



AUTOMATED MULTI-CRITERIA EVALUATION OF MALWARE OBFUSCATION ON COMPILED BINARIES

D.A. Khaman^{1*}, L.N. Abdullah², A. Shaikhanova³

¹Department of Cyber Security, Astana IT University, Astana, Kazakhstan, ORCID: <https://orcid.org/0009-0007-3450-9381>

²Department of Multimedia, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Serdang, Selangor, Malaysia, ORCID: <https://orcid.org/0000-0001-8704-2390>

³Department of Information Security, L.N. Gumilyov Eurasian National University, Astana, Kazakhstan, ORCID: <https://orcid.org/0000-0001-6006-4813>

*Corresponding author: Danial Khaman, 255085@astanait.edu.kz.

Abstract

Obfuscation is routinely applied to malicious code, yet its effectiveness is usually argued rather than measured. In practice it is assessed by the drop in the number of engines that flag a sample – a figure that changes as signature databases do. Existing quantitative frameworks measure the program instead, but operate on source code and cannot be applied to the samples an analyst receives. This paper presents an automated pipeline that measures obfuscation quality on compiled Windows binaries and reports two quantities: an index Q of the cost imposed on analysis and a separate measure S of how much of the original byte sequence survives. Every sample is verified by execution, and the pipeline was applied to thirteen purpose-built modules under ten transformation combinations, giving 130 verified pairs. Over these pairs Q and S prove not to be interchangeable – their relationship reverses direction depending on whether transformations are compared on one sample or samples are compared with each other. The clipped normalisation of relative change in common use reaches its ceiling as soon as a measured quantity doubles, so it assigns its maximum to four fifths of the dataset although the underlying structural growth spans two orders of magnitude; a logarithmic normalisation restores the distinction. And no transformation removes a single imported function: the import table is produced by the linker, beyond the reach of a source-level obfuscator, so the strongest static indicators available to a scanner survive any amount of added control-flow complexity.

Keywords: code obfuscation, malware analysis, software metrics, symbolic execution, signature detection, multi-criteria evaluation.

1. Introduction



Modern malicious software is almost always obfuscated. The transformation families involved – control-flow flattening, virtualisation, opaque predicates, literal encoding. And the tools that apply them are freely available to defenders and attackers alike. What remains unsettled is how to state, quantitatively, that one obfuscation is better than another.

In operational practice the question is answered by proxy. A sample is submitted to a multi-engine scanning service, the number of engines that flag it is compared before and after transformation, and whichever transformation lowers the count further is declared superior; Aditiawarman et al. (2023) [1] demonstrate how far automated obfuscation can push that count down, which is precisely the practice at issue. The procedure has two defects that are rarely acknowledged. It conflates the quality of an obfuscation with the present state of a particular set of signature databases, and it is not reproducible, because those databases change. The same experiment repeated six months later returns different numbers, and nothing in the reported result tells the reader which part of it was a property of the obfuscation and which was a property of the calendar.

Collberg et al. (1997) [5] proposed the conceptual alternative almost three decades ago: Potency, the degree to which a transformation complicates the program for a reader; Resilience, its resistance to automated analysis; and Cost, the overhead it imposes. The decomposition is now standard. What it does not supply is a procedure – the categories are defined analytically, and how the quantities are to be obtained from a real program is left open.

Jin et al. (2024) [6] closed part of that gap for source code. Their framework embeds twelve measurement indicators into the LLVM compiler and emits their values as the program is compiled: cyclomatic complexity, control-flow-graph size and depth, program length and instruction count for Potency; symbolic-execution time and coverage, static-analysis time and optimisation ratio for resistance to analysis; execution time, memory and file size for Cost. It is the most complete operational instrument of its kind, and it is by construction unavailable to malware analysis. The framework observes the program as the compiler sees it, whereas the analyst confronting a suspicious sample has a binary, not a build tree. A second gap remains open even for source code: the framework reports raw indicator values beside those of a baseline, without normalising them to a common range and without combining them, so the judgement of which obfuscation is better is still left to the reader.

The aim of this work is to determine whether the Collberg decomposition can be measured on a compiled binary with sufficient accuracy to support comparison between transformations, and what happens to the resulting metrics when they are exercised on a dataset large enough for their behaviour to become visible. The object of the study is obfuscating transformations applied to compiled Windows binaries; the subject is the set of quantitative indicators of obfuscation quality recoverable from such a binary in the absence of source code. Two things the

analyst may lack must be distinguished here. The method needs no source code, and in that sense, it applies to compiled artefacts. It does need a reference build of the same program without the transformation, because every indicator is a relative change measured against that baseline. It therefore serves the evaluation of an obfuscator whose input is available, which is the position of a defender assessing tools, and not the scoring of an isolated sample captured in the wild.

Two hypotheses are tested. The first is that resistance to automated analysis and resistance to signature matching are distinct properties describing different adversaries – one that attempts to understand the code, and one that never does and merely matches patterns. Folding them into a single scalar index discards information rather than summarising it. The second is that the ratio-based normalisation on which every metric of this kind rests silently assumes transformations to be multiplicative in the quantity being measured, whereas some add a fixed amount of structure and others an amount proportional to what was there.

The novelty of the approach lies in three elements: measurement on the compiled artefact rather than in the compiler, which is what makes the instrument applicable to the samples an analyst actually receives; the separation of signature resistance from the analysis-cost index, supported empirically rather than assumed; and an execution-based verification step which revealed that an entire transformation family had been producing binaries that could not run.

2. Material and research methods.

2.1 Normalisation of relative change.

Every metric in this work is built from the change in a measured quantity between a clean and an obfuscated build of the same program. Let x_c denote a measurement on the clean sample and x_o the corresponding measurement on the obfuscated one. The formulation in common use is the clipped relative difference

$$\Delta_{lin}(x) = \min \left(1, \max \left(0, \frac{(x_o - x_c)}{\max(x_c, \epsilon)} \right) \right) \quad (1)$$

with ϵ a small positive constant. It maps to $[0, 1]$, which a weighted aggregation requires, and it is monotonic. It also reaches its ceiling as soon as the measured quantity doubles. Section 3 shows this to be fatal in practice, and the logarithmic normalisation is used instead:

$$\Delta(x) = \min \left(\frac{1, \ln \left(\frac{x_o}{x_c} \right)}{\ln K} \right), x_o > x_c > 0 \quad (2)$$

with $\Delta(x) = 0$ when $x_o \leq x_c$. The constant K declares the growth factor treated as maximal. Throughout this work $K = 100$. Both formulations map to $[0, 1]$ and both are monotonic, but (2) distributes its resolution evenly across orders of magnitude rather than exhausting it below a factor of two. The boundary case $x_c = 0, x_o > 0$ is qualitative rather than a guard against division



by zero: the metric jumps to 1, so a function with no structure at all attains maximal Potency from any transformation whatsoever. In the dataset used here $x_c \geq 1$ throughout, but the condition must be checked when transferring the method to other data.

2.2 The component metrics

Potency is computed over the target function only, not over the whole binary. This is a correctness requirement rather than a refinement. A dynamic-link library produced by the MinGW toolchain contains approximately fifty-five runtime support functions contributing roughly 230 units of cyclomatic complexity before any user code is considered. Against that baseline the transformation of module code is a rounding error. Measured on one module of the dataset, whole-binary complexity grew by a factor of 1.09 while the target function grew by 7.50. A Potency metric summed over the binary therefore reports, to first order, the complexity of the compiler's runtime. Following the indicator set of Jin et al. (2024) [6] and adapting it to the binary level,

$$P = \frac{1}{4} [\Delta(CC) + \Delta(|V|) + \Delta(IC) + \Delta(SZ)] \quad (3)$$

where $CC = E - V + 2$ is McCabe's cyclomatic complexity [7] computed on the recovered control-flow graph, $|V|$ is the number of basic blocks, IC the instruction count and SZ the size of the function in bytes, the last standing in for source-code length, which does not survive compilation.

Resilience is measured by symbolically executing the target function under a fixed budget. The natural indicator is elapsed time, and it is the one the source-level framework uses, but elapsed time has a ceiling: once both samples exhaust the budget, both report the same value and their difference vanishes – not because the obfuscation is weak but because the instrument has run out of range. The number of solver steps completed within the same budget is used instead:

$$R = \frac{1}{3} [\Delta(Steps) + \Delta(T_{static}) + (1 - Cov_o)] \quad (4)$$

where T_{static} is the time taken by the static analyser to reconstruct the control-flow graph and $Cov_o \in [0, 1]$ is the fraction of the target function's basic blocks the solver reached. Two samples that both exhaust a sixty-second budget may differ by thousands of steps within it.

A step is one iteration of the exploration loop. The engine advances every currently active state by one basic block, so a step is a unit of solver work. The loop ends when no active state remains, or when the sixty-second budget expires. On a clean function the exploration usually finishes, so the count records the work needed to complete it. But obfuscation multiplies the number of distinct paths and the loop keeps iterating. The indicator therefore measures work performed under a fixed budget.

Coverage is the proportion of basic blocks of the target function that the solver reached in the obfuscated build, not the difference compared to the clean build. This approach has a drawback. The solver does not always fully cover the clean build either, it succeeds in 90 out of 130 cases

and in the worst case reaches only 62.8%. Where it falls short on the clean build, the missing coverage is attributed to obfuscation, which is not its cause. The median is 100% on the clean builds against 86.4% on the obfuscated ones. Thus the indicator is better interpreted as the share of the obfuscated function covered by the solver, rather than as the share of coverage lost to obfuscation.

Cost is

$$C = \frac{1}{3} [\Delta(T_{exec}) + \Delta(Mem) + \Delta(Size)] \quad (5)$$

where T_{exec} is the measured execution time of the obfuscated function, Mem the image size the module occupies when loaded and Size is its size on disk. All three transfer without modification. Reporting only the last, as is common in binary-level work, understates the cost by orders of magnitude.

The three are aggregated into a quality index

$$Q = w_P \cdot P + w_R \cdot R - w_C \cdot C, \quad w_P + w_R + w_C = 1, w_i \geq 0 \quad (6)$$

All three terms describe an analyst or automated tool attempting to recover the program's behaviour, and the overhead the defender accepts to obstruct that recovery. Q is bounded above by $w_P + w_R$. It is a comparative quantity, meaningful between transformations measured under the same weighting rather than on an absolute scale.

The weights are an expert assignment, stated as fractions to satisfy the constraint exactly: $w_P = w_R = 5/12$ and $w_C = 2/12$. The two benefit terms enter symmetrically, since neither structural complication nor resistance to automated analysis is prior to the other, and cost enters at less than half the weight of each. No claim is made that this assignment is uniquely correct. The pipeline records every raw indicator value alongside the derived metrics, so the index can be recomputed under any other weighting without repeating the measurements.

2.3 Signature resistance

A byte signature is a contiguous sequence of bytes drawn from a known-malicious sample. It matches a new sample if that sequence occurs in it. The question to ask of an obfuscating transformation is therefore how long is the longest sequence from the original malicious function that still occurs in the obfuscated binary. Let L_f be the length in bytes of the target function in the clean build and L_s the length of the longest of its byte sequences still present anywhere in the obfuscated binary. Then

$$S = 1 - \frac{L_s}{L_f} \quad (7)$$

$S = 0$ means the function was not changed at all; $S \rightarrow 1$ means no recognisable fragment survives. L_s is computed by binary search on sequence length, since if no sequence of length n survives then none of length $n + 1$ can.



Because S normalises by the length of the clean target function, two functions retaining fragments of the same absolute length receive different scores, the longer function scoring higher. Comparisons across modules therefore have to be read together with the absolute surviving length, which is reported alongside S throughout Section 3.

The bytes of the target function are taken from the compiled DLL with radare2 from function's entry address and size. Three properties of the function range matter for the validity of the measure:

1. The range carries no alignment padding: radare2 reports size equal to `realsz` for all thirteen target functions.
2. No base-relocation entry falls inside any target function range. The modules reach the Win32 API through the import table, so absolute addresses live in `.idata` and not in the function body. A binary carrying jump tables or statically embedded pointers would not have this property, and the measure would then need relocation-aware masking.
3. A surviving sequence is searched anywhere in the obfuscated binary rather than inside the corresponding function alone, which admits matches in compiler runtime code shared by both builds. Recomputing S over all 130 pairs with the search restricted to the obfuscated target function moves the mean from 0.9036 to 0.9076 and leaves the ranking intact, so the wider scope is reported.

2.4 The dataset

Thirteen modules were written for this study, each implementing a recognisable malicious technique as a Windows dynamic-link library. Purpose-built modules were used rather than samples captured in the wild. The reason is, the normalised metrics require a clean baseline, which for a captured sample does not exist. Ten have small target functions; three were written specifically with functions of realistic size, in order to test whether the results transfer. Table 1 describes the dataset.

Table 1. The modules of the dataset.

"Fns" is the number of functions passed to the obfuscator. CC, Blocks, Instrs and Bytes describe the target function of the clean build, which is the single function all metrics in this work are measured on.

Module	Target function	Fns	CC	Blocks	Instrs	Bytes
Registry persistence	InstallRegistryPersistence	2	2	3	40	161
Scheduled-task persistence	InstallPersistence	2	1	1	23	110
Dropper	BuildDownloadUrl	3	2	4	30	112



Shellcode runner	DecryptPayload	2	2	4	26	76
Process enumerator	FindTargetProcess	2	5	8	44	191
Keylogger stub	BuildLogPath	3	1	1	18	79
File enumerator	EnumerateFiles	2	7	12	70	317
Network beacon	BuildC2Address	2	1	1	27	102
Memory scanner	ScanProcessMemory	2	11	19	91	371
Environment fingerprint	BuildProfile	3	1	1	37	156
PE export resolver	ExportPE	3	29	53	149	412
Configuration decoder	DecodeConf	3	27	66	232	515
Command-protocol parser	ParseCP	4	43	78	309	811

The ten small modules total 607 lines of C; their clean builds are between 40 930 and 43 756 bytes and import between 26 and 34 functions, most belonging to the C runtime. The dataset spans the range of behaviour the metrics must distinguish. Some target functions are pure computation – DecryptPayload is a byte loop, BuildDownloadUrl concatenates strings – while others are dominated by system calls, such as FindTargetProcess, which iterates a process snapshot. Section 3 shows this distinction to be the dominant factor in the execution cost of obfuscation, so it matters that both are present. Control-flow structure varies from straight-line functions (CC = 1) to a nested scanning loop (CC = 11) among the small modules, and from 27 to 43 among the large ones. The three large modules are pure computation or read-only, so all thirteen remain verifiable by execution.

Each module contributes ten observations that share one source file, one clean build and one target function, so the 130 rows are not 130 independent samples. Correlations reported below are therefore descriptive of this dataset rather than estimates for a population of programs, and where the relationship between two metrics carries an argument, as it does for Q and S in Section 3.6, it is examined at two levels separately: within a module, where the target function is fixed



and only the transformation varies, and across the thirteen module averages, where it is the function that varies.

2.5 Transformation combinations

Ten combinations of Tigress transformations [4] were applied to each module, giving 130 clean/obfuscated pairs (Tab. 2).

Table 2. Transformation combinations, ordered by the mean growth in cyclomatic complexity of the target function, averaged over all thirteen modules

Combination	Constituent transformations	CC growth
encode_literals	literal encoding	1.91
opaque	opaque predicate insertion	1.98
flatten	control-flow flattening	4.18
encode_flatten	literal encoding, then flattening	5.07
flatten_opaque	flattening with opaque predicates	5.26
full_stack	encoding, opaque predicates, flattening	7.27
virtualize	translation to a custom stack machine	19.43
virtualize_opaque	virtualisation with opaque predicates	20.36
encode_virtualize	literal encoding, then virtualisation	42.68
virtualize_flatten	virtualisation, then flattening	72.71

The set separates the effects of the three primitive transformations and their pairwise interactions rather than enumerating the obfuscator's options. Literal encoding is the only one that touches static data. Flattening and virtualisation both restructure control flow but by different mechanisms: flattening rewrites the graph into a dispatch loop over the original basic blocks, while virtualisation replaces the code entirely with bytecode and an interpreter.

2.6 Measurement pipeline and tooling

For each pair the pipeline performs, in order:

1. Transformation of the module source; post-processing of the obfuscator's output to reconcile the two data models; compilation of both builds through the shared wrapper.
2. Recovery of the control-flow graph and extraction of the structural indicators from each binary.
3. Symbolic execution of the target function in each binary under a sixty-second budget.
4. Parsing of the PE headers for size, image size and imports measurement.
5. Execution of both builds for the equivalence check and the timing measurement.
6. Computation of the byte-sequence survival length.

The analysis of the clean build is performed once per module and reused across its ten transformations, since it does not depend on them.



The pipeline writes all raw indicator values alongside the derived metrics in its output tables, so alternative formulations of Δ , of the component metrics or of the weighting can be evaluated by post-processing without repeating the experiment. Every comparison between formulations reported in Section 3 was produced in exactly that way.

Structural indicators are extracted with Radare2 6.1.0 [8] and symbolic execution uses angr 9.2.203 [9] under Python 3.14. Obfuscation was performed with Tigress 3.3.3 on Ubuntu 24.04 under WSL2, invoked with '`--Environment=x86_64:Linux:Gcc:13`'. Both builds were compiled on Windows with MinGW-w64 GCC 15.2.0 as shared libraries, linked against kernel32, user32 and advapi32, with no optimisation flag. All measurements were taken on one machine, an Intel Core i5-10500H with six cores and 16 GB of memory running Windows 11.

The post-processing of the obfuscator's output reconciles two data models. Tigress is driven with a Linux target, where long occupies eight bytes, while its output is compiled for Windows x64, where it occupies four. Virtualisation stores pointers in long-typed interpreter cells, so without this step every pointer would be truncated to 32 bits.

Symbolic execution starts from a blank state placed at the entry of the target function, with no arguments constrained and no preceding path. Libraries are not loaded and calls into them are left unconstrained. The analysis ends when no active state remains or when sixty seconds have elapsed, whichever comes first, and coverage is counted over the basic blocks the engine attributes to the target function.

Execution time is measured with QueryPerformanceCounter around a loop of fixed length, chosen per module so that the loop runs long enough to be measurable: 200 000 iterations for the short string and byte-loop functions, down to three for the memory scanner. The reported figure is the total divided by the iteration count, taken from a single run of each build. Its dispersion was not estimated, which bounds how finely two similar slowdowns can be told apart.

3. The results and discussion

3.1 Semantic preservation

All 130 pairs were built, executed and compared against their clean counterparts. Each module is exercised through one fixed invocation of its target function with constant inputs – DecryptPayload on the byte sequence 1 to 8, BuildDownloadUrl into a caller-supplied buffer, FindTargetProcess on "explorer.exe". And the observable results are the contents of the output buffer or the return value. For the two modules whose target function has external side effects, the registry and scheduled-task installers, link-time stubs record the arguments of each Win32 call into a string instead of performing it, and that string is the compared output. A pair counts as preserved when the obfuscated build reproduces the clean build's line exactly. This is testing on one input per module rather than a proof of semantic equivalence, so the supported claim is that behaviour is preserved on the scenarios tested. That was enough to reveal that every sample of one transformation family could not run at all, which is why the step was added.



We conclude that any evaluation of obfuscation performed with an obfuscator external to the compiler must verify its samples by running them.

Table 3. Evaluation by transformation combination, averaged over the 13 modules.

Q aggregates P, R and C; S is reported alongside it

Transformation	P	R	C	Q	S
encode_literals	0.274	0.196	0.110	0.177	0.896
opaque	0.200	0.121	0.093	0.118	0.850
flatten	0.247	0.189	0.042	0.175	0.871
encode_flatten	0.403	0.246	0.130	0.249	0.910
flatten_opaque	0.348	0.226	0.110	0.220	0.889
full_stack	0.432	0.291	0.126	0.280	0.910
virtualize	0.519	0.346	0.149	0.335	0.924
virtualize_opaque	0.540	0.378	0.170	0.354	0.926
encode_virtualize	0.670	0.399	0.200	0.412	0.933
virtualize_flatten	0.694	0.418	0.202	0.430	0.928

Table 4. Evaluation by module class, averaged over the ten transformation combinations (n = 10 each).

Module class	P	R	C	Q	S
Environment fingerprint	0.530	0.277	0.013	0.334	0.919
Scheduled-task persistence	0.517	0.262	0.031	0.320	0.865
Network beacon	0.549	0.244	0.080	0.317	0.807
Keylogger stub	0.544	0.213	0.037	0.309	0.813
Registry persistence	0.489	0.241	0.077	0.292	0.886
Dropper	0.488	0.268	0.163	0.288	0.895
File enumerator	0.400	0.280	0.013	0.281	0.945
Memory scanner	0.378	0.300	0.160	0.256	0.943
Process enumerator	0.458	0.160	0.014	0.255	0.929
Command-protocol parser	0.251	0.431	0.291	0.236	0.963
PE export resolver	0.319	0.378	0.346	0.233	0.981



Shellcode runner	0.454	0.231	0.318	0.233	0.814
Configuration decoder	0.248	0.364	0.191	0.223	0.986

Target-function cyclomatic complexity ranges from 1 to 11 in the clean small-function builds and from 1 to 191 after obfuscation; basic-block counts range from 1–19 to 4–295. Growth in cyclomatic complexity spans $\times 0.5$ to $\times 180$ with a median of $\times 6.1$.

The lower end of that range deserves comment. In one pair, the shellcode runner under `full_stack`, the measured complexity of the target function fell from 2 to 1 while its instruction count more than doubled, from 26 to 55, and the block count was unchanged at four. What disappeared was a single edge, the back-edge of the byte loop, which the compiler eliminated when the transformed body was emitted. Since $CC = E - V + 2$, losing one edge without losing a block lowers the metric by one irrespective of how much code was added. This illustrates a general hazard of structural metrics recovered from binaries: they describe the graph the compiler emitted and the disassembler recovered, not the graph the transformation nominally produced. Cyclomatic complexity is particularly exposed because it is a difference of two counts and can move in either direction. Aggregating several structural indicators, as Eq. (3) does, bounds the influence of any one such case. This pair's instruction and size indicators both rise, and its Potency remains positive at 0.103.

3.2 Cost

File size grows by a factor between 1.002 and 1.822 across the dataset. Execution time behaves quite differently: the median slowdown is $\times 2.19$ and the maximum $\times 1839$. The two are poor proxies for one another – the rank correlation between slowdown and file-size growth is 0.41, far below the agreement that would let one stand in for the other. Structural growth predicts execution cost no better : the rank correlation between complexity growth and slowdown is -0.10, which is not distinguishable from zero. How much a transformation complicates the control-flow graph carries no detectable information about how much it costs to run. Since Potency is built from structural indicators and Cost from timing, this is a statement that two components of Q measure genuinely different things – which is what a multi-criteria index requires.

The cost of obfuscation is paid in instructions executed, so it is visible only where instructions dominate the running time. The shellcode runner's target function is a tight decryption loop and obfuscation ruins it. The enumerator's process spends its time inside system calls, and obfuscating the surrounding code costs nothing measurable. For syscall-bound routines heavy obfuscation is close to free, while for compute-bound ones it is ruinous. This is an operationally significant distinction – it determines where in a program obfuscation can be applied without consequence.

3.3 Signature resistance

The longest surviving byte sequence ranges from 6 to 114 bytes across the dataset, giving S between 0.539 and 0.992 over 50 distinct values. Ordered by median surviving length, `encode_virtualize` destroys the most (11 bytes surviving) and `opaque` the least (25 bytes). The



length at which a fragment becomes usable as a signature was measured. For each candidate length, we counted the fraction of the target function's fragments of that length that also occur in the binary of an unrelated module, since such a fragment is a compiler or runtime artefact and would produce false positives in any rule built on it. The fraction falls from 43.4% at four bytes to 21.1% at six, 9.8% at eight, 2.2% at twelve and 0.4% at sixteen, reaching zero at twenty-four. Therefore, eight bytes still carries roughly a 10% chance of being non-specific, and the practical knee lies between sixteen and twenty-four. On the small-function group the median surviving fragment is fourteen bytes, and 66 of the 100 pairs leave nothing longer than sixteen, so most fall below the length at which a fragment is specific enough to build a rule on, although only twelve fall below eight.

Two qualitative findings accompany this. Literal encoding removes every string indicator in all ten small modules, and no combination that omits it removes a single one. And no transformation removes a single import, in any pair: the import table is produced by the compiler and linker, and a source-level obfuscator does not reach it.

This last result is the sharpest boundary the work establishes, and it is where our findings and those of Cao et al. [2] diverge in consequence while agreeing in substance. Obfuscating WebAssembly binaries, they submitted eighteen labelled cryptominers to a multi-engine service under different transformations: control-flow flattening left every sample detected, while name and memory obfuscation applied together left none detected, from which they conclude that most detectors match function names and string literals rather than recovering semantics by static analysis. We agree on a different platform and toolchain that control-flow transformation does nothing for signature surface. But their tool rewrites binaries and can therefore rename, whereas a source-level obfuscator cannot reach the import table at all. If detection rests largely on names and strings, then source-level obfuscation reaches only the strings, and the more detectable half of the surface is structurally beyond it.

3.4 Dependence on function size

The ten modules discussed so far have small target functions, with cyclomatic complexity between 1 and 11, median 2. The three modules written with functions of realistic size: complexities of 29, 27 and 43, between 53 and 78 basic blocks.

Table 5. Median absolute increase in cyclomatic complexity, and the corresponding growth factor, by transformation and function-size group

Transformation	Absolute increase, small	Absolute increase, large	Growth factor, small	Growth factor, large
opaque	2.0	2.0	×2.0	×1.1
flatten	6.0	120.0	×4.0	×5.2
encode_flatten	7.5	132.0	×5.0	×5.7
full_stack	12.5	137.0	×8.0	×5.8
virtualize	45.5	46.0	×23.8	×2.7
virtualize_opaque	47.5	48.0	×24.8	×2.8



encode_virtualize	102.0	100.0	×55.8	×4.4
virtualize_flatten	172.5	104.0	×88.0	×4.9

Virtualisation adds essentially the same absolute amount of structure to a function. The reason is that what it adds is an interpreter, whose dispatch structure does not depend on the code being interpreted. Flattening adds twenty times more structure to the large functions than to the small ones, because large functions have more blocks. Expressed as ratios, this inverts the apparent ranking: on small functions virtualisation looks six times stronger than flattening (×23.8 against ×4.0), while on large functions it looks half as strong (×2.7 against ×5.2).

The saturation result is therefore size-dependent. On the small-function group the clipped normalisation reaches its ceiling in 85 of 100 pairs. On the large-function group it reaches the ceiling in 20 of 30 pairs. Pooled over all 130 pairs the difference remains clear. The practical statement is accordingly narrower than the small-function data alone would support: clipping fails when baseline functions are small, because a roughly fixed structural addition then produces an enormous ratio. Synthetic datasets tend to contain small functions, which makes this a methodological warning about evaluation design as much as about the formula.

Cost rises sharply with function size. Median slowdown is ×1.64 on the small-function group and ×91.4 on the large one. Obfuscating a large function is roughly fifty-five times more expensive than obfuscating a small one. It means that measurements of Cost taken on small functions understate what obfuscation costs in practice by a wide margin.

3.5 Measurement noise and threats to validity

Repeating the complete measurement on an unchanged binary returns identical values for cyclomatic complexity, basic-block count, instruction count, file size, entropy, coverage and solver-step count. Wall-clock symbolic-execution time varies by 3-5%, and solver steps by about 1%, and only when the budget is exhausted. This is the second argument for building Resilience on solver steps rather than elapsed time: besides retaining resolution at the ceiling, the quantity is reproducible.

Several further limitations bound what the measurements support. Cyclomatic complexity is a proxy for the difficulty a human analyst experiences, not a measurement of it, and the human-subject results of Ceccato et al. [3] caution directly against reading P as analyst effort. S models byte-signature matching specifically and says nothing about behavioural or machine-learning detection. Structure is recovered from machine code rather than from the compiler's own representation, so any mis-resolved indirect jump or misidentified function boundary propagates into Potency. Flattening and virtualisation produce precisely the indirect-dispatch patterns disassemblers resolve least reliably, making the measurement least trustworthy exactly where the transformation is strongest. Solver steps are counted within a fixed budget, so a sample that would require ten times the budget and one that would require a hundred times are not distinguished.

3.6 Why S is reported separately from Q

Over all 130 pairs the correlation between Q and S is +0.16, close to zero, so on its own this number would suggest the two are unrelated.



Compare transformations within one module, where the function is fixed and only the obfuscation changes, and the two move together: the correlation inside a module averages +0.66 and is clear in nine of the thirteen modules. Compare the modules with each other, using their averages, and the two move apart: the correlation is -0.61. The reason is function size. A larger function loses a smaller fraction of its length to any one surviving fragment, so S , which normalises by that length, comes out higher. Also, its complexity grows relatively less under the same transformation, so its Q comes out lower.

This is what makes a single fixed-weight combination of the two difficult to interpret rather than merely imprecise. Such an index would mean one thing when transformations are ranked on one sample and something different when samples are ranked against each other. And also, we do not claim that aggregation is impossible in principle. What these data support is the narrower conclusion that the two are best reported separately, with the level of comparison stated.

4. Conclusion

This work set out to determine whether the Collberg decomposition can be measured on a compiled binary with sufficient fidelity to support comparison between transformations, and what happens to the resulting metrics when they are exercised on a dataset large enough for their behaviour to become visible. Both halves of that question have been answered. An automated pipeline was implemented that evaluates obfuscation quality on compiled Windows binaries. Applied to thirteen purpose-built malicious modules under ten transformation combinations, it produced 130 verified clean/obfuscated pairs, on which the behaviour of the measurement model was examined.

The first hypothesis stated in the introduction is supported. Resistance to automated analysis and resistance to signature matching describe different adversaries – one that attempts to recover the program's behaviour and one that never does, merely matches patterns. And over the 130 pairs neither quantity can be recovered from the other. A weighted index cannot preserve two components whose relationship changes with the comparison being made, so reporting them as one number discards the information the assessment is being made for.

The second hypothesis is also supported, and it qualifies the measurement results rather than merely adding to them. Transformations divide into two structural families: virtualization adds a fixed amount of control-flow structure and the interpreter, while flattening adds an amount proportional to the original function. Any metric built on the ratio of obfuscated to original, therefore, ranks the two families differently depending on the baseline. Virtualization is six times stronger than flattening on the small functions and half as strong on the large ones.

The practical significance of the work lies in two places. The instrument operates on compiled binaries rather than on a build tree, so it can be applied where source-level frameworks cannot. But, it still requires a clean reference build of the same program, so it evaluates transformations rather than scoring isolated samples. The measure of signature resistance depends on no detection product and therefore returns the same value whenever it is repeated.



Three directions follow directly. The resilience indicator is bounded by the analysis budget, and replacing that budget with a quantity that grows without limit would remove the ceiling rather than merely raise it. The dataset consists of functions written for the purpose, so applying the pipeline to samples captured in the wild would test whether these relationships survive at scale. And the signature-resistance measure models byte matching, whereas detection increasingly rests on learned representations of static features, for which the relevant question is not whether a byte sequence survives but whether a feature vector crosses a decision boundary. Extending the measure to that setting requires a different construction and a different validation, which is the continuation of this work.

Thanks: The authors thank the Department of Cyber Security of Astana IT University.

Financing. This research received no external funding.

Conflict of interest: The authors declare that there is no conflict of interest.

References

1. Aditiawarman, U., Dody, A., Mantoro, T., Maarif, H.A.Q., & Pradiftha, A. (2023). Evading antivirus software detection using Python and PowerShell obfuscation framework. *MATRIK: Jurnal Manajemen, Teknik Informatika dan Rekayasa Komputer*, 22(3), 519-528.
2. Cao, S., He, N., Guo, Y., & Wang, H. (2024). WASMixer: Binary obfuscation for WebAssembly. In *Computer Security — ESORICS 2024* (pp. 88-109). Springer.
3. Ceccato, M., Di Penta, M., Falcarin, P., Ricca, F., Torchiano, M., & Tonella, P. (2014). A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19(4), 1040-1074.
4. Collberg, C. (2024). *The Tigress C diversifier/obfuscator* (Version 3.3.3). URL: <https://tigress.wtf/>
5. Collberg, C., Thomborson, C., & Low, D. (1997). *A taxonomy of obfuscating transformations* (Technical Report 148). Department of Computer Science, The University of Auckland.
6. Jin, H., Lee, J., Yang, S., Kim, K., & Lee, D.H. (2024). A framework to quantify the quality of source code obfuscation. *Applied Sciences*, 14(12), 5056.
7. McCabe, T.J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), 308-320.
8. Radare2 Contributors. (2024). *radare2: Libre reversing framework for UNIX geeks*. URL: <https://rada.re/>
9. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., & Vigna, G. (2016). SoK: (State of) the art of war: Offensive



techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)* (pp. 138-157).

Information about authors

Khaman Danial Askaruly –
Master's degree, *Astana IT
University*, Astana, Kazakhstan
e-mail:
255085@astanait.edu.kz
ORCID: <https://orcid.org/0009-0007-3450-9381>

Abdullah Lili Nurliana – PhD,
Associate Professor,
Universiti Putra Malaysia,
Serdang, Selangor,
Malaysia;
e-mail:
liyana@upm.edu.my
ORCID:
<https://orcid.org/0000-0001-8704-2390>

Shaikhanova Aigul – PhD,
Associate Professor, Professor
of the Department of
Information Security, L.N.
Gumilyov Eurasian National
University, Astana, Republic
of Kazakhstan
e-mail:
aigul.shaikhanova@gmail.com
ORCID:
<https://orcid.org/0000-0001-6006-4813>