

Improving Fuzzing Using Software Complexity Metrics

Maksim O. Shudrak^(✉) and Vyacheslav V. Zolotarev

IT Security Department, Siberian State Aerospace University,
Krasnoyarsky Rabochy Av. 31, 660014 Krasnoyarsk, Russia
mxmssh@gmail.com, amida@land.ru

Abstract. Vulnerable software represents a tremendous threat to modern information systems. Vulnerabilities in widespread applications may be used to spread malware, steal money and conduct target attacks. To address this problem, developers and researchers use different approaches of dynamic and static software analysis; one of these approaches is called fuzzing. Fuzzing is performed by generating and sending potentially malformed data to an application under test. Since first appearance in 1988, fuzzing has evolved a lot, but issues which addressed to effectiveness evaluation have not fully investigated until now.

In our research, we propose a novel approach of fuzzing effectiveness evaluation and improving, taking into account semantics of executed code along with a quantitative assessment. For this purpose, we use specific metrics of source code complexity assessment specially adapted to perform analysis of machine code. We conducted effectiveness evaluation of these metrics on 104 wide-spread applications with known vulnerabilities. As a result of these experiments, we were able to identify the best metrics that is more suitable to find bugs. In addition we proposed a set of open-source tools for improving fuzzing effectiveness. The experimental results of effectiveness assessment have shown viability of our approach and allowed to reduce time costs for fuzzing campaign by an average of 26–28 % for 5 well-known fuzzing systems.

Keywords: Fuzzing · Metrics · Complexity · Code coverage · Machine code

1 Introduction

Nowadays each software product should meet a number of conditions and requirements to be useful and successful on the market. Despite this fact, software engineers and developers keep making mistakes (bugs) during software development. In turn, these bugs can create favorable conditions for emergence of serious vulnerabilities. This is particularly relevant for network applications because vulnerabilities in this type of software create great opportunities for an attacker, such as remote code execution or DoS attack. However, practice has shown that vulnerabilities in local applications may also present a serious threat

to information systems if they allow to execute arbitrary code in the context of vulnerable application. This severely endangers commercial success of the product and can considerably decrease the security rate of infrastructure as well. Critical vulnerabilities in widespread products deserve special attention because they are often a target for mass malware attacks and persistent threats. Suffice it to say that in 2014, US National Vulnerability Database registered 26 new vulnerabilities per day on average [1].

There are two fundamentally different approaches for bugs detection in binary executables: static and dynamic analysis. Static analysis is aimed at finding bugs in applications without execution, while dynamic analysis performs bugs detection at runtime.

In our research, we consider only binary code of the program. Binary code (machine code, executable code) is a code (a set of instructions) executed directly by a CPU. The reason of this is due to the presence of proprietary software that is distributed in binary form only. The second problem related to transformations performed by compilers and optimizer tools that may significantly change actual behavior of the program in the binary form. This problem is called What You See Is Not What You eXecute [2].

In the paper, we will use technique of dynamic analysis called fuzzing. Fuzzing is performed by generating and sending potentially malformed data to an application. The first appearance of fuzzing in software testing dates back to 1988 by Professor Barton Miller [7]; since then the fuzzing has evolved a lot and it is now used for vulnerabilities detection and bugs finding in a large number of different applications. There are a lot of instruments for fuzzing, such as Sulley [3], Peach [4], SAGE [5] and many others. However, issues which addressed to effectiveness evaluation have not fully investigated until now.

Today researchers often use several basic criteria for effectiveness evaluation: the number of errors found, the number of executed instructions, basic blocks or syscalls as well as cyclomatic complexity or attack surface exposure [6–9].

During the last several decades, the theory of software reliability has proposed a wide range of different metrics to assess source code complexity and the probability of errors. The general idea of this assessment is that more complex code has more bugs. In this paper, our hypothesis is that source code complexity assessment metrics could be adapted to use them for binary code analysis. Thus it would allow to perform analysis based on semantics of executed instructions as well as their interaction with input data.

We will provide an overview of technique, architecture, implementation, and effectiveness evaluation of our approach. We will carry out separate tests to compare effectiveness of 25 complexity metrics on 104 wide-spread applications with known vulnerabilities. Moreover, we will perform assessment of our approach to reduce time costs of fuzzing campaigns for 5 different well-known fuzzers.

The purpose of this research was to increase effectiveness of the fuzzing technique in general, regardless of the specific solutions. Thus, we did not develop our own fuzzer, but focused on flexibility of our tools by making them easy to use with any fuzzers. Thus we did not try to improve test cases generation or

mutation to find more bugs but we try to make fuzzing campaign more efficient in terms of time costs required to detect bugs in software.

The contributions of this paper are the following:

1. We adapted a set of source code complexity metrics to perform fuzzing effectiveness evaluation by estimating complexity of executable code.
2. We conducted the comparative experimental evaluation of proposed metrics and identified the most appropriate ones to detect bugs in executable code.
3. We implemented a set of tools for executable code complexity evaluation and executable trace analysis. In addition, we also made our tools and experimental results accessible for everyone in support of open science [28].

The paper is structured as follows. In Sect. 2 we illustrate short overview of fuzzing and problems of its effectiveness evaluation. Section 3 covers details of metrics adaptation. Then, Sect. 4 provides an in-depth description of system implementation. Detailed results of metrics effectiveness evaluation and their comparison are presented in Sect. 5. Section 6 used to present experimental results of system integration with well-known fuzzers. Further, we outline related works in Sect. 7 and describe the direction of our future research in Sect. 8. Finally, we use Sect. 9 to present conclusions.

2 Problem Statement

In Sect. 1, we mentioned that fuzzing is performed by generating and sending potentially malformed data to an application. Nowadays, fuzzing is used for testing different types of input interfaces such as: network protocols [10], file formats [11], in-memory fuzzing [12], drivers and many others software and hardware products that process input data. Moreover, fuzzing is not limited to pseudorandom data generation or mutation, but includes a mature formal data description protocol and low-level analysis of binary code for generating data and monitoring results. However, the question still remains: *How can we evaluate fuzzing effectiveness?* Of course, we can assess it by the number of bugs detected in an application. But this is not a flexible approach, since it does not provide any information on how well the testing data was generated or mutated in case when the analysis showed no errors at all. On the other hand, for this purpose, we can use code coverage, assuming that the higher is code coverage, the more effective the testing. Code coverage is a measure used to describe the degree to which the code of a program is tested by a particular test suite. In most cases, researchers assess code coverage by calculating the total number of instructions, basic blocks or routines that have been executed in the application under test. However, they do not take into account the complexity of tested code. For example, different code paths may have equal values of code coverage but their complexity may be different. Let us consider the example in Fig. 1.

Listing A	Listing B
<pre> push eax push 0Ah lea eax, [ebp+Source] push eax call fgets add esp, 0Ch lea eax, [ebp+Source] push eax lea ecx, [ebp+Format] push ecx call strcpy add esp, 8 cmp [ebp+var_34], 0 jnz short loc_4135B6 </pre>	<pre> push eax push offset Format call scanf add esp, 8 mov eax, [ebp+b] imul eax, 6 add eax, 3 mov ecx, [ebp+b] imul ecx, 6 add ecx, 3 imul eax, ecx add eax, [ebp+a] mov [ebp+a], eax jnz short loc_4135AD </pre>

Fig. 1. Two different code blocks with equal code coverage measure

The code in Listing A handles user data and may contain buffer overflow, whereas the code in Listing B reads an integer and performs some calculations by using this value. Code coverage for these examples is the same, but the code in Listing A is more interesting for analysis.

Basili [13], Khoshgoftaar [14], Olague [15] and other researchers have shown that in general, increasing of code complexity leads to increase in the probability of an error. This contention is supported by experimental results [6–9].

In this paper, we propose to adapt source code complexity assessment metrics so as to take into account semantics of binary code. We propose the following hypothesis: *“There is a more effective complexity metric for fuzzing effectiveness assessment than the number of executed instructions, basic blocks and routines, as well as than cyclomatic complexity”*. Thus, we need to adapt complexity metrics for binary code and then perform analysis of their effectiveness in comparison with traditional metrics.

In our research, we consider the following types of errors: buffer and heap overflows, format string errors, read and write to invalid or incorrect memory address, null pointer dereferences, use after free, as well as use of uninitialized memory.

3 Metrics Adaptation

In the article, we adapted 25 metrics of source code complexity assessment. Without getting into description of each metric, let us describe symbols and references to the authors of each measure.

- Lines of code count (*LOC*), basic blocks count (*BBLs*), procedure calls count (*CALLS*);
- Jilb metric (*Jilb*) [16], ABC metric (*ABC*), Cyclomatic complexity (*CC*) [17], Modified cyclomatic complexity (*CC-mod*) [16], density of CFG (*R*) [18], Pivovarsky metric (*Pi*) [16], Halstead metrics for code volume (*H.V*), length and calculated length (*H.N*, *H.N**), difficulty (*H.D*), effort (*H.E*), the number of delivered bugs (*H.B*) [19];

- Harrison and Magel metric (*Harr*) [20], boundary values metric (*Bound*), span metric (*Span*), Henry and Cafura metric (*H&C*) [21], Card and Glass metric (*C&G*) [22], Oviedo metric (*Oviedo*) [23], Chapin metric (*Chapin*) [24];
- Cocol metric (*Cocol*) [16].

The detailed description of each adapted metric is also given in the Appendix A. Metrics that take into account high level information such as source code comments, name of variables or some object oriented information were excluded from the scope of this analysis.

It should be noted that for most of the metrics we need to perform conversion of routines code into control flow graph (CFG). CFG has only one entry and one exit. A path in the CFG can be represented as an ordered sequence of node numbers. In terms of binary code analysis, graph nodes are represented as a basic block of instructions and edges describe control flow transfer between basic blocks. Basic block (linear block) is a set of machine instructions without conditional or unconditional jumps excluding function calls. Algorithm 1 allows to perform such conversion.

Algorithm 1. Routine to CFG translation

Data: Address of the first instruction, an empty set of links

Result: A set of nodes, A set of edges

```

while not end of routine do
  Read instruction;
  if First instruction in the node then
    | Save instruction address as the first address of the node;
  Get links of the instruction;
  if Number of links > 0 then
    | Save instruction address as the last address of the node;
    | Save edges in a set of edges;
  | Move the pointer to the next instruction;

```

The algorithm passes through all basic blocks in the routine. A link is conditional or unconditional jump to some address within routine code. Note that the link is not considered for call instructions. Each instruction at some address may have from 0 up to n outgoing links. Unconditional jump always has two links, the first one refers to the address of unconditional jump, and the second one is the link to the address following immediately after jump instruction. Thus each node is associated with the following information: address of the head, address of the end, edge address 1 (optional) and edge address 2 (optional).

Note that bugs may arise from the use of unsafe library functions, such as *strcpy*, *strcat*, *lstrcat*, *memcpy* and etc. These functions are banned or not recommended to use, since they may cause overflows in the memory. Efficient

fuzzing campaign should take into account this fact and firstly cover the routines that call these functions. In the article, we propose to use the following experimental measure based on Halstead B metric (rationale for the choice of this metric is proposed in Sect. 5):

$$Exp = H.B \times \sum_{i=1}^n (v_i + 1) \quad (1)$$

n - a total number of banned or not recommended functions used in the routine. v_i is calculated as the total number of calls of banned or not recommended functions in the routine, multiplied by the coefficient of the potential danger associated with this syscall. This coefficient calculated by using the banned functions list proposed by Microsoft within their secure development lifecycle concept [25]. In our research, a function can take only two values: 0.5 for dangerous and 1 for banned syscalls. It should be noted that multiplication is used to prioritize routines that calls unsafe functions.

4 System Overview

4.1 Fuzzing Strategy

Let's describe all basic blocks in a program as an ordered set of nodes: $CFG = \{node_0, node_1, \dots, node_n\}$, where $node$ is a basic block and n - total number of basic blocks. Let's define an array of test data as $TD = [td_0, td_1, \dots, td_\nu]$, ν - an array size and td - one instance of test data (file, network packet, etc.) to make one fuzzing iteration. Then code coverage for one test iteration may be written as:

$$Cover = [cov_0, \dots, cov_\nu] \quad (2)$$

Then, let's assign weight for each test case and sort them in descending order of weight. Weights for test cases is assigned using complexity of trace which is calculated using metrics described above. Further we will send test cases according with their position in the sorted array.

In the case of adding new test data in TD without associated coverages, new instances take the highest priority with respect to existing elements, and passed to the program in random order before existing test cases.

4.2 Trace Analysis

As it was noted in the second section, we need to save addresses of instructions, basic blocks or routines to assess complexity of code that has been executed during analysis. In this research, we used technique called dynamic binary instrumentation to perform code coverage analysis. Dynamic Binary Instrumentation (DBI) is a technique of analyzing the behavior of a binary application at runtime through the injection of instrumentation code. The main advantage of DBI is the ability to perform binary code instrumentation without switching

the processor context, which significantly improves performance. In our research we use DBI framework called Pin [26]. Pin provides API to create the dynamic binary analysis tools called PinTools. Pin performs dynamic translation of each instruction and adds instrumentation code, if it required. Note that dynamic translator performs code translation without intermediate stages in the same architectures (IA32 to IA32, ARM to ARM and etc.).

4.3 Metrics Evaluation Module

Let us describe basic scheme of the tool for binary code complexity assessment in Fig. 2.

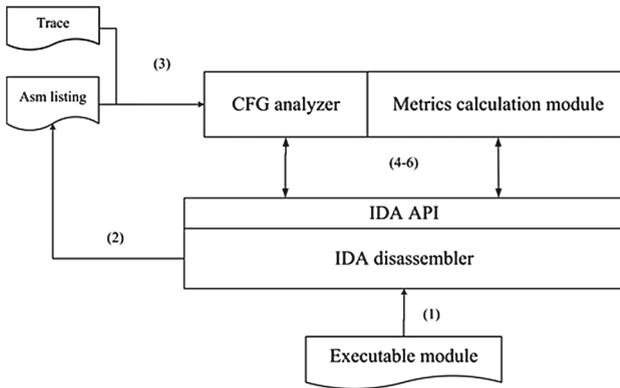


Fig. 2. Scheme of the tool for binary code complexity assessment

At the first stage, we use IDA disassembler to perform preliminary analysis and disassembling of executable module. Then assembler listing and trace is passed to the module of CFG analysis that sequentially iterates through each executed basic block in the program. The routine parser performs analysis of interconnections between basic blocks on the basis of which the tool builds graph of a routine. This graph is used in the module of metrics calculation that performs analysis and evaluation of each complexity measure for each required metric. Where necessary, this module also uses the binary code translation to get information required for some metrics. For example, the total number of assignments could be in turn obtained by using high level listing obtained by the translator, where operations like $eax = eax + 1$ may be considered as an assignment.

5 Metrics Effectiveness Evaluation

In Sect. 2, it was mentioned that we need to perform effectiveness comparison between adapted and traditional metrics. To meet this challenge, we decided to

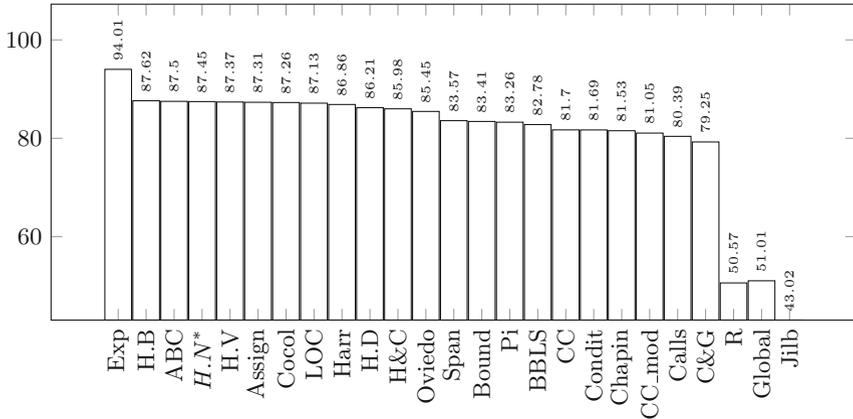


Fig. 3. Average effectiveness of each metric. Experimental metric demonstrates maximum effectiveness. Y - percent interval.

use open database with vulnerable applications called exploit-db supported by Offensive Security [27]. In our experiment, we randomly selected 104 different vulnerable applications. This is minimum sample size which is required to evaluate the effectiveness of all the metrics in the 95 % confidence interval within an error no more than 3%. As a result we randomly selected the following types of applications: video and audio players; FTP, HTTP, SMTP, IMAP and media servers; network tools; scientific applications; computer games; auxiliary tools (downloaders, torrent-clients, development tools and etc.); libraries (converters, data parsers and etc.); readers (PDF, DJVU, JPEG and etc.); archivers and etc. For details please visit [28].

Then exploit has been found for each program which allowed to locate vulnerable routine in the application. Each application was in turn analyzed by the tool of code complexity assessment. Complexity of each metric has been obtained for each routine in each vulnerable application. Then obtained measures were ranked in descending order. Lastly, we selected ranks of all vulnerable routines in each application (The results for each application may be found at [28]). Obviously that obtained results do not allow to assess and compare effectiveness of metrics, since they do not take into account total number of routines in the application. *An effective metric is a metric that takes a maximum complexity value for vulnerable routines.* Thus the following formula was used to solve this problem:

$$PR = \left(1 - \frac{frang}{TF}\right), \quad (3)$$

frang - a routine rank and *TF* - a total number of routines. This expression enables to answer the following question: *How many routines in a program have metric values less than for a vulnerable routine?* This value in percent may be obtained for each metric in each application. Now, it's possible to calculate average measures for each metric (Fig. 3).

According to Fig. 3, *Jilb*, *Global* and *R* metrics showed the lowest average values. Let's exclude these metrics from further analysis. Also, it makes sense to exclude *H.D*, *H.V* and *H.N** metrics, since they're used to calculate *H.B* and showed comparable results.

Let us compare metrics using coefficient of variation (Fig. 4). Coefficient of variation is used to show the extent of variability in relation to the mean of the value.

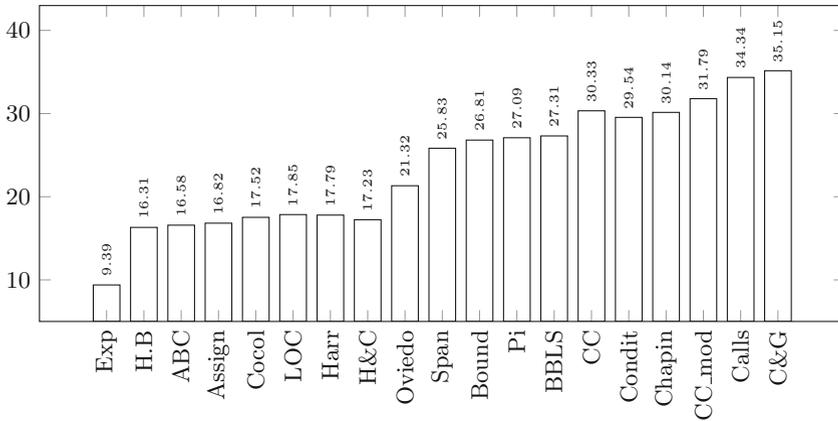


Fig. 4. Coefficients of variation for metrics (less is better). Y axis - coefficient of variation. Cyclomatic complexities, Chapin and Card & Glass demonstrate high level of variation.

The obtained statistical results have shown that experimental metric exceeds metrics based on cyclomatic complexity (by 12,31%) the number of basic blocks (by 11,23%), calls (by 13,62%), LOC (6.88%) and at the same time has the lowest coefficient of variation 9.4%. Note that the statistical error for the experimental metric is 2,54% at 95% confidence interval. Thus, all of these data prove that hypothesis proposed in the Sect. 2 is correct.

In Sect. 3 it was noted that the basis of experimental metric is Halstead B measure. We use this measure because Halstead B demonstrated the best effectiveness compared to other known metrics.

6 Experiments

6.1 Code Coverage Analysis

According to Sect. 5, the system is based on 2 modules: module of metrics calculation and module of trace analysis. Let's describe the general scheme of the system integration with fuzzer in Fig. 5.

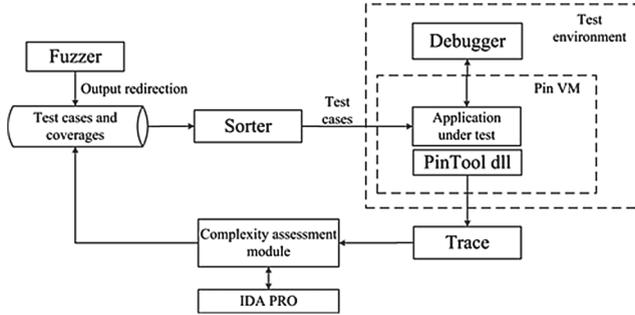


Fig. 5. General scheme of the system

The output of the fuzzer is redirected first in the database to perform test cases prioritization according to fuzzing strategy. Then the system starts fuzzing and executable code instrumentation. For each test case the system evaluates new code coverage using obtained trace. Calculated coverages are written in the database (to use them further) and results are visualized on the screen. It should be noted that the process of complexity evaluation is performed in parallel with fuzzing to increase performance of the system. The tools were developed taking into account support of several platforms, thus making them easy to port across different operating systems with minimal changes.

6.2 Experiments

For experimental analysis of proposed approach, it was decided to estimate time costs for fuzzing campaign before and after integration of our system with 5 well-known fuzzers. We randomly selected 14 popular applications with known bugs from exploit-db, so as to include each type of bug that is considered in the article (stratification technique was used). Also we added 4 randomly selected applications (2 for Linux and 2 for Windows) from exploit-db with two and more bugs in one application to analyze capability of the system reduce time costs for several bug detections. Each software product was deployed in the private virtual environments within the following configurations: Windows 7 x64 (Intel Core i7 2.4 GHz with 2 Gb RAM), Windows Server 2008 SP2 x64 (Intel Core i7 2.4 GHz with 4 Gb RAM), Ubuntu Linux 12.10 (Intel Core i7 2.4 GHz with 4 Gb RAM). Experimental results presented in Fig. 6.

Thus experimental results have shown that proposed system allowed to reduce time costs for testing by an average of 26–28% for any considered fuzzer. Detailed results may be found at [29].

7 Related Works

There are a lot of researches which performs fuzzing using some knowledge about testing application (white box fuzzing) to improve future tests generation, such as: symbolic execution or taint analysis [32–35]. Also, in several

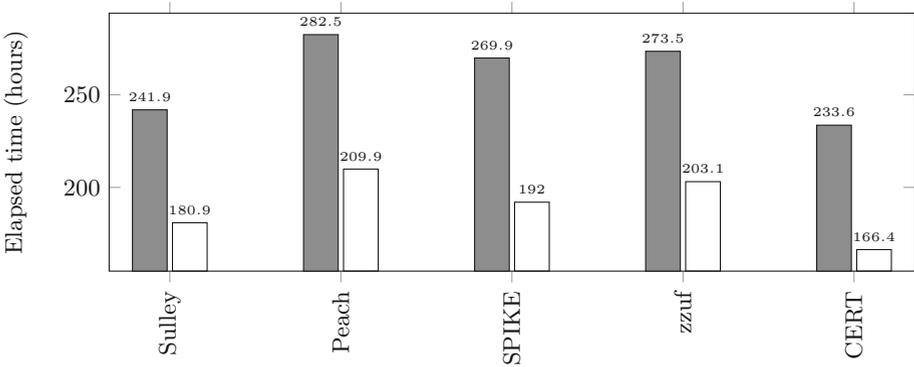


Fig. 6. The total time costs for fuzzing campaigns before and after integration of proposed system. The ordinate represents the total number of hours spent on testing all programs. White bar represents fuzzing campaign with proposed system. Zzuf [30], CERT fuzzer [31]

researches, authors try to use evolution algorithms [6, 36, 37] for effective data generation and increasing code coverage. Often, as an indicator of effectiveness is used the following metrics: the number of detected bugs, executed instructions, basic blocks and dangerous syscalls [6–9, 37–42]. Moreover, authors may apply special coverage criteria such as statements, decisions, and condition coverage [12, 38, 39]. In other case, researchers use input-based coverage criteria based on using input domain partitions and their boundary values [40].

In some way, our approach has certain features in common with this paper [37]. Authors used a set of variables based on disassembly attribute information and application for procedure, such as the number and size of function arguments and local variables, the number of assembler code lines, procedure frame stack size and also cyclomatic complexity. In [41], author uses cyclomatic complexity metric to perform in-memory fuzzing for more complex functions finding to increase a probability of bugs detection. In [12] author mentions about opportunity to apply cyclomatic complexity as a metric of effectiveness evaluation of the fuzzing technique. In [42] authors use basic blocks coverage to pick seed files to maximize the total number of bugs found during a fuzz campaign. In addition to coverage, they also consider other attributes, such as speed of execution, file size, etc. In [8] authors provide analysis of effective fuzzing strategies by using targeted taint driving fuzzing. Researchers used a different set of complexity metrics, such as cyclomatic complexity, attack surface exposure or static analysis for potentially vulnerable syscalls. The basic difference of our approach is that we use specially adapted metrics that take into account semantics of executed instructions as well as their interaction with input data.

8 Discussion and Future Work

While implementing the metrics evaluation module, we limited ourselves to only general-purpose x86 instructions. Thus, in future, the module should also support co-processor group of instructions as well as applications for x64 and ARM architectures. Also we did not consider obfuscated executables since analysis of obfuscated code is a separate direction of research.

Secondly, we plan to start using metrics to automatically improve the efficiency level of data generation. For example, it makes sense to perform in-memory fuzzing for routines that have the highest level of complexity. It is also possible to generate data using evolutionary algorithms, in which we could use our set of efficiency assessment metrics as parameters for the data fitness function to improve data generation. Certainly, this approach needs to be confirmed experimentally.

It should be noted that the limitation of our approach is the fact that to reduce time costs, we need to have coverages array for each test case even before fuzzing. However if we do not have such coverages, reducing of time costs is only achieved at the second fuzzing campaign. This is justified when the system is being integrated within existed secure development life cycle [25], when fuzzing is performed on the regular basis after new patch or functionality has been released. The system is also may be useful when existed set of test cases is applied for similar type of applications. Such fuzzing strategy makes sense, demonstrates positive results and is considered in this research [42].

9 Conclusion

In this article, we propose the novel approach to reduce time costs of fuzzing campaign. We adapted 25 source code complexity assessment metrics to perform analysis in binary code. Our experiments on the 104 vulnerable applications have shown that Halstead B metric demonstrates maximum effectiveness to find vulnerable routines in comparison with other metrics. We also proposed our own metric based on Halsted B which shows more stable results. The experimental results of effectiveness assessment have shown viability of our approach and allowed to reduce time costs for fuzzing campaign by an average of 26–28% for 5 well-known fuzzing systems. We have implemented our approach as a set of open-source tools that allows test cases prioritization, binary code complexity evaluation as well as performs code coverage analysis and results visualization.

This article is based upon work supported by the Russian Fund of Fundamental Research, research project 14-07-31350. This work was also supported by the research grant for young Russian scientists 14.Z56.15.6012-MK.

A Appendix: Adapted Metrics List

Metric	Symbol	Formula	Description
Halstead metric	H.V	$H.V = N \times \log_2 n$	Program volume $N = N_1 + N_2$ N_1 - the total number of operators. N_2 - the total number of operands. $n = n_1 + n_2$ n_1 - the total number of unique operators. n_2 - the total number of unique operands.
	$H.N^*$	$H.N^* = n_1 \times \log_2 n_1 + n_2 \times \log_2 n_2$	Calculated program length.
	$H.D$	$H.D = \frac{n_1}{2} \times \frac{N_2}{n_2}$	Program complexity.
Jilb's metric	$H.B.$	$H.B = \frac{E^{\frac{2}{3}}}{3000}$	The number of delivered bugs. $E = H.D \times H.B$
	Jilb	$cl = \frac{CL}{n}$	CL - the total number of condition operators (jmp, jxx, etc.). N - the total number of operators.
ABC metric	ABC	$ABC = \sqrt{A^2 + B^2 + C^2}$	A - assignments count. B - branches count. C - calls count.
Cyclomatic complexity	CC	$CC = e - v + 2$	e - the number of edges; v - the number of nodes (basic blocks).
Modified cycl. complex.	CC_mod	$CC_{mod} = e - v^* + 2$	v^* - the number of nodes (switch cases are considered as one node).
Pivovarskiy metric	Pi	$Pi = CC_{mod} + \sum_{i=0}^n p_i$	p_i - nesting level of predicate node i . n - the total number of predicate nodes.
Harrison & Magel metric	H&M	$H\&M = \sum_{i=0}^n c_i$	c_i - node complexity. n - the total number of predicate nodes.

Boundary values metric	Bound	$Bound = 1 - \frac{(n-1)}{S_a}$	<p>n - the total number of nodes. $S_a = \sum_i^n v_i$ - routine complexity $v_i = e_i - e_o$, e_i - the total number of input edges. e_o - the total number of output edges.</p>
Span metric	Span	$Span = \sum_{i=0}^n s_i$	<p>s_i - the number of statements containing the identifier. n - the total number of unique operators.</p>
Henry & Cafura metric	H&C	$H\&C = LOC \times (fan_{in} + fan_{out})^2$	<p>fan_{in} - the total number of input data flows. fan_{out} - the total number of output data flows.</p>
Card & Glass metric	C&G	$C\&G = S + D$	<p>$S = fan_{out}^2$, $D = \frac{v}{(fan_{out}+1)}$ v - the total number of input and output arguments.</p>
Oviedo metric	Oviedo	$Oviedo = \sum_{i=0}^n DEF(V_j)$	<p>$DEF(V_j)$ - a number of occurrences of variable V_j from $R(i)$ set. n - a set of variables which is used in $R(i)$. $R(i)$ - a set of local variables defined in a node i first time.</p>
Chapin metric	Chapin	$Chapin = P + 2M + 3C$	<p>P - the total number of output variables. M - the total number of local variables. C - the total number of variables which are used to manage CFG, such as: <i>cmp/test var</i> and then <i>jxx</i>.</p>
Cocol metric	Cocol	$Cocol = H.B + LOC + CC$	

References

1. NIST National Vulnerability Database. <http://nvd.nist.gov>
2. Balakrishnan, G., Reps, T., Melski, D., Teitelbaum, T.: WYSINWYX: what you see is not what you execute. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 202–213. Springer, Heidelberg (2008)
3. Sulley Fuzzing Framework. <http://code.google.com/p/sulley/>
4. Peach Fuzzing Framework. <http://peachfuzzer.com/>

5. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. *Queue* **10**(1), 20 (2012)
6. Miller, C.: Fuzz by number. In: *CanSecWest* (2008)
7. Woo, M., Cha, S.K., Gottlieb, S., Brumley, D.: Scheduling black-box mutational fuzzing. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pp. 511–522. ACM (2013)
8. Duran, D., Weston, D., Miller, M.: Targeted taint driven fuzzing using software metrics. In: *CanSecWest* (2011)
9. Weber, I.M.: Evaluation. In: Weber, I.M. (ed.) *Semantic Methods for Execution-level Business Process Modeling*. LNBP, vol. 40, pp. 203–225. Springer, Heidelberg (2009)
10. Banks, G., Cova, M., Felmetsger, V., Almeroth, K.C., Kemmerer, R.A., Vigna, G.: SNOOZE: toward a stateful NetwOrk prOtocol fuzZEr. In: Katsikas, S.K., López, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) *ISC 2006*. LNCS, vol. 4176, pp. 343–358. Springer, Heidelberg (2006)
11. Kim, H.C., Choi, Y.H., Lee, D.H.: Efficient file fuzz testing using automated analysis of binary file format. *J. Syst. Architect.* **57**(3), 259–268 (2011)
12. Takanen, A., Demott, J.D., Miller, C.: *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Norwood (2008)
13. Basili, V.R., Perricone, B.T.: Software errors and complexity: an empirical investigation. *Commun. ACM* **27**(1), 42–52 (1984)
14. Khoshgoftaar, T.M., Munson, J.C.: Predicting software development errors using software complexity metrics. *IEEE J. Sel. Areas Commun.* **8**(2), 253–261 (1990)
15. Olague, H.M., Etzkorn, L.H., Gholston, S., Quattlebaum, S.: Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Trans. Softw. Eng.* **33**(6), 402–419 (2007)
16. Abran, A.: *Software Metrics and Software Metrology*. Wiley-IEEE Computer Society, Hoboken (2010)
17. McCabe, T.J.: A complexity measure. *IEEE Trans. Softw. Eng.* **4**, 308–320 (1976)
18. Fenton, N.E., Ptleeger, S.L., Metrics, S.: *A Rigorous and Practical Approach*, 2nd edn, p. 647. International Thomson Computer Press, London (1997)
19. Halstead, M.H.: *Elements of Software Science*, p. 127. Elsevier North-Holland Inc., Amsterdam (1977)
20. Harrison, W.A., Magel, K.I.: A complexity measure based on nesting level. *ACM SIGPLAN Not.* **16**(3), 63–74 (1981)
21. Henry, S., Kafura, D.: Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.* **5**, 510–518 (1981)
22. Card, D., Glass, R.: *Measuring Software Design Quality*. Prentice Hall, Englewood Cliffs (1990)
23. Oviedo, E.I.: Control flow, data flow and program complexity. In: Shepperd, M. (ed.) *Software Engineering Metrics I*, pp. 52–65. McGraw-Hill, Inc., New York (1993)
24. Chapin, N.: An entropy metric for software maintainability. In: *Vol. II: Software Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, vol. 2, pp. 522–523. IEEE (1989)
25. Lifecycle, S.D.: List of banned syscalls. <https://msdn.microsoft.com/en-us/library/bb288454.aspx>
26. Intel Pin: A Dynamic Binary Instrumentation Tool. <http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>

27. Vulnerable applications, exploits database. <http://www.exploit-db.com/>
28. The set of tools, experimental results, the list of selected applications. <https://github.com/MShudrak/ida-metrics>
29. Detailed results of experiments for each application. <https://goo.gl/3dRMEx>
30. Zzuf fuzzer. <http://caca.zoy.org/wiki/zzuf>
31. CERT fuzzer. <https://www.cert.org/vulnerability-analysis/tools/bff.cfm?>
32. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software (2005)
33. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. *ACM SIGPLAN Not.* **43**(6), 206–215 (2008). ACM
34. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 317–331. IEEE (2010)
35. Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: IEEE 31st International Conference on Software Engineering, ICSE 2009, pp. 474–484. IEEE (2009)
36. Sparks, S., Embleton, S., Cunningham, R., Zou, C.: Automated vulnerability analysis: leveraging control flow for evolutionary input crafting. In: Twenty-Third Annual Computer Security Applications Conference, ACSAC 2007, pp. 477–486. IEEE (2007)
37. Seagle Jr., R.L.: A framework for file format fuzzing with genetic algorithms. Ph.D. thesis, University of Tennessee, Knoxville (2012)
38. Myers, G.J., Sandler, C., Badgett, T.: *The Art of Software Testing*. Wiley, Hoboken (2011)
39. Clarke, L.A., Podgurski, A., Richardson, D.J., Zeil, S.J.: A formal evaluation of data flow path selection criteria. *IEEE Trans. Softw. Eng.* **15**(11), 1318–1332 (1989)
40. Tsankov, P., Dashti, M.T., Basin, D.: Semi-valid input coverage for fuzz testing. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, pp. 56–66. ACM (2013)
41. Iozzo, V.: 0-knowledge fuzzing. <http://resources.sei.cmu.edu/assetfiles/WhitePaper/2010.019.001.53555.pdf>
42. Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Brumley, D.: Optimizing seed selection for fuzzing. In: Proceedings of the USENIX Security Symposium, pp. 861–875 (2014)