); Mem(rbp-21)=Reg(rdi,3);
    Mem(rbp-22)=Reg(rdi,2); Mem(rbp-23)=Reg(rdi,1); Mem(rbp-24)=Reg(rdi,0);
    rip = 4196794),
mov(480,Memory,Numeric) = 1 -> <> (Mem(rbp-1,0)=0; rip = 4196798),
jmp(481,Numeric) = 1 -> <> (rip = 4196800),
mov(482,rax,Memory) = 1 -> <> (Reg(rax,7)=Mem(rbp-17); Reg(rax,6)=Mem(rbp-18);
    Reg(rax,5)=Mem(rbp-19); Reg(rax,4)=Mem(rbp-20); Reg(rax,3)=Mem(rbp-21);
    Reg(rax,2)=Mem(rbp-22);Reg(rax,1)=Mem(rbp-23);Reg(rax,0)=Mem(rbp-24);
    rip = 4196804),
lea(483,rdx,Memory) = 1 -> <> (rip=4196808; rdx=rax+1; MemAddr2=rax+1; MemPtr=8),
mov(484,Memory,rdx) = 1 -> <> (Mem(rbp-24+7)=Reg(rdx,7); Mem(rbp-24+6)=Reg(rdx,6);
    Mem(rbp-24+5)=Reg(rdx,5); Mem(rbp-24+4)=Reg(rdx,4); Mem(rbp-24+3)=Reg(rdx,3);
    Mem(rbp-24+2)=Reg(rdx,2); Mem(rbp-24+1)=Reg(rdx,1); Mem(rbp-24+0)=Reg(rdx,0);
    rip = 4196812),
mov(485,edx,Memory) = 1-> <> (Reg(edx,3)=Mem(rbp-1+0); rip = 4196816;),
mov(486,Memory,dl) = 1 -> <> (Mem(rax,0)=dl; rip 4196818),
call_(487,Numeric) = 1 -> <> (rip=4265040; RipTop=RipTop+1; RipStack(RipTop)=4196823),
mov(488,Memory,al) = 1 -> <> (Mem(rbp-1,0)=al; rip=4196826),
cmp(489,Memory,Numeric) = 1 -> <> (rip=4196830; ZF=(Memory(rbp-1,0)-13=0);
    SF=((Memory(rbp-1,0)-13)<0); MemAddr1=rbp-1; MemPtr=1),
jne(490,Numeric) = (ZF = 0) -> <> (rip=196800),
mov(491,rax,Memory) = 1 -> <> (Reg(rax,7)=Mem(rbp-17); Reg(rax,6)=Mem(rbp-18);
    Reg(rax,5)=Mem(rbp-19); Reg(rax,4)=Mem(rbp-20); Reg(rax,3)=Mem(rbp-21);
    Reg(rax,2)=Mem(rbp-22); Reg(rax,1)=Mem(rbp-23);Reg(rax,0)=Mem(rbp-24);
    rip=4196836),
mov(492,Memory,Numeric) = 1 -> <> (Mem(rax,0)=0; rip=4196839),
ret(495) = 1 -> <> (rip = RipStack(RipTop); RipTop = RipTop-1)
```

***Рис. 3.*** Дії

завантажуються до регістрів, після чого виконується системний виклик.

Дії, які є в шаблоні, не містять семантики інструкцій, що включає умови, які мають бути істинними під час виконання алгоритмів зіставлення, що буде розглянуто далі. Розгляньмо умову для інструкції `call`:

```
call(12,MemoryOperand)=Forall(i:
int, 0 <= i < LengthSocket) ->
Input(ecx+i)=true
```

У цій дії пам'ять, що містить вхідне значення з адреси `ecx+i`, позначається значенням true, що визначає успішне виконання предиката `Input`. У передумові використовується квантор `Forall` для визначення циклу на всій довжині повідомлення, що приймається. Наступна частина відомої поведінки — це розподіл пам'яті стека:

```
allocateStack = push (1, ebp)
.mov(2,ebp,esp).sub(3,esp,N).
```

Ці три дії в поведінці `allocateStack` є стандартним розподілом пам'яті стека. Далі необхідно встановити атрибут, який визначає адресу стека в інструкції `mov(2,ebp,esp)`:

```
mov (2, ebp, esp) = 1-> StackAddr = ebp.
```

У постумовах визначаються новий атрибут `StackAddr` та предикат `Input`, які формують середовище шаблону. Поведінка переповнення буфера стека полягає в наступному:

```
Environment
(
        types: obj(Nil);
        attributes: obj
        (
                CF : bool, ZF : bool, SF : bool, OF : bool, PF : bool,
                Memory : (int)->int, BitAnd : (int,int)->int,
                r1 : int, r2 : int, r3 : int, r4 : int, r5 : int, r6 : int, r7 : int,
                r8 : int, r9 : int, r10 : int, r11: int, r12 : int, r13 : int,
                r14 : int, r15d : int, r15 : int,
                al : int, ax : int, ah : int, bx : int, bl : int, bh : int,
                bpl : int, bp : int, cs : int, cl : int, ch : int, cx : int,
                di : int, dil : int, dh : int, dx : int, dl : int, ds : int,
                es : int, fs : int, gs : int,
                rax : int, eax : int, ebx : int, ebp : int, esi : int, edx : int,
                edp : int, edi : int, ecx : int, rdi : int, rsi : int, rdx : int,
                rcx : int, rsp : int, rip : int, rbp : int, rbx : int,
                sil : int, si : int, eiz : int,
                MemAddr : int, MemPtr : int, RipStack : (int) -> int, RipTop: int
        );
        agent_types: obj ( ASSEMBLER: obj(Nil) );
        agents: obj ( ASSEMBLER: obj(x86) );
        instances: ( ExternalEnv );
);
```

***Рис. 4.*** Середовище

```
writeStack  =  mov(8,Memory,Memory)+
          mov(5,Memory,GenReg),
```

Ці дві альтернативи визначають різні записи для складання областей за допомогою інструкції `mov` або `movs`. Є дві відповідні дії:

```
movs(8,Memory,Memory)=Input(RefMemory
Src)&&(StackAddr==RefMemoryDest)->1
mov(5,Memory,regGen)=Input(RefMemory
Src)&&(StackAddr==RefMemoryDest)->1
```

Дії містять атрибути середовища `RefMemSrc` і `RefMemDest`, визначені як адреса `Memory-Operand`, що дорівнює адресі стека. Позначена вхідна пам'ять записується у верхню частину стека, що спричиняє її пошкодження.

Позначення вхідної пам'яті можна змінити під час виконання програми, якщо її можна скопіювати в іншу пам'ять. Для збереження маркування вхідної пам'яті ми визначаємо наступні інструкції:

```
mov (x, GenReg, Memory) = Input (RefMery-
Src) -> Input (GenReg), mov (x, Memory,
Memory) = Input (RefMemorySrc) -> Input
(RefMemoryDest).
```

Аналогічні дії визначаються під час переписування позначеної пам'яті.

Для зменшення помилкових позитивних результатів можливі також легкі обмеження алгебри поведінки. Наприклад, ми визначаємо предикат:

```
Proc (allocateStack; X2; writeStack);
```

це означає, що поведінки `allocateStack`, `X2` та `writeStack` містяться в одній асемблерній процедурі. Семантика поведінок у цій статті не розглядається.

## Побудова шаблонів вразливостей

Наведений приклад вразливості, пов'язаний з переповненням буфера, є надзвичайно поширеним. Суть цієї вразливості полягає в тому, що переповнення буфера за певних умов може призвести до того, що система почне виконувати зловмисний код. Зокрема, за розташування буфера в стеку, переповнення може призвести до спотворення адреси, до якої процесор повернеться при виході з відповідної процедури. Контролюючи цю адресу, зловмисник може примусити систему передати управління до впровадженого коду.

Вразливості, пов'язані з переповненням буфера, розташованого в «купі», описуються більш складно. Метью Коновер і Одед Горовіц аналізують цю категорію вразливостей у своїй презентації [9].

База даних *CWE* (*Common Weakness Enumeration database, https://cwe.mitre.org*) містить декілька розділів, що описують подібні вразливості. Це *CWE*-120–125, *CWE*-128, *CWE*-129, *CWE*-131, *CWE*-193, *CWE*-466, які описують узагальнені ситуації, що призведуть до вразливості програм. Є також велика кількість записів в інших джерелах, що описують специфічні випадки, такі як, наприклад, описана в базі даних *CVE* (*CVE Details, The ultimate security vulnerability datasource, https://www.cvedetails.com*) вразливість за певних обставин функції *strftime*, *CVE*-2015-8776. Іншим прикладом є група записів *CVE*-2000-0389 — *CVE*-2000-0392, які описують декілька можливостей отримання зловмисником *root*-привілеїв під час використання продукту *Kerberos* 5.

При цьому, хоча класифікатори часто розрізняють вразливості, пов'язані з переповненням буфера, за різними видами використовуваної пам'яті (стек, сегмент даних, сегмент коду, «купа»), по суті, до вразливості в усіх подібних прикладах ведуть однакові або подібні послідовності дій.

Таким чином, загальний шаблон поведінки, що описує вразливість, пов'язану з переповненням буфера, може виглядати як

```
BufferOverflowVulnerability = Input.
XX1. WriteBuffer,
```

тобто описується поведінка, що включає певний вхід (введення даних), який за деяких умов призводить до вразливості в поведінці `WriteBuffer`. Розгляньмо, як можна сформулювати таку поведінку.

Однією з проблем формулювання загального шаблону вразливості є питання методики створення такого шаблону. Оскільки для запису в буфер у більшості випадків використовуються, в кінцевому підсумку, функції копіювання, такі як `memcpy` або `memmove`, один із підходів до формування шаблону вразливості мав би полягати у виокремленні поведінок, що ведуть до виклику цих функцій. Поведінка для memcpy може виглядати, наприклад, так:

```
CallMemcpy=mov(1,eax,Memory).
mov(2,Memory,eax).X1.
mov(3,eax,Memory).
mov(10447,Memory,eax).call_
(5,80ea3f0)
```

Замість `mov` у багатьох випадках може бути використана інструкція `lea`:

```
CallMemcpy=mov(1,eax,Memory).
mov(2,Memory,Numeric).
X1.lea(3,eax,Memory)
mov(4,Memory,eax).call_(5,80ea3f0)
```

Тоді об'єднанням приведених шаблонів буде поведінка

```
CallMemcpy=mov(1,eax,Memory).(mov(2,
Memory,eax)+mov(3,Memory,
Numeric)).X1.(mov(4,eax,Memory)+
lea(5,eax,Memory)).mov(6,Memory,eax)
.call_(7,80ea3f0)
```

Нажаль, варіанти поведінок, що ведуть до виклику memcpy, цим не вичерпуються. До того ж, розташування складових частин поведінки до власне виклику `memcpy` може бути довільним. Наостанок, адреса `memcpy` (у прикладі — `80ea3f0`) може змінюватися відповідно до таблиці переміщень.

Таким чином, у намаганнях отримати певний «універсальний» шаблон виклику функції `memcpy` ми отримаємо досить складну та громіздку конструкцію, яка, до того ж, навряд чи покриватиме всі можливі поведінки, що ведуть до виклику цієї функції.

Інший спосіб створення загального шаблону вразливості — використання поведінок відповідних функцій. Зокрема, для функції memcpy така поведінка виглядатиме як

```
MemCpy=LoadMemcpyParam.X1.DoMemCpy,
LoadMemcpyParam=mov(1,eax,edi).
mov(2,edi,Memory). mov(3,edx,esi).
mov(4,esi,Memory).mov(5,ecx,edi).
xor(6,ecx,esi). and(7,ecx,3),
DoMemCpy = Proc(DoCheck. X1.DoCopy),
DoCheck = mov(8,edi,eax). mov(9,esi,
edx). mov(10,eax,Memory),
```

```
DoCopy = Proc(shr(11,ecx,1). X2.
mov(12,Memory,Memory).
shr(13,ecx,Memory). X3.
mov(14,Memory,Memory).
mov(15,Memory,Memory). DoCheck
```

Як видно, шаблон поведінки функції memcpy є доволі складним і включає рекурсію, але такий шаблон принаймні покриває всі можливі випадки використання функції.

Шаблон поведінки функції memcpy може бути частиною поведінки `WriteBuffer`. Для врахування використань інших методів запису буфера маємо розробити шаблони поведінки для таких методів. Зрештою, поведінка `WriteBuffer` виглядатиме як:

```
WriteBuffer=MemCpy+MemMove+
WriteLoop+…,
```

тобто як об'єднання всіх поведінок, що описують запис в буфер.

Аналогічна ситуація складається з поведінкою `Input` — врахувавши всі можливі методи введення даних, отримаємо кінцевий результат.

## Інші види вразливостей

Далі дається короткий огляд інших найпоширеніших вразливостей програмного коду. Треба зазначити, що наведений перелік не є повним і має ілюстративний характер: розгляд повного переліку відомих вразливостей, навіть в узагальненому вигляді, мав би бути темою окремої статті.

***Форматування рядків***. Цей клас вразливостей описано в *CWE*-134 (неконтрольований форматний рядок). У мовах *C* та *C++* використання введених користувачем даних без попередньої перевірки для форматування рядків може призводити до запису в довільні адреси пам'яті. Зловмисник може обходити захист стека та змінювати малі ділянки пам'яті. Яскравим прикладом є запис *CVE*-2000-0573, що описує ситуацію, коли продукт *wu-ftpd* 2.6.0 (*Linux*) завдяки відсутності перевірки форматних рядків може дозволити віддалене отримання *root*-привілеїв.

***Цілочисельне переповнення***. Суть цієї проблеми полягає в тому, що для практично будь-якого двійкового формату, обраного для представлення цілих чисел, наявні операції, результат виконання яких відрізнятиметься від результату аналогічних обчислень, виконаних вручну. Винятками є реалізації представлення цілих чисел змінного розміру, однак такі реалізації зустрічаються та застосовуються доволі рідко й до того ж пов'язані з великими накладними витратами.

Власне, про проблему переповнення під час виконання цілочисельних операцій відомо доволі давно, але її почали пов'язувати із вразливостями програмного забезпечення відтоді, коли у зловмисних діях щодо програмних систем почали використовуватися атаки, пов'язані з виокремленням пам'яті в «купі».

На цей час зафіксовано такі класи вразливостей, пов'язані з цілочисельним переповненням:

- *CWE*-682: Невірні обчислення.
- *CWE*-190: Цілочисельне переповнення або циклічний зсув.
- *CWE*-191: Цілочисельна втрата значення (простий або циклічний зсув).
- *CWE*-192: Помилки приведення до цілочисельного типу.

***Специфіка C++.*** Специфічні до *C++* види вразливостей програмного забезпечення використовують пошкодження або спотворення значень вказівників. Це може відбуватися двояко:

- пошкодження або спотворення вказівників на функції, що є членами класів;
- пошкодження або спотворення вказівників на таблицю віртуальних функцій, або всередині таблиці віртуальних функцій.

Обидва види ведуть до зміни способу виконання програми.

Ще одна потенційна вразливість пов'язується з повторним звільненням пам'яті, що може дозволити зловмиснику переписати в пам'яті нормально ініціалізований об'єкт.

Відповідні класи вразливостей описано в наступних категоріях:

- *CWE*-703: Відсутність обробки потенційних виняткових ситуацій;
- *CWE*-404: Некоректне звільнення ресурсів;
- *CWE*-457: Використання неініціалізованих змінних;

▪ *CWE*-415: Повторне звільнення пам'яті;

▪ *CWE*-416: Використання об'єктів після звільнення;

***Пошук вразливостей у бінарному коді.*** Із міркувань ефективності та прискорення визначення вразливостей задача їхнього пошуку поділяється на два етапи:

▪ зіставлення моделі коду з шаблонами вразливостей, коли визначаються регіони коду, що зіставляються із заздалегідь підготованими шаблонами вразливостей, та формуються поведінки, які відповідають поведінкам, визначеним у шаблонах. Системи розв'язання при цьому не використовуються, щоб скоротити час виконання;

▪ символьне виконання поведінок, зформованих на попередньому етапі, коли використовуються всі необхідні розв'язувальні та доказові процедури.

## Зіставлення моделі коду з шаблонами вразливостей

Алгебраїчне зіставлення відрізняється, але не значною мірою, від традиційного зіставлення, яке використовується в антивірусних програмах. Традиційні моделі відповідного коду можуть містити конкретні значення, які перевіряються, тоді як алгебраїчне зіставлення передбачає вирішення або підтвердження кожного кроку відповідності. Ця властивість алгебраїчного підходу дає змогу охопити більше випадків уразливості та значно підвищує точність виявлення, зменшуючи в такий спосіб кількість помилково позитивних випадків.

Для ілюстрації алгоритму алгебраїчного зіставлення розгляньмо шаблон вразливості

```
BufferOverflowVulnerability = Input.
XX1. WriteBuffer,
```

де, своєю чергою,

```
Input = ConsoleInput + SocketInput
WriteBuffer = MemCpy + MemMove + WhiteLoop
```

Таким чином, видно, що шаблон вразливості представляється як сукупність дерев. Назвімо поведінки, для яких визначено кінцеві ланцюжки поведінок, що відповідають бінарному коду, терміналами, або термінальними поведінками. Невизначені поведінки назвімо детермінаналами. Термінали можуть бути представлені деревами, що, можливо, містять циклічні зв'язки. Оскільки поведінка програми також може бути представлена у вигляді дерева з циклічними зв'язками, задача алгебраїчного зіставлення зводиться до зіставлення дерев, визначення «зіставлених» регіонів, та шляхів, що з'єднують ці регіони. Для цього мають бути виконані такі дії:

▪ відновлення дерева викликів програми; вочевидь, відновлена інформація про виклики буде неповною, адже в *C/C++* наявна можливість непрямих викликів, тобто обчислення адреси функції, яка буде викликана;

▪ власне алгебраїчне зіставлення, тобто визначення в коді програми регіонів, які зіставляються з термінальними поведінками, заданими в описі вразливості;

▪ визначення шляхів між регіонами, зіставленими для відповідних термінальних поведінок; вочевидь, такі регіони можуть бути розташовані в різних гілках програми, тобто алгоритм має рухатися вниз до точки повернення, а також вгору по дереву викликів із пошуком «точок перетину» таких шляхів, що формуються для регіонів, які зіставляються з різними термінальними поведінками.

Результатом дії наведеного алгоритму є поведінка, яка включає всі знайдені шляхи між регіонами коду, що зіставляються з термінальними поведінками, визначеними в описі вразливості. Наступним кроком є символьне виконання поведінок для доведення досяжності знайдених шляхів і визначення умов такої досяжності.

## Символьне виконання поведінок

Під час символьного моделювання застосовуються дії, для яких задовольняється вираз `Env && Prec`, де `Env` — символьне середовище моделі двійкового коду і `Prec` — передумова відповідних дій. Якщо ця передумова задовольняєтеся, мають виконуватися операції, визначені в постумові у середовищі шаблону й у середовищі моделі двійкового

коду. Моделювання виконується від точки введення даних. При досягненні точки вразливості в шаблоні ми отримуємо готовий сценарій спрацьовування вразливості. Символьне моделювання вимагає використання специфічних розв'язувачів, які підтримують побітові функції або байтові векторні операції.

Символьне середовище включає також множину формул, завдяки чому є можливість реалізувати конкретний сценарій на основі певної поведінки, або визначити вхідні значення, завдяки яким поведінка, що відповідає вразливості, стає досяжною. Також символьне моделювання може бути проведено у зворотному напрямку, тобто від точки вразливості до введення даних. У цьому разі використовуваний предикатний трансформер дає початкову формулу, яка охоплює значення, що дозволяють виконувати зловмисні дії.

## Отримані результати

Запропонований підхід реалізовано в науковому прототипі, який було протестовано на реальних прикладах програм, таких як старі версії *vi, ed* і *mc* (*ОС Linux*), а також інших програм, що містять вразливості. Тести показують, що в поведінках, згенерованих на стадії алгебраїчного зіставлення, трапляється багато помилкових позитивних результатів. Це є природним, тому що зіставлення поведінки відбувається без участі середовища моделі; проте запропонований підхід було доповнено також обмеженнями для поведінки моделі вразливості, наприклад, «поведінка» не має містити інструкції `ret`. Алгоритм зіставлення дерев поведінок є кінцевим, оскільки завжди розглядається лише один крок циклу і кожна дія відвідується принаймні один раз.

Перші експерименти з використанням алгебраїчного зіставлення виконано в рамках *APS* і платформи *Garuda AI*. Вони охоплювали 15 відомих вразливостей із бази даних *Common Vulnerabilities and Exposures* (*CVE*), переважно таких, що стосуються переповнення буфера, розташованого в стеку.

## Висновки

Основна перевага алгебраїчного підходу полягає в тому, що він надає можливість точнішого виявлення вразливості. Опис вразливості охоплює набір можливих його різновидів.

Перевагою реалізації є те, що спочатку виконується швидша процедура виявлення (алгебраїчна відповідність), а потім дорожча стадія (символьне моделювання). Здійснені експерименти «дворівневою» системою показали, що час виконання виявлення вразливостей за такого підходу значно зменшується порівняно із суто символьним моделюванням.

Головною проблемою цього підходу є те, що досяжність під час символьного виконання є невирішуваною в загальному випадку. Також під час символьної обробки моделей можуть виникати експоненційний вибух простору станів та інші проблеми, що є типовими для цього підходу. Ці проблеми можна вирішити, використовуючи альтернативні символьні методи, такі як інваріантна генерація, апроксимація або зворотне символьне моделювання. Різні параметри пошуку можуть обмежити розмір простору станів; наприклад, можемо забезпечити певне покриття коду, хоча це скорочення може призвести до того, що ми пропустимо деякі вразливості.

Важливою проблемою є узагальнення алгебраїчних шаблонів, яке базується на кількості розглянутих прикладів та аналізі можливих сценаріїв. Однак, універсальної техніки побудови шаблонів досі немає, і неможливо гарантувати, що наведана модель охоплює всі можливі поведінки вразливості даного типу.

Іншою проблемою, з якою зіштовхується моделювання бінарного коду, є моделювання середовища, що включає ядро *ОС*, статичні та динамічні бібліотеки, багатопотоковість, виклики функцій за адресами, що обчислюються під час виконання, та інших специфічних для бінарного коду питань.

Ці проблеми не є критичними, але трудомісткими, і їх можна подолати в рамках цього підходу.

ЛІТЕРАТУРА

1. *DARPA*, Cyber Grand Challenge. URL: https: //www.cybergrandchallenge.com/
2. *Cha S.K., Avgerinos T., Rebert A.* et al. Unleashing Mayhem on binary code. IEEE Symposium on Security and Privacy, 2012. P. 380–394.
3. *Nguyen-Tuong, Melski D., Davidson J.W.* et al. 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. URL: https://github. com/mechaphish/mecha-docs
5. *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.
6. *Algebraic* Programming System. APS. URL: www.apsystem.org.ua
7. *Howard M., LeBlanc D., Viega J.* Deadly Sins of Software Security: Programming Flaws and How to Fix Them. McGraw-Hill. 2010. 24
8. *Letychevskyi O., Letichevsky A.* Predicate transformers and system verification. Proc. Third International Workshop on Symbolic Computation in Software Science (SCSS 2010). Hagenberg. 2012.
9. *Matt Conover & Oded Horovitz.* Reliable Windows Heap Exploits. URL: http://xcon.xfocus.org/XCon2004/archives/14_Reliable%20Windows%20Heap%20Exploits_BY_SHOK.pdf

Надійшла 14.11.2019

REFERENCES

1. *DARPA,* “Cyber Grand Challenge.” [Online]. Available at:<https://www.cybergrandchallenge.com> [Accessed 21 Oct. 2018].
2. *Cha, S.K., Avgerinos, T., Rebert, A. and Brumley, D.,* 2012. “Unleashing Mayhem on binary code,” Proc. of IEEE Symposium on Security and Privacy, pp. 380–394.
3. *Nguyen-Tuong, Melski, J. W. Davidson, M. Co, W. Hawkins, J. D. Hiser, D. Morris, D. Nguen, and E. Rizzi,* 2008.“Xandra: An autonomous cyber battle system for the Cyber Grand Challenge,” IEEE Security & Privacy, vol. 16 , no. 2, pp. 42–53.
4. *Mechaphish,* “Github repository.” [Online]. Available at:<https://github. com/mechaphish/mechadocs> [Accessed 21 Oct. 2019].
5. *Gilbert, D., Letichevsky, A.,* 1999. “Interaction of agents and environments,” Recent trends in algebraic development technique, LNCS 1827, Springer-Verlag.
6. *Algebraic Programming System,* APS, [Online]. Available at:<www.apsystem.org.ua> [Accessed 2 Oct. 2019].
7. *Howard, M., LeBlanc, D., Viega, J.,* 2010. 24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them. McGraw-Hill.
8. *Letychevskyi, O. and Letichevsky, A.,* 2012. “Predicate transformers and system verification”. Proc. Third International Workshop on Symbolic Computation in Software Science (SCSS 2010).
9. *Matt Conover & OdedHorovitz.* Reliable Windows Heap Exploits. [Online]. Available at:<http://xcon.xfocus.org/XCon2004/archives/14_Reliable%20Windows%20Heap%20Exploits_BY_SHOK.pdf> [Accessed 2 Oct. 2019].

Received 14.11.2019

*O.O. Letychevskyi,* Doctor (Phys.-Math.), Head of department,
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine,
Glushkov ave., 40, Kyiv, 03187, Ukraine,
oleksandr.letychevskyi@garuda.ai

*Ya.V. Hryniuk*, Ph.D. student of Computer Science at
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine,
Glushkov ave., 40, Kyiv, 03187, Ukraine,
yaroslav.hryniuk@garuda.ai

*V.M. Yakovlev*, Lead Mathematician at
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine,
Glushkov ave., 40, Kyiv, 03187, Ukraine,
victoryakovlev@ukr.net

## ALGEBRAIC APPROACH TO VULNERABILITIES FORMALIZATION IN THE BINARY CODE

**Introduction**. The security and protection of the software resources is one of the most actual problems in the IT industry nowadays. The approaches to the threat modeling and vulnerabilities detection, based on the symbolic methods, became very popular and promising during the last decade. The article describes an approach to the vulnerabilities detection in the binary code, based on the formal methods of symbolic modeling and algebraic matching.

**Purpose**. Although the symbolic modeling could be effectively used for the code vulnerabilities detection, the relevant software tools are very slow. Also, the reachability problem during the symbolic program execution is unresolvable in the common case. In the article, the formalization of representation of binary code and vulnerabilities based on the behavior algebra, and an approach that allow speeding up the vulnerabilities search, and reducing the area of the symbolic modeling are proposed.

The behavior algebra used for the representation of the formal binary code behavior, as well as for describing the vulnerabilities behavior. However, while the representation of the binary code in the terms of behavior algebra could be automated, creation of the vulnerabilities description requires development of the correct and effective methodology. Using the behavior algebra representation, the task of vulnerabilities detection can be solved in two steps – relatively fast algebraic matching, and the symbolic modeling itself, based on the data provided by the algebraic matcher.

**Methods**. By the development of the vulnerabilities description in the terms of behavior algebra, and the algebraic matching algorithm the speed of detection of vulnerabilities in the binary code can be increased.

**Results**. The methodology of development of the vulnerabilities description in the terms of the behavior algebra has been proposed. The algebraic matching algorithm has been implemented in the scientific prototype system, which has been successfully used to check several programs for the known vulnerabilities.

**Conclusion.** The advantage of the algebraic approach is that the code vulnerabilities can be found more precisely, and the vulnerability description in the terms of behavior algebra can take in account different possible kinds of it. Also, the experiments with the implementation prototype shown that the "two-level" vulnerability detection system is faster than "pure" symbolic modeling: the fast matching step is executed first, and the slow modeling step is executed next on the results, provided by the matching step.

***Keywords***: *software vulnerabilities, symbolic modeling, algebraic matching, behavior algebra.*

*А.А. Летичевский*, доктор физ.-мат. наук, зав. отделом,
Ин-т кибернетики им. В.М. Глушкова НАН Украины,
03187, г. Киев, просп. Акад. Глушкова, 40, Украина,
oleksandr.letychevskyi@garuda.ai
*Я.В. Гринюк*, аспирант,
Ин-т кибернетики им. В.М. Глушкова НАН Украины,
03187, г. Киев, просп. Акад. Глушкова, 40, Украина,
yaroslav.hryniuk@garuda.ai
*В.М. Яковлев*, ведущий математик,
Ин-т кибернетики им. В.М. Глушкова НАН Украины,
03187, г. Киев, просп. Акад. Глушкова, 40, Украина,
victoryakovlev@ukr.net

## АЛГЕБРАИЧЕСКИЙ ПОДХОД К ФОРМАЛИЗАЦИИ УЯЗВИМОСТЕЙ В БИНАРНОМ КОДЕ

**Введение.** Безопасность и защита программных ресурсов в настоящее время — одна из наиболее актуальных проблем в ИТ-индустрии. Подходы к моделированию угроз и поиску уязвимостей, основанные на символьных методах, в последнее десятилетие стали весьма популярными и многообещающими. Данная статья описывает подход к поиску уязвимостей в двоичном коде, основанный на формальных методах символьного моделирования и алгебраического сопоставления.

**Цель статьи.** Хотя символьное моделирование может быть эффективно использовано для поиска уязвимостей в программном коде, соответствующие программные инструменты работают весьма медленно. Кроме того, проблема достижимости при символьном выполнении программ является неразрешимой в общем случае. В статье предлагается формализация представления двоичного кода и уязвимостей на основе алгебры поведений, а также подход, позволяющий ускорить поиск уязвимостей и сократить область символьного моделирования.

Алгебра поведений используется для представления поведения как двоичного кода, так и уязвимости. Однако, хотя получение представления бинарного кода в терминах алгебры поведений может быть автоматизировано, создание описания уязвимостей требует разработки корректной и эффективной методологии. При использовании представлений в терминах алгебры поведений задача поиска уязвимостей может быть решена в два этапа — относительно быстрого алгебраического сопоставления и собственно символьного моделирования на основе данных, полученных на этапе сопоставления.

**Методы**. Разработка описаний уязвимостей в терминах алгебры поведений и алгоритма алгебраического сопоставления позволяет ускорить алгоритмы поиска уязвимостей в двоичном программном коде.

**Результаты.** Предложена методика разработки описаний уязвимостей двоичного кода в терминах алгебры поведений. Разработан алгоритм алгебраического сопоставления, включенный в научный прототип, который успешно использован для проверки известных уязвимостей в нескольких программах.

**Выводы.** Преимущество алгебраического подхода состоит в том, что уязвимости в коде могут быть определены более точно, а описание уязвимости в терминах алгебры поведений позволяет учесть ее различные варианты. Кроме того, эксперименты с прототипом показали, что двухуровневая система поиска уязвимостей работает быстрее, чем чистая система символьного моделирования: вначале выполняется быстрый этап сопоставления, а затем медленный этап символьного моделирования на данных, полученных на предыдущем этапе.

***Ключевые слова:*** *уязвимости программного обеспечения, символьное моделирование, алгебраическое сопоставление, алгебра поведений.*