An Analysis of Code Obfuscation Effectiveness

Andreas Felder
andreas.felder@students.fhnw.ch
FHNW University of Applied Sciences and Arts Northwestern Switzerland
Module: Cyber Security Lab (cysL)

November 3, 2025

Abstract

This report evaluates the effectiveness of various code obfuscation techniques in hindering the reverse engineering of a C++ application. The primary objective is to analyze the practical impact of "security through obscurity" by implementing a layered, manual obfuscation strategy. This manual obfuscation strategy is then contrasted with a binary produced by an automated, compiler-based tool to evaluate its relative strengths.

The methodology involved subjecting a baseline C++ password-checking program to a sequence of escalating manual obfuscations: layout obfuscation (symbol stripping), junk code insertion (utilizing opaque predicates and volatile sinks), data obfuscation (runtime XOR encryption synergistically linked to the junk code), and control-flow flattening (CFF). This multi-layered binary was then contrasted with a binary generated by the DeClang (Obfuscator-LLVM) tool. The efficacy of each stage was assessed using Ghidra for static analysis, measuring quantitative metrics (e.g., cyclomatic complexity, CFG node count) and qualitative readability.

The analysis demonstrated that isolated techniques provided minimal resistance. Simple layout or junk code obfuscation was trivially bypassed by analyzing clear-text string literals. The critical point in the manual strategy was the implementation of data obfuscation. This created an interdependent defense by forcing an analyst to first defeat the junk code to acquire the decryption key. This was contrasted with the automated DeClang tool, which generated exceptionally high logical complexity (a 182 cyclomatic complexity vs. 37 for the manual CFF) but, by design, focuses on logical transformations and does not obfuscate program data.

The presence of clear-text string literals in the automated build, despite its formidable logical complexity, highlighted that logical obfuscation alone is insufficient. The findings conclude that a "defense in depth" strategy is only successful when it is holistic. Effective obfuscation requires a synergistic application of both logical and data-centric techniques to significantly challenge the analysis of a reverse engineer.

Contents

1 Introduction

The primary objective of this project, as assigned in the Cyber Security Lab module, is to perform an in-depth analysis of a specific attack or security improvement. This report focuses on security through obscurity, specifically by applying code obfuscation techniques to a C++ application to hinder reverse engineering efforts.

1.1 Usage of AI

AI tools were used as an assistant throughout this project, both for code development and for authoring this report.

Report Authoring The primary text of all sections was originally drafted by the author to ensure all analysis and conclusions were based on the project's findings. Following this initial composition, the "Writefull" plugin for Overleaf (a online LaTeXeditor) was utilized to refine the wording. This process involved reviewing and accepting automated suggestions to improve word choice and, in some cases, restructure sentences for greater clarity. The Abstract was the only section completely generated by an AI (Gemini Pro 2.5), and this was done after the main report was completed to provide a concise summary of the finished work.

Code Development The C++ test program (Baseline) was developed entirely by the author without AI assistance. The subsequent obfuscation techniques were conceptualized and selected by the author following a brainstorming process that involved consulting both AI and the cited literature. The manual implementation of these techniques was coded by the author, who utilized a code auto-completion tool (GitHub Copilot) for inline code suggestions during the implementation phase.

2 Methodology

This section outlines the experimental setup, the techniques applied, and the metrics used to evaluate their effectiveness.

2.1 Test Program (Baseline)

The baseline is a simple C++ program that asks for a password and grants access to a "secure area" upon successful authentication. The full source code can be found in Appendix A.

2.2 Tools and Environment

All binaries were compiled on an Apple Silicon Mac (`arm64-apple-darwin25.0.0`) and analyzed with `Ghidra version 11.4.2` using default auto-analysis settings.

- Manual Obfuscation Compiler: `Apple Clang version 17.0.0` was used.
- Automated Obfuscation Compiler: `DeClang (v1.0.0)`, a maintained fork of `obfuscator-llvm`, was used.

2.3 Applied Manual Obfuscation Techniques

The baseline program was progressively altered using several manual obfuscation layers. Each layer was implemented by the author without using an automated obfuscator. The selection of obfuscation follows common transformation categories used in obfuscation research (opaque predicates, data hiding, control-flow transforms) [1, 2]

2.3.1 Layout Obfuscation

Layout obfuscation was applied by removing symbol and debug metadata from the binary to eliminate informative identifiers. In practice `strip` was used on the compiled executable (equivalent to compiling with the -s compile flag) to replace readable names with generic labels, preventing immediate name based inference during static inspection [3]. This step changes only metadata. The program logic remains unchanged (see discussion in [1]).

2.3.2 Junk Code Insertion

To obscure control and inflate analysis effort, non-functional "junk" code was injected into the program. The goal was to introduce plausible-looking computation that an analyst must examine but which does not affect the legitimate program outcome [1, 2]. To ensure this junk code was not removed by the compiler's dead-code elimination optimizer, a multi-part strategy was used, as detailed below.

The Volatile Sink Anchor. A global `volatile int g_junkSink` was used as an anchor. The `volatile` keyword is a C++ keyword to the compiler that the variable's value can change at any time from outside the programs control. This strictly forbids the compiler from optimizing away any code that writes to this variable. This forces it to keep all junk calculations that "feed the sink".

Opaque Predicates. Opaque predicates were created to guard the junk code branches. These are conditions that always evaluate to a constant value at runtime but are computationally difficult for a static analyzer to prove [1, 2]. The implementation in this report is deterministic, based on hashing a constant string: `std::hash<std::string>("fhnw_cysl")`. The result is then scrambled with a series of bitwise operations, including an `XOR` with the constant `0x5f3759df`. This seemingly random magic number is famous for the fast inverse square root algorithm used in the Quake III Arena game engine [4]. Its presence serves as a potential red herring to mislead an analyst into thinking the code is performing complex mathematics. The final deterministic result is then compared to a constant to produce a boolean value that Ghidra cannot pre-calculate.

Live Junk Code Blocks. The opaque predicates were then used to guard "live" junk code blocks that perform useless, but complex, calculations on real program data (like `userInput`) and feed their results into the volatile sink.

An example is the `noisyChecksum()` function. It is a non-cryptographic hash function designed to look legitimate. Its construction uses several techniques from cryptography and hashing:

- **Hashing with a Red Herring Constant:** It uses the large integer `1843216759u` to multiply with the loop index. The deliberate choice of a non-significant, random-looking number serves as a subtle red herring. By placing this unknown constant alongside other well-known magic numbers (like the Quake III and TEA constants), it exploits an analysts pattern recognition. A reverse engineer is likely to waste significant time attempting to identify the origin or mathematical significance of this number, only to find that it has none.
- **Circular Bit Shift:** It performs a 32-bit circular left shift using `(acc << 5) | (acc >> 27)`. A left shift alone discards high bits. Pairing it with a right shift and OR re-injects the spilled bits, so information is not lost but patterns are broken up.
- **Golden Ratio Constant:** It XORs the result with `0x9e3779b9`. This magic number is derived from the golden ratio and is famous for its use in the Tiny Encryption Algorithm (TEA) [5] due to its excellent bit-mixing properties [2, 6].

By building the junk code from legitimate (but, in this context, useless) hashing algorithms, the analyst is forced to waste significant time reverse engineering it, only to find it has no effect on the program's true logic.

Example: Obscuring Logic in the main function A practical example of these three components working together can be seen in the `main` function:

Obscuring logic in the main function

```
//...
if (opaqueFalse()) {
  int t = 0;
        for (int i = 0; i < 100 +
    (static_cast<int>(userInput.size()) & 3); ++i) {
            t ^= (i * 2654435761u) ^ static_cast<int>(userInput.size());
            if ((t & 7) == 5) { t += 3; }
        }
        g_junkSink ^= t;
    } else {
  g_junkSink += noisyChecksum(userInput);
  }

if (checkPassword(userInput)) {
        accessGranted();
} else {
        accessDenied();
}
```

This single `if/else` structure is an implementation of the before described techniques:

- **The Analyst's View:** An analyst in Ghidra sees a conditional branch based on a complex predicate (`opaqueFalse`). They cannot easily determine the outcome. They must assume both paths are possible and analyze them.
- **The Runtime Reality:** The `opaqueFalse()` function always returns `false`. This means the `if` block is "dead code" and the `else` block is "live code" that will always execute.
- **The Deception:** The "dead" `if` block is intentionally filled with a complex-looking loop. This

is a "rabbit hole" designed to lure the analyst into wasting significant time and effort trying to understand a piece of code that will never run.

- **The "Live" Pollution:** The run `else` block, which is part of the real execution path, also calls a junk function (`noisyChecksum`). This pollutes the actual program flow with more useless calculations that must be analyzed, even after the analyst determines the first path was dead.

In this way, the obfuscation wastes the analyst's time twice: once on a dead path, and again on the polluted real path.

2.3.3 Data Obfuscation

The third technique, Data Obfuscation, was designed to eliminate the critical weakness identified in the prior analyses: the presence of clear-text string literals. The first idea was to just split strings. This was not effective at all. With just a little bit of investigative work, a reverse engineer would be able to reconstruct them. Therefore, a more robust, multi-part strategy based on runtime encryption was implemented.

Repeating-Key XOR Encryption. Instead of storing strings in plain text, all meaningful string literals (the password, UI prompts, and secret messages) were encrypted before compilation. This was accomplished using a repeating-key XOR (effectively a Vigenère-style byte XOR [7]) with the key `"bananenbrot"`. This transforms human-readable strings into meaningless byte arrays that are stored in binary. This technique is designed to render Ghidra's "Defined Strings" window completely ineffective, as the byte arrays do not resemble text.

Hiding the Decryption Key. A simple XOR cipher is weak if the key itself is stored in plain text. To avoid this, the decryption key `"bananenbrot"` is never stored as a contiguous string. Instead, it is constructed on the stack at runtime by the `getKey()` function. This function builds the key, character by character (e.g., `key += (char)(opaqueFalse() + 'b');`).

Synergistic Defense. This method of key construction creates a powerful synergistic defense by linking the data obfuscation layer to the junk code layer. Because the construction of each character is dependent on the output of the `opaqueFalse()` function (which is 0), an analyst cannot find the key by simply reading the code. To reconstruct the key `"bananenbrot"`, the analyst must first successfully and completely reverse engineer the opaque predicate, including its complex hashing and "magic number" calculations. This effectively hides the data behind the full complexity of the junk code.

2.3.4 Control-Flow Flattening with Instruction Substitution

As a last technique, Control-Flow Flattening was implemented. This technique was applied on top of the binary that already contained layout, junk code, and data obfuscation.

CFF is designed to fundamentally obscure the program's logical flow, and was therefore targeted at the `checkPassword` function, which contains the core program logic. It operates by breaking the function's natural control flow into a series of independent code blocks. These blocks are then placed inside a single, large `while(true)` loop, which contains a `switch` statement that acts as a central dispatcher [1, 2]. A state variable is used to determine which block to execute next. After a block runs, it updates the variable to jump to the next logical block. This effectively flattens the natural, readable control flow into a more complex and difficult to trace state machine.

To further complicate the analysis, instruction substitution was also added within the flattened logic blocks. Simple, direct comparisons were replaced with mathematically identical but more complex opaque operations (e.g. `((a ^ 0xAA) ^ 0xAA) != ((b + 13) - 13)`). This forces an analyst to not only reverse engineer the state machine itself, but also to analyze and defeat opaque operations inside each state.

2.4 Applied Automatic Obfuscation Technique

While the manual, layered approach is excellent for understanding and learning each technique, it is not a scalable solution for a standard software development lifecycle. It requires specialized security knowledge, is error-prone, and makes the code base significantly more difficult to read, debug, and maintain. Therefore, the second strategy investigated was a "push-button" tool that integrates obfuscation directly into the compilation process.

2.4.1 DeClang (Obfuscator-LLVM)

For this analysis, `DeClang` was chosen. `DeClang` [8] is a well-maintained, modern fork of the original `obfuscator-llvm` (OLLVM) project [9]. Unlike the original OLLVM, which is no longer maintained and failed to compile on the modern `arm64` test environment, DeClang provides pre-compiled binaries for Apple Silicon and is actively developed.

The binary was compiled using `DeClang -O2` and `strip`. The `-O2` optimization is a requirement, as the DeClang documentation states that obfuscation passes only run on optimized code. DeClang was configured via its `config.pre.json` file for maximum logical obfuscation:

- `"overall_obfuscation": 100`: Enables the equivalent of Bogus Control-Flow (`-bcf`) and Instruction Substitution (`-sub`) across the entire program.
- `"flatten": [{"name": ".*"}]`: This regex targets all functions for aggressive Control-Flow Flattening.

2.5 Evaluation Methodology

To evaluate the effectiveness of each obfuscation technique, a set of quantitative and qualitative metrics was defined. Each binary version, starting with the baseline, was assessed against these criteria.

2.5.1 Quantitative Metrics

The following metrics were counted by analyzing the program's entry point and all functions it calls directly. This analysis was non-recursive, meaning it only included the first level of function calls and did not trace the call graph any deeper.

- **Cyclomatic Complexity:** A measure of the number of linearly independent paths through the code.
- **Control Flow Graph (CFG) Node Count:** The total number of basic blocks in a function's graph.
- **Visible String Count:** The number of human-readable strings detected in the binary.
- **Decompiler Line Count:** The total number of lines in the decompiled functions' bodies, serving as a direct measure of code bloat.
- **Ghidra-Generated Variable Count:** The number of local variables declared by the decompiler.

2.5.2 Qualitative Metrics

A qualitative analysis of code complexity is by its definition challenging, as the assessment can be highly subjective. To mitigate this, the assessment is based on a structured evaluation of the code's readability, focusing on:

- The clarity of the control flow.
- The ease of identifying the program's core logic (the password check).
- The effort required to trace data flow between variables.

To complement this, an experimental metric using an LLM was used. For each binary, the decompiled code was presented to Google's AI Studio with a deterministic prompt (found in Appendix A). It was then compared if the response and qualitative analysis of the author overlap.

3 Results and Analysis

3.1 Baseline Analysis

The unobfuscated baseline binary was analyzed first to establish the reference values for all evaluation metrics.

Qualitative Analysis: Anatomy of the Baseline Binary

The flow is obvious and clear to follow. The programs core logic is identifiable without any problems as well as there is zero effort required to trace data flow between variables. Ghidra's analyzer was highly effective, providing a clear and readable output that can be found in Appendix A. The initial decompilation in Ghidra immediately reveals several key characteristics of a standard C++ binary:

C++ Name Mangling C++ compilers encode function signatures into cryptic strings. Using Ghidra's built-in demangler, the encoded C++ symbols were automatically resolved back to their readable forms (for instance, `checkPassword`).

Successful Type Propagation Ghidra also propagated C++ types. Instead of showing a generic stack buffer (`undefined1 auStack_30[28]`), the decompiler correctly identified the variable as a `std::string` object.

Boolean Logic Simple `if` statements are translated into efficient bitwise operations, such as checking the least significant bit.

External Imports Code from the C++ standard library is handled as imports. The `Imports` section in Ghidra lists these dependencies, such as `libc++.1.dylib` (the C++ Standard Library) and `libSystem.B.dylib` (the core macOS system library).

Data Section String literals are not in the code but are stored in a read-only data section. All six strings from the source code, including the password, were visible in clear text in this section.

To fully appreciate the value of Ghidra's automatic demangling, Table 1 dissects the original mangled string for the `checkPassword` function.

Mangled Component	Explanation
`_Z`	A standard prefix for a mangled name (Itanium ABI [10]).
`13checkPassword`	The function's name (`checkPassword`) prefixed by its length.
`R`	A code for a reference parameter (`&`).
`K`	A code for a const parameter.
`N...E`	A block indicating a nested name (e.g., a type in a namespace).
`St3__1`	An encoding for the `std::__1` namespace (used by Clang).
`12basic_string...`	The full template name for `std::string`.

Table 1: Demangling of the `checkPassword` Function Name

AI Analysis

The de-compiled baseline code was provided to the LLM along with the prepared prompt. The AI's response (see Appendix A) agreed with the qualitative analysis of the author.

Quantitative Metrics

The quantitative metrics for the baseline binary are summarized in Table 2.

Metric	**Value**
Total Cyclomatic Complexity	8
Total CFG Node Count	27
Visible String Count	6
Total Decompiler Line Count	109
Ghidra-Generated Variable Count	17

Table 2: Quantitative Metrics for the Baseline Binary

3.2 Analysis of Layout Obfuscation

The first obfuscation technique applied was layout obfuscation. As defined in the methodology, this was achieved by using the `strip` command to remove the symbol table from the compiled baseline binary. The primary goal of this technique is to remove human-readable function and variable names. The underlying code logic remains unchanged, so the quantitative metrics are expected to be nearly identical to the baseline.

Qualitative Analysis

After stripping the binary, all semantic identifiers for functions and variables were lost. Ghidra compensated by substituting these with generic, non-descriptive placeholders (e.g., `FUN_1000005e4`) as can be seen in Appendix A.

While the program's control flow and logic remained identical to the baseline, the decompiled code was no longer self-explanatory. The "at-a-glance" readability was significantly reduced, as it was not obvious what each function's purpose was, before opening them.

However, this technique was only a minor obstacle. A more detailed inspection quickly revealed a significant weakness: the string literals were not affected by the `strip` command. Readable strings like `Access Granted` and `Access Denied` remained intact, making it easier to infer the code's purpose. By examining the cross-references to these strings, you can easily deduce the purpose of their corresponding functions, thereby defeating this layer of obfuscation.

AI Analysis

The response from the AI (see Appendix A) strongly supports the findings of the qualitative analysis. Even with all function names stripped and replaced by generic placeholders, the AI rated the "Core Logic Identification" as easy.

Quantitative Metrics

As expected, since there was no code change, most of the metrics compared to the baseline have not changed at all. The decompiler line code reduced a bit. The results can be found in table 3

Metric	Baseline	Layout Obfuscation
Total Cyclomatic Complexity	8	8
Total CFG Node Count	27	27
Visible String Count	6	6
Total Decompiler Line Count	109	94
Ghidra-Generated Variable Count	17	17

Table 3: Quantitative Metrics for the Layout Obfuscated Binary

3.3 Analysis of Junk Code Insertion

The second technique applied was the junk code strategy, which was designed to survive compiler optimization and actively pollute the binary with complex, but irrelevant, code.
Unlike layout obfuscation, which only removed metadata, this technique was expected to have a significant and measurable impact on the binary's complexity. The hypothesis was that all quantitative metrics would increase (besides the String count), reflecting the new code bloat.

Qualitative Analysis

The before just reduced "at-a-glance" readability is now almost lost. The programs control flow, while still being manageably small, has significantly changed compared to the baseline. It is difficult to follow the control flow as it branches much more compared to the previous obfuscation. Since the string literals are not affected by any of the obfuscation techniques, it remains that while the control flow is much more difficult to follow, the correct path may be found quite quickly by finding the string literals. This demonstrates that while the junk code was effective at complicating the program's logic, it is not a complete solution. It must be combined with data obfuscation to fully hide the program's purpose.

AI Analysis

The AI's response (see Appendix A) provides strong qualitative evidence of the junk code's effectiveness. All ratings increased. The AI identified the `"modified global variable"` as a primary source of confusion. This is a direct reference to the `g_junkSink` anchor, confirming that the volatile sink not only defeats the compiler but also serves as a significant source of confusion for analysis.

Quantitative Metrics

The qualitative findings are further supported by the quantitative metrics (see table 4). Besides the visible strings, every category significantly increased.

Metric	Baseline	Layout Obfuscation	Junk Code Insertion
Total Cyclomatic Complexity	8	8	22
Total CFG Node Count	27	27	82
Visible String Count	6	6	6
Total Decompiler Line Count	109	94	228
Ghidra-Generated Variable Count	17	17	39

Table 4: Quantitative Metrics for the Layout and Junk Code obfuscated Binary

3.4 Analysis of Data Obfuscation

The third technique, Advanced Data Obfuscation, was applied on top of the existing junk-polluted and stripped binary. This step was designed to eliminate the single most critical weakness identified in both prior analyses: the presence of clear-text string literals. The hypothesis was that this would render all string-based analysis impossible, forcing any analyst (human or AI) to engage with the fully complexity of the junk code.

Qualitative Analysis

This technique was a complete success. An inspection of Ghidra's `Defined Strings` window, which was previously the primary help for finding the core logic, now shows no meaningful strings. The 6 critical strings have vanished. The impact on the analysis process is devastating. With the strings gone, the analyst is left with no "easy wins". Their only remaining path is to perform a full, brute-force analysis of the program's logic.

The clarity of the control flow is now incredibly difficult:

1. The analyst is forced to trace the program's logic from the `entry` function.
2. They must navigate the polluted, complex control flow from the junk code (step 2).
3. They will eventually find the decryption routine (the XOR loop).
4. To use it, they must find the key. They will discover the key is not a constant, but is built by the `getKey()` function.
5. To understand `getKey()`, they are forced to fully reverse engineer the opaque predicate (the hash, the magic numbers, etc.).

The analyst cannot decrypt the data (step 3) until they have completely defeated the junk code (step 2). The obfuscation layers now protect each other. However, this implementation is not infallible and introduces its own, new weakness: The `decrypt()` function. Since it is a simple and distinct XOR loop, it becomes the new central point of failure. A skilled human analyst would likely identify this characteristic function by its bitwise operations. Once identified, the analyst could simply place a breakpoint at the end of this function and dump all the decrypted strings from memory as they are created. This would then completely bypass the key finding complexity. This concludes that for a dynamic analysis, a more robust data obfuscation would be needed.

AI Analysis

The AI's response (see Appendix A) confirms this loss of context. All three qualitative ratings increased to Difficult or Extremely Difficult.

The justification text is the most telling part, as the AI was able to precisely identify every layer of obfuscation applied. In the previous steps, the AI could always bypass the logic by "cheating" with the string literals. With those strings removed, the AI was forced to analyze the junk-filled, complex logic head-on. As a result, it was unable to determine the program's true purpose.

Quantitative Metrics

The quantitative data in table 5 shows the impact of this technique. The most important change is in the Visible String Count, which has dropped from 6 to 0. In Addition, the Total Decompiler Line Count, Ghidra-Generated Variable Count and Cyclomatic Complexity increased again. This secondary effect is due to the introduction of the decryption function, the runtime key construction, and the large byte arrays, which all add to the code bloat and complexity an analyst must investigate.

Metric	Baseline	Layout	Junk	Data Obf.
Total Cyclomatic Complexity	8	8	22	32
Total CFG Node Count	27	27	82	138
Visible String Count	6	6	6	0
Total Decompiler Line Count	109	94	228	353
Ghidra-Generated Variable Count	17	17	39	73

Table 5: Quantitative Metrics for the Data Obfuscated Binary

3.5 Analysis of Control-Flow Flattening

The final technique applied was Control-Flow Flattening, which was layered on top of all previous obfuscations. This method replaces the core logic of the `checkPassword` function with a state machine.

Qualitative Analysis

This technique had a profound impact on code readability. At this point, neither the control flow nor the core functionality is findable without extensive analysis work. The `checkPassword` function, which was already polluted but still structurally intact, is now qualitatively unrecognizable. The decompiler output of the function `checkPassword()`(see Appendix A) compared to the version before (see A) has much more branching.

This forces a shift in the analyst's approach. They can no longer "read" the function. they must now meticulously trace the state variable and manually reconstruct the entire state machine diagram just to understand the original, simple logic.

Note that this flattening was deliberately applied only to the `checkPassword` function. The main `entry` function was left clear.

AI Analysis

The AI's response (see Appendix A) demonstrates the power of all techniques combined. The rating for Control Flow Clarity increased to Extremely Difficult, with the Core Logic Identification staying at Difficult as well as the Data Flow Tracing staying at Extremely Difficult.

The justification provided by the AI is a significant finding. It no longer offers a vague guess. It provides a high-level summary of the exact obfuscation strategy employed:

- It explicitly identifies the Control-Flow Flattening, describing it as a `"state machine-like do-while loop with non-obvious state transitions."`
- It explicitly identifies the Data Obfuscation, stating that the logic is hidden because `"all relevant strings are decrypted at runtime."`

This result proves the power of defense in depth. In the previous steps, the AI could always find a "thread to pull". First the function names then the string literals. With all simple identifiers and data hidden, the AI was forced to confront the spaghetti code of the CFF and the noise of the junk code.

Quantitative Metrics

The quantitative data in Table 6 presents a critical finding. While the metrics for Total Decompiler Line Count and Total Cyclomatic Complexity did increase, the magnitude of this increase is notably smaller than the "explosion" caused by the junk code and data obfuscation steps. The reason for this is that the previous two steps added a significant volume of new code. In contrast, the CFF step did not add much new code. It primarily transformed the existing `checkPassword` logic. It replaced a simple for-loop with a state machine. This resulted in a relatively low net increase in lines and nodes.

This finding highlights the severe limitations of purely quantitative metrics. While the CFF step's numerical impact was the smallest, its qualitative impact was huge.

Metric	Baseline	Layout	Junk	Data	CFF
Total Cyclomatic Complexity	8	8	22	32	37
Total CFG Node Count	27	27	82	138	152
Visible String Count	6	6	6	0	0
Total Decompiler Line Count	109	94	228	353	374
Ghidra-Generated Variable Count	17	17	39	73	76

Table 6: Final Quantitative Metrics Comparison Across All Techniques

3.6 Analysis of Automated DeClang Obfuscation

This final binary represents the "all-in-one" automated approach, compiled with all logical obfuscations from DeClang.

Qualitative Analysis

The result is a binary that is, from a pure logical perspective, almost completely unanalyzable. Every single function was transformed into a complex, multi-state state machine. The decompiler output is a mix of `while(true)` loops and `switch` statements, making a manual trace of the logic a prohibitively time-consuming and tedious task.

However, this logical "fortress" highlights a critical limitation of relying solely on logical obfuscation. DeClang, as a tool focused on logic, is not designed to obfuscate the program's data section. As a result, all the original plain-text strings were still present and readable. This shows that, despite extensive logical obfuscation, accessible data such as unencrypted strings can still guide an analyst's interpretation.

AI Analysis

The AI analysis (see A) further supports the qualitative impression of the author. While the ratings for Control Flow Clarity and Data Flow Tracing are Extremely Difficult, it was able to identify the core logic, most likely due to the literal strings still fully visible and not encrypted.

Quantitative Metrics

An interesting finding is revealed when comparing the quantitative impact of the manual CFF versus the automated one in table 7. While the manual CFF increased the Total CFG Node Count to 152 and the Total Cyclomatic Complexity to 37, DeClang's approach yielded a similar Total CFG Node Count (159) but a monumental Total Cyclomatic Complexity (182). This discrepancy gives an insight into the tool's methodology. It suggests that while both techniques flatten the program into a state machine (resulting in a similar number of states or nodes), the DeClang binary is far more complex. DeClang first injects massive amounts of spaghetti code via Bogus Control-Flow (BCF) and Instruction Substitution, and then flattens that polluted logic. The manual approach, by contrast, only flattened the clean logic. This results in an automated binary where each state in the state machine is, on its own, a convoluted and complex block of logic, exploding the cyclomatic complexity.

Metric	Baseline	Layout	Junk	Data	CFF	DeClang
Total Cyclomatic Complexity	8	8	22	32	37	182
Total CFG Node Count	27	27	82	138	152	159
Visible String Count	6	6	6	0	0	6
Total Decompiler Line Count	109	94	228	353	374	837
Ghidra-Generated Variable Count	17	17	39	73	76	117

Table 7: Final Quantitative Metrics Comparison Across All Techniques

4 Conclusion

The results of this analysis confirm that effective code obfuscation is a game of economic attrition. The goal is not to create an unbreakable binary, but one that is too costly, time-consuming, and complex for an attacker to analyze. This report found that isolated, simple techniques fail this objective. Layout Obfuscation was a trivial hurdle. Even the more sophisticated Junk Code, which dramatically increased quantitative complexity, was strategically useless as long as clear-text strings provided a fast lane straight to the core logic for both the human analyst and the AI.

The defensive capabilities shifted fundamentally with the introduction of a manual, synergistic strategy. By encrypting all string literals (Data Obfuscation), the path of least resistance was removed. By protecting the decryption key with the opaque predicates from the Junk Code layer, an attacker is forced to engage with the junk code to find the key. The final addition of Control-Flow Flattening completed this defense by hiding the logic itself, forcing the attacker to analyze and defeat every single layer in sequence.

This manual strategy was then compared against the automated DeClang obfuscator, which yielded a interesting finding. The automated tool was demonstrably superior at brute-force logical obfuscation. The metric analysis revealed that while the manual CFF and DeClang produced a similar number of CFG nodes (152 vs. 159, respectively), DeClang's Total Cyclomatic Complexity exploded to 182, compared to the manual approach's 37. This discrepancy suggests DeClang's synergistic passes, such as Bogus Control-Flow and Instruction Substitution, heavily polluted the logic before flattening it. Consequently, each resulting state becomes a dense, interwoven block of logic, resembling classic spaghetti code.

This automated approach, is not a complete solution by itself. DeClang (by design) focuses on logical obfuscation and does not attempt to obfuscate data, leaving all 6 strings in plain text . This is not a vulnerability of the tool, but rather a crucial clarification of its capabilities. It highlights that even a perfectly obfuscated logical flow can be rendered analyzable if plain-text data is left as an anchor. The manual analysis was invaluable not for competing with the tool, but for identifying the critical, separate domain of data obfuscation that most automated logical obfuscators do not cover.

In conclusion, this project proves that the most effective, real-world strategy is not a competition between manual and automated techniques, but a synergy of both. An automated tool like DeClang provides a "brute-force" logical fortress that is unfeasible to replicate by hand . This fortress should then be combined with a separate, human-driven data obfuscation strategy to protect the "easy win" vulnerabilities. This true Defense in Depth, which combines automated logical complexity with targeted data-hiding, is the most effective approach . It successfully transforms the task of reverse engineering from a simple puzzle into a "needle in a haystack" problem, hopefully raising the cost of analysis far beyond the potential reward.

References

[1] Collberg, C., Thomborson, C., and Low, D., "A taxonomy of obfuscating transformations," *Proceedings of the 14th Annual Computer Security Applications Conference*, IEEE, 1997, pp. 3–12.

[2] Sikorski, M. and Honig, A., *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, No Starch Press, 2012.

[3] Free Software Foundation, "GNU Binutils: strip," https://sourceware.org/binutils/docs/binutils/strip.html, [Accessed 15-10-2025].

[4] Lomont, C., "Fast inverse square root," Technical report, Purdue University, 2003.

[5] Wheeler, D. J. and Needham, R. M., "TEA, a tiny encryption algorithm," *Fast Software Encryption: Second International Workshop*, Springer, 1994, pp. 363–366.

[6] "Magic number in boost::hash_combine — stackoverflow.com," https://stackoverflow.com/questions/4948780/magic-number-in-boosthash-combine, [Accessed 16-10-2025].

[7] Katz, J. and Lindell, Y., *Introduction to Modern Cryptography*, CRC Press, 2020.

[8] "GitHub - DeNA/DeClang: An anti-hacking compiler forked from the ollvm (https://github.com/obfuscator-llvm/obfuscator) — github.com," https://github.com/DeNA/DeClang, [Accessed 28-10-2025].

[9] Junod, P., Rinaldini, J., Wehrli, J., and Michielin, J., "Obfuscator-LLVM – Software Protection for the Masses," *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, edited by B. Wyseur, IEEE, 2015, pp. 3–9.

[10] Multiple Authors, "Itanium C++ ABI," https://itanium-cxx-abi.github.io/cxx-abi/abi.html, 2001, [Accessed 12-10-2025].

A Appendix

The appendix contains some code and ghidra excerpts. The full output can be found within the files as it was too long to include here.

Full Source Code for Baseline Program

Here is the complete C++ source code for the baseline password checker used for this report's analysis.

Baseline C++ Source Code

```cpp
int main() {
    std::cout << "Please enter the password to access the secure system: ";
    std::string userInput;
    std::getline(std::cin, userInput);

    if (checkPassword(userInput)) {
        accessGranted();
    } else {
        accessDenied();
    }

    return 0;
}

void accessGranted() {
    std::cout << "Access Granted. Welcome to the secure area." << std::endl;
    std::string secretMessage = "The nuclear launch codes are: Hamburger Banana
    ";
    std::cout << "Secret Information: " << secretMessage << std::endl;
}

void accessDenied() {
    std::cout << "\n[-] Access Denied. Incorrect password." << std::endl;
}

bool checkPassword(const std::string& userInput) {
    const std::string correctPassword = "cysLFHNW25!123";

    if (userInput.length() != correctPassword.length()) {
        return false;
    }

    for (int i = 0; i < correctPassword.length(); ++i) {
        if (userInput[i] != correctPassword[i]) {
            return false;
        }
    }
    return true;
}
```

Ghidra Decompiled Output

Baseline

This section contains the full, unmodified output from the Ghidra decompiler for the baseline binary with name mangleling.

Ghidra Decompiled Code

```
undefined4 entry(void)

{
    uint uVar1;
    undefined1 auStack_30 [28];
    undefined4 local_14;

    local_14 = 0;
    __ZNSt3__11sB8ne200100INS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EES6_PKc
              (PTR___ZNSt3__14coutE_100004060,
               "Please enter the password to access the secure system: ");
    __ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEC1B8ne200100Ev();
    __ZNSt3__17getlineB8ne200100IcNS_11char_traitsIcEENS_9allocatorIcEEEERNS_13basic_istrea
     _RNS_12basic_stringIS6_S7_T1_EE
              (PTR___ZNSt3__13cinE_100004058,auStack_30);
    uVar1 =
    __Z13checkPasswordRKNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEE
                      (auStack_30);
    if ((uVar1 & 1) == 0) {
        __Z12accessDeniedv();
    }
    else {
        __Z13accessGrantedv();
    }
    local_14 = 0;
    __ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEED1Ev(auStack_30);
    return local_14;
}
```

Ghidra Decompiled Code Demangled

```
undefined4 entry(void)

{
  uint uVar1;
  string asStack_30 [28];
  undefined4 local_14;

  local_14 = 0;
  std::TEMPNAMEPLACEHOLDERVALUE<[abi:ne200100]<std::operator<
            ((ostream *)PTR_cout_100004060,"Please enter the password to access the secure system: "
            );
  std::string::string[abi:ne200100](asStack_30);
  std::getline[abi:ne200100]<>((istream *)PTR_cin_100004058,asStack_30);
  uVar1 = checkPassword(asStack_30);
  if ((uVar1 & 1) == 0) {
    accessDenied();
  }
  else {
    accessGranted();
  }
  local_14 = 0;
  std::string::~string(asStack_30);
  return local_14;
}
```

Ghidra Output State Machine

Both an example from the state machine as well as the same code without the state machine

Ghidra Decompiled Partial State Machine Code

```
//...
  if ((uVar4 & 1) == 0) {
    local_9d = false;
    local_a4 = 0;
    local_b0 = 0;
    do {
      while( true ) {
        while( true ) {
        if (local_a4 != 1) break;
        uVar5 = FUN_1000016f8(asStack_78);
        if (local_b0 < uVar5) {
          pcVar6 = (char *)FUN_100001668(local_b0 - uVar5,local_48,local_b0);
          cVar1 = *pcVar6;
          pcVar6 = (char *)FUN_1000016b0(asStack_78,local_b0);
          if (cVar1 == *pcVar6) {
            local_b0 = local_b0 + 1;
            local_a4 = 1;
          }
          else {
            local_9d = false;
            local_a4 = 99;
          }
        }
        else {
          local_9d = true;
          local_a4 = 99;
        }
      }
    } while (local_a4 != 99);
    DAT_100008000 = DAT_100008000 ^ 0xbeef;
    local_39 = local_9d;
  }
  else {
    local_90 = 0;
    local_98 = 0;
    while( true ) {
      uVar5 = FUN_100000bd8(local_48);
      bVar2 = false;
      if (local_98 < uVar5) {
        uVar5 = FUN_100000bd8(asStack_78);
        bVar2 = local_98 < uVar5;
      }
      if (!bVar2) break;
      pcVar6 = (char *)FUN_100001668(local_48,local_98);
      cVar1 = *pcVar6;
      pcVar6 = (char *)FUN_1000016b0(asStack_78,local_98);
      local_90 = local_90 + (ulong)(cVar1 == *pcVar6);
      local_98 = local_98 + 1;
    }
//...
```

Ghidra Output checkPassword function without State Machine

Ghidra Decompiled Partial normal checkPassword funciton

```
// ...
if ((uVar4 & 1) == 0) {
    lVar7 = FUN_100001610(local_38);
    lVar8 = FUN_100001610(asStack_68);
    if (lVar7 == lVar8) {
      local_90 = 0;
      while( true ) {
        uVar5 = FUN_100001610(asStack_68);
        if (uVar5 <= (ulong)(long)local_90) break;
        pcVar6 = (char *)FUN_100001580((long)local_90 - uVar5,local_38,(long) local_90);
        cVar1 = *pcVar6;
        pcVar6 = (char *)FUN_1000015c8(asStack_68,(long)local_90);
        if (cVar1 != *pcVar6) {
          local_29 = false;
          goto LAB_100000fac;
        }
        local_90 = local_90 + 1;
      }
      local_29 = true;
    }
    else {
      local_29 = false;
    }
  }
  else {
    local_80 = 0;
    local_88 = 0;
    while( true ) {
      uVar5 = FUN_100000bd8(local_38);
      bVar2 = false;
      if (local_88 < uVar5) {
        uVar5 = FUN_100000bd8(asStack_68);
        bVar2 = local_88 < uVar5;
      }
      if (!bVar2) break;
      pcVar6 = (char *)FUN_100001580(local_38,local_88);
      cVar1 = *pcVar6;
      pcVar6 = (char *)FUN_1000015c8(asStack_68,local_88);
      local_80 = local_80 + (ulong)(cVar1 == *pcVar6);
      local_88 = local_88 + 1;
    }
// ...
```

Layout Obfuscation

Ghidra Decompiled Layout Obfuscation Code

```

undefined4 entry(void)

{
  uint uVar1;
  string asStack_30 [28];
  undefined4 local_14;

  local_14 = 0;
  FUN_10000059c(PTR_cout_100004060,"Please enter the password to access the
    ↪ secure system: ");
  FUN_1000005e4();
  FUN_100000610(PTR_cin_100004058,asStack_30);
  uVar1 = FUN_100000668(asStack_30);
  if ((uVar1 & 1) == 0) {
    FUN_100000848();
  }
  else {
    FUN_100000794();
  }
  local_14 = 0;
  std::string::~string(asStack_30);
  return local_14;
}
```

AI Studio Prompt for Code Analysis

This prompt was used for AI studio. The C-like code was pasted below it.

AI Studio Prompt for Code Analysis

You are an expert reverse engineer and security analyst specializing in static analysis of C code. You will be given a function's source code that has been generated by the Ghidra decompiler from a C++ binary. This code may contain non-standard data types (like `undefined4`), mangled function names, and low-level programming constructs.

Your task is to analyze only the provided code and return a report in the following strict format. You must choose one of the following ratings for each metric: **Easy**, **Moderate**, **Difficult**, or **Extremely Difficult**.

---

1. Control Flow Clarity:
[Your Rating]
2. Core Logic Identification:
[Your Rating]
3. Data Flow Tracing:
[Your Rating]
4. Justification:
[A one to two-sentence justification for your ratings.]

CODE FOR ANALYSIS:
`[PASTE DECOMPILED C CODE HERE]`

AI Studio Response for Baseline

Control Flow Clarity: Easy Core Logic Identification: Easy Data Flow Tracing: Easy Justification: The function exhibits a simple, linear control flow with a single if/else branch determining the outcome. The core logic is immediately identifiable as a standard password authentication routine, and tracing the user-provided string from input to the validation function is trivial.

AI Studio Response for Layout Obfuscation

- Control Flow Clarity: Easy
- Core Logic Identification: Easy
- Data Flow Tracing: Easy

Justification: The core logic is a simple password check, made obvious by explicit string literals, but the control flow within the checking function (`FUN_100000668`) is slightly obscured by a while(true) loop and a goto. Despite the generic function names, tracing the user's input from the prompt to the character-by-character comparison is straightforward.

AI Studio Response for Junk Code Insertion

- Control Flow Clarity: Moderate
- Core Logic Identification: Difficult
- Data Flow Tracing: Extremely Difficult

Justification: The overall control flow is discernible with standard conditional branches and loops, but the core password-checking logic is obfuscated by complex arithmetic and bitwise operations. Tracing data is extremely difficult due to the heavy use of a global variable modified in multiple, non-obvious ways across different functions, combined with calculations involving magic numbers.

AI Studio Response for Data Obfuscation

- Control Flow Clarity: Difficult
- Core Logic Identification: Difficult
- Data Flow Tracing: Extremely Difficult

Justification: The core logic is heavily obfuscated through runtime string decryption, and the control flow is complicated by conditional branches that select between different complex algorithms for processing. Tracing data is extremely difficult due to the runtime generation of comparison values and constant, intricate modifications to a global state variable.

AI Studio Response for Control Flow Flattening

- Control Flow Clarity: Extremely Difficult
- Core Logic Identification: Difficult
- Data Flow Tracing: Extremely Difficult

Justification: The control flow is deliberately convoluted, especially within the main checking function which employs a state machine-like do-while loop with non-obvious state transitions, making the execution path exceptionally hard to follow. Identifying the core logic is difficult because all relevant strings are decrypted at runtime, and a comparison logic is hidden within the complex state machine.

AI Studio Response for DeClang

- Control Flow Clarity: Extremely Difficult
- Core Logic Identification: Difficult
- Data Flow Tracing: Extremely Difficult

Justification: The control flow is an obfuscated state machine driven by magic numbers, making the execution path extremely difficult to follow. The core logic of a password check is heavily fragmented across numerous states, requiring tedious analysis to trace the user input to each scattered comparison point.