The ISC Int'l Journal of Information Security

ISeCure

January 2019, Volume 11, Number 3 (pp. 1–7)

http://www.isecure-journal.org

Selected Paper at the ICCMIT'19 in Vienna, Austria

Algebraic Matching of Vulnerabilities in a Low-Level Code☆

Oleksandr Letychevskyi [1,*], Yaroslav Hryniuk [1], Viktor Yakovlev [1], Volodymyr Peschanenko [2], and Viktor Radchenko [2]

[1] *Glushkov Institute of Cybernetics of National Academy of Sciences of Ukraine 40, Glushkova prospect, Kyiv, Ukraine*
[2] *Garuda AI B.V., 54-62, Beachavenu, Schiphol-Rijk, Netherlands*

ARTICLE INFO.

Keywords:
Algebraic Matching; Symbolic Modeling; Behavior Algebra; Rewriting Rules; Vulnerability Detection

Abstract

This paper explores the algebraic matching approach for detection of vulnerabilities in binary codes. The algebraic programming system is used for implementing this method. It is anticipated that models of vulnerabilities and programs to be verified are presented as behavior algebra and action language specifications. The methods of algebraic matching are based on rewriting rules and techniques with usage of conditional rewriting. This process is combined with symbolic modeling that gives a possibility to provide accurate detection of vulnerabilities. The paper provides examples of formalization of vulnerability models and translation of binary codes to behavior algebra expressions.

© 2019 ISC. All rights reserved.

1 Introduction

The algebraic approach in cybersecurity was demanded over the past two decades with the appearance of efficient solving and deductive tools. Different techniques like symbolic modeling [1] and concolic computations [2] use this approach because it has created more possibilities for detection in cybersecurity.

The Defense Advanced Research Projects Agency (DARPA) launched the Cyber Grand Challenge [3] to create defensive systems with purposes of automated, scalable and fast detection of vulnerabilities and cyber threats. In 2016, there were three winners that used the symbolic modeling technique as a detection algorithms, which gives the possibility to provide more efficient implementation of program paths traversal.

However, there are still a number of problems related to efficiency and accuracy of vulnerability detection solutions. Specially, complex symbolic modeling algorithms require using deductive tools, like solving and proving machines that are much slower than a heuristic search or matching for compliance with vulnerability indicators. The other problem is the inaccuracy of these algorithms because of insufficient formalization of vulnerability signatures that entails a fake detection or impossibility.

This paper considers the algebraic approach that we apply when searching for vulnerability by combining the rewriting rules technique and symbolic modeling.

2 Main Approach

The main idea of the method is to present the vulnerability description in a formal language and match it

☆ The ICCMIT'19 program committee effort is highly acknowledged for reviewing this paper.

* Corresponding author.

Email addresses: oleksandr.letychevskyi@garuda.ai (O. Letychevskyi), yaroslav.hryniuk@gmail.com (Y. Hryniuk), victoryakovlev@ukr.net(V. Yakovlev), volodymyr.peschanenko@garuda.ait(V. Peschanenko), viktor.radchenko@garuda.ai(V. Radchenko)

ISSN: 2008-2045 © 2019 ISC. All rights reserved.

with binary code in the verification phase. Both the code and vulnerability model are presented by using *behavior algebra expressions* [4].

Behavior algebra is a two-sorted universal algebra. The main sort is a set of behaviors, and the second one is a set of actions. The algebra has two operations, three terminal constants and an approximation relation. The operations has the prefix $a.u$ (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 can be among successful termination Δ, deadlock 0 and unknown behavior $\perp$. This algebra is also enriched by two operations: parallel ($||$) and sequential (;) compositions of behaviors. Examples of behavior expressions are given below:

$$1B0 = a1.a2.B1 + a3.B2 \tag{1}$$

$$B1 = a4.\Delta \tag{2}$$

$$B2 = \perp \tag{3}$$

These imply that behavior $B0$ could be interpreted as a sequence of actions $a1$ and $a2$ followed by behavior $B1$, or as action $a3$ followed by unknown behavior $B2$. Behavior $B1$ will finish after action $a4$.

This action language has been developed within the scope of the Algebraic Programming System (APS) [5] that was implemented at the Glushkov Institute of Cybernetics. It is built over an attribute environment that changes its state under some conditions formed by values of attributes. Every action is defined by the precondition and postcondition as a formula in some basic logic language. As a basic logical language, we consider the set of formulas of first-order logic over polynomial arithmetic. As a whole, the semantic of the action means that the environment can change its state if the precondition is satisfiable and the state will change correspondingly to the postcondition. The postcondition can also contain an assignment statement that defines new state of the environment.

In the proposed solution, the general scheme of the algebraic approach implies translation of binary code into behavior algebric expressions and sets of actions. During the first stages, we should disassemble the input binary code. Reading the instructions of compiled and linked programs, we can process the part of the system that contains third-party tools or libraries that can also be the source of vulnerabilities. In the development environment, assembler code can be directly produced by the compiler. Having disassembled the code, we can translate it into the behavior algebra expressions and set of actions defining the semantic of instructions through action language.

The next stage is to use the database of vulnerability models that are created from the description of known vulnerabilities. These models can be derived from a standard database of vulnerability, like Common Vulnerabilities and Exposures [6]. After preparation, we use the algebraic programming system and its component for further implementation of algebraic matching. The translated code and database of vulnerabilities are the inputs of the algebraic matching.

3 Formal Model of Code

We consider the low-level code as the set of instructions of the Intel 64 and IA-32 processors. It shall also be considered as the interaction between the processor and memory in an algebraic environment. The architecture is composed of the attribute environment, where attributes are the set of general-purpose registers (AH, AL, AX, EAX, RAX, etc.) of different types (byte, word, double word, etc.) and different bit capacities. Moreover, we consider as attributes the set of flags that is contained in the EFLAGS/RFLAGS register. In a large amount of instructions, we distinguish:

- control flow instructions (e.g. JCC, JMP, CALL, etc.)
- instructions that change the attribute environment. 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 algebraic expressions with actions with preconditions containing predicates and postconditions that define changing attributes. For example, branch instruction $60c984 : jne\ 60cb50$ can be converted to a behavior algebra expression, covering possible outcomes based on the state of the ZF flag in EFLAGS:

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

where the actions are given as follows:

$$a_jne1 = (ZF = 0) \rightarrow 1 \tag{4}$$

$$a_jne2 = \sim (ZF = 0) \rightarrow 1 \tag{5}$$

We denote behavior identifiers together with the hexadecimal address of instructions in a program segment for traceability to the assembly code. The expression above means that the instruction at the address $60c984$ will pass the control to the instruction at the address $60cb50$ if flag ZF is equal to 0; otherwise the next instruction at the address $60c98a$ will be performed. The preconditions of action are equalities $(ZF = 0)$ and $\sim (ZF = 0)$. The postcondition is absent, so it is equal to 1. This means that the environment state is not changed after the instruction is performed.

The instructions that change the environment and its attributes can be presented as actions with a postcondition containing this change. For example, the instruction:

$$60c99d : add\ DWORD\ PTR[r13 + 0x15c], r8d$$

will be transformed to

$$B_60c99d = a_add_3480.B_60c9a4$$

where

$$a_add_3480 = 1 \rightarrow \quad (6)$$

$$Memory(r13 + 348) := Memory(r13 + 348) + r8d; \quad (7)$$

$$ZF := (Memory(r13 + 348) + r8d = 0); \quad (8)$$

$$SF := (Memory(r13 + 348) + r8d < 0) \quad (9)$$

This instruction performs the addition of a memory element that is available at the given address in register $r13$ to the content of double word register $r8d$. The given flags will be set to bit 1 or 0 corresponding to the truth of the given equality or inequality. There are other flags (e.g., CF, OF, AF, PF, etc.) that are affected, but these are not illustrated for simplicity.

All the semantics of instructions have been defined directly following specification in the data sheet, and we see that formalization of executable code is not that complicated for representation with formal logic language. Consider the following code fragment:

```
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>
```

Figure 1. Example of code.

which can be translated to algebra behavior expressions:

```
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
```

Figure 2. Behavior expressions.

The actions in behavior can be presented as the following in specialized syntax:

```
a_push_33766 = Operator(1->("x86:action 'push 425060';")
(rip := 4345954)),
a_push_33767 = Operator(1->("x86:action 'push 425062';")
(rip := 4345956)),
a_push_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_push_33771 = Operator(1->("x86:action 'mov 425069';")
(rip := 4345964; r12 := rdi)),
a_push_33772 = Operator(1->("x86:action 'mov 42506c';")
(rip := 4345966; ebp := edx)),
a_push_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),
```

Figure 3. Actions of behaviors.

Therefore, 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. Further, they will be considered as the input of algebraic matching.

4 Formal Model of Vulnerability

Vulnerability implies undesired behavior of a program that an attacker can exploit for malicious actions. It can be caused by development errors, developerâĂŹs backdoors, or bad design. Algebraic matching performs a smart search of vulnerability into the binary code, so together with an algebraic model of binary code we shall develop a model of vulnerability.

We consider the example of model creation studying the buffer overflow vulnerability. It is a known vulnerability described by [7]. Considering the program behavior, we can see that access to addresses of bytes that are larger than the declared buffer length can offer access to automatic memory - especially the stack that leads to execution of code. An attacker can copy the address of malicious code to this part of the stack and launch it.

The model of vulnerability will be created in a behavior algebra expression model and derived from the assembler (x86) code. Our solution is applicable for C-language with the corresponding assembler set of instructions:

In the above rectangles, there is the assembler instructions that correspond to the C-code and present the given vulnerability. The process of creating the model begins with the first stage which is the creation of the behavior algebra expression. The given example is simple and we can define the sequence of actions that corresponds to the sequence of instruc-

```
#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 4889E5       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, BYTE 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
22    01
23 0020 488345F8     add QWORD PTR [rbp-8], 1
23    01
24                   .L2:
25 0025 488B45F0     mov rax, QWORD PTR [rbp-16]
26 0029 0FB600       movzx eax, BYTE 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
33                   .cfi_def_cfa 7,8
```

Figure 4. Code of vulnerability "buffer overflow".

tions such as:

$$mov.movzx.mov.mov.add.add.mov.movzx.test.jne$$

The sequence expresses copying a buffer in any memory cell. The features of the example are that the buffer is declared as a parameter of function and it shall be allocated in the stack. Also, in the set of instructions, we have the cycle that checks the end of the byte sequence as the end of the string (0). These two features are the sources of vulnerability.

Thus, we have the behavior algebra expression that implies the cycle:

$$B1 = a1.a2.a3.a4.a5.a6.a7.a8.a9.(a10.B1 + !\,a10.B2)$$

with the following actions, that contain the following preconditions:

$$a1 : (mnem(a1) = mov)\&(arg1(a1) = rax)\&$$
$$(arg2(ptr) = rbp - 16)\&$$
$$(arg2(format) = QWORD)$$
$$a2 : (mnem(a2) = movzx)\&(arg1(a1) = edx)\&$$
$$(arg2(ptr) = rax)\&(arg2(format) = BYTE)$$
$$\ldots$$

where $mnem(x), arg1(x), arg2(x)$ are the predicate that define the semantic of every instruction and state of environment before its being performed.

This expression shows the very concrete situation that was performed in corresponding C-code. To create the general algebraic model, we should parametrize the model. For example, the call of parameters can be arbitrary so $arg2(ptr) = rbp - PP * 8$, where PP shall be calculated correspondingly to the stack reserved automatic memory. The other situation is that the rax-register can be occupied and the other general-purpose registers can be used. In this case we should define $arg2(ptr) = XX$ where $XX = rax || XX = rbx || \ldots$ or define predicate $regType(arg2(ptr))$.

In addition, there can be the situation when the vulnerability code is hidden inside the other code, or it can alternate with the independent sequence of statements in C-code or instructions in the assembler code. We can denote the average behavior and obtain the following sequence:

$$B1 = a1.X1.a2.X2.a3.X3.a4.X4.a5.X5.a6.X6.$$
$$a7.X7.a8.X8.a9.X9.a10.X10.B1$$

where Xi âĂŞ is a behavior. This behavior also can be restricted by environment state. For example, the value of rax-register between $a1$ and $a2$ shall not be changed. We can define a precondition for behavior that we will use for algebraic matching. For example, for $X1$:

$$(var = rax),$$

where var is a special variable for the storing of our rax value. Then, in the precondition of $a2$ we will use $(rax = var)$.

The process of model creation can be automated fully or particularly after correct preparation of the source and assembler code. The more we can parametrize the algebraic model of vulnerability, the more effectively we can detect additional varieties of the given vulnerability in a code.

5 Algebraic Matching

Algebraic matching is based on the non-deterministic system of rewriting that was implemented within the scope of the algebraic programming system. This system has been implemented in the Glushkov Institute of Cybernetics of the National Academy of Science of Ukraine in 1987. Historically, APS is the first system that has started to use the technology of term rewriting in combination with user-defined strategies of rewriting.

Algebraic programming is based on rewriting. It extends functional programming and has applications in solving algebraic compute problems (word problems in finite algebras, or completion algorithms like Knuth-Bendix or Buchberger) and in operational semantics of programming languages (executable algebraic specifications of software components, definitions of operational semantics of programming languages, or developing interpreters and prototypes of software components).

The rewriting system contains the set of rewriting rules. We specialized the rewriting rule that can be presented as the following equality:

$$A(x, y, \ldots) = B(x, y, \ldots)$$

where $A(x, y, \ldots)$ and $B(x, y, \ldots)$ are algebraic expressions over the variables $x, y, \ldots$ that are behaviors and actions.

The algebraic system matches two expressions and rewrites A correspondingly to B, performing the substitution of matched attributes. Rewriting rules can be also conditional such as:

$$C(a, b, \ldots) \rightarrow A(x, y, \ldots) = B(x, y, \ldots)$$

where $a, b, \ldots$ are the attributes of environment. This means that rewriting can be performed if condition $C(a, b, \ldots)$ over attributes is true.

Strategy is a function that defines the strategy of rewriting; for example, left-side or right-side rewriting. There also can be user-defined rewriting defining the coverage of states of environment.

With the above descrition, we can present the model for detecting vulnerabilities with the following system of rewriting rules:

$$precond(a1) \rightarrow a1.y = 1, y,$$
$$precond(X1) \rightarrow 1, X1.y = 2, y,$$
$$precond(a2) \rightarrow 1, a2.y = 2, y,$$
$$precond(X2) \rightarrow 2, X2.y = 3, y,$$
$$\ldots$$
$$precond(a7) \rightarrow 10, a10.y = Delta$$

In thid model, we have the conditional rewriting rules that are defined by preconditions of corresponding actions and average behaviors $(X1, X2, \ldots)$. The rewriting starts in a program to be verified from the first occurrence of action $a1$ where its precondition is satisfiable. If such a case exists, then we start the rewriting from this point and it means that the first rule $a1.y$ is matched. The variable y indicates that the rest of the program must be verified and we rewrite $a1.y$ as $1, y$. The numbering means the order of the matched behaviors. The rewriting process will be performed if the corresponding number is matched also.

The next step is to match y, the tail of the program, with other left parts of rewriting systems that contain number 1. It can bean arbitrary behavior $X1$ that satisfied precondition $precond(X1)$ or action $a2$ that satisfied precondition $precond(a2)$. We continue matching before we match $a10.y$ and rewrite it as $Delta$. This means that the vulnerability has been detected.

We can continue algebraic matching due to the different strategies and with selection of different coverage of code. For example, we can find all the shortest behaviors Xi or get only the first matched sequence of actions leading to vulnerability.

When providing algebraic matching, it is necessary to mention that checking of preconditions is the most expensive procedure. With parameters of the program we have the algebraic expression with unknown variables where we should detect the satisfiability with usage of proving or solving tools. For efficiency reasons we can separate the algebraic matching procedure into two stages. The first stage is the algebraic matching where the precondition contains only concrete values and is easy to calculate. The second stage is to provide symbolic modeling of the obtained trace.

Symbolic modeling starts from the initial action of the vulnerability sequences that can be presented by an initial formula. Then, we can apply the action corresponding to the behavior algebra expression. The action is applicable if its precondition is satisfiable and consistent with the current state. With the initial state S_0 and from the behavior B_0, we select the next action. In the first step we check the satisfiability of the conjunction

$$S_0 \wedge P_{a1}$$

where $B_0 = a_1.B_1$, and P_{a1} is a precondition of $a1$. The next state of the environment will be obtained by means of the predicate transformer [?]; that is, the function over the current agent state, precondition, and postcondition:

$$PT(S_i, Q_{ai}) = S_{i+1}$$

where Q_{ai} is a postcondition of $a1$. By applying the predicate transformer function to different environment states, we obtain the sequence $S_0, S_1, \ldots$ of formulas that express the environment states changing from the initial state.

The second stage of the algebraic matching is implemented by comparing the environment state of the program to be verified with the preconditions of every action from the vulnerability model. If their intersection is satisfiable then it is matched. The criterion of vulnerability detection is the matching of the whole trace on both levels.

6 Discussion and Conclusions

The first experiments with algebraic matching have been performed within the scope of APS and the Garuda AI platform. It covered 15 known vulnerabilities from the Common Vulnerabilities and Exposures (CVE) database.

The main benefit of an algebraic approach is that we have more accurate detection of vulnerability. The description of the vulnerability covers a set of its possible varieties. One of the advantages is that the faster procedure of detection (algebraic matching) shall be implemented first and the more expensive stage (symbolic modeling) afterwards.

The main problem or shortcoming of the approach is that the problem of reachability is undecidable in a general way. There can be exponentially resulting to explosion of the state space and possible program scenarios during algebraic matching. These are typical problems in the model-checking community. These challenges shall be resolved by using alternative symbolic methods like invariant generation, approximation, or backward symbolic modeling. The different settings of searching can reduce the state space; for example, we can provide some coverage of code lines. However, this reduction can cause us to miss vulnerabilities.

The other problem is insufficient formalization of vulnerabilities and lack of generalization in the similar work. Different compilers in different operation environments can use other registers and differ orders of instructions. In this case, the automation of a vulnerability model and generation of its varieties in different environment shall be provided.

Acknowledgements

The project was carried out with the financial support of Garuda AI. Science research and development are based on the results of research of the Glushkov Institute of Cybernetics of the National Ukrainian Academy of Sciences. It used an algebraic engine from the Garuda AI platform for the symbolic modeling stage and the Algebraic Programming System for algebraic matching algorithms. Special thanks to Kherson State University for support of the site of APS.

References

[1] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 269–278. IEEE, 2006.

[2] Maryam Mouzarani, Babak Sadeghiyan, and Mohammad Zolfaghari. Detecting injection vulnerabilities in executable codes with concolic execution. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 50–57. IEEE, 2017.

[3] https://www.darpa.mil/program/cyber-grand-challenge,, 2019.

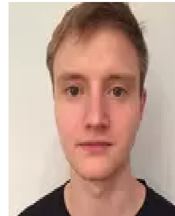
[4] Alexander Letichevsky and David Gilbert. Interaction of agents and environments, 1999.

[5] http://apsystems.org.ua,, 2019.

[6] https://cve.mitre.org,, 2019.

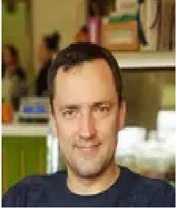
[7] John Viega, M Howard, and D LeBlanc. Deadly sins of software security-programming flaws and how to fix them". 19.

Oleksandr Letychevskyi is a leading research fellow in GIC. He finished PhD degree in 2005 and Doctor Degree in 2016. He is an author of technology of symbolic methods usage in model-based testing and software verification especially symbolic white-box and integration model-based testing methods, methods of invariants generation, methods of reengineering (business rules extraction from program code, languages migration, model transformation). In 2010-2012 he acted as a subcontractor of Missouri University (Rola, USA) developing symbolic methods in verification and model-based testing. Dr. Letychevskyy has published 65 papers (including 35 in reviewed international conferences and journals). His main research interests include formal methods in software engineering (testing, verification, reengineering); insertion; algebraic programming; and cybersecurity.



Yaroslav Hryniuk is a Ph.D. student of Computer Science at V.M. Glushkov Institute of Cybernetics of NAS of Ukraine. His major research interest involves insertion programming, formal verification, computer system's vulnerabilities. He received his MSc in Software Engineering from Taras Shevchenko National University of Kyiv.

Viktor Yakovlev is a lead mathematician in the Glushkov Institute of Cybernetics of National Academy of Sciences of Ukraine. He gained his MS degree from Kyiv Shevchenko National University in 1985. His major interests are in data processing, distributed systems and modeling software systems. He earlier worked as a team leader of the Ukrainian First Stock Trading System (PFTS) and also developing the Verification of Requirements System (VRS) project in an Information Software Systems team.



Volodymyr Peschanenko is a faculty of science, head of the department of Informatics, Software Engineering and Economic Cybernetics at the Kherson State University. In 2007 he was awarded the PhD (mathematical and software support of computing systems) in Glushkov Institute of Cybernetics of the NAS in Ukraine. Theme âĂŞ "Design and Development of School Systems of Computer Algebra". 2010 âĂŞ awarded the Academy Rank of Associate Professor of the Department of Informatics. 2015 - awarded the Doctor of Science (Mathematical and Software Support of Computing Systems) in Glushkov Institute of Cybernetics of NAS in Ukraine. Theme âĂŞ "Implementation methods of Insertion Modeling Systems". He has more than 50 publications (including papers in scientific journals, conference proceedings and methodical recommendations etc.). Co-author of the next software: Algebraic Programming System APS, Insertion Modeling System IMS. Participated in the following international projects: Austrian-Ukrainian project "CENREC" (finished), Tempus JEP-27237-2006 "CC4U2 Educational Program for Computer Sciences for Ukrainian Universities" (finished), MASTIS Project: establishing modern master-level studies in information systems (Funded by Erasmus+ Programme of the European Union, in process).

Viktor Radchenko is a skilled solution architect and product manager. He has over 20 years experience in software development and management. He took part in more than 30 projects as R&D leader and CTO, and CEO of Garuda AI company which develops platforms for formal verification and security research.