Algebraic Patterns of Vulnerabilities in Binary Code

Dr. OLEKSANDR LETYCHEVSKYI

V. Peschanenko, V. Radchenko, Y. Hryniuk, V. Yakovlev

Glushkov Institute of cybernetics
National Academy of Sciences of Ukraine,
Kyiv, prospect Glushkova, 40,
Ukraine

Garuda AI B.V.
54-62, Beachavenue, Schiphol-Rijk,
Netherlands
www.garuda.ai
The Defense Advanced Research Projects Agency (DARPA) launched the Cyber Grand Challenge to create defensive systems with purposes of automated, scalable, machine-speed detection of vulnerabilities and cyber infections.

MAYHEM
ForAllSecure team from Pittsburg

XANDRA
TECH team From Ithaka New York

Mechanical Phish
Shellphish team from Santa Barbara, California,

ALL THESE TEAMS USE SYMBOLIC (ALGEBRAIC APPROACH) IN DETECTION OF VULNERABILITIES IN BINARY CODE
Problems:
1. Pattern Matching.
    **Redundant.** Too many False Positive.
2. Simulation in isolated environment.
    **Insufficient.** Deep-hidden vulnerabilities cannot be detected.
3. Analysis of high-level programming languages.
    **Insufficient.** Security issues are on the level of third-party (non-compiled) libraries.
Our approach:

Use the algebraic matching and symbolic execution of binary code model.

1. Create algebraic model of binary code.
2. Formalize the existed vulnerabilities as algebraic patterns.
3. Provide algebraic matching of given vulnerability models with binary code model and detect the vulnerability candidates.
4. Prove the reachability of the detected vulnerability by symbolic execution.
Algebra of Behaviors

Algebra of behaviors was developed by D. Gilbert and A. Letichevsky (Senior) in 1997. It considers the operations over actions and behaviors.

Prefixing operation $a.B$ means that action $a$ follows behaviour $B$. The operation of nondeterministic choice of behaviours $u + v$ establishes alternative behaviours. The algebra has three terminal constants: successful termination $\Delta$, deadlock 0, and unknown behaviour $\perp$. The parallel and sequential composition are defined on the behaviors.

Example:

$$B_0 = a_1.a_2.B_1 + a_3.B_2,$$

$$B_1 = a_4.\Delta,$$

$$B_2 = \ldots$$

The example defines the order of event. The behavior $B_0$ has two alternative - first is two actions $a_1$ and $a_2$ and then the rest behavior $B_1$ or action $a_3$ and rest behavior $B_2$. Behavior $B_1$ is action $a_4$ and end of behavior etc.
Algebra of Behaviors

Every action is also defined by a couple, namely, the **precondition** and **postcondition** of an action, given as an expression in some formal theory.

\[
Action(A,B) = (A > B) \land \neg (A == 0) \rightarrow B = (B + 1)/A
\]

The semantic of the action presented in C-like syntax means that if precondition \((A > B) \land \neg (A == 0)\) is true for concrete values of \(A\) and \(B\) or is satisfiable for symbolic (arbitrary) values of \(A\) and \(B\), then we can change attribute \(B\) by the assignment \(B = (B + 1)/A\). The action can be parametrized by the attributes used in the action’s conditions.
Translation of x86 Assembler to Algebra of Behaviors

Listing after disassembling:

```
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
```

Model of binary code behavior:

```
B8049865 = sub(1,eax,0x81d01e0).B804986a,
B804986a = sar(1,eax,0x2).B804986d,
B804986d = mov(1,edx,eax,mov).B8049876,
B8049876 = jmp(1,jne).B8049879
```
Translation of x86 Assembler to Algebra of Behaviors

ALGEBRAIC ENVIRONMENT:
- the set of general- and special-purpose registers.
  Some attributes are identified as names of registers:
  \( ax,al,bx,bl,\ldots,eax,ebx,\ldots,rax,rbx,\ldots,ebp,esp,rbp,rsp,rip \)
- physical memory that can be considered as the function \( Memory(addr) \), where \( addr \) is the available memory address

\[
\begin{align*}
Bx1 &= cjne.Bz + \neg cjne.Bx2 \\
Bx2 &= \ldots \\
cjne(n,A,B) &= !(A == B) \rightarrow PI = PI + z + 3; \text{FLAG}_C = (B > A) \\
\neg cjne(n,A,B) &= (A == B) \rightarrow PI = PI + 3;
\end{align*}
\]
Translation of x86 Assembler to Algebra of Behaviors

Set of Instructions is converted to

Set of Algebra Behavior Expressions

Set of Algebra Behavior Actions

```
0000000000425060 <SSL_CTX_use_certificate_file>:
  425060: 41 55    push r13  
  425062: 41 54    push r12  
  425064: 49 09 f5  mov r13,rsi  
  425066: 55        push rsp  
  425068: 59        push rbx  
  425069: 48 99 fc  mov r12,rdi  
  42506b: 89 d5     mov ebp,edx  
  42506d: 48 83 ec 00  sub rsp,0x8  
  425072: e8 d9 24 fe ff  call 407550 <BIO_s_file8plt>  
  425078: e8 a1 31 fe ff  call 408220 <BIO_new8plt>  
  42507f: 48 05 c0  test rax,rax  
  425082: 0f b4 00 00 00 00  je 425138  
  425089: 4c 89 e5    mov rcx,r13  
  42508b: a_push_33766.b425062,  
  a_push_33767.b425064,  
  a_move_33760.b425067,  
  a_move_33762.b425068,  
  a_move_33770.b425069,  
  a_move_33771.b42506c,  
  a_move_33772.b42506e,  
  a_sub_33773.b425072,  
  a_call_33774.call b407550.b425077,  
  a_move_33775.b42507a,  
  a_call_33776.call b408220.b42507f,  
  a_call_33777.b425082,  
  a_call_33778.b425088,  
  a_move_33780.b42508b,
  a_move_33781.b42508f,
  a_move_33782.b425092,
  a_move_33783.b425095,

a_push_33767 = Operator1 -> ("m686: action 'push 425062';")  
(a_rpl := 4345964),  
  a_push_33767 = Operator1 -> ("m86: action 'pusg 425062';")  
(a_rpl := 434595e),  
  a_move_33760 = Operator1 -> ("m86: action 'mov 425064';")  
(a_rpl := 4345958; r13 := rdi),  
  a_push_33769 = Operator1 -> ("m86: action 'push 425067';")  
(a_rpl := 4345960),  
  a_push_33770 = Operator1 -> ("m86: action 'push 425068';")  
(a_rpl := 4345961),  
  a_push_33771 = Operator1 -> ("m86: action 'mov 425069';")  
(a_rpl := 4345964; r12 := rdi),  
  a_move_33772 = Operator1 -> ("m86: action 'mov 42506e';")  
(a_rpl := 4345966; ebp := edx),  
  a_sub_33773 = Operator1 -> ("m86: action 'subb 425066';")  
(a_rpl := 4345976; r8 := rax),  
  a_call_33774 = Operator1 -> ("m86: action 'call 425072';")  
(a_rpl := 4345975),  
  a_move_33775 = Operator1 -> ("m86: action 'mov 425077';")  
(a_rpl := 4345978; rdi := rax),  
  a_call_33776 = Operator1 -> ("m86: action 'call 425079';")  
(a_rpl := 4345963),  
  a_test_33777 = Operator1 -> ("m86: action 'test 42507f';")  
(a_rpl := 434598e),  
  a_je_33778 = Operator[SF = 1] -> ("m86: action 'je 425087';")  
(a_rpl := 4345992),  
  a_alr_33779 = Operator[SF = 1] -> ("m86: action 'jne 425095';")  
(a_rpl := 4345992),  
  a_move_33780 = Operator1 -> ("m86: action 'mov 425088';")  
(a_rpl := 4345995; rcx := r13).```
Algebraic Patterns of Vulnerabilities

\[ \text{VulnerabilityPattern} = \text{IntruderInput}; \text{ProgramBehavior}; \text{VulnerabilityPoint} \]
Algebraic Patterns of Vulnerabilities

**BUFFER OVERFLOW VULNERABILITY**

$vulnerabilityBufferOverflow = input; X1; allocateStack; X2; writeStack,$

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

\[
allocateStack = push(1,ebp).mov(2,ebp,esp).sub(3,esp,N),
\]

\[
writeStack = movs(8, MemoryOperand, MemoryOperand) + mov(5, MemoryOperand, regGen)
\]
Algebraic Patterns of Vulnerabilities

ACTIONS:

\[
\begin{align*}
call(12, \text{MemoryOperand}) &= \text{Forall}(i: \text{int}, 0 \leq i < \text{LengthSocket}) \rightarrow \text{Input}(ecx + i) = \text{true}, \\
\text{mov}(2, \text{ebp}, \text{esp}) &= 1 \rightarrow \text{StackAddr} = \text{ebp}, \\
\text{movs}(8, \text{MemoryOperand}, \text{MemoryOperand}) &= \text{Input}(\text{RefMemorySrc}) \land (\text{StackAddr} = \text{RefMemoryDest}) \rightarrow 1, \\
\text{mov}(5, \text{MemoryOperand}, \text{regGen}) &= \text{Input}(\text{RefMemorySrc}) \land (\text{StackAddr} = \text{RefMemoryDest}) \rightarrow 1, \\
\text{mov}(x, \text{GenReg}, \text{MemoryOperand}) &= \text{Input}(\text{RefMemorySrc}) \rightarrow \text{Input}(\text{GenReg}), \\
\text{mov}(x, \text{MemoryOperand}, \text{MemoryOperand}) &= \text{Input}(\text{RefMemorySrc}) \rightarrow \text{Input}(\text{RefMemoryDest})
\end{align*}
\]

Sufficient conditions of vulnerability:
1. The bytes are written in address that is equal to the top stack pointer;
2. The bytes shall be received from the registers after system call.
Behavior Matching

The first level of matching is to find behavior expressions that will find the behavior corresponding to the algebraic pattern of a vulnerability.

Behavior matching anticipates solving behavior expressions.

The task of solving behavior equations is formulated as follows. Let $B_0$ be the system of behavior equations:

$B_0 = R(a_1, a_2, \ldots, B_1, B_2, \ldots)$,

$B_1 = R(a_{11}, a_{12}, \ldots, B_{11}, B_{12}, \ldots)$, ...

where $B_1, B_2, \ldots B_{11}, B_{12}, \ldots$ are the behaviors and $a_1, a_2, \ldots$ are actions. The behavior $B_0$ shows translated binary code.

Let behavior $X$ be an unknown behavior containing a vulnerability. The task is to find $X$, that is $B_0 = Y; X; Z$ and $X = vulnerabilityBufferOverflow$
Model Matching

Model matching is performed by the symbolic modelling of a given behavior that was obtained by behavior matching.

During symbolic modelling, we apply the actions for which we detect the satisfiability of the expression $Env \land Prec$, where $Env$ is a symbolic environment of the model of the binary code and $Prec$ is the precondition of the matched action.

If it is satisfiable, then we perform the postcondition operations in the pattern environment and in the environment of the binary code model.

If we reach the vulnerability point in the pattern, then we have a scenario that leads from the input point.
Exploit Generation

With the symbolic environment represented by the set of formulae, it is possible to realize a concrete scenario or to define input values that cause stack buffer overflow.

Backward symbolic modeling to the input point gives the initial formula that covers the values enabling exploits to perform malicious actions.
Detection of Vulnerabilities in Binary Code

Source Code → compiling, program linking → Conversion of code to algebraic model → Data Base of Algebraic Patterns of Vulnerabilities (from CVE)

Algebraic Matching:
- Behavior matching (model of code and patterns);
- Action matching (symbolic modeling of detected behaviors)

Verdict, Counterexample
Analysis of Suspicious Process

- OS
- Process1
- Process2
- New Process

Network or desktop

Analyser

Conversion to algebra.

Algebraic Matching

Verdict, Counterexample

Intruder's Behaviors Patterns
Platform for Detection of Vulnerabilities in Binary Code

- Upload the binary code;
- Disassembling;
- Conversion to behavior algebra;
- Matching for vulnerabilities;
- OPEN for creation the new patterns of vulnerabilities.

www.garuda.ai

TWO-LEVEL ORGANIZATION IS MORE EFFICIENT:

- Reduce the time of matching;
- Reduce the number of FALSE POSITIVE;
- The accuracy is high
THANK YOU FOR ATTENTION

oleksandr.letychevskyi@garuda.ai