The 9th IEEE International Conference on Dependable Systems, Services and Technologies, DESSERT'2018
24-27 May, 2018, Kyiv, Ukraine

Algebraic Methods for Detection of Vulnerabilities in Software Systems

Oleksandr Letychevskyi
Glushkov Institute of cybernetics of
National Academy of Sciences of Ukraine,
40, Glushkov propspect, http://www.icyb.kiev.ua/
Senior Researcher, lit@litsoft.com.ua

***Abstract*—The paper presents an algebraic approach for finding vulnerabilities in a program system that is given as the sequence of processor instructions. The is the transformation of code to algebraic specifications and providing its symbolic modeling for the detection of vulnerability cases that are presented as formulas in logic language. The method anticipates the usage of solving and proving systems integrated with the Algebraic Programming System.**

Keywords—cybersecurity; cyberattacks; formal methods; behavior algebra

I. INTRODUCTION

Nowadays in conditions of cyberwar the topicality of increasing of cybersecurity level is extremely high. The cyberattacks are becoming both more impudent and more mature. Banking system mischief, election interference, theft of confidential information, data destruction and other examples of cyberattacks arise the most challenging problem of counteractions to intruders.

Static analysis of code for searching for vulnerability, different hardware like firewalls and huge amount of software application are insufficient for repelling of hackers attacks and countering of viruses penetration.

Though the growth of the different theories, methods, techniques in computer science gives the hope to win this race in the most cases. The last achievements in deductive tools, model checking, symbolic computations caused the creation of more efficient formal methods for detection of vulnerabilities statically and online.

There are many kinds of vulnerabilities fixed in CVE [1] data base like SQL injection, operation system injections, buffer overflow, cross-site scripting, missing authentication.

Also there are two majority flaws in microprocessors that could allow hackers to steal confidential information from memory of computers, mobile devices, cloud networks.

The vulnerability called Meltdown [2] exists in current architecture more then 20 years and there are no reliable protection. Such companies as Intel, Google, Amazon provide by security advisory and patches for different OS but the development of new kinds of attacks make these measures less efficient. The same is for Spectre vulnerability that breaks the isolation between different applications and allows an attacker to trick error-free programs. Spectre is harder to exploit than Meltdown.

Usage of formal algebraic methods to verify system vulnerabilities has become relevant due to the growth in complexity of the designs common in hardware and software industries. Usually vulnerability detection is by a reachability of properties of an abstract mathematical model of the system. Examples of such mathematical models are finite state machines, labeled transition systems, Petri nets, or process algebras. There are two main approaches to establish properties of such models: Model checking [3] is an exhaustive exploration of the states and transitions of the mathematical model. It subsumes different techniques such as abstract interpretation [4], symbolic simulation [5], abstraction refinement [6], and others. The other approach is automated proving where the model is presented as a set of assertions and the properties are established by theorem provers such as HOL, Z3, or Isabelle.

Static analysis tools detect vulnerabilities by consideration the application source code and examining all possible execution paths and variables values without its execution. Static analysis tools present a set of warnings that have to be inspected manually and classified as true weaknesses or false alarm.

II. ALGEBRA OF BEHAVIORS AND INSERTION MODELING

The theory of interaction between agents and environments is known as Insertion Modelling [7]; it was developed as a generalization of the interaction of transition systems or automaton networks.

Algebra of behavior was developed by D. Gilbert and A. Letichevsky in 1997 [8]. It was realized in the scope of the Insertion Modeling System (IMS) as an extension of Algebraic Programming System (APS) developed in Glushkov Institute of Cybernetics. Behavior algebra is a two-sorted universal algebra. The main sort is a set of behaviors and the second sort is a set of actions. The algebra has two operations, three terminal constants, and a relation of approximation. The operations are the prefixing *a.u* (where *a* is an action, and *u* is a behavior) and non-deterministic choice of behaviors *u* + *v* (associative, commutative, and idempotent operations on the set of behaviors). The terminal constants are successful termination Δ, deadlock *0*, and non-determinate behavior $\perp$. The relation of approximation $\sqsubseteq$ is a partial order on the set of behaviors with minimal element $\perp$. The following example of behavior expressions

B0 = a1.a2.B1 + a3.B2,
B1 = a4.⊥,
B2 =…

means that behavior *B0* could be interpreted as sequence of action *a1*, *a2* and the behavior *B1* afterwards, or the action *a3* with the next behavior *B2*. Behavior *B1* will finish after action *a4*.

The Basic Protocols language has been developed in the scope of the Verification of Requirements Specifications (VRS) project [9], implemented together with Motorola and the Glushkov Institute of Cybernetics. The language is built over some attribute environment, where agents interact one with another. Every agent is defined by a set of attributes. An agent changes its state under some conditions formed by values of attributes. Every agent's actions defines some basic protocol that is a triple: *B* = <*P, A, S*>, where *P* is a precondition of basic protocol presented as a formula in some basic logic language, *S* is a postcondition, and *A* is a process that illustrates agent transition. As a basic logical language, we consider the set of formulas of first-order logic over linear arithmetic. As a whole, the semantic of a basic protocol means that the agent could change its state if the precondition is true and the state will change correspondingly to the postcondition, which is also a formula of first order logic. The postcondition could also contain an assignment statement. The process of basic protocol depends on the subject domain and illustrates the sequence of basic protocols application. In a telecommunications domain, it could be the sending or receiving of signals with corresponding parameters.

III. Algebraic Presentation of Machine Instructions

The description of the architecture of the Intel 64 and IA-32 processors is presented in [10]. The architecture is composed of the attribute environment, where attributes are the set of general purpose registers (AH, AL, AX, EAX, RAX,…) of different types (byte, word, double word,…), and different bit capacities. Moreover, we consider as attributes the set of flags that are contained in the EFLAGS/RFLAGS register. In a huge amount of instructions we distinguish: - control flow instructions (JCC, JMP, CALL, …) that provide navigation via program code corresponding to attributes values; - instructions that change the attributes environment. This is a set of ALU instructions. These instructions change the values of registers or memory, can provide calculation, and compare values in registers with settings of corresponding flags. We transform the sequence of instructions into behavior algebra expressions with actions that are the basic protocols with preconditions containing predicates and postconditions that define changing attributes. Consider the following fragment of code (Figure 1)

```
0000000000425060 <SSL_CTX_use_certificate_file>:
  425060:  41 55                 push   r13
  425062:  41 54                 push   r12
  425064:  49 89 f5              mov    r13,rsi
  425067:  55                    push   rbp
  425068:  53                    push   rbx
  425069:  49 89 fc              mov    r12,rdi
  42506c:  89 d5                 mov    ebp,edx
  42506e:  48 83 ec 08           sub    rsp,0x8
  425072:  e8 d9 24 fe ff        call   407550 <BIO_s_file@plt>
  425077:  48 89 c7              mov    rdi,rax
  42507a:  e8 a1 31 fe ff        call   408220 <BIO_new@plt>
  42507f:  48 85 c0              test   rax,rax
  425082:  0f 84 b0 00 00 00     je     425138
<SSL_CTX_use_certificate_file+0xd8>
  425088:  4c 89 e9              mov    rcx,r13
```

Figure 1. Example of code

which could be translated to algebra behavior expressions (Figure 2).

```
B425060 = a_push_33766.B425062,
B425062 = a_push_33767.B425064,
B425064 = a_mov_33768.B425067,
B425067 = a_push_33769.B425068,
B425068 = a_push_33770.B425069,
B425069 = a_mov_33771.B42506c,
B42506c = a_mov_33772.B42506e,
B42506e = a_sub_33773.B425072,
B425072 = a_call_33774.call B407550.B425077,
B425077 = a_mov_33775.B42507a,
B42507a = a_call_33776.call B408220.B42507f,
B42507f = a_test_33777.B425082,
B425082 = a_je_33778.B425138 + a_alt_je_33779.B425088,
B425088 = a_mov_33780.B42508b,
```

Figure 2. Behavior expressions

The actions in behavior could be presented as the following (Figure 3).

```
a_push_33766 = Operator(1  -> ("x86: action 'push 425060';")
(rip := 4345954)),
a_push_33767 = Operator(1  -> ("x86: action 'push 425062';")
(rip := 4345956)),
a_mov_33768 = Operator(1  -> ("x86: action 'mov 425064';")
(rip := 4345959; r13 := rsi)),
a_push_33769 = Operator(1  -> ("x86: action 'push 425067';")
(rip := 4345960)),
a_push_33770 = Operator(1  -> ("x86: action 'push 425068';")
(rip := 4345961)),
a_mov_33771 = Operator(1  -> ("x86: action 'mov 425069';")
(rip := 4345964; r12 := rdi)),
a_mov_33772 = Operator(1  -> ("x86: action 'mov 42506c';")
(rip := 4345966; ebp := edx)),
a_sub_33773 = Operator(1  -> ("x86: action 'sub 42506e';")
(rip := 4345970; rsp := rsp - 8; ZF := (rsp - 8 = 0); PF :=
((rsp - 8) = 0); SF := (rsp - 8) < 0))),
a_call_33774 = Operator(1  -> ("x86: action 'call 425072';")
(rip := 4345975)),
a_mov_33775 = Operator(1  -> ("x86: action 'mov 425077';")
(rip := 4345978; rdi := rax)),
a_call_33776 = Operator(1  -> ("x86: action 'call 42507a';")
(rip := 4345983)),
a_test_33777 = Operator(1  -> ("x86: action 'test 42507f';")
(rip := 4345986)),
a_je_33778 = Operator((ZF = 1)  -> ("x86: action 'je
425082';")(rip := 4345992)),
a_alt_je_33779 = Operator(( ~(ZF = 1))  -> ("x86: action 'je
425082';")(rip := 4345992)),
a_mov_33780 = Operator(1  -> ("x86: action 'mov 425088';")
(rip := 4345995; rcx := r13)),
```

Figure 3. Actions of behaviors

Thereby the behavior expressions present the control flow of the program, and the actions define the changing of the attributes by means of the basic language.

IV. Insertion Model of Intruder Attacks

On the high level of abstraction in the terms of insertion modelling we can see the parallel composition of the following agents processing in computer environment.

CPU || P1 || P2 || Attacker || OS Kernel || Scheduler|| ...

CPU processes with all kinds of memory and communicates with all agents. It has its cash memory for efficiency of computations. P1, P2 are the processes launched in the environment by different users. Attacker is also a process that tries to access the secured memory. OS Kernel, Scheduler and other OS components can work with authorized memory.

Physical memory is an environment for all agents and every its cell is a byte located correspondingly to address. It is defined by function:

Mem(Address) = Byte

In the modern architecture the memory is paged and every agent works with its own memory segment. Memory can be authorized with limited access for other agents. *Access(Address)* is Boolean function that defines authorized memory.

Every process in the environment has the model given as expressions of behavior algebra. The parallel composition of agents can be modeled by means of Insertion Modeling System.

The modeling is performed symbolically. It means that interpretation of behavior algebra expressions is performed on the abstract level without concrete values of environment attributes. Symbolic modeling is in the basis of the behavior algebra transformations used in formal methods.

Consider the behavior of attacker on the example of the known vulnerability caused by exploit Meltdown. The given attack uses the speculative executions techniques that are implemented in a current OS. This technique anticipates parallel or pseudoparallel execution of program branches that are independent. These branches could be used or not correspondingly to the state of the program environment. The result of preliminary execution is located in cash memory of processors. Cash can also contain the access to authorized memory. Intruder uses this situation and detects the values of bytes from authorized memory by means of side channel technique. It especially measures time of access to the cells.

Presentation of possible intruder behavior on the high level of abstraction is the following.

B0 = a1.(a2.B0 + a3.B1)

where *a1* - measuring of access time, *a2* - if time is close to quick access to cash then fixes the current side channel value (index of array), *a3* - if time is much more then time of access to cash, then continues measure access time.

On the level of instructions the presentation of behavior algebra expression is more detailed and it expresses the situation when “someone measures in a cycle the time of access to cash”. Such model could be recognized by special formal methods and detected as an intruder.

Complexity of the model and the degree of its abstraction can be changed and refined if we consider other similar behaviors. For this purposes the deep learning technique could be used for the set of generated and real cases.

V. Methods of Detection of Intruder Attacks

Static detection of attack is implemented by means of symbolic modeling. Given a model of executable code and the property that expresses the case of vulnerability, we use the symbolic methods for checking it that are implemented in IMS (APS).

The initial state of the program, especially values of registers and initial flags, can be presented by an initial formula. This formula can contain known (or predefined) values of attributes and unknown (arbitrary symbolic) values. Starting from the initial formula, we can apply the basic protocol corresponding to the control flow that is expressed in behavior algebra. The basic protocol is applicable if its precondition is satisfiable and consistent with the state of the environment. Starting from the formula of the initial state *S0* and from the initial behavior *B0*, we select the action and move to the next behavior. We check on the first step the satisfiability of the conjunction

S0 /\ Precodnition(a1)

if *B0 = a1.B1*. The next state of the environment will be obtained by means of the predicate transformer; that is,

the function over the state of the environment and the postcondition

PT(S0,Postcondition(a1)) = S1

The output is the new state of the environment expressed by the formula over the attributes.

Moving from the initial formula and applying the basic protocols, we will obtain the sequence of states or formulas over the attributes that define some possible trace of program execution. For a large system, this method might be unsuccessful due to the exponential explosion, so the suspected point might never be reached. To overcome this limitation, we can use backward symbolic modeling from the suspected point to the initial state. If all traces from the state presenting vulnerability lead to deadlocks, then vulnerability is unreachable.

If the APS dynamic method can be combined with static methods, the computation of invariants is possible in APS by different manners, especially the use of static detection of cycle invariants and methods of approximation of the invariant formula. In the case of approximation, we can compare the approximated formula with the vulnerability cases, and detect the issues earlier then the invariant will be computed. It should be taken into account that the problem of reachability is unresolved in a general way, so complete absence of vulnerability is not guaranteed.

Reduction states that the traversal could also be reached by use of those parts of the code that affect the formula of vulnerability. For this purpose, we can consider a slice (or subset) of behavior expressions and use only vulnerability formula attributes and its dependencies.

Static detection of possible attack is used for detection of vulnerability of program and obtaining of counterexample for its demonstration. Moreover by means of IMS(APS) the set of traces can be generated that is to be considered as the set of tests for testing of cybersecurity software or firewalls.

Dynamic analysis is implemented by creation of special agent - listener which analyses incoming instructions and matches with abstract intruder behavior model. It can signalize when the behaviors are equivalent or provide corresponding protection. Checking for equivalence of abstract model and concrete agent's behavior is implemented by deductive tools with automated proving.

VI. Conclusion and Future Works

The given technology can resolve two main shortcomings of formal methods usage. First is the problem of involving of non-compiled part of program project such as libraries or third party modules. Usually the checking is implemented in the scope of the high level programming language like C++ or Java. The given technology is implemented on the level of instructions and behavior of non-compiled component is taken into account.

The second issue is that possible fake attacker can be presented. The accuracy of symbolic methods is much higher then heuristics methods or analyzing of patterns of environment signalizing about intruder actions.

Now the number of intruder behavior models are created. There are different examples from CVE and CWE data bases and especially models of Meltdown and Spectre exploits. It is anticipated to create huge data base of exploits for efficient security. For this purpose the formal methods are improved for greater efficiency and adjusted for large scaled projects with processing of millions code lines.

References

[1] Common Vulnerabilities and Exposures (CVE®) https://cve.mitre.org/

[2] Bright Peter, "Meltdown and Spectre: Here's what Intel, Apple, Microsoft, others are doing about it". Ars Technica. Retrieved January 6, 2018.

[3] Doron Peled, Patrizio Pellicone, Paola Spoletini, "Model Checking", Wiley Encyclopedia of Computer Science and Engineering, 2009.

[4] Patrick Cousot, "Formal Verification by Abstract Interpretation", Lecture Notes in Computer Science, 2012, vol. 7211, pp. 3—7, Springer.

[5] Robert B. Jones, "Symbolic Simulation Methods for Industrial Formal Verification", 2002, Springer.

[6] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith, "Counterexample-Guided Abstraction Refinement", Lecture Notes in Computer science Volume, 1855, 2000, pp. 154-169.

[7] Letychevskyi O.,Letichevsky A., Peschanenko V. Insertion Modeling System And Constraint Programming, CEUR–WS : [Workshop Proceedings V. Ermolaev eds.]: http://ceur-ws.org, 2011. – № 716. pp. 51–64.

[8] A. Letichevsky and D. Gilbert, "A Model for Interaction of Agents and Environments," in: Resent trends in Algebraic Development technique, LNCS 1827 (D. Bert and C. Choppy, eds.), Springer-Verlag, pp. 311-328, 1999.

[9] A. Letichevsky, O. Letychevskyi, T. Weigert, J. Kapitonova, V. Volkov, S. Baranov, V. Kotlyarov, "Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specications," Computer Networks, vol. 47, 2005, pp. 662-675.

[10] Intel 64 and IA-32 Architectures Software Developer's Manual, Intel Corporation, 1997-2016.