Method and Prototype of Utility for Partial Recovering Source Code for Low-Level and Medium-Level Vulnerability Search

Mikhail Buinevich*, Konstantin Izrailov*, Andrei Vladyko*
*The Bonch-Bruevich 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 describes a automated method for searching of low-level and medium-level vulnerabilities in machine code, which is based on its partial recovering. Vulnerability search is positioned in the field of telecommunication devices. All various and typical vulnerabilities in source code and algorithms for its search is given. The article contains examples of usage method and its utility. There is forecast to develop methods and utilities in the near future.**

***Keywords*— machine code, reverse-engineering, static analyzer, telecommunication devices, vulnerability**

I. INTRODUCTION

Accidental and intentional errors that lead to vulnerabilities in software (hereinafter referred to as the SW) are one of the top challenges of the modern worlds that have taken the informatization path of development. Though the errors that have been made by hackers are seen less, they make the SW more unsafe, as long as such errors are the ultimate goal of the attackers. With view to the fact that critical information is usually shared via telecommunication devices, with the functional of such devices implemented with the help of the SW, the task of vulnerability search is one of the principal tasks of information security. This task is sophisticated and depends on an underdeveloped search methodology, which is defined via a set of methods used for such search. In this context, the efficiency of such methods is a function of purely practical application aspects, i.e. speed and complexity, and is now estimated as extremely unsatisfactory.

Therefore, development of new and highly efficient methods for vulnerability search for telecommunication device SW is of theoretical (in terms of methodology) and, obviously, practical interest.

II. ANALYZING

In order to analyze available methods for SW vulnerability search, we would like to break such methods into the following groups by their application target.

Any methods that are applied exclusively to the SW source code fall into the first group and they are most abundant. Such methods are quite developed and are used very efficiently. A large base of typical source code vulnerabilities and methods for identification of such vulnerabilities has been collected. CppCheck, Lint and Cland may serve as implementation examples for the C/C++ code.

If the source code is not available, you will have to search for vulnerabilities using the final representation, i.e. machine code. Such methods form the second group and they usually rely on code disassembling and manual analysis by security experts (hereinafter referred to as the Expert). Individual implementations of such methods you may found in such products as Binary Static Analysis (SAST) by Veracode and Software Static Analysis Toolset by MALPAS. However, such methods have a different ultimate goal and may not be used as a comprehensive solution for the search of machine code vulnerabilities. However, a certain quantity of theoretical works, that are close to the solution of this problem still exists and is based on the use of the character programming [1].

It should be mentioned that the source code here means any code that must be compiled in a platform-dependent machine code for running such code – so called unmanaged code (e.g., C++, Pascal). A managed code (e.g. Java, C#) is an alternative and is compiled into an interim bytecode, which is then run on a virtual machine and is unambiguously converted into the source code. Availability and search for any vulnerabilities in the managed code are, obviously, a different task, which is less popular and more simple.

Manuscript received on June 19, 2015. This work is a follow-up of the invited journal to the accepted conference paper of the 17th International Conference on Advanced Communication Technology.

M. V. Buinevich is with The Bonch-Bruevich Saint-Petersburg State University of Telecommunications, Russian Federation, Saint-Petersburg, 22-1 Prospekt Bolshevikov (e-mail: bmv1958@yandex.ru).

K. E. Izrailov is with The Bonch-Bruevich Saint-Petersburg State University of Telecommunications, Russian Federation, Saint-Petersburg, 22-1 Prospekt Bolshevikov (corresponding author to provide phone: +7(921)555-2389; fax: none; e-mail: konstantin.izrailov@mail.ru).

A. G. Vladyko is with The Bonch-Bruevich Saint-Petersburg State University of Telecommunications, Russian Federation, Saint-Petersburg, 22-1 Prospekt Bolshevikov (e-mail: vladyko@bk.ru).

As long as a telecommunication device is usually supplied with the SW in the form of the machine code already installed on such device and one does not have any access to the source code, it seems so that the search for any vulnerabilities in the telecommunication device is done at the moment exclusively using manual techniques.

We should also mention a vast majority of telecommunication device models, modifications and SW versions leading to a tremendous number of various machine code images. And such machine code images may be of a rather big size – up to hundreds of megabytes. Also, it is obvious that each and every image must be analyzed "from scratch".

Thus and so, there is a critical task of search for vulnerabilities in the telecommunication device machine code in the above area of interest, and its solving efficiency is obviously not satisfactory. In this context, efficiency means a total number of vulnerabilities found in the machine code for a reasonable time, but not in the defined volume (i.e. vulnerability density). So, the manual technique would allow for identification of a larger number of vulnerabilities in small machine code volumes vs. any automated techniques. However, application of such technique for any line of telecommunication device machine code will be so time-consuming in practice, even for a medium-sized volume, that the results will just not keep up with the release of new upgrades. Highly qualified Experts must also always be available for application of the manual technique, which is not always possible.

Design of a method and means of automated search for vulnerabilities in the machine code may be an obvious solution of the task, in which case involvement of human factor, especially highly qualified, will be minimized. Operational capability and vulnerability identification for large code volumes exactly for a small time at the expense of quality and density of the vulnerabilities found is the basic requirement to such method. However, the last factor is not critical, as long as you may always add any manual search methods, with the findings of the automated search simply facilitating such work.

Comparison of manual and automatic methods for vulnerability search, using various criteria, is presented in Table I.

TABLE I
COMPARISON OF METHODS FOR VULNERABILITY SEARCH BY CRITERIA

Criteria	Manual	Automatic
Search time	Long	<u>Short</u>
Number of detected vulnerabilities	Many	Few
Detected vulnerability patterns	All	Template-based
Amount of code processed	Small	<u>Large</u>
Required qualification	Expert	<u>Engineer</u>
Ability to bypass anti-detection mechanisms	Yes	Sometimes
Source code information	Preferable	<u>Not needed</u>
Formalization of results	Possible	<u>Always</u>

Advantages of the automatic method are highlighted in Table I, which brings us to the following conclusion: fully automated search methods can be used efficiently to detect telecommunication devices machine code vulnerabilities.

For better review of the object domain, let us divide all vulnerabilities into 4 types, according to their layout at the software build-up levels. Such division is presented in Figure 1.

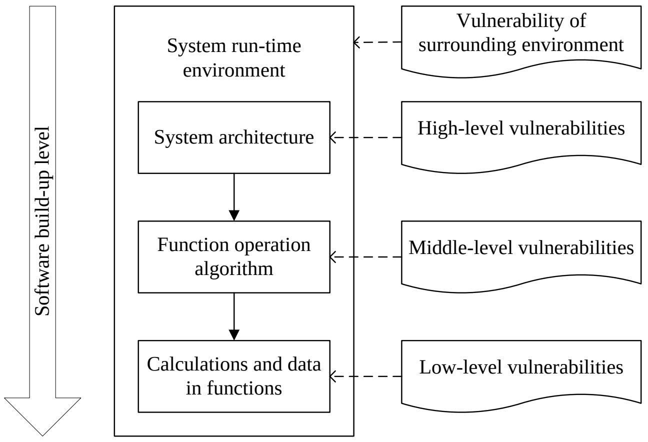
Software build-up level
System run-time environment
System architecture
Function operation algorithm
Calculations and data in functions
Vulnerability of surrounding environment
High-level vulnerabilities
Middle-level vulnerabilities
Low-level vulnerabilities

Fig. 1. Types of vulnerabilities according to SW build-up level

First of all, these include low-level vulnerabilities, such as calculation errors, data structure, data access etc. For example, dividing by zero, or incorrect structure field processing due to non-standard alignment of the members, may fall into this type. Second of all, these include medium-level vulnerabilities, such as incorrect implementation of algorithms and functions, transfer of input parameters, function returns etc. For example, occurrence of infinite cycles and recursions, appearance of an unreachable code (despite the fact that such code does not usually lead to any adverse effects), and IF-ELSE transition test errors may fall into this type. Third of all, these include high-level vulnerabilities, such as program system architecture errors [2]: violation of general principles of system functioning and security, incorrect implementation of various mechanisms and protocols etc. For example, errors in implementation of algorithm of common secret key importation, according to Diffie-Hellman protocol may fall into this type. And, forth of all, these include vulnerabilities of the surrounding environment, such as errors in run-time modules of the active program system etc. For example, DLL injection (running code within the address space of another process by forcing it to load a dynamic-link library) and operating errors of objective code loader into the system address area (ld.so for Unix-systems) may fall into this type.

The following available developments may be used as initial data for the task solution. First of all, as mentioned before, the development of methods for vulnerability search has allowed us to gather certain information about the vulnerabilities in the source code and search specifics. Second of all, the required vulnerability search may be implemented via conversion of the machine code of the device into the source code and application of the available search algorithms to such code. Therewith, any previous work of the authors [3] also touched this task directly, as long as it aimed at machine code algorithm recovery in the telecommunication device, which may be considered a partial source code recovery. Having combined such developments and with view to our goal, we hereby offer a solution of the task, which includes two major steps. We must systemize typical source code vulnerabilities and adapt such vulnerabilities to the machine code, i.e. select the vulnerabilities that may be found and identified in the machine code first. We must then develop a method for machine code vulnerability search (hereinafter referred to as the Method) based on the original recovery method for the algorithms and adapted vulnerabilities in the machine code. The Method offered is designed for searching two types of vulnerabilities, such as: low-level vulnerabilities and partially medium-level vulnerabilities, which is the most important peculiarity of this Method. In this case, the basic method of the authors [3] is suitable for full-sized search of medium-level and high-level vulnerabilities only. Machine code of the telecommunication device is usually a detached binary image and is run completely, using special hardware. Therefore, search of vulnerabilities of the surrounding environment is not covered by this task. Let us further consider vulnerabilities that are connected to the proposed technique.

A. Adapted vulnerabilities in the machine code.

As long the vulnerability in the SW is basically an integral part of the SW contents (i.e. functional), any code presentation shape conversions may not affect vulnerability in any way, i.e. make it disappear or change significantly. Therefore, any source code vulnerability will be reflected in the machine code to some degree. However, due to any operations that reduce the program “structurization” (compilation, assembling), the final presentation of the vulnerability may become "washed out", which may make it practically impossible to identify such vulnerability, even in cases of reversed engineering (disassembling, decompilation). Let us mark out the vulnerabilities that can be found in the source code, and are still integral in the machine code, which will be indicative of the potential ability of them being identified in the code. For this purpose there is a sufficient quantities of theoretical studies and practical implementations, for example, in the works [4] and [5].

Vulnerability 1 – Dividing by 0

This vulnerability can be found in operations of dividing by 0, which should not occur in any correct programs at all. The reasons for such vulnerability may include missed check by zero value of denominators of expressions, and operational logic mistakes. Exclusion of division by zero will be called up as a result, which may lead to incorrect program exit. This vulnerability search algorithm is based on defining any possible variable value ranges, which are included in the expression with dividing and signalization of the possibility of the dominator equal to 0. Example of a code with this vulnerability is given at the following listing.

```
if(y == 0)
  z = x / y;
else
  z = x * y;
```

Vulnerability 2 – Using a non-initialized variable

This vulnerability occurs during the first use of a variable that was not assigned any initial value. The reasons for such vulnerability may include any programmer's error, which usually includes missing initial value of the variable or assumption of such value equalling to 0 by default. As a result, the variable may take random values (so called 'garbage'), which will lead to incorrect calculations. The vulnerability search algorithm is based on determining locations of the initial variable assignment/use and signalization, if such use occurred before the assignment. . An example of code, which involves this vulnerability, is presented in the following listing.

```
int x, y;
y = x * 2;
```

Vulnerability 3 – Buffer overflow

This vulnerability occurs due to no control over going out of the object stored in the buffer, which usually constitutes an array. Overwriting of the memory area, which is physically located out of the buffer, may occur and it may lead to writing off the contents of any other objects and change of the program code, or simply exclusion into the protected memory based on the record. The vulnerability search algorithm is based on detecting the buffer memory range, operation algorithms for the indexes of buffer objects, and possible values of such pointers, and signalization in case of index going out of the range [6]. Example of a code with this vulnerability is given at the following listing.

```
int arr[10];
…
arr[10] = 0;
```

Vulnerability 4 – Handling incorrect memory pointers

This vulnerability can be found, while dereferencing of pointer containing incorrect memory paths. The reasons for such vulnerability may include errors in function operation algorithm or incorrect pointer initialization. Reading and writing using the pointer may be done on the area of the executed code as a result. Dereferencing of the 0x0th pointer is a special case. The vulnerability search algorithm is based on determining pointer values and dereference locations. Example of a code with this vulnerability is given at the following listing.

```
int *ptr_1 = 0x0;
int *ptr_2 = &funct;
*ptr1 = 0;
*ptr2 = 0;
```

Vulnerability 5 – Memory leaks

This vulnerability can be found in case of unlimited memory use. The reasons for such vulnerability may include missing required operations to free up the memory, in particular in case of cyclic execution. Memory shortage exclusion or write-off of the contents of used objects (due to incorrect stack handling) may occur as a result. Although such situation does not make the full-featured vulnerability, it may interrupt program operation in case of prolonged use, which typical just for the telecommunication device. Unfortunately, there is not any common vulnerability search algorithm available (as long as memory selection and freeing-up are phenomena that are defined exclusively within the program and program libraries). Nevertheless, manual algorithm set-up, such as obvious function and memory operation template determination will allow for identifying vulnerabilities of such type with the help of management flow graph analysis, including call-ups of such functions in the graph. An example of code, which involves this vulnerability, is presented in the following listing ('malloc()' is a memory allocation function).

```
funct(int x){
  int *ptr;
  while(x){
    ptr = malloc(10);
    if(x % 2 == 0)
      dummy(ptr);
    --x;
  }
}
```

Vulnerability 6 – Infinite loops and recursions

This vulnerability can be found in case of looping of function management flows or calling up the function as a result of incorrect program logics. Thus and so, the program may go, under certain circumstances, into a cycle that does not have any executable conditions for completion, or call up the same function indefinitely. This vulnerability can be identified by building complete management flow graphs and call-ups with further analysis of the conditions for simultaneous execution of their paths. It should be mentioned that the software of the telecommunication device often has an architecture, which consists of the single processing cycle for the incoming network packages, and such case must be processed separately. An example of code, which involves this vulnerability, is presented in the following listing.

```
int funct(int x){
  return funct(x+1);
}
…
bool flag = false;
do{
  …
  flag = true;
}while(flag);
```

Vulnerability 7 – Unused code

This situation happens, if there are code areas in the programs with their instructions never to be run, and it corresponds to the destructed function algorithm structure. The reasons for such vulnerability may include errors in function algorithm operation, and a consequence of "rough" introduction of an alien code in the program. Although such situation does not make the full-featured vulnerability, it is indicative of an abnormality in the machine code, which is a potential vulnerability of the "bookmark" type. An example of code, which involves this vulnerability, is presented in the following listing.

```
int funct(int x){
  if(x)
    return 1;
  else
    return 2;
  x += 1;
  return x;
}
```

B. Method for vulnerability search in the machine code.

As offered, the method for vulnerability search in the machine code must contain consecutive steps for source code recovery and vulnerability search using such code with the help of the existing algorithms. Although complete source code recovery (i.e. decompilation) is not practically possible, it may be recovered partially, which is a minimum requirement to search performance. Thus, for example, it is not crucially necessary to recover names of variables and points of return from the function in order to identify buffer overload vulnerability.

As mentioned before, a number of sub-tasks (algorithm recovery to be specific) were solved partially in the original method. Therefore, some phases of this method may be used in this Method. In particular, it is reasonable to use the IDA Pro product [7], which has the following functional. First of all, this product is a full-scale machine code disassemble for a large variety of processors. And second of all, this product performs a partial machine code recovery, as long as it divides memory areas into functions and data blocks, uses debugging information, if any in the machine code, and recognizes library functions by their signatures. It should be mentioned that crucial product specifics, such as interactivity and debugging capacity, may not be used for this Method in practice.

The rest of the Method relies on implementation of adapted machine code vulnerability search algorithms using the interim representation including output of any findings.

If we sum up the above, basic phases of the Method are:

Phase 1 – Disassembling machine code

The machine code is converted to the assembler for disassembly and analysis. This phase may be fully implemented based on the IDA Pro. This phase may be implemented completely on the basis of IDA Pro, which includes an external API and supports internal scripts. A rather correct division of code and data sections, definition of function body and global variables, which may be used at further phases, are the operational specifics of this product.

Phase 2 – Recovering partially source code

The assembly code is disassembled, an internal program representation is built, necessary conversions are made, source code elements are assumed, and algorithms and other necessary information is recovered – this phase and all further phases must be carried out using a specialized software application (hereinafter referred to as the Utility). It is obvious that SW analysis for a certain processor is not possible, unless its machine code is supported in the IDA Pro and FrontEnd of Utility (i.e. input parser) is available for its assembler. The basics code recovery is made using the IDA Pro immediately after the decompilation and it is reflected in the assembler, which is generated in Phase 1. This phase may be implemented in most part using algorithms based on the original machine code algorithm recovery method, as complemented by the algorithms gathering the information that is required for searching the adapted vulnerabilities. First of all, aliases must be supported, i.e. information about common code areas that are indicated by several different pointers or objects. Second of all, calculation of any possible value ranges, which are used in the object program, including pointers is required. Third of all, variable life time graphs must be plotted, including points of their first and last initialization / use.

Phase 3 – Vulnerability search

Search for each of adapted vulnerabilities is made using the internal code presentation obtained. It is obvious that such presentation must not be a sophisticated text (usually used for SW development), but a set of specialized graphs, tables and structures determining program run. Such presentation will allow for more effective searching. Generalized search algorithms for each vulnerability have been provided earlier.

Phase 4 – Gathering findings

Machine code analysis findings are generated, i.e. information about the code (scope, number of functions), identified vulnerabilities or suspected locations of vulnerabilities etc. This processed SW code for the telecommunication device is especially characterized by its possible large size, various variations, and contents. Thus and so, this Method must be applicable multiple times with summarization of the findings obtained. Therefore, the presentation format of the vulnerabilities found must be suitable for program processing (i.e. its syntax must be strict) and for investigation by the Expert (i.e. human friendly). YAML [8] is a suitable presentation option for this task, which is characterized by such peculiarities as formalization and readability.

Diagram of Method phases and data used for such phases can be found in Fig. 2.

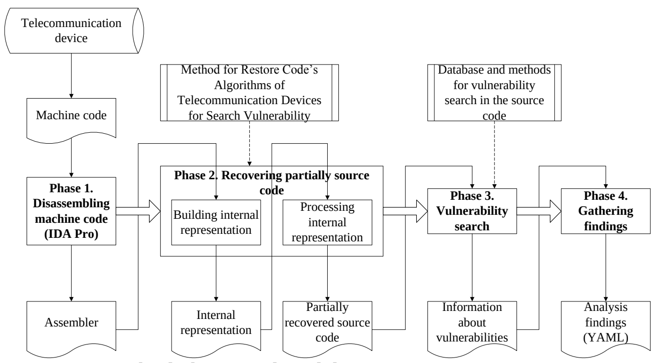
Telecommunication device
Method for Restore Code's Algorithms of Telecommunication Devices for Search Vulnerability
Database and methods for vulnerability search in the source code
Machine code
Phase 1. Disassembling machine code (IDA Pro)
Phase 2. Recovering partially source code
Building internal representation
Processing internal representation
Phase 3. Vulnerability search
Phase 4. Gathering findings
Assembler
Internal representation
Partially recovered source code
Information about vulnerabilities
Analysis findings (YAML)

Fig. 2. Method phases and used data

Note. The process of obtaining the telecommunication device firmware and reducing such firmware to the machine code is out of scope of this Method, as long as it is a separate and purely technical task that may be solved quite effectively.

III. Evaluating the Method

An imaginary experiment of application of individual phases of the offered Method on typical abstract machine codes, containing each of vulnerabilities in the original source code was done to study the specifics of such Method. We will get an unambiguous assembly code representation after the Phase 1. This representation will also include vulnerabilities of all types, as long as IDA Pro converts binary processor instructions into text lines including operations unambiguously. The machine code does not usually include any debugging information, and, therefore, a partial code recovery in the Phase 2 will be done without any source code meta information, such as function names and arguments. This does not appear to be rather significant; however, it may still affect the accuracy of our findings. To the contrary, machine code obfuscation could worsen recovery efficiency significantly, as long as it usually destructs the algorithm structure, albeit it is used quite rarely. Machine code optimization (source code compilation involving optimization options to be specific) is not applied to all SW types; in particular, it is not used often the telecommunication device. Various steganography messages embedded in the executable file, at this phase, will be lost; nevertheless, it is likely note rather than a disadvantage [9]. Efficiency of the Phase 3 depends on the outcomes of the Phase 2 completely, i.e. on the quality of the partially recovered source code. Therefore, assigning of some degree of reliability to the found vulnerabilities may be reasonable. This may prove useful for a semi-automatic application of the Method, i.e. including further analysis done by the Expert. Phase 4 involves flow-by-flow output of vulnerability search findings in a special format and does not depend on the previous phases or analyzed machine code greatly. YAML does not have any application limitations. Therefore, any vulnerability details (if any) for each typical machine code will be gathered in one place and converted into the uniform base suitable for the manual analysis.

According to our imaginary experiment, application of the Method and its individual phases for vulnerability search in the telecommunication device machine code can be justified strictly and logically. The machine code for the processor supported by IDA Pro and Utility will be Method input, and a list of vulnerabilities in YAML will be Method output. Unsuitability for any obfuscated machine codes and average efficiency for any optimized code are among Method limitations.

IV. A Hypothetical Example

Let us consider a hypothetical example of operation of this technique, and pay special attention of Phase 2, which is most complicated. As it was mentioned before, phase realization must be in the form of a separate automating utility and may be taken partially from the utility of the basic technique [3]. At the moment, development of a utility prototype for the technique is in progress; however, we may already predict a scheme and operational details of such utility. Various program representations at all phases of the technique and utility (from a source code to a formalized list of vulnerabilities) are presented below.

A. *Input data (source code)*

Let us assume we have a program that comprises 'funct()' function, which involves a vulnerability in the form of a possible division by zero (a code of such function is presented in the following listing, using the high-level C language).

```
01:    int funct(int x, int y){
02:        int z = 0;
03:        if (y == 0) {
04:            z = x / y;
05:        } else {
06:            z = x * y;
07:        }
08:        return z;
09:    }
```

The vulnerability is located in line '04:' and it occurs, if the IF-ELSE condition is fulfilled in line ’03:’, which leads to dividing variable ‘y’ equalling to 0.

B. Input data (machine code)

Machine code of this example for the PowerPC processing has the following listing.

```
94 21 FF D0 93 E1 00 2C  7C 3F 0B 78 90 7F 00 18
90 9F 00 1C 38 00 00 00  90 1F 00 08 80 1F 00 1C
2F 80 00 00 40 9E 00 18  81 3F 00 18 80 1F 00 1C
7C 09 03 D6 90 1F 00 08  48 00 00 14 81 3F 00 18
80 1F 00 1C 7C 09 01 D6  90 1F 00 08 80 1F 00 08
7C 03 03 78 39 7F 00 30  83 EB FF FC 7D 61 5B 78
```

It is obvious that we will not be able to discover the fact of vulnerability existence by an expert manual technique in this representation. Application of automated search algorithm may be successful; nevertheless, realization of such algorithms will be highly non-trivial in this representation.

C. Phase 1 – Disassembling machine code

Application of the IDA Pro will result in an assembly representation of the program, like the one in the following listing.

```
0x00:    stwu      r1, -0x30(r1)
0x04:    stw       r31, 0x2C(r1)
0x08:    mr        r31, r1
0x0C:    stw       r3, 0x18(r31)
0x10:    stw       r4, 0x1C(r31)
0x14:    li        r0, 0
0x18:    stw       r0, 8(r31)
0x1C:    lwz       r0, 0x1C(r31)
0x20:    cmpwi     cr7, r0, 0
0x24:    bne       cr7, loc_3C
0x28:    lwz       r9, 0x18(r31)
0x2C:    lwz       r0, 0x1C(r31)
0x30:    divw      r0, r9, r0
0x34:    stw       r0, 8(r31)
0x38:    b         loc_4C
0x3C: loc_3C:
0x3C:    lwz       r9, 0x18(r31)
0x40:    lwz       r0, 0x1C(r31)
0x44:    mullw     r0, r9, r0
0x48:    stw       r0, 8(r31)
0x4C: loc_4C:
0x4C:    lwz       r0, 8(r31)
0x50:    mr        r3, r0
0x54:    addi      r11, r31, 0x30
0x58:    lwz       r31, -4(r11)
0x5C:    mr        r1, r11
0x60:    blr
```

Despite the fact that such representation may already be analyzed by experts, required efforts for such process are critically high.

D. Phase 2 – Recovering partially source code

This phase and all further phases must be implemented, using a utility that is being developed at the moment. Most of its algorithms and representations will be similar to those of the basic technique, which uses the operational utility prototype [10].

First of all, the assembly representation will be converted into an abstract syntax tree that describes the source program in a formalized and structured way. Then, it will be converted into a similar internal representation. Complete independency from the run-time processor and source assembler is the specifics of such representation, as long as all operations and variables are fully abstract. The appearance of such trees is quite similar. A text form of such trees is presented in the following listing, where left indent is indicative of the depth of an element.

```
IrList()
  IrFunct('funct')  // Function 'funct()'
    IrArgs
      IrReg('r1')   // Function arg N_1
      IrReg('r2')   // Function arg N_2
    IrLocalVars
      IrReg('r3'), value='0' // Local var N_1
    IrList()        // Function body
      IrBranch      // if (r2 == 0) goto label_1
        IrCond('beq'), kind='=='
          IrReg('r2')
          IrInteger('0')
        IrLabel('label_1')
      IrOperation('mr'), kind='='    // r3=r1*r2
        IrReg('r3')
        IrOperation('mul'), kind='*'
          IrReg('r1')
          IrReg('r2')
      IrGoto('b')                 // goto label_2
        IrLabel('label_2')
      IrLabel('label_1')              // label_1:
      IrOperation('mr'), kind='='    // r3=r1/r2
        IrReg('r3')
        IrOperation('div'), kind='/'
          IrReg('r1')
          IrReg('r2')
      IrLabel('label_2'),    name='label_2'    //
label_2:
      IrReturn('blr')    // return r3
        IrReg('r3')
```

Second of all, a graph of basic blocks, which reflects the sequence of operations and conditional transfers will be constructed, based on the internal representation tree. Third of all, the internal representation tree will be analyzed for any possible values of the variables. We will obtain the following graph, as a result.

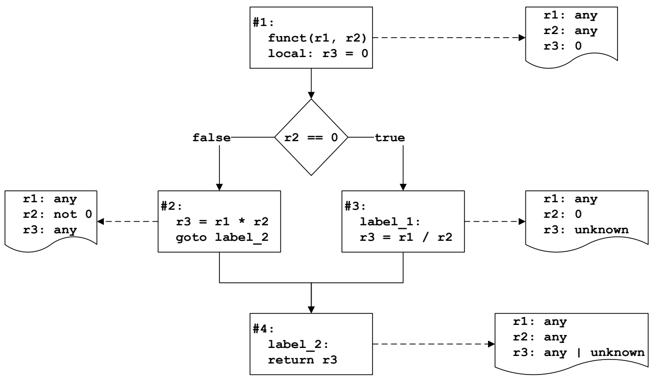

Fig. 3. Intermediate Representation Graph with variables value ranges

According to the graph in Figure 3, Block#1 is a function header, including arguments ‘r1’, ‘r2’ and local variable ‘r3’, which is initialized with ‘0’ value. Conditional check ‘r2 == 0’ and transitions to blocks #2 and #3, depending on the results of such conditional check, will form the first block of the function. Block #2 performs a multiplication of variables ‘r1’ and ‘r2’, and the result is entered into variable ‘r3’. Thus and so, according to the conditional transition, variable 'r2' may take any values, except for '0', and variable ‘r3’ may take any values. Block #3 performs a division of variables ‘r1’ and ‘r2’, and the result is entered into variable ‘r3’. In this case, according to the conditional transition, variable ‘r2’ equals to 0. Therefore, ‘Divide by zero’ exclusion occurs after the division in the program, and value of variable 'r3' will not be defined. Block #4 performs an exit from the function and return of the value in variable 'r3'.

E. Phase 3 – Vulnerability search

Vulnerability search algorithms are applied at this phase. According to the internal representation graph and possible variable values, one of the search algorithms may detect the fact of dividing by 0 in block #3, which leads to the exclusion. This means an incorrect program behaviour in general. This way the error in the form 'Vulnerability – Dividing by 0' is discovered. Thus and so, the algorithm will find the vulnerability in the code and will add details about such vulnerability to the resulting list.

F. Phase 4 – Gathering findings

The final phase of the technique will gather all information about the vulnerabilities and will generate such information in a formalized, but operator-friendly way, i.e. in the YAML format (see the following listing).

```
Sources:
  - Name: "test.asm"
  - Vulnerabilities:
    -
      - Type: "Dividing by 0"
      - Machine address: 0x00000030
      - Machine instruction: "divw r0, r9, r0"
      - BasicBlock: 3
      - IRSubTree: "
        IrOperation('div'), kind='/'
          IrReg('r1')
          IrReg('r2')"
```

This example justifies to some extent the accuracy of implementation of the proposed technique, as well as the automating utility that is being developed.

V. CONCLUSION

This methods aims at solving the most important task of machine code vulnerability search, which is critical specifically for the telecommunication device, as long as the SW of such device affects the security the information transferred. Application of the available developments in the area of vulnerability search by the source code as combined with the previous research of the authors [11] are the specifics of implementation of this Method. Completion of implementation of a software tool to operate the Method automatically and approbate such Method using existing telecommunication device software will be the only next logical step and will allow to prove Method feasibility, in which case the theoretical value of the Method will be the development of the methodology of the highly popular machine code security area in terms of vulnerability search. Combination of the offered and previous original methods forms a methodological basis for search of vulnerabilities of all possible types in the machine code. Practical value of the Method is priceless, as long as identification of a vast number of machine code vulnerabilities for a tremendous number of existing telecommunication device firmware is foreseen upon automated application of such Method. Some of such vulnerabilities are still in action, which reduces significantly the overall security of information transferred via telecommunication networks.

ACKNOWLEDGMENT

In the end, we would like to express our gratitude to the Bonch-Bruevich Saint-Petersburg State University of Telecommunications for the possibility and support in development of the area of scientific studies, which was described in this and other articles.

The authors are thankful to their parents and Teachers for their craving for knowledge cultivated since their childhood and scientific achievements, which have been demonstrated multiple times using their own examples. First of all, to Evgeny Izrailov for development of source beams ultracold polarized H-atoms for metrology and physical investigations. Second of all, to Vladimir Rosenberg for developing the special mathematical management support methodology.

REFERENCES

[1] Cova. M, Felmetsger V., Banks G., and Vigna G., "Static Detection of Vulnerabilities in x86 Executables," in *Proc. 22nd Annu. Computer Security Applications Conference*, Miami Beach, 2006, pp. 269-278
[2] Buinevich M.V. and Izrailov K.E., “Architectural software vulnerabilities,” theses, *6th Congressional Research Undergraduate and Graduate Students “Engecon-2013”*, 2013
[3] Buinevich M.V. and Izrailov K.E., “Method and Utility for Recovering Code Algorithms of Telecommunication Devices for Vulnerability Search,” in *Proc. IEEE 16th Int. Conf. on Advanced Communications Technology*, PyeongChang, 2014, pp. 172-176
[4] Xin L. and Wandong C., “A program vulnerabilities detection frame by static code analysis and model checking,” in *Proc. IEEE 3rd Int. Conf. on Communication Software and Networks*, Xi'an, 2011, pp. 130-134
[5] Ivannikov V. P., Belevantsev A. A., Borodin A. E., Ignatiev V. N., Zhurikhin D. M., and A. I. Avetisyan, “Static analyzer Svace for finding of defects in program source code,” in *Proc. Institute for System Programming of Russian Academy of Sciences*, vol. 26, no. 1, pp. 231-250, 2014
[6] Rawat S. and Mounier L., “Finding Buffer Overflow Inducing Loops in Binary Executables,” in *Proc. IEEE 6th Int. Conf. on Software Security and Reliability*, Gaithersburg, 2012, pp. 177-186
[7] The IDA Pro website [Online]. Available: https://www.hex-rays.com/products/ida/
[8] The official YAML website [Online]. Available: http://yaml.org
[9] Shterenberg S.I. and Krasov A.V., “Variants of embedding information in the executable file with format .Intel HEX,” *Spb.: Information Technology and Telecommunications*, no. 4, pp. 52-64, 2013
[10] Izrailov K.E., “The internal representation of a prototype utility for the algorithmization of the code,” in *Proc. 2th Int. Scientific and Practical Conf. on Fundamental and Applied Research in the Modern World*, Saint Petersburg, 2013, pp. 79–90
[11] The research area and method of the authors [Online]. Available: http://www.demono.ru

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.
Dr. Prof. Buinevich, at the present time, is the professor of the Protected Communications System Chair of Saint Petersburg State University of Telecommunications (SPbSUT).

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 postgraduate student of the Protected Communications System Chair of Saint Petersburg State University of Telecommunications (SPbSUT). He has about 20 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 the Candidate of Sciences at Komsomolsk-on-Amur State Technical University, Russia in 2001. At present he is a head of the Scientific Work Organization and Researchers Training Administration of Bonch-Bruevich Saint-Petersburg State University of Telecommunications, Saint-Petersburg, Russia. His major interests include control systems, soft computing, communication networks, network security management.