An Algebraic Approach for the Detection of Vulnerabilities in Software Systems

Oleksandr Letychevskyi and Vadim Sukhomlinov
Glushkov Institute of Cybernetics of National Academy of Sciences (Ukraine), Intel Corporation (USA)
lit@iss.org.ua
vadim.sukhomlinov@intel.com

ABSTRACT

The paper presents an algebraic approach for finding vulnerabilities in a program system that is given as the sequence of processor instructions. The main result of the paper 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 developed by the authors. A given example illustrates the method.

KEYWORDS

Cybersecurity, algebraic programming, predicate transformer, vulnerability, safety, symbolic modeling.

1 INTRODUCTION

Detection of vulnerability and software security analysis is now one of the most challenging problems in software engineering. A great amount of tools have been developed since the 1990’s to meet the requirements for prevention of intrusion and security violations. One of the most popular methods of vulnerability detection is a static method that analyzes the code and offers a set of possible points where a problem could occur. A good survey of static tools is presented in [1]. The static method has been realized in a number of tools, such as Parasoft C++ [2], Klocwork [3], and is available today in mainstream development tools such as Clang (LLVM) [4] and Microsoft Visual Studio [5]. It can detect various types of security vulnerabilities in programs like buffer overflow and null pointer assignment, but many other types of vulnerabilities are very difficult to find. Moreover, the static method can list a big variety of false issues and can fail to detect the issues that are not presented in the given code, such as library functions.

One of the ways to find all vulnerabilities is to run an exhaustible execution of all program scenarios by taking into account all input values. For this purpose, the symbolic execution approach was used in analysis of a Java program [6]. Such an approach demands the usage of powerful solving and proving machines that are actively used in a model-checking domain. Nevertheless, symbolic execution of the program on the level of the Java language still does not resolve the vulnerability detection problem due to the impossibility for checking compiled components like libraries or third-party tools.

Fuzzing, another automated method for software testing, has roots from the 1950’s, when data were still stored on punched cards. Programmers used punched cards that were pulled from the trash or card decks of random numbers as input to computer programs [7]. If an execution revealed undesired behavior, a bug had been detected and was fixed. In the urge to enhance security, fuzzing is commonly used today in automated building/integration solutions. Compilers provide support in instrumenting code to detect leaks, out-of-bound accesses, etc., such as AddressSanitizer in Clang. Special fuzzers such as AFL and libFuzzer [8] feed inputs.
Academic research focused on improving coverage and automating crash analysis. We propose to resolve this problem by the symbolic execution of the instructions of a program executable code by using special slicing methods that are popular in model-checking techniques. Symbolic execution framework is based on the Algebraic Programming System (APS) [9] that has been developed and maintained since the 1990’s at the Glushkov Institute of Cybernetics of the National Academy of Sciences of Ukraine. In this paper, we will adhere to the following content. First, we will formulate the problem statement and give the high level scheme of the method. We then will present the main algebraic notions that will be necessary for the understanding of the method and the technology chain to be used. Finally, we will illustrate the algorithm by the example of vulnerability detection presented in the Common Vulnerabilities and Exposures (CVE) database [10].

2 PROBLEM STATEMENT AND HIGH LEVEL DESIGN OF METHOD

Our goal is to prove that the known vulnerability can be reached or not in a given system. We consider as “known” the vulnerabilities from CVE. The formalized case of vulnerability, together with a model of executable code are the input of the technology.

It is anticipated that the case of vulnerability will be formalized manually as a formula in some formal language. The examples of vulnerability formalization will be considered further, and the language for presentation of the formula should express the static properties (logic language), behavioral properties (process algebra) or time properties (temporal logic).

The executable code itself also is presented by means of formal specifications that are the model of the program. These specifications are the inputs for technology and corresponding transformation that should be provided.

Here, shown below, is the scheme that illustrates the process of detection of vulnerability. It consists of two stages.

![Figure 1. The first stage of method. Input data preparation.](image)

During the first stage we should disassemble the input code that is uploaded to the memory. Reading the instructions directly from memory, we can process the part of the system that contains third-party tools or libraries as is. In the development environment, assembler code can be directly produced by the compiler.

After the preparation, we can use the Algebraic Programming System and its component for further processing in stage 2.

![Figure 2. The second stage of method. Symbolic execution and detection of presence of vulnerability.](image)

Before direct symbolic modeling we should detect the possible points of intrusion for security of data. Usually these are the points of input in the program system. The explored property of vulnerability also defines the subset of processor instructions to be
processed, so we can symbolically execute only the interested part of program by selecting the method of symbolic modeling.

Symbolic modeling of the selected program slice is performed in the scope of the APS system, that uses as the third-party solvers as the integrated ones.

3 ALGEBRAIC MODEL OF EXECUTABLE CODE

3.1 Behavior Algebra

Algebra of behavior was developed by D. Gilbert and A. Letichevsky in 1997 [11]. It was realized in the scope of the Insertion Modeling System (IMS) as an extension of APS. 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 $\Delta$, deadlock $0$, and non-determinate behavior $\bot$. The relation of approximation $\sqsubseteq$ is a partial order on the set of behaviors with minimal element $\bot$. The following example of behavior expressions

$$B_0 = a_1.a_2.B_1 + a_3.B_2,$$
$$B_1 = a_4.\bot,$$
$$B_2 = \ldots$$

means that behavior $B_0$ could be interpreted as sequence of action $a_1, a_2$ and the behavior $B_1$ afterwards, or the action $a_3$ with the next behavior $B_2$. Behavior $B_1$ will finish after action $a_4$.

3.2 Basic Protocols Language

The Basic Protocols language has been developed in the scope of the Verification of Requirements Specifications (VRS) project [12], 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. In the current case it is given as an identifier of a basic protocol.

3.3 Semantic of Processor Instructions by Means of Behavior Algebra and BP-Language

The description of the architecture of the Intel 64 and IA-32 processors is presented in [13]. We consider the interactions of the following agents: processor, memory, and external environment. 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, doubleword,...), 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. For example, branch instruction:

40a984: jne 40ab50

could be converted to a behavior algebra expression, covering possible outcomes based on the state of the ZF flag in EFLAGS:

B40a984=a_jne1.B40ab50+a_jne2.B40a98a

where the actions are the following:

\[
\begin{align*}
\text{a\_jne1} &= (ZF=0) \rightarrow \langle"\text{jne1}\rangle \ 1 \\
\text{a\_jne2} &= \neg(ZF=0) \rightarrow \langle"\text{jne2}\rangle \ 1
\end{align*}
\]

We denote behavior identifiers together with the hexadecimal address of instructions in a program segment for traceability to assembler code. The expression above means that the instruction by the address 40a984 will pass the control to the instruction by the address 40ab50 if flag ZF is equal to 0; otherwise the next instruction by the address 40a98a will be performed. The condition of transition is in a predicate that is the precondition of the basic protocol. The postcondition is absent so it is equal 1. The process of basic protocol is given as some string with the name of an action containing the instruction. It will be used for traces defining the program behavior. The instructions that change the attributes could be presented also as basic protocols with postcondition containing this changing. For example, the instruction:

40a99d: add DWORD PTR [r13 + 0x15c],r8d

will be transformed to

\[
B40a99d = a\_add\_3480.B40a9a4
\]

where

\[
a\_add\_3480 = 1\rightarrow \langle"a\_add\_3480\rangle
\]

Memory(r13 + 348) := Memory(r13 + 348) + r8d;
ZF := (Memory(r13 + 348) + r8d = 0);
SF := (Memory(r13 + 348) + r8d < 0);

This instruction performs the adding of a memory element that is available by the given address in register r13 to the content of doubleword 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 this semantic of instructions has been defined directly following specification in the data sheet, and we see that formalization of executable code is not that complicated for representation by formal logic language. Consider the following code fragment:

```
425060: 41 55 push r13
425062: 41 54 push r12
425064: 40 05 rs mov r13,r31
425067: 55 push ebp
425068: 53 push rdx
425069: 48 99 0f mov r12,rdx
42506c: 99 05 mov esp,edx
42506e: 48 43 cc 00 sub esp,edx
425072: e8 df 24 fe ff call 607550 <___32_s_file>print>
425077: 48 89 c7 mov rdx,esp
425079: e8 13 31 ff ff call 002220 <___32_new>print>
42507d: 48 89 09 test rax,rax
425082: 0f 44 00 00 00 je 425138
425088: 4c 59 69 mov rcx,r13
```

**Figure 3. Example of code which could be translated to algebra behavior expressions**

```
B425060 = a_push_3376.B425062,
B425062 = a_push_3376.m425064,
B425064 = a_mov_3376.B425067,
B425067 = a_push_3376.B425069,
B425069 = a_push_3376.m42506a,
B42506a = a_mov_3376.B42506c,
B42506c = a_mov_3376.B425068,
B425068 = a_sub_3373.m425072,
B425072 = a_call_3374.call B407550.B425077,
B425077 = a_mov_3375.B42507a,
B42507a = a_call_3376.call m425020.m42507f,
B42507f = a_test_3377.B425082,
B425082 = a_re_3377.B425136 + a_alt_re_3379.B425090,
B425090 = a_mov_3378.m42508b,
```

**Figure 4. Behavior expressions**
The actions in behavior could be presented as the following.

```
a_push_33766 = operator(1 -> ("@x6: action 'push 425060';")
(a_push_33767 = operator(1 -> ("@x6: action 'push 425062';")
(a_mov_33768 = operator(1 -> ("@x6: action 'mov 425064';")
(rip := 4345560); r13 := rax))
(a_push_33769 = operator(1 -> ("@x6: action 'push 425067';")
(rip := 4345560))
(a_push_33770 = operator(1 -> ("@x6: action 'push 425068';")
(rip := 4345561))
(a_mov_33771 = operator(1 -> ("@x6: action 'mov 425069';")
(rip := 4345564); r12 := rdi))
(a_mov_33772 = operator(1 -> ("@x6: action 'mov 42506c';")
(rip := 4345564); edbp := edx))
(a_sub_33773 = operator(1 -> ("@x6: action 'sub 42506e';")
(rip := 4345567); rip := rip - 0x1e; SF := (rip < 0));
(a_call_33774 = operator(1 -> ("@x6: action 'call 425072';")
(rip := 4345575))
(a_mov_33775 = operator(1 -> ("@x6: action 'mov 425077';")
(rip := 4345576); rdi := rax))
(a_mov_33776 = operator(1 -> ("@x6: action 'mov 42507a';")
(rip := 4345576))
(a_mov_33777 = operator(1 -> ("@x6: action 'mov 42507c';")
(rip := 4345576))
(a_xc_33778 = operator(SF = 1) -> ("@x6: action 'je 425082';")
(rip := 4345592))
(a_xc_33779 = operator(SF = 1) -> ("@x6: action 'je 425082';")
(rip := 4345592))
(a_mov_33780 = operator(1 -> ("@x6: action 'mov 425088';")
(rip := 4345595); rcx := r13))
```

**Figure 5.** 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. Such conversion gives the possibility of APS use where behavior algebra expressions are the input. In the scope of APS, we can provide symbolic modeling, analysis, generation of the scenario of program execution, and proving of the properties.

### 4 FORMAL PRESENTATION OF VULNERABILITY PROPERTY

Any property could be presented as some formula over attributes. The safety property states that something undesirable will never happen. In the context of vulnerability of a program, it could be reformulated that in some points of the program the truth of some formula over program attributes will never be achieved.

One of the easiest examples of the safety formula is the prohibition of the reference to not initialized memory, or “null pointer assignment.” In the instruction’s parameters `QWORD PTR[rax]`, the value in register rax shall not equal 0.

\[(\neg (rax = 0))\]

When software does not validate input data properly, the attacker can craft the input with unexpected values, leading to undesirable behavior – arbitrary code execution, leakage of data or a crash. The condition for leakage of confidential data would be if the program allows memory access out of admissible bounds:

\[(\text{Admissible}_0 \leq rax \leq \text{Admissible}_N)\]

The famous “Heartbleed” vulnerability in OpenSSL library is an example of a lack of boundary check when a program reads adjacent memory. This example will be studied in detail in the next chapters.

Formalization of the safety property is the most complicated problem, and every case of vulnerability should be formalized separately. Some of the properties can be generated automatically, whereas some of them require manual formalization. This problem is open, but significant parts of known cases of vulnerabilities can be represented in formal logic.

### 5 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 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 `S_0` and from the initial behavior `B_0`, we select the action and move to the next behavior. We check on the first step to satisfiability of the conjunction

\[S_0 \land \text{Precondition}(a_1)\]
if $B_0 = a_1.B_1$. 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

$$\text{PT}(S_0, \text{Postcondition}(a_1)) = S_1$$

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. Using different traversal algorithms we can obtain the set of traces covering complete program behavior. Searching for the intersection of the vulnerability formula with the satisfiable states of the program leads to a trace at an intersection, the results of which are potentially exploitable.

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 than 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.

### 6 EXAMPLE

We consider the example of the “Open SSL Heartbleed vulnerability” that allowed an attacker to read sensitive data, such as authentication credentials, and secret keys by incorrect memory handling via Transport Layer Security (TLS) extension. The client requested the server to send strings from memory of a given length. The server sent the requested bytes, but the intruder could request lengths that were much greater than the buffer, so that the adjacent memory could be read and could contain data in which the intruder could be interested.

The problematic point is located in the following C-code in OpenSSL 1.0.1f:

```c
int
    tlsls_process_heartbeat(SSL *s)
    {
        unsigned char *p = ls->ssl3-recv.data[0], *pl;
        unsigned short htype;
        int padding = -16 ; /* Use minimum padding */
        /* Read type and payload length first */
        htype = *p++;
        n12(p, payload);
        pl = p;
        if (e->msg_callback)
            s=tlsls_callback(0, s->version, TLSSL_IO HEARTBEAT, 
               &s->ssl3-recv.data[0], a->ssl3-recv.length, 
               s, s->msg_callback_arg);
    }
```

It corresponds to the following sequence of instructions:

```
61590:  40 80 69 00 00 00 00 00 00 00 00 00 mov rax,WORD PTR [esp+0x0]
61597:  01 6c 00 00 00 00 00 00 00 00 00 00 mov rdx,WORD PTR [esp+0x6c]
6159f:  63 7a 1c cmp rdx,0x1c
615a0:  76 0d jnz 4387Eh <tlsls_process_heartbeart+0x6f>
615a4:  44 6f 67 65 61 79 01 movzx EI,WORD PTR [esp+0x6f]
615a9:  66 41 61 00 00 rcl rsi,0x7e
615ab:  45 5f 67 6c mov si,[edi+rsi]  
615b0:  45 5f 67 7e mov si,[edi+rsi]  
615b5:  45 5f 67 7e mov si,[edi+rsi]  
615b6:  44 39 1c cmp rsi,0x1c
615b8:  73 18 jnz 4387Eh <tlsls_process_heartbeart+0x6f>
615bc:  08 65 55 00 mov svx,erdx,TPHY [edi=rsi]
615c0:  46 64 7a 01 jae rdx,0x41850h <tlsls_process_heartbeart+0x7a01>
615c5:  6e 63 fa 0e cmp rdx,0xl
615c7:  4e 47 jnz 4159b0h <tlsls_process_heartbeart+0x47>
615c9:  6b 63 fa 0e cmp rdx,0xl
615ca:  74 11 jnz 4159b0h <tlsls_process_heartbeart+0x7411>
615c3:  60 63 63 10 add rdx,0x10
615c9:  7b pop rip
615cb:  50 pop rbx
615cd:  50 pop rdx
615cf:  50 pop r14
615d0:  50 pop r13
615d2:  50 pop r12
615d4:  50 pop r11
615d6:  50 pop r10
615d8:  50 pop r9
615db:  50 pop r8
615dc:  50 pop r7
615dd:  66 99 call edx
615e0:  46 41 89 fa 12 cmp rdx,0x12
615e3:  79 40 pop eax
615e5:  31 89 75 00 movzx edx,WORD PTR [rip+0x3]
```

This piece of code has been selected manually for demonstration and for proving of
vulnerability absence we should move from instruction 415d97 to instruction 415d40 by backward symbolic modelling. The initial state for backward moving is defined as the vulnerability case when length of buffer to be read is exceed the admissible one.

We define this state of vulnerability manually as the following formula:

\[
\text{Memory}(\text{rdi}+128) < 19 \text{ } || \text{ } \text{Memory}(\text{rdi}+128) > \text{Memory}(\text{r12d}) + 19
\]

In future, this formula would be derived from known semantics of memory allocation methods (malloc(), stack pointer, etc).

Since, this vulnerability was already resolved in OpenSSL 1.0.1g, we can compare how detection works. The older version does not contain input parameter checks so moving from vulnerability place to the initial state is possible and corresponding trace-counterexample has been generated. We will consider the second version of code where protection of the unauthorized access to the adjacent memory has been implemented. Providing symbolic modeling from vulnerability state we should not reach the initial state so the absence of vulnerability is proved.

Firstly, we need select the necessary slice of assembler code for symbolic modeling. It means that we will execute symbolically only those instructions that performs buffer access. In the point of buffer access we have the following state of environment

\[
(\text{Memory}(\text{rdi}+128) < 19 \text{ } || \text{ } \text{Memory}(\text{rdi}+128) > \text{Memory}(\text{rol}(8,\text{Memory}(\text{rbp}+1))+19 \text{ } )\& \text{ } \text{rol}(8,\text{Memory}(\text{rbp}+1))+19 < = \text{edx} \& \text{ } (\text{edx} > 19)
\]

This is disjunction of two states. If to consider every disjunction member we can see that they are not satisfiable, so the next basic protocols are non-applicable. We have deadlock by backward modeling so the absence of vulnerability is proved. The satisfiability of logical expressions was proved by Microsoft Z3 proving machine.

7 CONCLUSIONS

This simple example demonstrates the technology in a nutshell. This is the first step in big research that has started at the Glushkov Institute of Cybernetics together with Intel specialists.

The primary challenge is the formalization of the situation as a vulnerability formula under certain circumstances. Today it requires manual analysis of the code and vulnerability case, but it’s possible to automate it. Another challenge is definition of the points where symbolic modeling should start for detection of the issues. There were two ways presented - automatic searching of possible suspected points in the code that present some input from external environment. The second way also could be implemented by symbolic modeling on a higher level of abstraction with computation of possible places where more detailed modeling should start, probably with using traces of actual execution.

Overall, the results are encouraging and demonstrate the perspective of the algebraic approach. The corresponding modules are developed in the scope of APS and will be updated to cover semantics of vulnerabilities exploited in the last virus attacks as an algebraic approach.

REFERENCES

www.apsystem.org.ua


