The Ghidra Metrics Toolkit for obfuscated native code

Jonathan Gobbo[1,*], Federica Sarro[2] and Paolo Falcarin[1]

[1]Ca' Foscari University of Venice, Venice, Italy
[2]University College London, London, UK

Abstract

Software Obfuscation is commonly used by companies to protect their software assets from reverse engineering and code analysis. While many types of obfuscations have been developed and combined, it is not clear how to assess the additional security brought by a particular set of obfuscation transformations. Some approaches rely on code complexity metrics to represent the difficulty in understanding code. Moreover many metrics are computed by tools on source code, which is different from the native code that the attacker will analyse: such metrics might not hold on to the binary code due to the variability introduced by compiler options and optimizations, and the underlying hardware. To this end, we developed GhidraMetricsToolkit, an open-source Ghidra plug-in to compute a collection of complexity metrics on native code. We tested the plug-in on a set of 61 obfuscated binaries generated from two programs using a set of 9 transformations provided by Tigress, each using four different configurations, from which we found that cyclomatic complexity is mostly influenced by EncodeArithmetic, Flatten and EncodeLiterals obfuscations, while entropy is affected by EncodeArithmetic, Inline and Split. These results show how our plug-in can help developers to choose among different obfuscations by assessing the complexity of code.

Keywords

Software Metrics, Binary code analysis, Obfuscation, Code complexity, Reverse engineering

1. Introduction

Software Obfuscation refers to the practice of transforming the code of a program in such a way that makes MATE (i.e. Man At The End) attacks [1] harder by slowing or thwarting manual and automated reverse-engineering efforts. This is achieved by applying to the code a sequence of obfuscating transformations that preserve the semantics of the original program. According to the taxonomy introduced by Collberg et al. [2], transformations can be classified in three categories: layout, control and data transformation. Layout obfuscations are the simplest transformations, like variable renaming or functions reordering. Control obfuscations alter the control flow of the program by inlining, flattening, splitting and merging different code portions [3], while Data obfuscations transform data structures, for example with string and constant encryption, splitting and merging data.

Obfuscators can also differ in the phase in which the obfuscation is applied: given the input source code of a program, source-to-source obfuscators like Tigress [4] output a new transformed source file, while source-to-binary obfuscators like Obfuscator-LLVM [5] directly output the obfuscated binary. Different measures have been defined to assess the quality of the obfuscation: *potency* measures how hard it is to reverse-engineer the code for humans, *resilience* estimates the difficulty of analysis with automated tools, while *stealth* measures how well hidden the obfuscation is across the code [6]. Software complexity metrics from the domain of software engineering have been used to evaluate the quality of Java obfuscations [7] as they can represent the perceived complexity of software from the reverse engineer's viewpoint, as some protections increase the manual effort needed to comprehend or reverse-engineer the code under analysis; other approaches relied on simulations [8], empirical experiments [9], and various analysis tools [10][11].

In this paper we introduce an open-source plug-in for the Ghidra reverse-engineering tool that provides a collection of complexity metrics that can be computed on native binary code, with no access

Joint National Conference on Cybersecurity (ITASEC & SERICS 2026), February 09-13, 2026, Cagliari, IT
*Corresponding author.
jonathan.gobbo@unive.it (J. Gobbo); f.sarro@ucl.ac.uk (F. Sarro); paolo.falcarin@unive.it (P. Falcarin)
0009-0002-3017-8638 (J. Gobbo); 0000-0002-9146-442X (F. Sarro); 0000-0003-1933-5348 (P. Falcarin)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-04-23

CEUR-WS.org/Vol-4198/paper58.pdf

to the original source code. We introduce each metric that was implemented and provide examples of the results they provide on obfuscated binary code. Then, we test the plug-in on a set of 61 obfuscated binaries generated using Tigress, and evaluate how each obfuscation affects the metrics. We identified EncodeArithmetic, Flatten and EncodeLiterals as the obfuscations that most increase the cyclomatic complexity and EncodeArithmetic, Inline and Split that most increase the entropy: a more thorough examination is reported in Section 4.

The paper is organized as follows: Section 2 introduces our Ghidra plug-in project, Section 3 introduces the background on complexity metrics and an example of the ones we have implemented; Section 4 describes the experiments, while Section 5 draws the conclusions.

2. Ghidra plug-in Project

We developed a plug-in for the Ghidra[1] reverse-engineering tool that aims to provide a collection of complexity metrics in a single package.

Figure 1 shows an example of the user interface of our plugin, featuring the whole-binary and per-function McCabe Cyclomatic Complexity values for an unobfuscated `factorial` program.

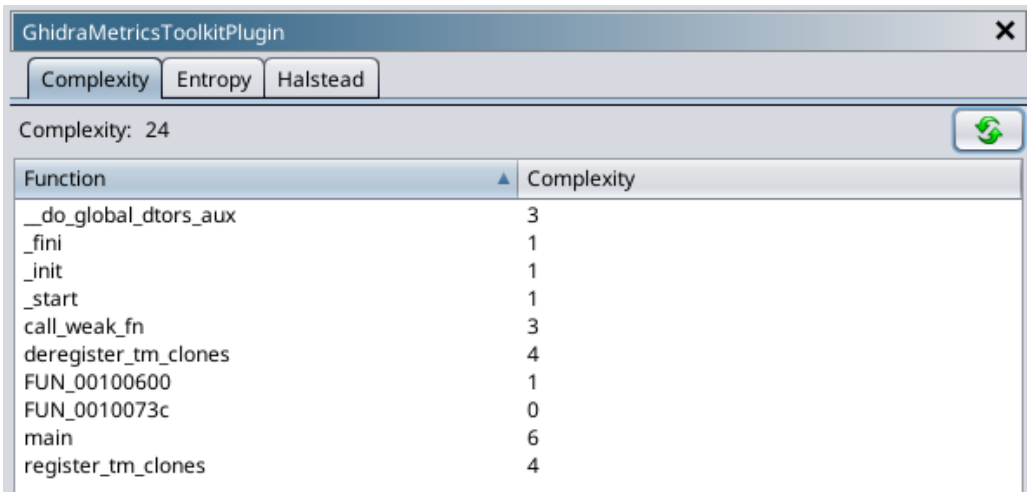

Figure 1: A screenshot of the tool, picturing the McCabe Cyclomatic Complexity metric

The plug-in leverages on Ghidra's APIs to navigate and process the structure of the disassembled binaries to allow the computation of the metrics without requiring access to the original source code of the analysed programs.

This is especially useful in scenarios where the source code is not available, for example when dealing with a source-to-binary obfuscator (e.g. Obfuscator-LLVM) which directly outputs an obfuscated binary without an obfuscated source code.

Even in the presence of the source code, the advantage of using binary-level metrics is that the results can potentially represent the actual program more faithfully, given the fact that the compilation process can alter the original structure of the code by means of optimization practices.

The functionality of the plug-in is provided both through a Graphical User Interface accessible from Ghidra's Codebrowser, and through the headless interface. This allows our Ghidra plug-in to be used in automated scenarios, to compute metrics on various programs and on many different versions.

The plug-in and its source code are open-source and have been published on Github[2].

Any subsequent example featured in Section 3 has been performed using a C implementation of the `factorial` algorithm from the TheAlgorithms[3] dataset, compiled in a clean and obfuscated version

[1] www.nsa.gov/ghidra
[2] https://github.com/UniVE-SSV/GhidraMetricsToolkit
[3] https://github.com/TheAlgorithms/C/blob/master/math/factorial.c

on an ARM64 platform. The obfuscated variant has been generated using Hikari-LLVM15[4], which is a source-to-binary LLVM-based obfuscator that evolved from the older Obfuscator-LLVM project by providing additional transformations on a more recent LLVM platform.

2.1. Related Work

Measuring the strength of software protections is an open research problem. The first categorization of different aspects of protection strength has been proposed by Collberg's metrics: resilience, stealth, and potency [2, 6]. While resilience measures the strength of a protection against automatic analysis tools, potency is the ratio between any complexity metric on an obfuscated program and the same metric on the original clean program; stealth describes how hidden a protection is and, thus, how hard it is to evaluate if the software under analysis has been obfuscated at all.

In literature, different complexity metrics have been used [12, 13, 14] for a generic evaluation of obfuscation potency, covering a wide range of protections and attacks.

Metrics such as the cyclomatic complexity [15] that were originally developed for assessing testability in the software engineering community, lend themselves to estimate the potency of protections, i.e., the capability of protections to increase the perceived complexity of code, and increase the manual effort necessary to understand, analyse, or reverse-engineer the code to be protected: recent experiments have shown that such metric can be indeed used as a proxy for assessing the attacker's reverse engineering effort [16].

Other works have tried to assess the effectiveness of Java obfuscations using source code metrics [17] or by means of controlled experiments with students [18, 9]. More recently, the quality of software protections has been assessed via qualitative analysis of reports of professional Penetration Testers [19, 20] on native code of three industrial use-cases with eight selected combination of transformations: while these works were the first approaches involving human subjects, they cannot provide a general way in assessing the quality of many combinations of obfuscation transformations.

Measuring code complexity has been shown to be helpful to assess how likely a successful attack is in a Man at the End scenario [16]. Additionally, complexity has applications in the malware detection problem [21] [22], [23] [24].

Table 1 presents a comparison between our solution and other binary analysis tools, with respect to the provided features. The angr[5] project, a binary analysis framework that provides both static and dynamic analyses [25], includes an API to compute the Cyclomatic Complexity of a function at binary level. Ghidra also natively provides an implementation of the cyclomatic complexity, but limited on a single function, as will be described in Section 3.1. The IDAmetrics[6] project is a plug-in for the commercial IDA Pro reverse-engineering tool that provides McCabe cyclomatic complexity and Halstead metrics computation, both at function and whole program level. While it doesn't compute entropy, it provides several more metrics, e.g. ABC metric, Oviedo metric [26], Henry and Kafura metric [27] and others. It should be noted that this project has not been updated since IDA Pro's 5.5 version.

Sonarqube[7] is a platform that provides static code analysis for quality and security purposes. Sonarqube can calculate the cyclomatic complexity of individual functions at source code level.

Finally, Frama-C[8] is a framework for the static analysis of C programs that can calculate cyclomatic complexity both for individual functions and for the entire program. Additionally, Frama-C can compute the Halstead metrics but exclusively for the whole program. Both the metrics are computed on source code level.

[4]https://github.com/61bcdefg/Hikari-LLVM15 – Last Accessed Feb 6, 2025
[5]https://angr.io/
[6]https://github.com/mxmssh/IDAmetrics
[7]https://www.sonarsource.com/products/sonarqube/
[8]https://frama-c.com

Project	Binary Level	C. Complexity		Entropy		Halstead	
		Whole	Function	Whole	Section	Whole	Function
GhidraMetricsToolkit	✓	✓	✓	✓	✓	✓	✓
angr	✓	–	✓	–	–	–	–
Ghidra	✓	–	✓	–	✓	–	–
IDAmetrics	✓	✓	✓	–	–	✓	✓
SonarQube	–	–	✓	–	–	–	–
Frama-C	–	✓	✓	–	–	✓	–

Table 1
Feature comparison between GhidraMetricsToolkit and related tools.

3. Complexity Metrics

Software complexity metrics try to quantify how difficult to understand some code is, and are often employed in the software engineering process as an indicator of low maintainability and understandability. Additional use cases include evaluating the quality of software obfuscation, or detecting instances of malware, encryption or obfuscation. The two most important complexity metrics are the McCabe Cyclomatic Complexity and the Halstead Metrics, which will be described in Sections 3.1 and 3.3, respectively. Other metrics include Program Length, Nesting Complexity, Fan-in/out complexity, as well as several other metrics that count some properties of the program, e.g. Instruction Count and Branch Count [28].

3.1. McCabe Cyclomatic Complexity

The McCabe Cyclomatic Complexity metric [29] measures the number of linearly independent paths in the Control Flow Graph of a function or program. As such, this metric can give an indication on how complex a program is, which can be helpful for developers to write maintainable code. This metric can also give an indication of the usage of obfuscation methods and could be employed to optimize it.

Given the Control Flow Graph of the function / program p, the complexity is computed as:

$$C(p) = E - V + 2 * P \tag{1}$$

with E being the number of edges in the graph, V the number of vertices and P the number of connected components.

While it is common to compute this metric on source code, we instead leverage Ghidra's API to compute it without needing access to the source code of the program. This provides the advantage of representing the structure of the program more closely, which might be affected by the compilation process. Additionally, this choice makes the metric compatible with any compiled program, regardless of the original language, as long as it is compiled for one of the hardware platforms supported by Ghidra.

Although Ghidra natively provides an implementation of the McCabe metric in the `CyclomaticComplexity` class, it is limited by the fact that it is computed per function, while in our case it could be useful to have an indication of the complexity of the entire binary.

This is especially useful when examining obfuscated binaries, where the obfuscating transformations might disrupt Ghidra's ability to assign some blocks of code to any of the detected functions. Obfuscated binaries infact tend to be much larger, often with a very complex control flow graph, that might hinder the reverse-engineering tools ability to analyze the code.

Our implementation, featured in Listing 1, relies on Ghidra's `BasicBlockModel` class to iterate over all basic blocks found in the program to build a directed graph representing the whole binary. Each block is added as a vertex in the graph, and for each of its destinations an edge between the two is

```
private int computeMcCabe(ConsoleTaskMonitor ctm, Program program) throws CancelledException {

    GDirectedGraph<String, GEdge<String>> graph = GraphFactory.createDirectedGraph();

    BasicBlockModel blockModel = new BasicBlockModel(program);
    CodeBlockIterator blockIter = blockModel.getCodeBlocks(ctm);

    for (CodeBlock block : blockIter) {

        graph.addVertex(block.getName());

        CodeBlockReferenceIterator destRefIter = block.getDestinations(ctm);
        while (destRefIter.hasNext()) {
            CodeBlockReference ref = destRefIter.next();
            if (ref.getFlowType().isCall()) {
                continue;
            }
            String destName = ref.getDestinationBlock().getName();

            graph.addEdge(new GEdge<>() {
                @Override
                public String getStart() {
                    return block.getName();
                }

                @Override
                public String getEnd() {
                    return destName;
                }
            });
        }
    }

    int nVertices = graph.getVertexCount();
    int nEdges = graph.getEdgeCount();
    int nComponents = countConnectedComponents(graph);

    return nEdges - nVertices + 2 * nComponents;
}
```

Listing 1: Implementation for the McCabe Cyclomatic Complexity metric

added. Edges representing function calls are filtered out as the complexity for a program is given by the sum of the complexity of each individual function.

Once the entire control flow graph is built and the number of vertices V, edges E and connected components C are extracted, the complexity value is computed as specified in Equation 1. Note that since the `GDirectedGraph` class does not provide a method to count the connected components, we implemented that algorithm separately.

Figure 2 shows the reconstructed control flow graph for the clean `factorial` program, compiled on an ARM64 platform with gcc.

To demonstrate how our implementation is better than Ghidra's, we devised an example experiment involving the measurement of an obfuscated program to demonstrate how the latter fails to capture the structure of the program. The program under investigation is again `factorial`, which has been obfuscated on an ARM64 platform using Hikari-LLVM15 with the default configuration (i.e. using the `-enable-allobf` flag), which enables all the available transformations (i.e. BogusControlFlow, BasicBlockSplit, ConstantEncryption, Flattening, FunctionWrapper, IndirectBranches, Substitution, StringEncryption) with the default parameters.

Table 2 lists the complexity values calculated for each function using the built-in Ghidra implemen-

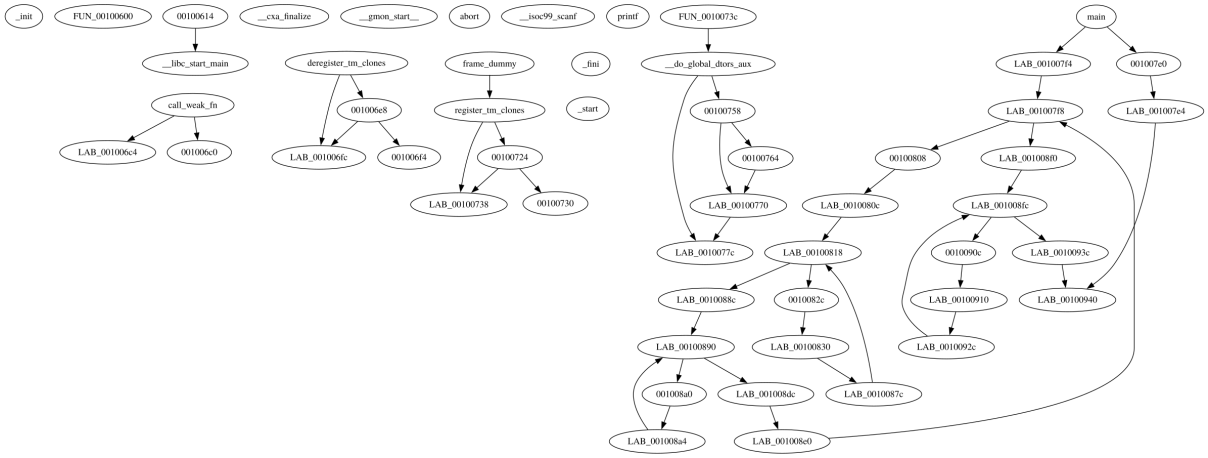

Figure 2: Reconstructed Control Flow Graph for the clean factorial program

Function	Complexity
`__do_global_dtors_aux`	3
`_fini`	1
`_init`	1
`_start`	1
`call_weak_fn`	3
`deregister_tm_clones`	4
`FUN_00106f00`	1
`FUN_0010703c`	0
`HikariFunctionWrapper`	1
`HikariFunctionWrapper.467`	1
`HikariFunctionWrapper.468`	1
`HikariFunctionWrapper.469`	1
`main`	0
`register_tm_clones`	4
Total	22

Table 2
Complexity values for the obfuscated factorial program as calculated by Ghidra's implementation

```
void main(void)
{
    /* WARNING: Could not recover jumptable
        at 0x00109020. Too many branches */
    /* WARNING: Treating indirect jump as call */
    (*(code *)PTR_LAB_00130370)();
    return;
}
```

Listing 2: The `main` function of the obfuscated factorial program decompiled by Ghidra

tation of the cyclomatic complexity. In particular, the `main` function amounts to a complexity of 0, because Ghidra fails to analyze the program correctly. This is also evident in the decompiled code for the `main` function, which we report in Listing 2.

Instead, our implementation, which does not rely on the concept of functions, yields a result of 505, much higher than the total of 22 listed in Table 2.

Our plug-in provides a whole-program complexity figure using our implementation as well as a list of the complexity values of each function using the native implementation.

Section	Clean Entropy	Obf. Entropy
`_elfSectionHeaders`	1.43	1.53
`.comment`	5.14	5.14
`.data`	**0.67**	**3.61**
`.dynamic`	1.51	1.56
`.dynstr`	4.65	4.65
`.dynsym`	0.79	0.78
`.eh_frame`	3.71	3.78
`.eh_frame_hdr`	3.06	3.29
`.fini`	3.75	3.75
`.fini_array`	1.55	1.55
`.gnu.hash`	0.49	0.49
`.gnu.version`	1.46	1.46
`.gnu.version_r`	2.18	2.18
`.got`	2.03	2.07
`.got.plt`	1.27	1.32
`.init`	3.91	3.91
`.init_array`	1.55	1.55
`.interp`	4.21	4.21
`.note.ABI-tag`	1.56	1.56
`.plt`	4.38	4.49
`.rela.dyn`	1.82	2.45
`.rela.plt`	1.28	1.34
`.rodata`	4.40	1.50
`.shstrtab`	4.24	4.24
`.strtab`	4.88	5.00
`.symtab`	1.74	1.88
`.text`	**5.62**	**6.38**
`segment_2.1`	1.84	1.91
Overall	**0.56**	**3.45**

Table 3
Whole-program and per-section entropy comparison between the clean and obfuscated factorial program

3.2. Entropy

Shannon Entropy [30] measures the randomness of a byte sequence. This measure is often used in malware analysis to identify obfuscated or encrypted sections in a program [31][32], as these are characterized by a higher degree of randomness.

Shannon Entropy is defined as:

$$H(X) = -\sum_{x \in X} P(x) \log_2 P(x) \tag{2}$$

with $P(x)$ being the probability of the outcome x for the random variable X.

The equation simply amounts to iterating over the byte sequence of the program, building a normalized frequency histogram of every state a byte can have. Then each element in the histogram is multiplied by its logarithm in the specified base and all the resulting values are accumulated in a negative sum.

The plug-in provides both a figure for the whole binary entropy and for each program section. The list of sections and their byte content is retrieved through the `Memory` class, filtering out external and uninitialized ones, and the entropy for each one is calculated as stated in Equation 2.

Table 3 lists the entropy value for each section in the original and obfuscated `factorial` programs, as well as the overall binary values. While most sections share the same value across the two programs, we underline an increase in the obfuscated variant for the `.data` and `.text` sections, which respectively

	Clean program	Obf. Program
Dist. Operators (n1)	77	92
Dist. Operands (n2)	133	1244
Tot. Operators (N1)	192	2116
Tot. Operands (N2)	443	5412
Program Vocabulary (n)	210	1336
Program Length (N)	635	7528
Estimated Length (∼N)	1420.90	13389.45
Volume (V)	4898.55	78168.53
Difficulty (D)	128.24	200.12
Effort (E)	6.28E+05	1.56E+07
Time to Program (T)	34898.56 s	869069.80 s
Delivered Bugs (B)	2.44	20.85

Table 4
Halstead metrics for the clean and obfuscated factorial programs

contain global variables and the executable code, from 0.67 to 3.61, and 5.62 to 6.38. These increases show how this metric can aid in detecting encryption of data and obfuscation of code.

3.3. Halstead Metrics

Halstead Metrics [33] are a collection of metrics that provide some insights on the quality of code. All the metrics can be computed by extracting the following features from the code:

- n_1: n. of distinct operators
- n_2: n. of distinct operands
- N_1: n. of total number of operators
- N_2: n. of total number of operands

The number of operators and operands is usually computed on the source code of a program but our plug-in computes them instead on the disassembly listing. For example, the instruction `MOV RBP, RSP`, is composed by the operator `MOV` and the two operands `RBP`, `RSP`. As such, the results may differ if the metrics were computed on the source code.

Once these figures are extracted, the metrics are computed as follows:

- Vocabulary: $n = n_1 + n_2$
- Program length: $N = N_1 + N_2$
- Estimated program length: $\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$
- Volume: $V = N * \log_2 n$
- Difficulty: $D = \frac{n_1}{2} * \frac{N_2}{n_2}$
- Effort: $E = D * V$
- Time to program: $T = \frac{E}{18}$
- Delivered bugs: $B = \frac{E^{\frac{2}{3}}}{3000}$

The plug-in provides all the measures both for the whole program, computed by summing the operators/operands of each function, and for the currently highlighted function in the code listing.

Table 4 contains once again a comparison between the Halstead metrics for the clean and obfuscated `factorial` programs, showing how obfuscation can impact the code.

4. Experiments

4.1. Experiment Setup

To evaluate the capabilities of our plug-in we set up an experiment involving 2 C programs [34], `hash_blake2b` and `naval_battle`, which were obfuscated using the Tigress (version 4.0.11) obfuscator with multiple obfuscation scripts.

Tigress is a source-to-source C obfuscator, meaning that given a source file, it returns an obfuscated source file. Tigress is also defined as a whole-program transformer, therefore it can only obfuscate programs that are defined in a single source file.

We selected 9 transformations among the ones provided by Tigress, excluding most notably the dynamic ones (i.e. Virtualize, Jit, JitDynamic, SelfModify), and obfuscated each program by enabling one transformation at a time. Additionally, for each transformation we defined 4 configurations which vary according to the values of each parameter related to the transformation. In particular, we defined the following configurations:

- *Default*: uses the default values for each parameter, as defined by Tigress;
- *Minimum*: boolean parameters are set to False, multiple choice parameters are set to pick the first value as defined by Tigress, and integers are set to a low value, which in most cases is 1;
- *Average*: boolean parameters are set to True, multiple choice parameters are set to the first half of the choices, while integers are set to the average value between the *Minimum* and *Maximum* configurations;
- *Maximum*: boolean parameters are set to True, multiple choice parameters are set to pick all the available choices and integers are set to some arbitrary values we picked around the default values.

The list of transformations, as well as the values used for each parameter, is detailed in Table 5.

Additionally, for select obfuscations and parameters we added some initialization transformations with the parameters set to their default values or matching the value of a related obfuscation parameter, when required by Tigress. For example, some obfuscations may require initializing a source of randomness using the `InitEntropy` transformation, while other obfuscations that employ Opaque Predicates, e.g. `AddOpaque`, require adding the `InitOpaque` transformation.

As Tigress requires each obfuscation to have one or more target functions, we specify the target as `--Target=100%` so that all functions are selected, instead of specifying their name manually. Additionally, we explicitly exclude the `main` function from the targets with the `--Exclude=main` parameters as it cannot be the target of obfuscations, as required by Tigress.

Of the 72 expected obfuscated source codes, only 63 were successfully produced, with the rest either failing or stuck in a loop. To address the latter case, we set a timeout of 5 minutes before forcefully terminating the obfuscation process.

Then, the obfuscated source files were compiled using GCC 15.2.1, along with their corresponding clean programs, yielding a total of 61 binaries and two compilation failures.

Finally, we load all the binaries into a Ghidra (version 11.2.1) project using the headless interface and compute all the metrics implemented in our plug-in on each executable and report the results in the following subsection.

All experiments were conducted on an x86-64 machine running Fedora Workstation 43 with an AMD Ryzen 7 5700X CPU and 16GiB of Ram.

4.2. Results

Table 6 lists the results obtained by computing the complexity metrics over all the binaries.

There are several instances of configurations achieving identical or similar metrics in the context of the same transformation, e.g. all the complexity metrics are identical for the *Min*, *Avg* and *Max* Inline

Parameter	Type	Def.	Min.	Avg.	Max.
AddOpaque					
Count	INT	1	1	5	10
Kinds	LIST	call,bug,true,junk	call	call,bug,true	call,bug,true,junk,fake_call,question
Obfuscate	BOOL	True	False	True	True
SplitBasicBlocks	BOOL	False	False	True	True
Inline	BOOL	True	False	True	True
SplitKinds	LIST	top,block,deep,recursive	top	top,block,deep	top,block,deep,recursive,level,inside
SplitLevel	INT	1	1	5	10
Structs	LIST	list,array	list	list	list,array,env
AntiAliasAnalysis					
ObfuscateIndex	BOOL	True	False	True	True
BogusEntries	BOOL	True	False	True	True
AntiBranchAnalysis					
Kinds	LIST	branchFuns	branchFuns	branchFuns,goto2call	branchFuns,goto2call,goto2push,goto2nopSled
OpaqueStructs	LIST	list,array	list	list	list,array,env
ObfuscateBranchFunCall	BOOL	False	False	True	True
BranchFunFlatten	BOOL	False	False	True	True
BranchFunAddressOffset	INT	8	4	10	16
EncodeArithmetic					
Kinds	LIST	builtin	builtin	builtin	builtin,plugins
MaxLevel	INT	1	1	4	8
MaxTransforms	INT	1	1	4	8
RepeatTimes	INT	1	1	4	8
MaxSplit	INT	1	1	4	8
AddImplicitFlow	BOOL	False	False	True	True
AddOpaques	BOOL	False	False	True	True
EncodeLiterals					
Kinds	LIST	integer,string	integer	integer	integer,string
MaxLevel	INT	100	10	505	1000
MaxTransforms	INT	100	10	505	1000
IntegerKinds	LIST	opaque	opaque	opaque	opaque
Flatten					
SplitBasicBlocks	BOOL	False	False	True	True
RandomizeBlocks	BOOL	False	False	True	True
Dispatch	LIST	switch	switch	switch, goto	switch,goto,indirect,call,concurrent
NumberOfThreads	INT	1	1	5	10
ConditionalKinds	LIST	branch	branch	branch	branch,compute,flag
ObfuscateNext	BOOL	False	False	True	True
OpaqueStructs	LIST	array	list	list	list, array
ImplicitFlowNext	BOOL	False	False	True	True
Inline					
OptimizeKinds	LIST	–	constProp	constProp,copyProp	constProp,copyProp,mergeLocals,gotos
Merge					
ObfuscateSelect	BOOL	True	False	True	True
OpaqueStructs	LIST	array	list	list	list,array
Flatten	BOOL	True	False	True	True
FlattenDispatch	LIST	switch	switch	switch	switch
SplitBasicBlocks	BOOL	False	False	True	True
RandomizeBlocks	BOOL	False	False	True	True
ConditionalKinds	LIST	branch	branch	branch	branch
Split					
Kinds	LIST	top,block,deep,recursive	top	top,block	top,block,recursive,level,inside
Count	INT	1	1	5	10
Level	INT	1	1	5	10
LocalsAsFormals	BOOL	True	False	True	True

Table 5
List of the Tigress transformations and their parameter values used for the experiment

configurations. This can be explained by the fact that some parameters can have little to no impact to the properties of the code captured by the metrics.

The most impactful obfuscation on the `hash_blake2b` program, according to the cyclomatic complexity metric, appears to be the Encode Arithmetic *Avg* configuration, achieving a score of 2999. The score is especially high when compared to other two Encode Arithmetic configurations, which only scored 71: this behaviour is justified by the recursive nature of the obfuscation, defined by the MaxLevel, MaxTransforms, RepeatTimes and MaxSplit parameters, which were all set to 4. Considering this, the corresponding *Maximum* configuration would theoretically have achieved much higher complexity with the parameters set to 8, but its obfuscation process was cancelled due to the timeout we set. The same

Program	Transformation	Conf.	Compl.	Entropy	Halstead											
					Operators		Operands		Vocab.	Length	Est. Len.	Volume	Difficulty	Effort	Time	Bugs
					Dist. (n_1)	Tot. (N_1)	Dist. (n_2)	Tot. (N_2)	(n)	(N)	($\hat{N}$)	(V)	(D)	(E)	(T)	(B)
hash_blake2b	Clean	N/A	52	4.02	166	1119	436	2476	602	3595	5047.18	33194.86	471.35	1.56E+07	8.69E+05	20.85
	Add Opaque	DEF	77	4.73	217	2068	1395	4792	1612	6860	16256.50	73090.80	372.71	2.72E+07	1.51E+06	30.18
		MIN	82	4.77	215	2104	1409	4874	1624	6978	16404.64	74422.71	371.86	2.77E+07	1.54E+06	30.50
		AVG	131	4.63	247	2591	1494	5831	1741	8422	17717.42	90668.73	482.01	4.37E+07	2.43E+06	41.36
		MAX	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	Anti Alias Analysis	DEF	88	4.86	209	2192	1420	5022	1629	7214	16480.62	76971.73	369.58	2.84E+07	1.58E+06	31.06
		MIN	71	4.72	197	2035	1364	4743	1561	6778	15705.73	71902.75	342.51	2.46E+07	1.37E+06	28.22
		AVG	88	4.86	209	2192	1420	5022	1629	7214	16480.62	76971.73	369.58	2.84E+07	1.58E+06	31.06
		MAX	88	4.86	209	2192	1420	5022	1629	7214	16480.62	76971.73	369.58	2.84E+07	1.58E+06	31.06
	Anti Branch Analysis	DEF	72	4.65	202	1985	1348	4642	1550	6627	15561.58	70233.29	347.81	2.44E+07	1.36E+06	28.06
		MIN	72	4.65	202	1985	1348	4642	1550	6627	15561.58	70233.29	347.81	2.44E+07	1.36E+06	28.06
		AVG	–	–	–	–	–	–	–	–	–	–	–	–	–	–
		MAX	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	Encode Arithmetic	DEF	71	4.62	234	2710	1365	6263	1599	8973	16057.71	95499.23	536.83	5.13E+07	2.85E+06	46.00
		MIN	71	4.62	234	2710	1365	6263	1599	8973	16057.71	95499.23	536.83	5.13E+07	2.85E+06	46.00
		AVG	2999	5.49	273	186884	14396	431964	14669	618848	201066.75	8565155.16	4095.80	3.51E+10	1.95E+09	3572.13
		MAX	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	Encode Literals	DEF	589	5.06	267	8718	2511	17653	2778	26371	30511.56	301679.78	938.54	2.83E+08	1.57E+07	143.73
		MIN	564	5.12	256	7012	2420	13474	2676	20486	29250.72	233250.78	712.67	1.66E+08	9.24E+06	100.78
		AVG	564	5.12	256	7012	2420	13474	2676	20486	29250.72	233250.78	712.67	1.66E+08	9.24E+06	100.78
		MAX	589	5.06	267	8718	2511	17653	2778	26371	30511.56	301679.78	938.54	2.83E+08	1.57E+07	143.73
	Flatten	DEF	294	4.86	224	2714	1614	6078	1838	8792	18948.32	95339.75	421.77	4.02E+07	2.23E+06	39.12
		MIN	295	4.87	229	2739	1587	6138	1816	8877	18668.30	96107.27	442.85	4.26E+07	2.36E+06	40.63
		AVG	1983	5.31	262	10814	3232	21562	3494	32376	39784.10	381087.01	873.95	3.33E+08	1.85E+07	160.16
		MAX	596	5.40	911	6164	3100	13262	4011	19426	44910.28	232524.29	1948.66	4.53E+08	2.52E+07	196.64
	Inline	DEF	143	5.08	74	14566	1896	35817	1970	50383	21104.56	551390.54	698.96	3.85E+08	2.14E+07	176.53
		MIN	129	5.06	74	12292	1875	29175	1949	41467	20845.76	453172.87	575.72	2.61E+08	1.45E+07	136.10
		AVG	129	5.06	74	12300	1875	29195	1949	41495	20845.76	453478.87	576.11	2.61E+08	1.45E+07	136.22
		MAX	127	5.02	74	12102	1258	28899	1332	41001	13413.02	425564.89	849.97	3.62E+08	2.01E+07	169.22
	Merge	DEF	76	4.73	92	2648	1221	6006	1313	8654	13120.12	89643.77	226.27	2.03E+07	1.13E+06	24.79
		MIN	77	4.82	92	2786	1114	6356	1206	9142	11875.56	93577.64	262.46	2.46E+07	1.36E+06	28.16
		AVG	77	4.82	92	2786	1114	6356	1206	9142	11875.56	93577.64	262.46	2.46E+07	1.36E+06	28.16
		MAX	77	4.82	92	2786	1114	6356	1206	9142	11875.56	93577.64	262.46	2.46E+07	1.36E+06	28.16
	Split	DEF	295	5.34	1452	4959	3050	10030	4502	14989	50554.07	181911.76	2387.47	4.34E+08	2.41E+07	191.17
		MIN	68	4.95	279	2355	1499	5509	1778	7864	18080.76	84900.06	512.68	4.35E+07	2.42E+06	41.25
		AVG	322	5.24	1650	5346	3341	10706	4991	16052	56745.57	197200.64	2643.65	5.21E+08	2.90E+07	215.92
		MAX	381	5.40	2060	6276	4048	12470	6108	18746	71184.52	235758.78	3172.95	7.48E+08	4.16E+07	274.68
naval_battle	Clean	N/A	228	4.73	209	2572	671	5547	880	8119	7911.64	79414.86	863.88	6.86E+07	3.81E+06	55.86
	Add Opaque	DEF	286	5.16	268	3967	1858	9210	2126	13177	22338.73	145657.58	664.23	9.68E+07	5.38E+06	70.25
		MIN	283	5.15	263	3960	1860	9196	2123	13156	22315.86	145398.65	650.15	9.45E+07	5.25E+06	69.17
		AVG	344	5.01	287	4442	1950	10129	2237	14571	23655.38	162136.61	745.39	1.21E+08	6.71E+06	81.48
		MAX	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	Anti Alias Analysis	DEF	329	5.05	249	4516	1973	10212	2222	14728	23578.84	163740.65	644.40	1.06E+08	5.86E+06	74.43
		MIN	269	5.16	241	3967	1804	9226	2045	13193	21420.84	145095.10	616.26	8.94E+07	4.97E+06	66.65
		AVG	329	5.05	249	4516	1973	10212	2222	14728	23578.84	163740.65	644.40	1.06E+08	5.86E+06	74.43
		MAX	329	5.05	249	4516	1973	10212	2222	14728	23578.84	163740.65	644.40	1.06E+08	5.86E+06	74.43
	Anti Branch Analysis	DEF	268	5.09	248	3841	1799	8984	2047	12825	21425.19	141065.96	619.24	8.74E+07	4.85E+06	65.62
		MIN	268	5.09	248	3841	1799	8984	2047	12825	21425.19	141065.96	619.24	8.74E+07	4.85E+06	65.62
		AVG	–	–	–	–	–	–	–	–	–	–	–	–	–	–
		MAX	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	Encode Arithmetic	DEF	269	5.47	325	6886	1875	15846	2200	22732	23098.16	252399.94	1373.32	3.47E+08	1.93E+07	164.48
		MIN	269	5.47	325	6886	1875	15846	2200	22732	23098.16	252399.94	1373.32	3.47E+08	1.93E+07	164.48
		AVG	–	–	–	–	–	–	–	–	–	–	–	–	–	–
		MAX	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	Encode Literals	DEF	1320	5.34	314	17558	4015	35182	4329	52740	50668.82	637089.61	1375.73	8.76E+08	4.87E+07	305.28
		MIN	1241	5.27	303	14469	3832	27731	4135	42200	48113.36	506976.94	1096.36	5.56E+08	3.09E+07	225.34
		AVG	1241	5.27	303	14469	3832	27731	4135	42200	48113.36	506976.94	1096.36	5.56E+08	3.09E+07	225.34
		MAX	1320	5.33	314	17558	4015	35182	4329	52740	50668.82	637089.61	1375.73	8.76E+08	4.87E+07	305.28
	Flatten	DEF	1268	5.24	267	7097	2789	15554	3056	22651	34073.80	262240.34	744.52	1.95E+08	1.08E+07	112.18
		MIN	1271	5.25	272	7123	2754	15617	3026	22740	33670.61	262947.08	771.21	2.03E+08	1.13E+07	115.06
		AVG	1116	5.28	304	6374	1662	12504	1966	18878	20288.62	206545.10	1143.57	2.36E+08	1.31E+07	127.37
		MAX	1720	5.81	2916	22357	9357	47789	12273	70146	156998.46	952807.17	7446.44	7.10E+09	3.94E+08	1230.78
	Inline	DEF	1223	5.56	73	17680	8265	42949	8338	60629	108002.64	789722.17	189.67	1.50E+08	8.32E+06	94.01
		MIN	717	5.41	73	13372	7581	34069	7654	47441	98157.09	612083.70	164.03	1.00E+08	5.58E+06	72.01
		AVG	–	–	–	–	–	–	–	–	–	–	–	–	–	–
		MAX	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	Merge	DEF	272	5.01	102	5262	1619	11993	1721	17255	17940.56	185474.54	377.79	7.01E+07	3.89E+06	56.65
		MIN	275	5.01	102	5284	1575	12049	1677	17333	17408.88	185665.32	390.16	7.24E+07	4.02E+06	57.92
		AVG	275	5.01	102	5284	1575	12049	1677	17333	17408.88	185665.32	390.16	7.24E+07	4.02E+06	57.92
		MAX	275	5.01	102	5284	1575	12049	1677	17333	17408.88	185665.32	390.16	7.24E+07	4.02E+06	57.92
	Split	DEF	273	5.14	412	5866	1224	14260	1636	20126	16133.88	214864.31	2399.97	5.16E+08	2.86E+07	214.35
		MIN	264	4.81	341	7207	1218	18359	1559	25566	15353.91	271163.36	2569.96	6.97E+08	3.87E+07	262.01
		AVG	297	5.44	597	6235	1509	14937	2106	21172	21439.39	233745.01	2954.73	6.91E+08	3.84E+07	260.45
		MAX	372	5.32	1157	7820	2775	18042	3932	25862	43515.04	308819.37	3761.19	1.16E+09	6.45E+07	368.33

Table 6
Results for the complexity metrics computed on all obfuscated variants of the two programs

configuration also achieved the highest Entropy score of 5.49, as well as by far the highest Halstead metrics across all other obfuscations.

The second most effective obfuscation on the first program is the Flatten *Avg*, scoring a cyclomatic complexity of 1983. Interestingly, this value is lower than the corresponding *Max* configuration, which achieved a score of 596. A possible explanation is given by the difference of the Dispatch and ConditionalKinds methods used between the two configurations: the additional strategies for the two

parameters used in the maximum configuration, as listed in Table 5, might not reflect on the control flow graph of the program as much as the medium configuration does, thus reducing the cyclomatic score.

Regarding the entropy, the Flatten *Max* and Split *Max* configurations both achieved a score of 5.40, slightly smaller than that of Encode Arithmetic *Avg*. The Halstead metrics make for a harder comparison, considering that each configuration might excel in only a subset of metrics, according to their usage of operators and operands (e.g. the Inline obfuscation, no matter the configuration, achieves a higher Volume score than the Split obfuscations, which in turn got higher scores for the Estimated Length metric).

As for the `naval_battle` program, the EncodeArithmetics *Avg* configuration, which excelled in the previous program, did not survive the obfuscation process, just like the *Max* configuration. Instead, the highest cyclomatic complexity score is achieved by the Flatten *Max* configuration, which also excels in the Entropy and Halstead metrics. In general, EncodeLiterals and Flatten obfuscations achieved higher complexity than the other obfuscations, regardless of the specific configurations, except for the Inline *Def*, which scored similarly.

The second most higher entropy score is achieved by the two Encode Arithmetic configurations, both with a value of 5.47. Again, when dealing with the Halstead metrics, different obfuscations will affect each metric with a different strength.

These results show how our plug-in can assist developers in assessing the effect of the chosen obfuscations, providing helpful insights to select the most effective configuration.

5. Conclusions

In this paper we introduced GhidraMetricsToolkit, a Ghidra plug-in that provides three complexity metrics that can be computed on binary code, namely the McCabe Cyclomatic Complexity, Shannon Entropy and Halstead metrics.

We extended the McCabe complexity metric which is usually defined on a single function to instead allow the computation over the entire call graph of the program. Our approach is more resilient to situations where Ghidra fails to correctly assign some basic blocks to functions, which can happen in presence of obfuscation.

We tested our plug-in on 61 obfuscated binaries, generated from 2 C programs using 9 obfuscations, with 4 configurations each. We found that generally EncodeArithmetic, Flatten and EncodeLiterals obfuscations increase the cyclomatic complexity the most across the two tested programs, while entropy is maximized by EncodeArithmetic, Inline and Split. These results show how our plugin-in can provide insights into the effects of the different obfuscations.

We plan to extend our tool by implementing additional complexity metrics, e.g. the ones featured in IDAmetrics as mentioned in Section 2.1. Additionally, we plan to also add support for similarity metrics, which can be useful to measure how much an obfuscated program differs from the original one, and in the assessment of software diversification [35]. Finally, we plan to extend the experiment setup by investigating how the metrics perform when dealing with combinations of multiple obfuscations.

Acknowledgments

This work was supported by project SERICS (PE00000014) under the National Recovery and Resilience Plan of the Italian Ministry of University and Research (MUR), funded by the European Union - NextGenerationEU (https://serics.eu/).

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] P. Falcarin, C. Collberg, M. Atallah, M. Jakubowski, Software protection, IEEE Software 28 (2011) 24–27. doi:10.1109/MS.2011.34.

[2] C. Collberg, C. Thomborson, D. Low, A taxonomy of obfuscating transformations, Technical Report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[3] C. S. Collberg, C. Thomborson, Watermarking, tamper-proofing, and obfuscation-tools for software protection, IEEE Transactions on software engineering 28 (2002) 735–746.

[4] C. Collberg, Tigress: A source-to-source-ish obfuscation tool, in: Proc. 8th Workshop on Software Security, Protection, and Reverse Engineering, 2018.

[5] P. Junod, J. Rinaldini, J. Wehrli, J. Michielin, Obfuscator-LLVM – software protection for the masses, in: P. Falcarin, B. Wyseur (Eds.), Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015, IEEE, 2015, pp. 3–9. doi:10.1109/SPRO.2015.10.

[6] C. Collberg, C. Thomborson, D. Low, Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs, in: Proc. 25th Symp. Principles of Programming Languages, 1998, pp. 184–196.

[7] M. Ceccato, A. Capiluppi, P. Falcarin, C. Boldyreff, A large study on the effect of code obfuscation on the quality of java code, Empirical Software Engineering 20 (2015) 1486–1524.

[8] G. Zhang, P. Falcarin, E. Gomez-Martinez, S. Islam, C. Tartary, B. De Sutter, J. d'Annoville, Attack simulation based software protection assessment method, in: Cyber Security, 2016.

[9] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, P. Tonella, The effectiveness of source code obfuscation: An experimental assessment, in: IEEE 17th International Conference on Program Comprehension (ICPC), 2009, pp. 178–187. doi:10.1109/ICPC.2009.5090041.

[10] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, A. Pretschner, Code obfuscation against symbolic execution attacks, in: Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016, pp. 189–200.

[11] M. Talukder, S. Islam, P. Falcarin, Analysis of obfuscated code with program slicing, in: 2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security), 2019, pp. 1–7. doi:10.1109/CyberSecPODS.2019.8885094.

[12] H. Wang, D. Fang, N. Wang, Z. Tang, F. Chen, Y. Gu, Method to Evaluate Software Protection Based on Attack Modeling, in: Proc. 10th IEEE Int. Conf. High Performance Computing and Communications & IEEE Int. Conf. Embedded and Ubiquitous Computing, 2013, pp. 837–844.

[13] B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, B. Preneel, Program obfuscation: a quantitative approach, in: Proc. ACM workshop Quality of protection, 2007, pp. 15–20.

[14] A. Capiluppi, P. Falcarin, C. Boldyreff, Code defactoring: Evaluating the effectiveness of java obfuscations, in: 2012 19th Working Conference on Reverse Engineering, 2012, pp. 71–80. doi:10.1109/WCRE.2012.17.

[15] T. J. McCabe, A complexity measure, IEEE Trans. Software Eng. 2 (1976) 308–320.

[16] L. Regano, D. Canavese, C. Basile, M. Torchiano, Empirical assessment of the code comprehension effort needed to attack programs protected with obfuscation, 2025. URL: https://arxiv.org/abs/2511.21301. arXiv:2511.21301.

[17] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, P. Tonella, A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques, Empirical Software Engineering 19 (2014) 1040–1074.

[18] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, P. Tonella, Towards experimental evaluation of code obfuscation techniques, in: Proceedings of the 4th ACM workshop on Quality of protection, QoP '08, ACM, New York, NY, USA, 2008, pp. 39–46. URL: http://doi.acm.org/10.1145/1456362.1456371. doi:http://doi.acm.org/10.1145/1456362.1456371.

[19] M. Ceccato, P. Tonella, C. Basile, P. Falcarin, M. Torchiano, B. Coppens, B. De Sutter, Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge, Empirical Software Engineering (2018) 1–47. doi:10.1007/s10664-018-9625-6.

[20] M. Ceccato, P. Tonella, C. Basile, B. Coppens, B. De Sutter, P. Falcarin, M. Torchiano, How

professional hackers understand protected code while performing attack tasks, in: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), 2017, pp. 154–164. doi:10.1109/ICPC.2017.2.

[21] S. K. S. Kumar, S. P. Kulyadi, P. Mohandas, M. S. Raman, V. Vasan, Computation of cyclomatic complexity and detection of malware executable files, in: 2021 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI), IEEE, 2021, pp. 1–5.

[22] M. Protsenko, T. Müller, Android malware detection based on software complexity metrics, in: International conference on trust, privacy and security in digital business, Springer, 2014, pp. 24–35.

[23] A. Calleja, J. Tapiador, J. Caballero, A look into 30 years of malware development from a software metrics perspective, in: International Symposium on Research in Attacks, Intrusions, and Defenses, Springer, 2016, pp. 325–345.

[24] A. Calleja, J. Tapiador, J. Caballero, The malsource dataset: Quantifying complexity and code reuse in malware development, IEEE Transactions on Information Forensics and Security 14 (2018) 3175–3190.

[25] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al., Sok:(state of) the art of war: Offensive techniques in binary analysis, in: 2016 IEEE symposium on security and privacy (SP), IEEE, 2016, pp. 138–157.

[26] E. I. Oviedo, Control flow, data flow and program complexity, 1984.

[27] S. Henry, D. Kafura, Software structure metrics based on information flow, IEEE transactions on Software Engineering (1981) 510–518.

[28] S. A. Ebad, A. A. Darem, J. H. Abawajy, Measuring software obfuscation quality–a systematic literature review, IEEE Access 9 (2021) 99024–99038.

[29] T. J. McCabe, A complexity measure, IEEE Transactions on software Engineering (1976) 308–320.

[30] C. E. Shannon, A mathematical theory of communication, The Bell system technical journal 27 (1948) 379–423.

[31] K. Lee, S.-Y. Lee, K. Yim, Machine learning based file entropy analysis for ransomware detection in backup systems, IEEE access 7 (2019) 110205–110215.

[32] M. Bat-Erdene, H. Park, H. Li, H. Lee, M.-S. Choi, Entropy analysis to classify unknown packing algorithms for malware detection, International Journal of Information Security 16 (2017) 227–248.

[33] M. H. Halstead, Elements of Software Science (Operating and programming systems series), Elsevier Science Inc., 1977.

[34] TheAlgorithms, Thealgorithms/c: Collection of various algorithms in mathematics, machine learning, computer science, physics, etc implemented in c for educational purposes., 2023. URL: https://github.com/TheAlgorithms/C.

[35] M. Ceccato, P. Falcarin, A. Cabutto, Y. W. Frezghi, C. A. Staicu, Search based clustering for protecting software with diversified updates, in: Proc of International Symposium on Search-Based Software Engineering, Springer, 2016, pp. 159–175.