

Testing of Utilities for Finding Vulnerabilities in the Machine Code of Telecommunication Devices

Mikhail Buinevich*, Konstantin Izrailov*, Andrei Vladyko*

**The Bonch-Bruевич Saint-Petersburg State University of Telecommunications, Russian Federation, Saint-Petersburg, 22-1 Prospekt Bolshevikov*

bmv1958@yandex.ru, konstantin.izrailov@mail.ru, vladyko@bk.ru

Abstract— The article is devoted to the authors' machine code algorithmization utility to find vulnerabilities in telecommunication devices. The utility is compared with its closest analogue – the IDA Pro software and its plug-in – Hex-Rays. The paper presents the key tests that are used for such comparison and the results of software tools operation for them. The formalized outcome of the comparative analysis justifies the advantages of the authors' utility for solving this task.

Keywords— vulnerability, machine code, algorithmization, IDA Pro, utility testing

I. INTRODUCTION

Information security is closely connected with the processes of its treatment, that are virtually completely allocated to specialized software (hereinafter referred to as the SW) in modern telecommunication networks. As a consequence, the absence of vulnerabilities in the SW code has a significant impact on information security.

It is possible to eliminate such vulnerabilities in different ways; yet the specifics of this kind of SW make it possible to effectively use only some of them. Thus, the SW execution on some telecommunication devices, as well as the availability of machine code (hereinafter referred to as the MC) in the absence of source code (hereinafter referred to as the SC) leads to the need to address the following task – creating a method to find vulnerabilities in the MC. One possible way, that stands out among others, is finding vulnerabilities with the use of MC algorithmization, wherefore the corresponding authors' method has been developed [1] (hereinafter referred to as the Method).

The basic idea of the Method is automatic processing of the telecommunication device MC and restoring its operating algorithms. The thus obtained algorithmic representation of the code may be analyzed by an expert in code security (hereinafter referred to as the Expert) for the presence of vulnerabilities. It should be noted that the Method is designed for finding medium-level and high-level vulnerabilities [2, 3], with a high level of subjective evaluation, unlike low-level ones, that may be found using many automatic tools.

The method has the following features. Firstly, it restores the very algorithm rather than the SC. Secondly, the restoration process is controlled by a human to obtain more (subjectively) «accurate» algorithms. Thirdly, ready-made solutions, well-developed and well-tested, are used for parsing the MC. The MC operation is based on a specialized software tool

(hereinafter referred to as the Utility) that is used for processing the assembler code (hereinafter referred to as the AC), building its internal model, converting it into an algorithmic presentation and generating the latter in a form that is convenient for Expert's analysis [4].

In addition to the authors' prototype of the Utility, there are other algorithmization tools that are applied for the task in question. The comparison of the Utility with a competing tool allows us to reasonably confirm the Utility operability and to assess its efficiency, as well as to suggest the way for its further development.

II. ORGANIZATION OF TESTING

Since the main purpose of the Method is obtaining an algorithmic presentation of the code, the criteria for tools comparison should characterize the quality of such restoration. The criteria selection is a particular challenging task – mainly due to the judgment subjectivity; however, their basic set may be partially obtained on the basis of the used PC metrics analysis (such as SLOC, OO metrics, Halstead metrics, cyclomatic metric, Chapin metrics, Gilb metrics, connectivity metrics, etc.). The general form of such a list of objective and subjective criteria should be as follows: the restoration errors; the identified simple and complex control structures (conditional and unconditional jumps, cycles, subroutine calls); correctly defined routine signatures; redundant variables; cyclic code complexity; the presence of too complex structures; the level of abstraction and algorithms description density; successfully found vulnerabilities. However, because of the lack of full-fledged testing tools, these criteria may be defined subjectively, based on the results of corresponding tests.

The process itself of software tools (hereinafter referred to as SWT) assessment should be as follows: a testing base is created from a group of key tests; for each test, the assessed SWT is applied to obtain an algorithmized presentation (or after reduction to the same): the obtained presentations are assessed using the proposed criteria; the algorithmization tools are selected that have shown the best result.

Obviously, one of the assessed SWTs should be the Utility itself – strictly speaking, the MC algorithmization is its intended purpose. The second tool – the closest one to the Utility among other similar tools – should be the IDA Pro software complete with the Hex-Rays plug-in.

IDA Pro is an interactive disassembler that supports a lot of execution processors: receiving a MC at the input, it shows its

assembly presentation, identifying routines and some of their shared data (global variables), while adding explanatory comments for all identified elements. Note that, at the first stage of the Method, it is this software tool that is used to convert the MC into the AC (through built-in scripts). Also, IDA Pro builds a flowchart of the subroutine control flow (with assembler inserts) and a graph of their calls. In terms of algorithmization, its plug-in – Hex-Rays (hereinafter referred to as IDA+Hex-Rays) is of primary interest, that is designed for code decompiling– i.e. the attempt to obtain its SC (in theory, the original one, but in practice, only a similar one). There are also third-party plug-ins in the interests of decompiling, such as reconstruction the hierarchy of C++ classes [5]. However, they are in addition to Hex-Ray.

Both the SC or the AC, obtained by compiling the latter, or the MC obtained by assembling, may be used as the program code for creating tests. Using the MC is extremely time-consuming tasks while creating tests, because the code has a binary, unreadable form. Using the SC does not allow for accurately specifying the essence of testing, because the used compilers do not generate an identical AC, by adding excessive instructions (for example, excessively saving some variables in the stack) or by changing some algorithms (for example, predicting variable values as a result of optimization). As for the AC, it will not allow for both creating tests manually (naturally, subject to appropriate skills available) and making the algorithmized MC fully compliant with them.

The base should include 9 key tests, each of them being intended to reflect the algorithmization criteria (as specified in the test name). The intended use and the operating algorithm of each test should be described, as well as the AC created manually for the PowerPC processor and the SC prototype to explain the test operation. The results of applying algorithmization tools and their comparison, both with each other and with the SC prototype, will be presented. The first comparison will make it possible to show the advantages of one test against the other one in various aspects; the second one, the general correctness of the produced algorithmization.

It should be mentioned the particular style of the algorithm text as generated by the Utility. Firstly, it uses the following designation of variable names: the prefix «_» means that the variable is an input parameter of a subroutine; the postfix «_», an output one, whereas the prefix «@» is used for local variables; variables without these designations are global ones. Secondly, the list of registers via which a function returns its values is specified in the signature before the function name. And, thirdly, the declaration of a global variable may include the absolute address of its location after the character «&».

For more clarity and convenience of comparison, the input data and the test results will be shown in a tabular form, where consequentially (from left to right, and from top to bottom) the following information is provided: the SC code prototype; the AC corresponding to the same, that is used immediately for testing; the Utility operation result; IDA+Hex-Rays operation result.

A. Test 1. Identifying Control Structures – Conditional Branches

The test consists of a function that returns various values in accordance with equality of its 3 parameters: 111, if the parameters are equal; 110, if the first two parameters are equal; 011, if the last two parameters are equal; and 101, if the first and the last parameters are equal (see Table 1).

TABLE 1. SC, AC AND ALGORITHMIZED CODE FOR TEST 1

SC Prototype	AC for Testing
<pre>int test1(int x, int y, int z){ if (x == y && y == z) return 111; if (x == y) return 110; if (y == z) return 11; if (x == z) return 101; return 0; }</pre>	<pre>test1(){ 0x00000000: test1: 0x00000000: cmpw r3, r4 0x00000004: bne loc_18 0x00000008: cmpw r4, r5 0x0000000C: bne loc_18 0x00000010: li r3, 0x6F 0x00000014: blr 0x00000018: loc_18: 0x00000018: cmpw r3, r5 0x0000001C: bne loc_28 0x00000020: li r3, 0x6E 0x00000024: blr 0x00000028: loc_28: 0x00000028: cmpw r3, r5 0x0000002C: bne loc_38 0x00000030: li r3, 0xB 0x00000034: blr 0x00000038: loc_38: 0x00000038: cmpw r3, r5 0x0000003C: bne loc_48 0x00000040: li r3, 0x65 0x00000044: blr 0x00000048: loc_48: 0x00000048: li r3, 0 0x0000004C: blr }</pre>
Outcome of the Utility	Outcome of IDA+Hex-Rays
<pre>(r3) test1(w1_, w2_, w3_) { if (w1_ == w2_ && w2_ == w3_) { _w4 = 0x6f; goto \$lr; } if (w1_ != w2_) { if (w1_ != w3_) { if (w1_ != w3_) { _w4 = 0; } else { _w4 = 0x65; } } else { _w4 = 0xb; } } else { _w4 = 0x6e; } \$lr; return _w4; }</pre>	<pre>signed int __fastcall test1(int a1, int a2, int a3) { if (a1 == a2 && a2 == a3) return 111; if (a1 == a2) return 110; if (a1 == a3) return 11; if (a1 == a3) return 101; return 0; }</pre>

As clearly seen from Table, the IDA+Hex-Rays operating result is more human-friendly due to the use of a plurality exit points (the «return» operator). As for the Utility, at first sight, it has restored the algorithm in a more «complicated» form – using a unique exit point, resulting in the presence of a «goto» operator and a cascade of nested conditions. The reason for

this behavior of the Utility is its striving to observe the procedural and structural paradigm for the obtained algorithms: the division into subroutines, the absence of unconditional jumps, one exit from the subroutine – as a result, a larger hierarchy of operation blocks. Such a strict form allows us to apply more successfully the automated techniques for algorithm analysis and to search for vulnerabilities. For example, the obligatory existence of only one exit point should guarantee the impossibility of subroutine completion until the end of its body as a result of a logical bomb, which increases the chances of its detection by the Expert.

However, it may be advisable to add into the Utility some additional algorithmization management capabilities, such as the support for multiple exit points from the subroutine, less stringent requirements for the absence of «goto», the maximum depth of nested conditions, etc.

B. Test 2. Identifying Control Structures – Cycles

The test consists of a function that returns the factorial of the numeral value transmitted by a single parameter (see Table 2).

TABLE 2. SC, AC AND ALGORITHMIZED CODE FOR TEST 2

SC Prototype	AC for Testing
<pre>int test2(int x){ int t = 1; for(int n = 1; n <= x; ++n) t *= n; return t; }</pre>	<pre>test2(){ 0x00000000: test2: 0x00000000: li r4, 1 0x00000004: li r5, 1 0x00000008: loc_8: 0x00000008: cmpw r5, r3 0x0000000C: bgt loc_1C 0x00000010: mullw r4, r4, r5 0x00000014: addi r5, r5, 1 0x00000018: b loc_8 0x0000001C: loc_1C: 0x0000001C: mr r3, r4 0x00000020: blr }</pre>
Outcome of the Utility	Outcome of IDA+Hex-Rays
<pre>(r3) test2(w1_) { _w2 = 1; @w3 = 1; loop (@w3 <= w1_) { _w2 *= @w3; ++@w3; }; return _w2; }</pre>	<pre>signed int __fastcall test2(signed int a1) { signed int v1; // r4@1 signed int i; // r5@1 v1 = 1; for (i = 1; i <= a1; ++i) v1 *= i; return v1; }</pre>

As clearly seen from Table, the restored algorithms are similar, with the sole exception: IDA+Hex-Rays has restored the cycle in a uniform manner – with initializing the variable, checking its value and incrementing («for (i = 1; i <= a1; ++i)»); on the other hand, the Utility used 5 separate operations for the same purpose («@w3 = 1», «loop (@w3 <= w1_)», «++@w3»). In this example, the capabilities of the Utility are inferior to those available for IDA+Hex-Rays; however, their implementation in the basic variant is a pure engineering task of coding.

C. Test 3. Defining the Subroutine Signature

The test consists of a function that calculates the value of a polynomial of the 3rd degree, where the variable is defined by the 1st parameter, and the coefficients, by the other 4 ones (see Table 3).

TABLE 3. SC, AC AND ALGORITHMIZED CODE FOR TEST 3

SC Prototype	AC for Testing
<pre>int test3(int x, int a, int b, int c, int d){ int t1 = a * x * x * x; int t2 = b * x * x; int t3 = c * x; int t4 = d; return t1 + t2 + t3 + t4; }</pre>	<pre>test3(){ 0x00000000: test3: 0x00000000: mullw r8, r4, r3 0x00000004: mullw r8, r8, r3 0x00000008: mullw r8, r8, r3 0x0000000C: mullw r9, r5, r3 0x00000010: mullw r9, r9, r3 0x00000014: mullw r10, r6, r3 0x00000018: mr r11, r7 0x0000001C: mr r3, r8 0x00000020: add r3, r3, r9 0x00000024: add r3, r3, r10 0x00000028: add r3, r3, r11 0x0000002C: blr }</pre>
Outcome of the Utility	Outcome of IDA+Hex-Rays
<pre>(r3) test3(w1_, w2_, w3_, w4_, w5_) { return (w2_ * w1_ **3) + (w3_ * w1_ **2) + (w4_ * w1_) + w5_; }</pre>	<pre>int __fastcall test3(int a1, int a2, int a3, int a4, int a5) { return a2 * a1 * a1 * a1 + a3 * a1 * a1 + a4 * a1 + a5; }</pre>

The both software tools have defined the subroutine signature in the same way, according to the SC: 5 input parameters and 1 output parameter. However, it should be noted that the effect of the code conciseness, as applied in the Utility, has allowed us to obtain a shorter and more human-friendly algorithm record – using the exponentiation operator «**» (a more detailed description will be given in Test 8).

D. Test 4. Complex Type Support – Structures

The test consists of a function that returns a structure-type variable, the fields of which are filled with the results of addition, subtraction, multiplication and division operations with two input parameters (see Table 4).

TABLE 4. SC, AC AND ALGORITHMIZED CODE FOR TEST 4

SC Prototype	AC for Testing
<pre>typedef struct { int add; int sub; int mul; int div; } OpRes; OpRes test4(int x, int y){ OpRes res; res.add = x + y; res.sub = x - y; res.mul = x * y; res.div = x / y; return res; }</pre>	<pre>test4(){ 0x00000000: test4: 0x00000000: add r7, r3, r4 0x00000004: subf r8, r4, r3 0x00000008: mullw r5, r3, r4 0x0000000C: divw r6, r3, r4 0x00000010: mr r3, r7 0x00000014: mr r4, r8 0x00000018: blr }</pre>
Outcome of the Utility	Outcome of IDA+Hex-Rays
<pre>(r3, r4, r5, r6) test4(w1_, w2_) { return (w1_ + w2_, w1_ - w2_, w1_ * w2_, w1_ / w2_); }</pre>	<pre>int __fastcall test4(int a1, int a2) { return a1 + a2; }</pre>

}	}
---	---

As clearly seen from Table, the Utility and IDA+Hex-Rays do not fully support the functions that return the structure type – i.e. variables stored in more than one register. However, as a result of IDA+Hex-Rays execution, the algorithm has been restored in an absolutely incorrect way, since, according to its logic, the function should return only the value of the first member of the structure – «OpRes::add» (via the register R3). In such a case, a situation where the returned structure is stored in several registers is more than realistic in modern processors and compilers. On the other hand, although the Utility has not clearly specified the structure type in the signature and for the variables, however, the algorithm has been restored correctly – the function returns a series of 4 values, according to all member of the OpRes structure. In fact, the structure type corresponds physically to an ordered sequence of variables, and such an interpretation may be considered as allowable. Also, the absence of support for variable types in the Utility (with a few exceptions in simple cases) meets one of its principles – eliminating «semantic garbage» from the algorithm description, which is unnecessary for understanding the essence of the algorithm operation.

E. Test 5. Handling Global Variable

The test consists of 3 global variables with given addresses and a function that returns their paired multiplication with the input parameters (see Table 5).

TABLE 5. SC, AC AND ALGORITHMIZED CODE FOR TEST 5

SC Prototype	AC for Testing
<pre>int x, y, z; int test5(int a, int b, int c) { int t = a*x + b*y + c*z; return t; }</pre>	<pre>var x & 0x0000003C; var y & 0x00000040; var z & 0x00000044; test5() { 0x00000000: test5: 0x00000000: lis r6, x@h 0x00000004: lwz r6, x@l(r6) 0x00000008: mullw r3, r3, r6 0x0000000C: lis r7, y@h 0x00000010: lwz r7, y@l(r7) 0x00000014: mullw r4, r4, r7 0x00000018: lis r8, z@h 0x0000001C: lwz r8, z@l(r8) 0x00000020: mullw r5, r5, r8 0x00000024: mr r9, r3 0x00000028: add r9, r9, r4 0x0000002C: add r9, r9, r5 0x00000030: mr r3, r9 0x00000034: blr }</pre>
Outcome of the Utility	Outcome of IDA+Hex-Rays
<pre>var x & 0x3c; var y & 0x40; var z & 0x44; (r3) test5(w1_, w2_, w3_) { return (w1_ * x) + (w2_ * y) + (w3_ * z); }</pre>	<pre>int __fastcall test5(int a1, int a2, int a3) { return a1 * x + a2 * y + a3 * z; }</pre>

As can be clearly seen, the Utility, in addition to the algorithm function, has generated a list of global variables with their absolute addresses – this can be useful in the analysis of their assignment, based on their location in the memory. In

other respects, the restored algorithms can be regarded as identical.

F. Test 6. Restoring a Sophisticated Algorithm

The test consists of a function that finds the senior non-zero bit of a number using a «quick» algorithm: by an iterative division of a segment of the bit sequence into fragments with a selection of the senior non-zero one (see Table 6).

TABLE 6. SC, AC AND ALGORITHMIZED CODE FOR TEST 6

SC Prototype	AC for Testing
<pre>int test6(int x) { int d1 = 0; int d2 = 32; for (int n = 4; n >= 0; --n) { int m = (d1 + d2) / 2; if ((x >> m) == 0) d2 = m; else d1 = m; } return d1; }</pre>	<pre>test6() { 0x00000000: test6: 0x00000000: mr r8, r3 0x00000004: li r3, 0 0x00000008: li r10, 0x20 0x0000000C: li r11, 4 0x00000010: loc_10: 0x00000010: add r0, r3, r10 0x00000014: srwi r9, r0, 1 0x00000018: srwi r0, r8, r9 0x0000001C: beq loc_28 0x00000020: mr r3, r9 0x00000024: b loc_2C 0x00000028: loc_28: 0x00000028: mr r10, r9 0x0000002C: loc_2C: 0x0000002C: addic r11, r11, -1 0x00000030: bge loc_10 0x00000034: blr }</pre>
Outcome of the Utility	Outcome of IDA+Hex-Rays
<pre>(r3, r10) test6(w1_) { _w2 = 0; _w3 = 0x20; @w4 = 4; loop { @w5 = (_w2 + _w3) / 2; if (w1_ >> @w5 == 0) { _w3 = @w5; } else { _w2 = @w5; } --@w4; } (@w4 >= 0); return (_w2, _w3); }</pre>	<pre>unsigned int __fastcall test6(unsigned int a1) { unsigned int v1; // r8@1 unsigned int result; // r3@1 unsigned int v3; // r10@1 signed int v4; // r11@1 v1 = a1; result = 0; v3 = 32; v4 = 4; do { if (v1 >> ((result + v3) >> 1)) result = (result + v3) >> 1; else v3 = (result + v3) >> 1; --v4; } while (v4 >= 0); return result; }</pre>

As clearly seen from Table, the logic and the overall presentation of the restored algorithms by both software tools fully comply with those specified in the SC; however, there are the following differences. Firstly, the function that is restored using the Utility returns the value in two variables (r3 and r10), instead of one (int type), as in the SC. However, it is quite logical, because, according to the AC, a situation where a function algorithm returns the values in two specified registers has every right to exist. This situation is solved after analyzing the algorithm and assigning a function by the Expert

and after making amendments to the assembly code by explicitly defining its signature («(r3) test6(w1_)»). Secondly, the algorithm after the Utility has a more optimized presentation due to the removal of the common expression «(w2+w3)/2», unlike the redundant one «(result+v3)>>1» in the code after IDA+Hex-Rays. Such a generation of a suboptimal algorithm presentation by the Utility's competitor is likely due to the absence of required modules therein. And, thirdly, a special record of the cycle in the optional post-prefix form «loop (pre-cond) { ... } (post-cond);» (where the *pre-cond* and *post-cond* conditions are not obligatory) allows us to create all possible variants of cycles in a unified manner: an endless cycle, with a precondition, a post-condition or with conditions, simultaneously. Consequently, there is a great opportunity for an abbreviated record of the algorithm iterative parts while minimizing the number of operations used for this.

G. Test 7. Restoring the Code Algorithm of a Non-Structural Programming Paradigm

The test consists of a function the algorithm of which is consciously built in violation of the structural programming paradigm – by using the «goto» operator to jump from one branch of a conditional branch to another branch on the same level in the hierarchy. The essence of the algorithm is incrementing or decrementing the second parameter on the basis of the sign of the first one, which is subsequently returned by the function. In such a case, if the first parameter is equal to 0 or the second parameter is less than 0 or more than 10, the error subroutine is called – «print_error()» (see Table 7).

TABLE 7. SC, AC AND ALGORITHMIZED CODE FOR TEST 7

SC Prototype	AC for Testing
<pre>void error(void); int test7(int x, int y) { if (x > 0) { y++; if (y > 10) goto label; } else if (x < 0) { y--; if (y < 0) goto label; } else { label; print_error(); } return y; }</pre>	<pre>test7(){ 0x00000000: test7: 0x00000000: cmpwi r3, 0 0x00000004: ble loc_18 0x00000008: addi r4, r4, 1 0x0000000C: cmpwi r4, 0xA 0x00000010: bgt loc_30 0x00000014: b loc_34 0x00000018: loc_18: 0x00000018: cmpwi r3, 0 0x0000001C: bge loc_30 0x00000020: addi r4, r4, -1 0x00000024: cmpwi r4, 0 0x00000028: blt loc_30 0x0000002C: b loc_34 0x00000030: loc_30: 0x00000030: bl print_error 0x00000034: loc_34: 0x00000034: mr r3, r4 0x00000038: blr }</pre>
Outcome of the Utility	Outcome of IDA+Hex-Rays
<pre>(r3) test7(w1_, w2_) { if (w1_ <= 0) { if (w1_ < 0) { --w2_; if (w2_ >= 0) { goto loc_34; } } } } else { </pre>	<pre>int __fastcall test7(int a1, int a2) { int v2; // r4@2 if (a1 > 0) { v2 = a2 + 1; if (v2 <= 10) return v2; </pre>

<pre>++w2_; if (w2_ <= 0xA) { goto loc_34; } } () = print_error(); loc_34:; return w2_; }</pre>	<pre>goto LABEL_6; } if (a1 >= 0 (v2 = a2 - 1, v2 < 0)) LABEL_6: print_error(); return v2; }</pre>
--	---

As clearly seen from Table, the control flow of the restored algorithms essentially differs from that given by the SC, which has a symmetrical form with a clear logic. Such a behavior of the assessed algorithmization tools is quite natural, since each of them tends, in varying degrees, to obtain a structured code (the Utility to a greater, and IDA+Hex-Rays to a lesser extent), which is naturally possible not for all cases. However, there is an obvious IDA+Hex-Rays predisposition to use several exit points, that is completely excluded when using the Utility. As noted for the previous tests, such a topology of the algorithm is less preferred while using in the Method. Also, the algorithm restored by the Utility is more symmetrical, as compared with the result of IDA+Hex-Rays.

H. Test 8. Algorithm Conciseness

The test consists of a global variable and a function that takes two parameters; the first one is partially used for calculating the values of the memory, the address of which is stored in the second one (i.e. the second parameter has the pointer type). Conciseness is understood as optimizing the algorithm visible part description by using special structures, without changing the details of its logic (see Table 8).

TABLE 8. SC, AC AND ALGORITHMIZED CODE FOR TEST 8

SC Prototype	AC for Testing
<pre>int g; // at 0x48 int test8(int x, unsigned char *y) { int t = 0x10; x++; *y = 1; y[1] += 2; y[2] = t + 0x38; y[3] = x * x * x; y[4] = x & 0x4; x++; return x; }</pre>	<pre>var g & 0x00000048; test8(){ 0x00000000: test8: 0x00000000: li r5, 0x10 0x00000004: addi r3, r3, 1 0x00000008: li r6, 1 0x0000000C: stb r6, 0(r4) 0x00000010: lbz r6, 1(r4) 0x00000014: addi r6, r6, 2 0x00000018: stb r6, 1(r4) 0x0000001C: addi r7, r5, 0x38 0x00000020: stb r7, 2(r4) 0x00000024: mr r8, r3 0x00000028: mullw r8, r8, r3 0x0000002C: mullw r8, r8, r3 0x00000030: stb r8, 3(r4) 0x00000034: rlwinm r9, r3, 0, 29, 29 0x00000038: stb r9, 4(r4) 0x0000003C: addi r3, r3, 1 0x00000040: blr }</pre>
Outcome of the Utility	Outcome of IDA+Hex-Rays
<pre>var g & 0x48; (r3) test8(w1_, w2_) { ++w1_; *w2_ = 0x1; w2_[1] += 2; w2_[2] = &g; w2_[3] = w1_ ** 3; w2_[4] = w1_ - 3; return w1_ + 1; }</pre>	<pre>int __fastcall test8(int a1, _BYTE *a2) { int v2; // r3@1 v2 = a1 + 1; *a2 = 1; a2[1] += 2; a2[2] = 64; a2[3] = v2 * v2 * v2; }</pre>

}	a2[4] = a1 & 0x4; return v2 + 1;}
---	--------------------------------------

Although the function algorithm, as restored by both SWT, is virtually identical, however, its description using the Utility, due to conciseness, may be considered as more compact and human-friendly. Conciseness in this example, which is absent in the IDA+Hex-Rays operation, is achieved due to the following techniques. Firstly, to increment the variable by 1, the increment operator «++» is used, that is common in the C/C++ language. Secondly, the value of 0x48 is replaced with the operator for taking the address of the global variable – «&», which may be useful in the analysis of implicit data flows. Thirdly, repeated multiplication of a variable by itself has been replaced with the exponentiation operator «**», which is non-standard for the C/C++ language, but which is largely used in such languages as Fortran, Haskell, Python, Ruby, etc. And, fourthly, the access to a bit of the number using bitwise operation with the «&» mask has been replaced with a more logical and obvious operator: «.» with the bit number specified, that is identical to the access to a structure member in the form of «S.x».

I. Test 9. Finding a Potential Vulnerability

The test consists of a function that increments the values of a global variable by a value passed through its parameter. A feature of the program is that, according to the alleged logic of its operation, the variable stores a secret predefined value that is read-only – i.e. the variable has a conditional attribute *ReadOnly*. Thus, changing this value by the function is considered as a vulnerability. Whereas changing the object value in the C language may be prevented using the qualifier «const», in the MC, it is almost impossible (see Table 9).

TABLE 9. SC, AC AND ALGORITHMIZED CODE FOR TEST 9

SC Prototype	AC for Testing
<pre>int secret_for_read; // Secret value, that is can't be rewritten int test9(int x) { secret_for_read += x; }</pre>	<pre>var secret_for_read & 0x0000001C; test9(){ 0x00000000: test9: 0x00000000: lis r4, se- cret_for_read@h 0x00000004: lwz r5, se- cret_for_read@l(r4) 0x00000008: add r3, r3, r5 0x0000000C: lis r4, se- cret_for_read@h 0x00000010: stw r3, se- cret_for_read@l(r4) 0x00000014: blr }</pre>
Outcome of the Utility	Outcome of IDA+Hex-Rays
<pre>var secret_for_read & 0x1c; () test9(w1_) { /* ATTENTION!!! Write to variable with readonly access. */; secret_for_read += w1_; return (); }</pre>	<pre>int __fastcall test9(int a1) { int result; // r3@1 result = a1 + secret_for_read; secret_for_read = result; return result; }</pre>

The vulnerability in this test, due to the specifics described, is subjective (generally, only the Expert can identify the variables that are read-only). Thus, no automatic tools cannot identify it – this required a human intervention.

IDA+Hex-Rays has no means to solve such tasks, in particular, due to the absence of embedded capabilities for finding vulnerabilities, whereas the use of embedded scripts for such an analysis can be a non-trivial task.

As for the Utility, it is designed just for solving this task and allows for making amendments into its algorithm operation immediately in the AC (as obtained from the tested MC).

Thus, adding to the AC of the following lines:

```
user_control {
    secret_for_read: p_readonly;
}
```

should indicate to the Utility that the object *secret_for_read* has the *ReadOnly* attribute and can only be read. The Expert may come to the same conclusion about the object attributes at the second iteration when running the Utility after the initial analysis of the restored algorithm code. If, following the MC analysis, it is found that an object with the *ReadOnly* attribute has been changed, the Utility will make appropriate warning: «/* ATTENTION!!! Write to variable with readonly access. */».

As mentioned above, IDA+Hex-Rays allows only to search for vulnerability directly, because it misses the respective modules and does not displays supporting information, as required for searching. Theoretically, the use of built-in scripts could solve this problem; however, although the required labor input for the current test is predicted as satisfactory, for other vulnerabilities it may be extremely high. On the other hand, the Utility, due to the possibility of making manual amendments and iterative execution, has correctly revealed, on the second iteration, an authorized access to the variable, which, according to the operating logic, cannot be changed. Note that due to the architecture of the Utility, it can easily be expanded module automatically search for vulnerabilities. Many of the algorithms used in such modules often use information derived from the IDA Pro [6].

III. ASSESSMENT RESULTS

Let us summarize the assessment results of the Utility and IDA+Hex-Rays by comparative analysis of the tests output. The results will be also presented in a tabular form, where for each SWT and test, there will be specified the success of algorithm restoring from the AC (corresponding to the SC) in the following way: «-2» – the restoration is not correct (including no vulnerability found), «-1» – there are non-critical differences, «0» – almost full compliance, «+1» – fully restored and has a more “successful presentation” (see Table 10).

TABLE 10. SWT ALGORITHMIZATION ASSESSMENT RESULTS

SWT	Test 1	Test 2	Test 3	Test 4	Test 5
Utility	-1	-1	+1	0	0
IDA+Hex-Rays	0	0	0	-1	-1
SWT	Test 6	Test 7	Test 8	Test 9	Total
Utility	0	-1	+1	0	-1
IDA+Hex-Rays	-1	-2	0	-2	-7

Therefore, conditionally believing that the weights of all tests in the general summary comparison of the SWT algorithmization are approximately equal, the Utility has a prefer-

able total number of -1 points against -7 points for IDA+Hex-Rays. This result is reinforced by the fact that almost all lags of the Utility from IDA+Hex-Rays may be solved by adding required functionality therein, being a purely engineering (programmer) task, requiring no changes in the existing implementation architecture. In the same time, the Utility has some possibilities that may be applied for automated search for medium-level vulnerabilities.

IV. CONCLUSIONS

This article is the final one in the series of authors' works devoted to the algorithmization method for finding vulnerabilities in the MC of TCD. Thus, these papers have initially provided the overall analysis of the problem and the ways to solve the related task, and then they have described the Method of its solution, including the architecture of the involved Utility. All this made it possible to speak only about theoretical results of the research, only assuming their authenticity in practice. The results of the comparative analysis of the Utility with its closet analog – IDA Pro with the Hex-Rays plug-in, as provided in the present article, in terms of their application in the Method, allow for reasonably arguing that the suggested Method and the SWT prototype cope well with their task, while surpassing the competitor in most aspects of algorithmization. The successful architecture of the Utility and its inherent extensibility allows us to quickly adapt the latter to the possible Expert's requirements for the code security, whereas the possibility of the Utility iterative execution with amendment of this process will provide a greater potential for finding vulnerabilities that require a subjective multi-stage analysis.

REFERENCES

- [1] Buinevich M., Izrailov K., Vladyko A. Method for Partial Recovering Source Code of Telecommunication Devices for Vulnerability Search // 17th International Conference on Advanced Communication Technology (ICACTION). IEEE. 2015. pp. 76-80
- [2] Buinevich M., Izrailov K., Vladyko A. The life cycle of vulnerabilities in the representations of software for telecommunication devices // 18th International Conference on Advanced Communication Technology (ICACTION). IEEE. 2016. pp. 430-435
- [3] Buinevich M., Izrailov K., Vladyko A. Method and Prototype of Utility for Partial Recovering Source Code for Low-Level and Medium-Level Vulnerability Search // 18th International Conference on Advanced Communication Technology (ICACTION). IEEE. 2016. pp. 700-707
- [4] Buinevich M., Izrailov K. Method and utility for recovering code algorithms of telecommunication devices for vulnerability search // 16th International Conference on Advanced Communication Technology (ICACTION). IEEE. 2014. pp. 172-176
- [5] Forin A., Troshina A., Chernov A. Reconstruction of Class Hierarchies for Decompilation of C++ Programs // 14th European Conference on Software Maintenance and Reengineering (CSMR). IEEE. 2010. pp. 240-243
- [6] Cheng S., Yang J., Wang Jia., Wang Jin., Jiang F. LoongChecker: Practical Summary-Based Semi-simulation to Detect Vulnerability in Binary Code // 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE. 2011. pp. 150-159



Mikhail Buinevich was born in 1958 in the USSR. He received education of the military engineer of electronic engineering. He served in the naval fleet and government agencies for information security. He held classes at various universities. His research interests include methods of information security. He has more than 100 scientific works. His primary publications are as follows:

1. M.V. Buinevich and others. Safety provision of high-security objects of the naval fleet in relation to damage effects in crisis and emergency situations in peacetime. / Under the editorship of the admiral V.S. Vysotskii. – Saint Petersburg: Publishing house ELMOR, 2008. – 300 p.
2. M.V. Buinevich and others. Provision of organizational and technical support of stability of function and safety of general communications network. / Under the general editorship of S.M. Dotsenko. – Saint Petersburg: Publishing house SPbSUT, 2013. – 142 p.

At the present time Dr. Prof. Buinevich is the professor of Bonch-Bruевич Saint-Petersburg State University of Telecommunications.



Konstantin Izrailov was born in 1979 in the city of Saint Petersburg (Russia). In 1996 he graduated from Saint Petersburg State Polytechnic University, Physical and Mechanical Department. At the moment he is a researcher of Bonch-Bruевич Saint-Petersburg State University of Telecommunications. He has about 30 published articles; he is an author of 3 scientific and research works and has a patent on the software

tool. His scientific interests include information security, search of vulnerabilities in machine code, reverse engineering and telecommunication devices. Mr. Izrailov has the title of the best postgraduate student of SPbSUT in 2012 and is the presidential scholar in 2013.



Andrei Vladyko (IEEE member (M'14)) acquired his degree of PhD at Komsomolsk-on-Amur State Technical University, Russia in 2001.

At present he is a head of R&D department of The Bonch-Bruевич Saint-Petersburg State University of Telecommunications, Saint-Petersburg, Russia.

His major interests include Control Systems, Internet of Things, Ubiquitous Sensor Network, Information & Network Security, Software-Defined Networking.