Assignment 5: Fortify
(CS 6510)

`

Submitted To:
Prof. Chester Rebeiro
Department of
Computer Science Engineering

Submitted By:
Arnav Karn (CS24M801)
Dipanshu Kumar (CS24M019)
Team: Kavach

1. Introduction

The goal of this project was to apply custom code obfuscation techniques to protect sensitive data and logic within a C program (`main.c`). This included concealing AES keys, egg parameters, and global flag calculations. The final deliverable was a statically compiled, self-contained ELF binary (`safe_main`) that produces the same output as the original program while being resistant to reverse engineering.

To achieve this, I explored various code obfuscation strategies and multi-layer binary encryption methods.

---

2. Objective

To implement a series of obfuscation strategies to make an AES-based C program more resistant to reverse engineering. This involves renaming identifiers, encrypting sensitive constants, and introducing code constructs that confuse automated tools and human analysts alike.

---

3. Literature Review

To begin, I referred to academic papers and resources on software obfuscation. Some useful papers I explored include:

1. **"Software Protection through Program Obfuscation" – Collberg et al.**
2. **"A Taxonomy of Obfuscating Transformations" – Collberg et al.**
3. **"Obfuscating C++ Programs via Control Flow Flattening" – László and Kiss**

These papers introduced several techniques such as:

- Control flow flattening
- Opaque predicates
- Dummy code paths
- String/data obfuscation
- Code and data encryption
- Junk code insertion

---

3. Overview of AES Implementation

The base implementation is a simplified AES encryption routine, including:

- Key expansion
- Round-based transformations (SubBytes, ShiftRows, MixColumns)
- XOR-based operations
- Use of S-box for substitution

The original code was written in a clean, readable, and modular fashion, making it a perfect candidate for complex obfuscation.

Analysis of main.c

After analyzing the provided `main.c` file, I identified key sensitive components:

- **AES Key**
- **Egg parameters**
- **Global flag calculations**

These components needed to be hidden or made harder to reverse-engineer.

---

4. Obfuscation Strategies inside main.c

4.1 Lyric-Based Identifier Obfuscation

Technique

All program identifiers (variables, functions, macros) were replaced with phrases from Hindi/English songs.

Implementation Examples

```
#define never_looking_back 4          // From "Closer" by The
Chainsmokers(AES block size)
void pull_me_closer(uint8_t roomate) // From "Closer" by The
Chainsmokers (bitwise decoder)
```

```
uint8_t abhi_na_jao(uint8_t idx)     // From "Abhi Na Jao" (S-box lookup)
```

Effectiveness

- Breaks semantic correlation between names and functionality
- Creates false lexical patterns (e.g., `bhak[]` array name means "go" but stores egg values)
- Requires cultural knowledge to interpret lyrics reference

4.2 Junk Functions and Dead Code

Several unused functions were added to simulate meaningful logic while having no real purpose. These functions are never or sometimes called but contribute to binary bloat and misleading disassemblers.

4.3 S-Box Obfuscation

Technique

Multi-layer encoding using XOR, bit rotation, and custom indexing.

Code Implementation

```
uint8_t abhi_na_jao(uint8_t idx) {
    uint8_t id = humsafar[idx];          // First-layer permutation (humsafar[] map)
    uint8_t val = abhi_abhi[id];         // Second-layer substitution (abhi_abhi[] S-box)
    val = (val >> 3) | (val << 5);       // Custom rotation
    return val ^ city_broke_down;        // Final XOR with 0x5A
}
```

Key Components

- `humsafar[256]`: Permutation table for index scrambling
- `abhi_abhi[256]`: Core S-box values
- `res[256]` array

Security Properties

1. Three transformation layers prevent direct S-box extraction
2. Non-linear composition of operations defeats pattern analysis
3. Index-dependent computation prevents batch processing

4.4 Key Schedule Obfuscation

Technique

On-demand key expansion with bitwise transformations.

Implementation Workflow

1. **Initial Key Processing**

```
memories[(i*4)+0] = pull_me_closer(key[(i*4)+0]);  // Uses ~, XOR 0x5A, rotation
```

2. **Round Key Generation**

```
tempa[0] = abhi_na_jao(tempa[1]);  // Uses S-box logic
tempa[0] ^= raat_baaki[i/4];       // XOR with round constants
```

3. **Key Chaining**

```
memories[j+0] = memories[k+0] ^ tempa[0];  // XOR-chain propagation
```

Anti-Analysis Features

- Keys never exist in plaintext form
- Mixed bitwise operations prevent algebraic simplification
- Round constants hidden as `raat_baaki[]` (0x8d,0x01...)

4.5 Easter Egg Parameter System

Data Flow Architecture

```
abhi_params[5][6] → abhi_decode() → Control Parameters → bhak[] Array → bose_rockstar()
```

Critical Components

1. **Obfuscated Parameter Store**

```
uint8_t abhi_params[5][6] = {0xb5,0xb5...};  // Encoded control values
```

2. **Runtime Decoding**

```
r = abhi_decode(abhi_params[channa][0]);  // Applies ~, XOR 0x5A, rotation
```

3. **Dynamic Value Collection**

```
bhak[channa++] = (*state)[i][j] ^ (*state)[l][m];  // XOR-state sampling
```

4.6 GF Calculation Obfuscation

```
uint8_t bose_rockstar(uint8_t a, uint8_t b) {
    uint16_t prod = 0;
    // Schoolbook multiplication with carry handling
    while (partial != 0) {
        carry = prod & partial;
        prod = prod ^ partial;
        partial = carry << 1;
    }
    return (uint8_t)prod ^ dk_bose[c];  // Final XOR with lookup
}
```

Obfuscation Layers

- Manual bitwise multiplication implementation
- Dynamic index `c = dk_bose[bhak[humsafar[210]]]`
- Mixed arithmetic/bitwise operations

4.7 Control Flow Obfuscation

Encrypted Workflow

```
void abhi_cipher(...) {
    abhi_add_key(0,state,memories);
    for(uint8_t round=1;;++round){
        // Parameter-controlled operations
        if(op == 1 && r == round) ...  // Dynamic behavior selection
        abhi_shift_rows(state);
        if(op == 2 && r == round) ...
        abhi_mix_columns(state);
        if(op == 3 && r == round) ...
        abhi_add_key(round,state,memories);
        if(op == 4 && r == round) ...
    }
}
```

Key Features

- Round-dependent operation triggering via decoded parameters
- State-dependent value collection (`bhak[]`)

Non-linear loop exit condition (`round==bose_bhaag`)

4.8. Anti-Debugging Techniques

1. State-Dependent Caching

```
static uint8_t rEs[256] = {0};  // Tracks initialized S-box values
if (!rEs[idx]) { ... }          // Prevents memory pattern analysis
```

2. Fragmented Memory Storage

- Egg indexes values of (x y z) from gf calculation comes as a value sttored in different arrays

3. Bitwise Arithmetic

- Uses non-standard operations:
  `(val << 2) | (val >> (3*2))` instead of standard rotation

4.9. Full Obfuscation Matrix

Component	Techniques Used	Protection Level
Identifiers	Lyric-based renaming	Medium
S-box	XOR + Rotation + Index permutation + Lookup	High
Key Schedule	Bitwise ops + Round constants	High
GF Calculation	Manual implementation + Lookup XOR	Medium-High
Control Flow	Parameter-driven operations	High
Constants	Multi-layer encoding	High

4.10. Validation

- Consistent ciphertext generation confirms functional integrity
- Easter egg parameters modify execution without affecting core output
- Python code was generated to check if obfuscation doesn't change the original functional logic

---

4.11. Security and Reverse Engineering Implications

- **Static Analysis Resistance**: Tools like Ghidra face difficulties due to junk code and lookup indirections.
- **Key Extraction Difficulty**: The actual AES key never appears in memory in plaintext.
- **Cognitive Overload**: Lyrics-based identifiers break standard naming conventions, slowing human reverse engineers.
- **Symbolic Execution Hardening**: Obfuscated control flow and heavy use of indirect addressing impede symbolic tools.

4.12. Challenges Faced

- Mapping song lyrics to plausible but meaningless C identifiers.
- Maintaining functional equivalence post-renaming.
- Ensuring that the junk code did not interfere with stack/memory behavior.
- Preserving 64-bit ELF binary compliance under all transformations.

5. Obfuscation strategy outside main.c

5.1 Key Obfuscation

Initial Attempt

I initially attempted to obfuscate the AES key using a chain of function calls to reconstruct the key at runtime. However, this method was too straightforward and did not significantly improve security.

Improved Strategy

I developed a **`key_generator.c`** program that:

- Accepts three XOR keys as inputs
- Produces an encrypted version of the AES key
- Outputs the result to `stdout`

In `main.c`, I used **pipes** to retrieve this key at runtime.

Using Pipes

A pipe in Unix allows inter-process communication. I created a child process that runs `key_generator`, then uses `popen()` and `fgets()` to capture the output in `main.c`, dynamically setting the AES key without embedding it in the code.

5.2. Bogus Functions and Control Flow Confusion

To further protect the key logic, I added:

- **Five fake key-related functions**, only one of which computes the real key.
- The other four include **opaque predicates** (e.g., `if (0 && some_condition)`), which mimic real logic but never execute.

I followed the same strategy for **egg parameters** using an `egg_generator.c` program and piped its output similarly into `main.c`.

5.3. Junk Code and Variable Renaming

- I inserted multiple **junk code blocks** that perform meaningless operations to confuse attackers.
- All variables were renamed using names of **animals and birds** to reduce semantic clarity.
- Multiple layers of **false `if` statements** and misleading logic were added inside functions dealing with keys and eggs.

5.4. Self-Decrypting Loader Chain

Inspired by class discussions on **self-extracting stubs**, I implemented a multi-stage binary loader pipeline:

Stage 1: `encrypt.c`

- Compiles and runs a C program that encrypts the `main` binary (`a.out`)
- Saves the output to `encrypted_a.out`

Stage 2: Embedding Binary into Source Code

- I used the `xxd` tool:
  xxd -i encrypted_a.out > payload.h
- This converts the binary into a byte array, which is embedded into `loader.c`

Stage 3: Loader Implementation

- **`loader.c`** includes `payload.h` and:
  - Decrypts the embedded binary
  - Writes it to a temporary file
  - Executes it at runtime
- This binary behaves identically to the original `main.c`

5.5. Additional Obfuscation in Loader

Upon analyzing `final_encrypt.out` in Ghidra, I noticed that it mostly revealed the loader logic. To confuse reverse engineers:

- I embedded a **clone of the obfuscated `main.c`** logic into `loader.c`
- I included **fake keys and eggs** to further mislead attackers

---

5.6. Multi-layer Encryption/Decryption Chain

To enhance security, I created:

- `encrypt_L2.c`: encrypts `encrypt.c`'s output
- `encrypt_L3.c`: encrypts `encrypt_L2.c`'s output
- `loader_L2.c`: decrypts `loader.c`'s output
- `loader_L3.c`: decrypts `loader_L2.c`'s output

Each encryptor simply encrypts the previous layer and each loader simply decrypts the previous layer and runs it, forming a **3-layer self-decrypting binary chain**.

5.7. Final Layer – Shift Cipher + XOR

In the final layer (`loader_L3.c`), I added:

- **Shift cipher encryption**: each byte is shifted by a constant
- **XOR encryption**: data is XORed with a dynamic key

This combined cipher technique makes static analysis even harder.

5.8. Obfuscation Automation Script

To streamline the build and encryption process, I created a shell script run.sh that automates all the steps:

```
#!/bin/bash

# Exit on any error
set -e

# Step 1: Compile key generator and egg encryptor
gcc -o keygen_encryptor keygen_encryptor.c
./keygen_encryptor
gcc -o egg_encryptor egg_encryptor.c
./egg_encryptor

# Step 2: Compile main program
gcc main.c -o a.out

# Step 3: First level encryption
gcc encrypt.c -o encrypt
./encrypt
xxd -i encrypted_a.out > payload.h

# Step 4: Loader for level 1
gcc loader_keygen_encryptor.c -o loader_keygen_encryptor
./loader_keygen_encryptor
gcc loader.c -o final_encrypt.out
./final_encrypt.out abc

# Step 5: Second level encryption
gcc encrypt_L88.c -o encrypt_L88
./encrypt_L88
xxd -i encrypted_a.out_L88 > payload_L88.h
```

```
# Step 6: Loader for level 2
gcc loader_keygen_encryptor_L88.c -o loader_keygen_encryptor_L88
./loader_keygen_encryptor_L88
gcc loader_L88.c -o final_encrypt_L88.out
./final_encrypt_L88.out abc

# Step 7: Third level encryption
gcc encrypt_L89.c -o encrypt_L89
./encrypt_L89
xxd -i encrypted_a.out_L89 > payload_L89.h

# Step 8: Compile final level encrypted binary
gcc loader_L89.c -o safe_main
./safe_main abc
```

Explanation Step 1: These lines compile the key and egg encryptor programs (`keygen_encryptor.c` and `egg_encryptor.c`) into executables `keygen_encryptor` and `egg_encryptor`.

Explanation Step 2: This line compiles the `main.c` program into an executable called `a.out`.

Explanation Step 3:

- The `encrypt.c` is compiled into an executable called `encrypt`.
- `./encrypt` runs the encryption on `a.out` (the main binary), producing `encrypted_a.out`.
- The `xxd -i` command converts the `encrypted_a.out` binary into a C header (`payload.h`), which can be embedded in the loader code.

Explanation Step 4:

- This compiles the loader program (`loader_keygen_encryptor.c`) into an executable.
- The `loader.c` is compiled into `final_encrypt.out`.

- `./final_encrypt.out abc` runs the loader, which decrypts the encrypted binary and executes it at runtime.

Explanation Step 5:

- Compiles `encrypt_L2.c` into an executable (`encrypt_L2`).
- `./encrypt_L2` encrypts the previously encrypted binary (`a.out`).
- Converts the new binary (`encrypted.a.out_L2`) to C header `payload_L2.h` for embedding in the next loader.

Explanation Step 6:

- Compiles the second-level loader (`loader_keygen_encryptor_L2.c`).
- Compiles `loader_L2.c` to produce `final_encrypt_L2.out`.
- `./final_encrypt_L2.out abc` runs the loader, which decrypts and executes the second layer of encrypted binary.

Explanation Step 7:

- Compiles `encrypt_L3.c` into `encrypt_L3`.
- `./encrypt_L3` encrypts the binary from the second layer (`encrypted.a.out_L2`).
- Converts the final encrypted binary (`encrypted.a.out_L3`) into C header `payload_L3.h` for embedding in the third loader.

Explanation Step 8:

- Compiles the final loader (`loader_L3.c`) into `final_encrypt_L3.out`.
- Executes `final_encrypt_L3.out` which decrypts and runs the third layer of encrypted binary.

---

6. Observations and Results

- The final `safe_main` binary is **under 5 MiB**, **statically compiled**, and **produces the correct output**.
- It passes all test cases and completes within **2 seconds**.
- Static analysis tools (like Ghidra) reveal misleading structures, junk logic, and bogus key paths.
- The loader chain and pipe-based key/egg loading increase the difficulty of reverse engineering substantially.

---

7. Dockerfile

During the development and testing of our obfuscated binary using Docker, we encountered issues when attempting to copy the final output binary (`safe_main`) from the container to the host machine using volume mounts. Despite ensuring correct paths and permissions, the expected file did not appear in the mounted directory on the host system.

To resolve this, we adopted an alternate and reliable method involving the use of `docker cp`:

1. Docker Image Build
   We built the Docker image from our Dockerfile:

   docker build -t img.

2. Run the Container

   We executed the container:

   docker run img

3. Retrieve Container ID

   After the container finished execution, we retrieved its ID using:

   docker ps -a

4. Copy Binary from Container to Host

   We then used the following command to manually extract the binary:

   docker cp container_id :path in container path in host

```
PS C:\Users\Arnav\Downloads\finalSafeMain> docker ps -a
CONTAINER ID   IMAGE          COMMAND       CREATED              STATUS                          PORTS     NAMES
6d642d52601f   img            "/bin/bash"   About a minute ago   Exited (0) 31 seconds ago                 suspicious_hodg
kin
4eeeb1d1c07b   img            "/bin/bash"   2 minutes ago        Exited (0) 2 minutes ago                  vibrant_noyce
716716e52d5b   img            "/bin/bash"   6 minutes ago        Exited (0) 6 minutes ago                  upbeat_edison
ea6ef57cd85a   71064105bf3a   "/bin/bash"   8 minutes ago        Exited (0) 7 minutes ago                  gifted_antonell
i
996399f6327c   71064105bf3a   "/bin/bash"   15 minutes ago       Exited (127) 12 minutes ago               trusting_roentg
en
d8409d889204   349f0a585196   "/bin/bash"   3 hours ago          Exited (0) 2 hours ago                    laughing_moser
0974f09cc5f8   6e981bb4caf1   "/bin/bash"   3 hours ago          Exited (0) 3 hours ago                    cool_visvesvara
ya
ce480959e96e   6e981bb4caf1   "/bin/bash"   3 hours ago          Up 3 hours                                goofy_margulis
PS C:\Users\Arnav\Downloads\finalSafeMain> docker cp 6d642d52601f:/workspace/output C:\Users\Arnav\Downloads\finalSafeMa
in
Successfully copied 99.8kB to C:\Users\Arnav\Downloads\finalSafeMain
PS C:\Users\Arnav\Downloads\finalSafeMain>
```

5. We have tested our binary **safe_main** in the docker's workspace using our **test_runner.py**.

```
root@6d642d52601f:/workspace# python3 test_runner.py

Plaintext: abcdefghijklmnopq
Expected CipherText: f65458304cc7bc881431e5b8781d96f7
Actual   CipherText: f65458304cc7bc881431e5b8781d96f7
Expected Egg 0      : 0x20
Actual   Egg 0      : 0x20
Expected Flag       : 0x5f
Actual   Flag       : 0x5f
Time of execution   : 0.019859 seconds
Match Result        : ✅ Matched

Plaintext: qrstuvwxyzabcdef
Expected CipherText: e4cdd1827eca97f616a6208a1851a242
Actual   CipherText: e4cdd1827eca97f616a6208a1851a242
Expected Egg 0      : 0xf2
Actual   Egg 0      : 0xf2
Expected Flag       : 0xc3
Actual   Flag       : 0xc3
Time of execution   : 0.009404 seconds
Match Result        : ✅ Matched

Plaintext: 1234567890abcdef
Expected CipherText: 7c1d469b54f224627817eb9ff260b7cb
Actual   CipherText: 7c1d469b54f224627817eb9ff260b7cb
Expected Egg 0      : 0x79
Actual   Egg 0      : 0x79
Expected Flag       : 0x60
Actual   Flag       : 0x60
Time of execution   : 0.012602 seconds
Match Result        : ✅ Matched

Plaintext: abcdef1234567890
Expected CipherText: d9678d4ed4f7400dca88910b9ccec573
Actual   CipherText: d9678d4ed4f7400dca88910b9ccec573
Expected Egg 0      : 0x9f
Actual   Egg 0      : 0x9f
Expected Flag       : 0x9a
Actual   Flag       : 0x9a
Time of execution   : 0.005974 seconds
Match Result        : ✅ Matched

Plaintext: ghijklmnopqrstuv
Expected CipherText: 258486e57e845a7c23ffd7da3f39ebad
Actual   CipherText: 258486e57e845a7c23ffd7da3f39ebad
Expected Egg 0      : 0x74
Actual   Egg 0      : 0x74
Expected Flag       : 0x0f
Actual   Flag       : 0x0f
Time of execution   : 0.012848 seconds
Match Result        : ✅ Matched

Size of executable: 97672 bytes
```

This approach allowed us to successfully retrieve the final obfuscated binary from within the container, bypassing the issues associated with volume mounting on our host system (Windows). Note: all the binaries must be present for the file to work. So copy the whole app directory.

---

7. Conclusion

This assignment successfully demonstrated a range of practical obfuscation techniques on a cryptographic C program. While the functionality of AES encryption was preserved, the code was heavily disguised through variable renaming, runtime key decoding, and misleading logic structures. The final result is a binary that poses a significant challenge to reverse engineers while still behaving correctly under execution.

Through a layered approach combining control flow obfuscation, junk logic, pipe-based key injection, self-decrypting stubs, and multi-layer binary protection, I was able to design a secure and obfuscated executable. The project strengthened my understanding of secure programming, obfuscation theory, and binary protection techniques.

---

8. Contribution

- **Arnav** focused on modifying `main.c` using multiple obfuscation techniques including bogus functions, key/egg parameter misdirection, variable renaming, junk code injection, and opaque control flows.
- **Dipanshu** worked on external binary protection, such as building key and egg generator tools, pipe-based injection, implementing multi-layer encryption, binary embedding via `xxd`, and constructing self-decrypting loaders.
- Finally, we **merged** the logics from both teams and stacked them into a **multi-layer encrypted binary**, resulting in a robust obfuscation framework.

---

10. References

- **Chester sir’s lectures**
- **Research papers**
    - **"Software Protection through Program Obfuscation" – Collberg et al.**
    - **"A Taxonomy of Obfuscating Transformations" – Collberg et al.**
    - **"Obfuscating C++ Programs via Control Flow Flattening" – László and Kiss**
- **chatGPT**
- **perplexity**