Hiding secrets in plain sight

A C++20 library for obfuscating secrets at compile time

February 10, 2026

Sebastien Andrivet <sebastien@andrivet.com>

Contents

1 | Introduction

Twelve years ago [1], I created and released an obfuscation library based on C++11 [2] (and updated later to C++14 and C++17) called **ADVobfuscator**. The goal was to provide a simple-to-use way to hide secrets in C++ code, such as API keys, passwords, or any sensitive data. In particular, I used it in a commercial product called ADVdetector, a library for detecting jailbroken iOS devices. The obfuscation techniques used in ADVobfuscator were based on template metaprogramming, which allowed it to generate complex code at compile time, making it harder to reverse engineer.

To my surprise, ADVobfuscator got some attention and was used in several programs, including some viruses [3], [4]. Some people even created tools to deobfuscate the code generated by ADVobfuscator [5], [6]. The library was far from perfect. Due to limitations in the C++ language, it used some macros, which made the code less readable. The obfuscation algorithm was simple (XOR-based) and could be broken by a determined reverse engineer or even some automated tools.

With the availability of C++20 [7], I decided to create a new version of the library, which takes advantage of the new features of C++20 to provide better obfuscation and a more user-friendly, natural interface no more macro-based. My goal is to make it harder to reverse engineer the code, while still being easy to use for developers. Another motivation for creating a new version of the library is to experiment with AES and determine if it can be used effectively at compile time to obfuscate secrets such as TLS certificates.

In this paper, I will present the design and implementation of the new version of a library and some of my experiments with compile-time AES.

2 | Obfuscation

Obfuscation is "the deliberate act of creating [...] code that is difficult for humans to understand" [8]. Obfuscated code has the same or almost the same semantics as the original and obfuscation is transparent to the system executing the application and to the users of this application.

Barak and al [9] introduced in 2001 a more formal and theoretical study of obfuscation: an obfuscator $\mathcal{O}$ is a function that takes as input a program $\mathcal{P}$ and outputs another program $\mathcal{O}(\mathcal{P})$ satisfying the following two conditions:

- (functionality) $\mathcal{O}(\mathcal{P})$ computes the same function as $\mathcal{P}$.
- ("virtual black box" property) "Anything that can be efficiently computed from $\mathcal{O}(\mathcal{P})$ can be efficiently computed given oracle access to $\mathcal{P}$."

Their main result is that general obfuscation is impossible even under weak formalization of the above conditions. This result puts limits on what we can expect from an obfuscator. In the remaining of this discussion, we will focus on obfuscators not as a universal solution but as a way to slow down reverse engineering of software. We will also focus on areas typically exploited by attackers. In other terms, we will follow a pragmatic approach, not a theoretical one. For a more theoretical presentation, see for example the thesis of Jan Cappaert [10].

2.1 | Types of obfuscators

It is possible to classify obfuscators in several ways depending on assumptions and intents. A possible classification is the following [11]:

- Source code obfuscators: transformation of the source code of the application before compilation.
- Binary code obfuscators: transformation of the binary code of the application after compilation.

This classification mimics the traditional phases of compilation: front-end (dependent on the source language) and back-end (independent on the source language, dependent on the target machine) [12].

Source code obfuscators can be further refined:

- Direct source code obfuscation: manual transformation of the source code by a programmer to make it difficult to follow and understand (including for other developers or for himself).
- Pre-processing obfuscators: automatic transformation of source code into modified source code before compilation.
- Abstract syntax tree (AST) or Intermediate representation (IR) obfuscators: compilers operate in phases. Some are generating an intermediate representation, a kind

of assembly language or virtual machine bytecode (as it is the case for LLVM). This class of obfuscators transforms this intermediate language.

- Bytecode obfuscators: transformation of bytecode generated by the compiler (Java, .NET languages, etc.) It is a special case and share similarities with Abstract syntax tree obfuscators. This class of obfuscators is in fact located between source code and binary code obfuscators. We classify it in source code obfuscators because it is dependent on the languages and not on the target machine.

Under some circumstances, software or a portion of it has to be released in source code. A typical example is JavaScript embedded in web pages. In this case, only some source code obfuscators are applicable.

Depending on the language, it is possible to further refine this classification or to add new classes of obfuscators. It is the case for the C++ language. Beyond the classical syntax and lexical analysis, C++ compilers incorporate other compilation phases: the pre-processor is well-known as it is directly inherited (almost without modifications) from the C language. But there is another one, specific to C++: templates instantiation and compile-time specifiers. It is this mechanism that will be used for the obfuscator described in this document. These techniques are described in an Appendix of this document.

3 | Design and Implementation

The objectives are:

- Use a simple and natural syntax for obfuscating secrets in C++ code, without using macros. Something like:

```cpp
auto secret = "This is a secret"_obf;
```

- Provide better obfuscation techniques (compared to the previous version of the library) to make it harder to reverse engineer the code.
- Do not rely on undefined behavior of the C++ language.

3.1 | Generation of random numbers at compile time

File `random.h`.

There is no function in C++ to generate random numbers at compile time. However, we can use the `__TIME__` macro, which expands to a string literal representing the time of compilation in the format "HH:MM:SS". We can parse this string to extract the hours, minutes, and seconds, and use them to generate random numbers:

```cpp
/// Use current (compile time) as a seed
static constexpr char time[] = __TIME__; // __TIME__ has the following
format: hh:mm:ss in 24-hour time

/// Convert a digit into the corresponding number
constexpr int digit_to_int(char c) { return c - '0'; }

/// Convert time string (hh:mm:ss) into a number
static constexpr unsigned seed =
  digit_to_int(time[7]) +
  digit_to_int(time[6]) * 10 +
  digit_to_int(time[4]) * 60 +
  digit_to_int(time[3]) * 600 +
  digit_to_int(time[1]) * 3600 +
  digit_to_int(time[0]) * 36000;
```

This number (`seed`) is then used to generate random numbers with a Lehmer random number generator [13]. It is certainly possible to use a better algorithm but it is simple to implement, well known and considered a minimal standard [14].

```cpp
/// Generate a (pseudo) random number.
/// \tparam T Type of the number to generate (std::size_t by default).
```

```
/// \param count The count for the generation of random numbers.
/// \param max The maximum value of the number generated (excluded).
/// \return A number generated randomly.
/// \remarks Inspired by 1988, Stephen Park and Keith Miller
/// "Random Number Generators: Good Ones Are Hard To Find", considered
as "minimal standard"
/// Park-Miller 31 bit pseudo-random number generator, implemented with
G. Carta's optimisation:
/// with 32-bit math and without division
template<typename T = std::size_t>
consteval T generate_random(std::size_t count, T max) {
  const uint32_t a = 16807;        // 7^5
  const uint32_t m = 2147483647;   // 2^31 - 1

  auto s = seed;
  while(count-- > 0) {
    uint32_t lo = a * (s & 0xFFFF); // Multiply lower 16 bits by 16807
    uint32_t hi = a * (s >> 16); // Multiply higher 16 bits by 16807
    uint32_t lo2 = lo + ((hi & 0x7FFF) << 16); // Combine lower 15 bits
    of hi with lo's upper bits
    uint32_t lo3 = lo2 + hi;
    s = lo3 > m ? lo3 - m : lo3;
  }

  // Note: A bias is introduced by the modulo operation.
  // However, I do believe it is negligible in this case (M is far lower
  than 2^31 - 1)
  return static_cast<T>(s % static_cast<uint32_t>(max));
}
```

3.2 | Obfuscation

File `obf.h`

The previous version of the library used one of a set of obfuscation algorithms (XOR, XOR with an incrementing key, shifting). The key and the choice of algorithm were random. Since there are only 3 algorithms and only 255 possible keys, it was possible to break the obfuscation with a brute-force attack. In the new version of the library, we will use a combination of two sets of algorithms:

- a set of data algorithms (caesar, XOR, rotation of bits, substitution) used to obfuscate a byte,

- a set of key algorithms (increment, inversion, substitution of bits, swapping high and low nibbles) used to compute the next value of the key to be used with the data algorithm.

```cpp
/// Algorithms to encode data
enum class DataAlgorithm {
  IDENTITY,   ///< Identity function, i.e. no change.
  CAESAR,     ///< Caesar algorithm, key is the displacement.
  XOR,        ///< XOR with the key.
  ROTATE,     ///< Bits rotation, key is the displacement.
  SUBSTITUTE, ///< Substitute bits, key % 8 is the displacement.
  NB_VALUES   ///< Number of values in this enum.
};

/// Algorithms to encode a key from a previous one
enum class KeyAlgorithm {
  IDENTITY,   ///< Identity function, i.e. no change.
  INCREMENT,  ///< Key is incremented at each step.
  INVERT,     ///< Key is inverted at each step.
  SUBSTITUTE, ///< Substitute bits (0 becomes 7, 7 becomes 0, ...) at each
  step.
  SWAP,       ///< Swap high and low nibbles at each step.
  NB_VALUES   ///< Number of values in this enum.
};

/// Parameters of an obfuscation algorithm.
struct Parameters {
  std::uint8_t key = 0; ///< Key to be used.
  KeyAlgorithm key_algo = KeyAlgorithm::IDENTITY; ///< Algorithm to
  compute the next key.
  DataAlgorithm data_algo = DataAlgorithm::IDENTITY; ///< Algorithm to
  encode data.
};
```

The combination of these two sets of algorithms allows to create a much larger number of obfuscation algorithms, making it harder to break the obfuscation with a brute-force attack. Of course, this is not a perfect solution. We are dealing with obfuscation, in order to slow down reverse engineering not to prevent it. A combination of these two sets of algorithms is called an `Obfuscation`:

```cpp
/// An obfuscation algorithm
struct Obfuscation {
```

```
    /// Construct an obfuscation with identity algorithms.
    consteval Obfuscation() = default;

    /// Construct an obfuscation with on the fly algorithms.
    /// \param counter Randomization counter.
    consteval explicit Obfuscation(std::size_t counter) noexcept
    : parameters_{
     .key = generate_random_not_0<std::uint8_t>(counter, 0x7F),
     .key_algo = generate_random(counter + 2, KeyAlgorithm::NB_VALUES), //
     Identity is acceptable here
     .data_algo = generate_random_not_0(counter + 1,
     DataAlgorithm::NB_VALUES)
    } {}

    /// Construct an obfuscation with explicit algorithms.
    /// \param params Parameters for the obfuscation (key and algorithms).
    consteval explicit Obfuscation(const Parameters &params) noexcept :
    parameters_{params} {}

    ...

    /// Parameters for the obfuscation (key and algorithms).
    Parameters parameters_;
};
```

The encoding and decoding are trivial:

```
    /// Encode a byte.                                                    C++
    /// \param key Key to be used for the encoding.
    /// \return The encoded byte.
    [[nodiscard]] consteval std::uint8_t encode(std::uint8_t c, std::uint8_t
    key) const {
      switch(parameters_.data_algo) {
        using enum DataAlgorithm;
        case IDENTITY: break;
        case CAESAR: return details::caesar(c, key);
        case XOR: return details::x0r(c, key);
        case ROTATE: return details::rotate(c, key);
        case SUBSTITUTE: return details::substitute(c, key);
        case NB_VALUES: throw std::exception(); // Invalid data encoding
      }
      return c;
```

```
  }

  /// Decode a byte.
  /// \param key Key to be used for the decoding.
  /// \return The decoded byte.
  [[nodiscard]] constexpr std::uint8_t decode(std::uint8_t c, std::uint8_t
  key) const {
    switch(parameters_.data_algo) {
      using enum DataAlgorithm;
      case IDENTITY: break;
      case CAESAR: return details::caesar_inverted(c, key);
      case XOR: return static_cast<std::uint8_t>(c ^ key);
      case ROTATE: return details::rotate_inverted(c, key);
      case SUBSTITUTE: return details::substitute(c, key);
      case NB_VALUES: throw std::exception(); // Invalid data encoding
    }
    return c;
  }
```

Each algorithm is implemented by a method. For example, the `substitute` algorithm is implemented as follows:

C++

```
  /// Substitute bits in a byte.
  /// \param b Input byte.
  /// \param d Number of bits for the substitution.
  /// \remark If d = 7, bits 0 and 7 are exchanged, bits 1 and 6 are
  exchanged, etc.
  /// If d = 6, bits 0 and 6 are exchanged, bits 1 and 5 are exchanged,
  etc.
  /// \result The result of the substitution.
  constexpr uint8_t substitute(uint8_t b, uint8_t d) {
    d %= 8;
    uint8_t result = 0;
    for(uint8_t i = 0; i < 8; ++i) {
      auto bit = (b >> i) & 0x01;
      result |= bit << (i <= d ? d - i : 8 - i + d);
    }
    return result;
  }
```

Given a key, we can compute the next key with the key algorithm:

```C++
/// Compute the next key from the current one.
/// \param key The current key.
/// \return The new key computed from the given key.
[[nodiscard]] constexpr std::uint8_t next_key(std::uint8_t key) const {
  switch(parameters_.key_algo) {
    using enum KeyAlgorithm;
    case IDENTITY: break; // This is acceptable here
    case INCREMENT: return static_cast<std::uint8_t>((key + 1) % 256);
    case INVERT: return details::x0r(key, 0xFF);
    case SUBSTITUTE: return details::substitute(key, 7);
    case SWAP: return details::swap(key);
    default: throw std::exception(); // Invalid key encoding;
  }
  return key;
}
```

The method `next_key` is used to compute the next key to be used for encoding/decoding the next byte. The first key is the one defined in the parameters of the obfuscation:

```C++
/// Encode a range of data.
/// \param begin_pos Relative position of the beginning of the range from the whole data.
/// \param begin Pointer to the first byte to encode.
/// \param end Pointer past the last byte to encode.
template<typename It>
consteval void encode(std::size_t begin_pos, It begin, It end) const noexcept {
  auto key = parameters_.key;
  while(begin_pos-- > 0) key = next_key(key);
  for(auto current = begin; current < end; key = next_key(key), ++current)
    *current = encode(*current, key);
}

/// Decode a range of data.
/// \param begin_pos Relative position of the beginning of the range from the whole data.
/// \param begin Pointer to the first byte to decode.
/// \param end Pointer past the last byte to decode.
template<typename It>
```

```
    constexpr void decode(std::size_t begin_pos, It begin, It end) const
    noexcept {
      auto key = parameters_.key;
      while(begin_pos-- > 0) key = next_key(key);
      for(auto current = begin; current < end; key = next_key(key), +
      +current)
        *current = decode(*current, key);
    }
```

In order to make the reverse engineering harder, the number of obfuscation algorithms are not always the same. They are randomly chosen between 2 and 4:

```C++
    /// Minimal number of algorithms
    static const std::size_t MIN_NB_ALGORITHMS = 2;
    /// Maximal number of algorithms
    static const std::size_t MAX_NB_ALGORITHMS = 4;

    ...

  /// A set of obfuscations
  struct Obfuscations {
    /// Construct a set of random generated obfuscations.
    /// \param counter Randomization counter.
    consteval explicit Obfuscations(std::size_t counter) noexcept
    : algos_{details::make_algorithms(
        counter,
        generate_random(counter, details::MIN_NB_ALGORITHMS,
        details::MAX_NB_ALGORITHMS),
        std::make_index_sequence<details::MAX_NB_ALGORITHMS>{}
    )} {}

    /// Construct a set of obfuscations with explicit parameters.
    /// \param params Parameters for the obfuscation (key and algorithms).
    consteval explicit Obfuscations(const Parameters &params) noexcept
    : algos_{details::make_algorithm(params)} {}

    /// Construct a set of obfuscations with explicit parameters.
    /// \param params Array of parameters for the obfuscation (key and
    algorithms).
    template<std::size_t A>
    consteval explicit Obfuscations(const Parameters (&params)[A]) noexcept
    : algos_{details::make_algorithms<A>(
```

```
      params,
      std::make_index_sequence<details::MAX_NB_ALGORITHMS>{})} {}

  ...

  /// A set of obfuscations
  std::array<Obfuscation, details::MAX_NB_ALGORITHMS> algos_;
};
```

Encoding and decoding is performed with one algorithm after the other:

```
  /// Encode a range of data.
  /// \param begin_pos Relative position of the beginning of the range
  from the whole data.
  /// \param begin Pointer to the first byte to encode.
  /// \param end Pointer past the last byte to encode.
  template<typename It>
  consteval void encode(std::size_t begin_pos, It begin, It end) const {
    for(std::size_t i = 0; i < details::MAX_NB_ALGORITHMS; ++i)
      algos_[i].encode(begin_pos, begin, end);
  }

  /// Decode a range of data.
  /// \param begin_pos Relative position of the beginning of the range
  from the whole data.
  /// \param begin Pointer to the first byte to decode.
  /// \param end Pointer past the last byte to decode.
  template<typename It>
  constexpr void decode(std::size_t begin_pos, It begin, It end) const
  noexcept {
    for(std::size_t i = 0; i < details::MAX_NB_ALGORITHMS; ++i)
      algos_[details::MAX_NB_ALGORITHMS - i - 1].decode(begin_pos, begin,
      end);
  }
```

3.3 | Obfuscation of Strings

File `string.h`.

String literals are one of the most important sources of information for an attacker when reverse engineering binaries. They are sometimes even more important than debugging information (when they are available). Thanks to those literals, the attacker will be able to quickly find interesting portions of code instead of trying to take a costly

top-down approach (reverse engineering from the entry point of the binary). Binaries often contains several different kind of string literals like:

- error messages
- log information (even if logs are not activated)
- name of functions or of classes
- URLs
- etc.

It is essential to obfuscate these literals in order to slow down reverse engineering. Some programmers obfuscate these literals manually (direct source code obfuscation) and maintain (manually) a list of correspondence between obfuscated strings and original ones. This kind of solution is difficult (if ever possible) to maintain. Others use a pre-processor to automate these modifications. But again, it is difficult to maintain and it makes debugging more difficult for the developer.

Our goal is to obfuscate string literals with the following constraints:

- use a developer-friendly syntax. In particular, the original string literal has to be present in source code.
- use only C++ without any external tool.
- obfuscate literals at compile time. De-obfuscation can be performed at runtime.
- the cost of obfuscation / deobfuscation has to be minimal.
- the original string must not be present in the binary in release builds. It is acceptable if it is present in debug builds.

C++11 introduced user-defined literals (UDL) that allow to define custom suffixes for literals. This is a perfect fit for our goal. C++20 further improved UDL with the introduction of *string literal operator template*. With these two features, we can define a UDL `_obf` that obfuscates the string literal at compile time and deobfuscates it at run-time when it is used. For example:

```C++
auto secret = "This is a secret"_obf;
use_secret(secret.decode());
```

We declare a string literal operator template `_obf`:

```C++
template<ObfuscatedString str>
consteval auto operator ""_obf() { return str; }
```

It is a `consteval` template UDL. It constructs (at compile-time) an instance of `ObfuscatedString`. The constructor of `ObfuscatedString` accepts a string literal (more precisely, a reference to a constant array of characters) and is also `consteval`:

```C++
/// An obfuscated string of characters.
/// \tparam N The number of bytes of the string (including the null
terminal byte).
```

```
template<std::size_t N>
struct ObfuscatedString {
  /// Construct an obfuscated string of characters.
  /// \param str The array of characters (including the null terminal
  byte).
  consteval ObfuscatedString(char const (&str)[N]) noexcept
  : algos_{generate_sum(str)} {
    encode(str);
  };

  ...

  /// Encoded or decoded data.
  std::array<char, N> data_{};
  /// Obfuscations used to encode the data.
  Obfuscations algos_;
  /// Is the data encoded (default) or decoded (i.e. used)?
  bool obfuscated_ = true;

  ...
};
```

The `count` parameter of the obfuscation is computed as the sum of the characters of the string, which means that the same string will always be obfuscated with the same algorithms and keys. The obfuscation (`encode`) is performed in-place on the data member of `ObfuscatedString`:

C++

```
  /// Encode an array of characters.
  /// \param str The string of characters to be encoded.
  consteval void encode(char const (&str)[N]) noexcept {
    std::array<std::uint8_t, N> buffer;
    std::copy(str, str + N, buffer.begin());
    algos_.encode(0, buffer.begin(), buffer.end());
    std::copy(buffer.begin(), buffer.end(), data_.begin());
  }
```

The deobfuscation is performed with an implicit cast operator to `const char*` that decodes the data in-place and returns a pointer to the decoded string:

C++

```
  /// Implicit conversion to a pointer to (const) characters, like
  a regular string.
  operator const char* () noexcept {
```

```
    constexpr auto random = call::generate_random(__LINE__);
    const ObfuscatedMethodCall call{random,
    &ObfuscatedString::decode_inplace};
    call(random, this);
    return data_.data();
  }
```

This method uses an instance of `ObfuscatedMethodCall` to call `decode_inplace`. This class is used to obfuscate the call to the method, making it harder to reverse engineer the code. This is explained later in the document. The method `decode_inplace` decodes the data in-place:

C++
```
  /// Decode an array of characters in-place.
  void decode_inplace() noexcept {
    if(!obfuscated_) return;
    std::array<std::uint8_t, N> buffer;
    std::copy(data_.begin(), data_.end(), buffer.begin());
    algos_.decode(0, buffer.begin(), buffer.end());
    std::copy(buffer.begin(), buffer.end(), data_.begin());
    obfuscated_ = false;
  }
```

3.4 | Obfuscation of data

File `bytes.h`.

The obfuscation of block of data (bytes) is similar to the obfuscation of strings. The difference is that the data is represented as a string of hexadecimal digits separated by spaces. For example, the string "01 02 03 1F" represents the block of bytes {0x01, 0x02, 0x03, 0x1F}. The obfuscation is performed in-place on the data member of `ObfuscatedBytes`.

We declare a user-defined literal (UDL) `_obf_bytes`:

C++
```
/// User-defined literal "_obf_bytes"
template<ObfuscatedBytes block>
consteval auto operator""_obf_bytes() { return block; }
```

Like for `ObfuscatedString`, the constructor of `ObfuscatedBytes` accepts a string literal (more precisely, a reference to a constant array of characters) and is also `consteval`:

C++
```
/// A block of obfuscated bytes
template<std::size_t N>
struct ObfuscatedBytes {
```

```
    /// Construct a compile-time block of bytes from a string.
    /// \lparam str Array of characters representing bytes to be encrypted
    at compile-time.
    /// The format of the string is: two hexadecimal digits seperated by
    spaces.
    /// Example: "01 02 03 1F".
    /// \remark A set of obfuscation algorithms are generated on the fly.
    consteval ObfuscatedBytes(const char (&str)[N])
    : algos_{generate_sum(str)} {
      parse(str);
      encode();
    }

    ...

    /// Obfuscated or decoded data.
    std::array<std::uint8_t, N / 3> data_{};
    /// Set of algorithms used for the obfuscation.
    Obfuscations algos_;
    /// Is the data obfuscated (default) or decoded (i.e. used)?
    bool obfuscated_ = true;
  };
```

The difference with `ObfuscatedString` is the calls to `parse` in the constructor. The `parse` method parses the string literal to extract the bytes:

```
    /// Parse a string representing bytes in hexadecimal separated by
    spaces.                                                          C++
    consteval void parse(const char (&str)[N]) {
      size_t byte_index = 0;
      uint8_t high = 0;
      bool half = false;

      // For each character (except the terminal null byte)...
      for(size_t i = 0; i < N - 1; ++i) {
        char c = str[i];
        // Is it a space?
        if(c == ' ') continue;
        // Get the corresponding nibble value
        uint8_t value = hex_char_value(c);
        // First nibble (half)?
        if(!half) {
```

```
        high = value << 4;
        half = true;
      }
      else {
        // We have the two nibbles, store them
        data_[byte_index++] = high | value;
        half = false;
      }
    }
  }
```

The deobfuscation is performed either with a subscript operator that decodes the data in-place and returns the decoded byte, or with a call to `decode` that decodes the whole data and returns it as an `std::array<uint8_t, N>`:

```
  /// Direct access to a byte in the block.
  /// \lparam pos Position of the byte in the block.
  /// \return The decoded byte.
  [[nodiscard]] constexpr std::uint8_t operator[](std::size_t pos) const {
    std::uint8_t b{data_[pos]};
    algos_.decode(pos, &b, &b + 1);
    return b;
  }

  /// Decode (deobfuscate) the block of bytes.
  /// \return The decoded bytes.
  [[nodiscard]] constexpr std::array<std::uint8_t, N / 3> decode() const
  noexcept {
    std::array<uint8_t, N / 3> buffer{};
    std::copy(data_.begin(), data_.end(), buffer.begin());
    algos_.decode(0, buffer.begin(), buffer.end());
    return buffer;
  }
```

3.5 | Obfuscation of function calls

File `call.h`.

A way to obfuscate a call to a function is to use a finite state machine (FSM). Instead of simply jump directly to the callee, we instantiate a FSM that will execute some steps before calling the callee. Such schema will slow down both static and dynamic analysis:

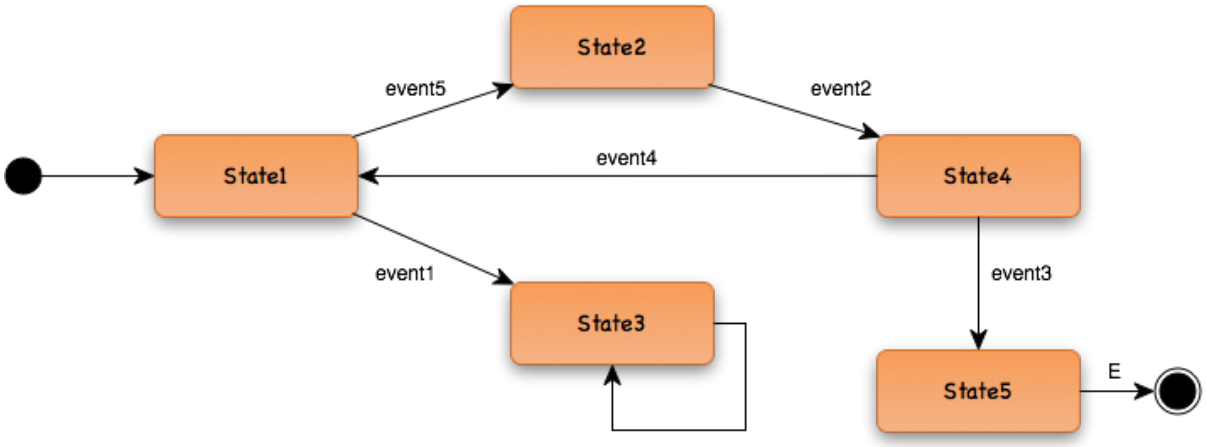

Figure 1 : Example of a finite state machine.

The `ObfuscatedCall` class is used to obfuscate a call to a function, making it harder to reverse engineer the code:

```C++
template<typename F>
struct ObfuscatedCall {
  consteval ObfuscatedCall(std::uint32_t recognize, F fn)
  : fsm_{recognize, fn} {
  }

  ...

  Fsm<F> fsm_;
};
```

It uses a finite state machine (FSM) to obfuscate the call. This is described later in the document. The call operator is used to call (indirectly) the function:

```C++
  template<typename... Args>
  decltype(auto) operator()(std::uint32_t value, Args... args) const {
    auto fn = fsm_.run(value);
    if constexpr (std::is_void_v<decltype(std::invoke(fn, args...))>) {
      std::invoke(fn, args...);
      return;
    }
    else
      return std::invoke(fn, args...);
  }
```

We first execute the FSM (`run`) then we check if the function returns void. In this case, we call the function and return (without returning a value). Otherwise, we call the function and return its result.

We declare a similar class `ObfuscatedMethodCall` to obfuscate a call to a method of a class:

```C++
template<typename F>
struct ObfuscatedMethodCall {
  consteval ObfuscatedMethodCall(std::uint32_t recognize, F fn)
  : fsm_{recognize, fn} {
  }

  template<typename O, typename... Args>
  decltype(auto) operator()(std::uint32_t value, O o, Args... args) const
  {
    auto fn = fsm_.run(value);
    if constexpr (std::is_void_v<decltype(std::invoke(fn, o, args...))>) {
      std::invoke(fn, o, args...);
      return;
    }
    else
      return std::invoke(fn, args...);
  }

  Fsm<F> fsm_;
};
```

3.5.1 | Finite State Machine (FSM)

File: `fsm.h`.

The actual finite state machine (FSM) is generated at compile time from a random number. This FSM is a recognizer for the binary digits of this random number. For example, if the random number is 11 (which is 1011 in binary), the FSM will have 17 states: the initial state, a state for recognizing the first bit (1), a state for recognizing the second bit (0), a state for recognizing the third bit (1) and a state for recognizing the fourth bit (1). In addition, there will be 3 states that will loop forever when the input bit is wrong. The FSM will transition from one state to another depending on the value of the input (the value passed to the call operator). If the input value is correct, the FSM will eventually reach a final state that will allow to call the callee. Otherwise, it will be stuck in states in a loop and will not call the callee.

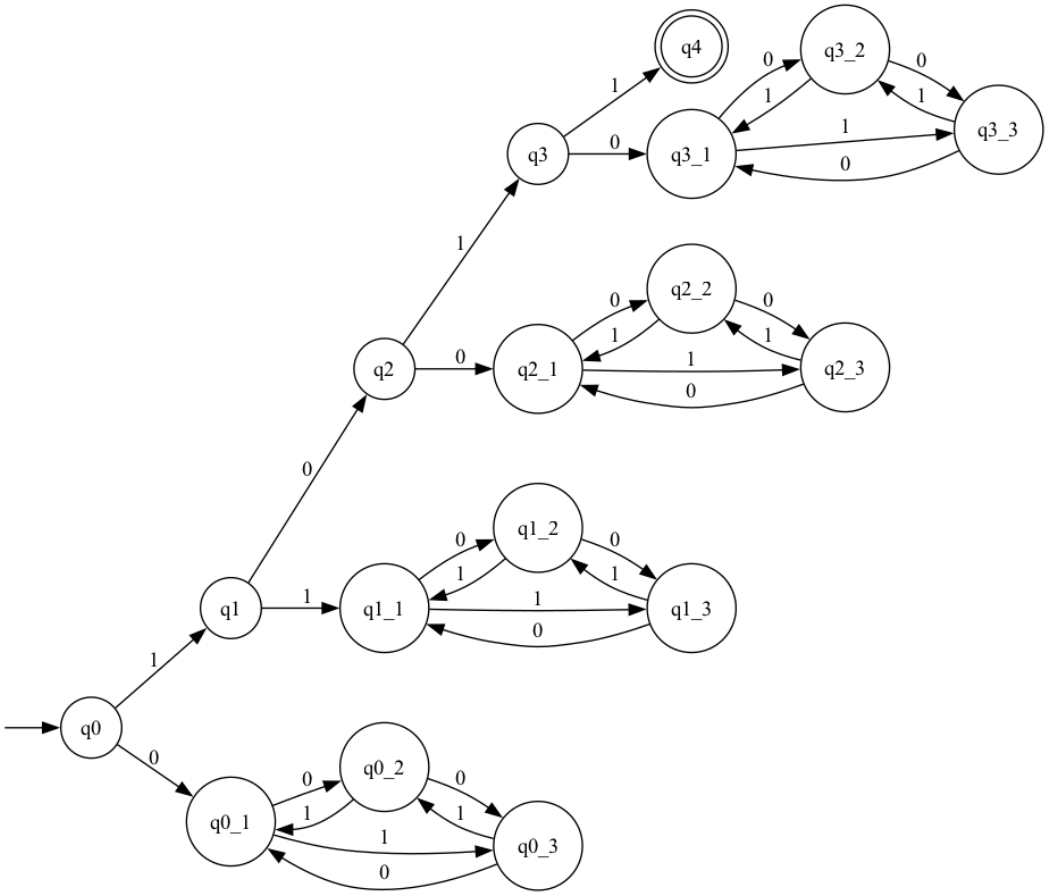

Figure 2 : FSM to recognize the binary digits of 11 (1011).

Note: Previous versions of the library relied on Boost Meta State Machine (MSM) library to implement the FSM. This is no more the case in the new version of the library. The FSM is implemented with C++20 features and is much simpler than the one implemented with Boost MSM. This removes the dependency on Boost and makes the code easier to understand and maintain.

We first declare some constants for the FSM like the number of bits (32), the number of transitions per bit (8) and the total number of transitions (256):

```C++
constexpr size_t NB_BITS = 32; ///< Number of bits recognized.
constexpr size_t TRANSITIONS_PER_BIT = 8; ///< Number of transitions per bit.
constexpr size_t MAX_TRANSITIONS = NB_BITS * TRANSITIONS_PER_BIT; ///< Total number of transitions.
```

The FSM is represented as an array of transitions. Each transition is a structure that contains the input bit (`input`), the source state (`from`), the destination state (`to`) and the object to be returned when the FSM reaches the final state (`o`):

```C++
    template<typename O>
```

```
  struct Transition {
    bool input;  ///< Input value (bit: 0 or 1).
    int from; ///< From this state.
    int to; ///< To this state.
    O o; ///< Object to return.
  };
```

One of the states is designed as the activation state. When the FSM reaches this state, it will call the callee and return its result. When the FSM is constructed, a random number is generated for the activation state. The transitions are then generated:

C++

```
/// A finite state machine that recognizes a number bit per bit,
template<typename O>
struct Fsm {
  /// Construct a new finite state machine that recognizes a number and
  stores an object.
  /// \param recognize The number to be recognized by this finite state
  machine.
  /// \param o The object stored in one of the transition (the active
  one).
  consteval Fsm(std::uint32_t recognize, O o) {
    // Get a random number for the activate transition of the recognizer.
    // The activate transition is the transition that stores the object.
    const std::uint32_t activate =
    generate_random_not_0<uint32_t>(recognize % 1000, NB_BITS - 1);
    auto bits = details::num_bits(recognize);

    // For each bit...
    for(int i = 0; i < bits; ++i) {
      // Get the bit's value
      bool bit = (recognize >> (bits - 1 - i)) & 0x01;
      // Transition to the next state of the recognizer and store (or not)
      the object
      add_transition(bit, 4 * i, 4 * i + 4, i + 1 == activate ? o : O{});
      // Transition to states in an infinite loop
      add_transition(!bit, 4 * i, 4 * i + 1, O{});
      add_transition(0, 4 * i + 1, 4 * i + 2, O{});
      add_transition(1, 4 * i + 1, 4 * i + 3, O{});
      add_transition(0, 4 * i + 2, 4 * i + 3, O{});
      add_transition(1, 4 * i + 2, 4 * i + 1, O{});
      add_transition(0, 4 * i + 3, 4 * i + 1, O{});
      add_transition(0, 4 * i + 3, 4 * i + 2, O{});
```

```
    }
  }
  ...
};
```

Adding a transition is just adding an element to the array of transitions:

```C++
  /// Add a transition to the finite state machine.
  /// \param input Input value.
  /// \param from From state.
  /// \param to To state.
  /// \param o Object to be stored in transition.
  consteval void add_transition(bool input, int from, int to, O o) {
    if(nb_transition_ >= MAX_TRANSITIONS) throw std::exception(); //
    MAX_TRANSITIONS is too small
    transitions_[nb_transition_++] = {.input = input, .from = from, .to =
    to, .o = o};
  }
```

Running the FSM is just following the transitions until we reach a final state (a state with an object to return):

```C++
  /// Run the finite state machine on a number.
  /// \param value The value to recognize.
  /// \remark The FSM will never return (infinite loop) if the number is
  wrong.
  /// This is by design to annoy reverse-engineering.
  decltype(auto) run(std::uint32_t value) const {
    auto bits = details::num_bits(value);
    int state = 0;

    // For each bit in reverse...
    for(int i = bits - 1; i >= 0; --i) {
      // Get the value of the bit.
      bool bit = (value >> i) & 1;
      // Find the transition
      auto &transition = find(state, bit);
      // Update the state.
      state = transition.to;
      // Treat the active transition as a final state.
      if(transition.o != O{}) return transition.o;
    }
```

```
    throw std::exception(); // Invalid FSM (i.e.bug);
  }
```

3.6 | Encryption of data with AES (Experimental)

3.6.1 | AES (Advanced Encryption Standard)

File: `aes.h`.

When I created this new version of the library, I was wondering if it would be possible to use a strong encryption algorithm like AES to obfuscate data. The idea was to encrypt the data at compile time with AES and decrypt it at runtime. In other terms, is it possible to implement AES at compile time in C++? And the answer is yes with some severe limitations (see later in this document).

Note: This implementation is directly inspired from the AES Proposal: Rijndael by Joan Daemen and Vincent Rijmen.

First we define the length of the encryption key (i.e. we will use AES-128):

```C++
/// Length of the cipher key
static constexpr std::size_t n_key{128}; // 128-bit or 192-bit or 256-bit
```

Then we define some type aliases for the different elements of the AES algorithm:

```C++
using Byte = std::uint8_t;
using Block = std::array<Byte, 128 / 8>;
using Key = std::array<Byte, n_key / 8>;
using Nonce = std::array<Byte, 8>;

...
    // A 32-bit word
  using Word = std::array<Byte, 4>;
  // An array of 4 columns, each column has 4 rows (s[column][row]).
  using State = std::array<Word, 4>;

  ...
  // Expanded key
  using EKey = std::array<Word, 4 * (n_rounds() + 1)>;
```

The `consteval` function `n_rounds` gives the number of rounds of AES depending on the key length (chapter 4.1 of the AES proposal):

```C++
  /// Number of rounds
```

```
  consteval std::size_t n_rounds() {
    static_assert(n_key == 128 || n_key == 192 || n_key == 256);
    return n_key == 128 ? 10 : n_key == 192 ? 12 : 14;
  }
```

We then declare the Rijndael S-Box and inverse S-Box as `constexpr` obfuscated arrays:

```
  // Rijndael S-Box (obfuscated)                                    C++
  static constexpr ObfuscatedBytes<16 * 3> sbox[16] = {
    "63 7C 77 7B F2 6B 6F C5 30 01 67 2B FE D7 AB 76"_obf_bytes,
    "CA 82 C9 7D FA 59 47 F0 AD D4 A2 AF 9C A4 72 C0"_obf_bytes,
    "B7 FD 93 26 36 3F F7 CC 34 A5 E5 F1 71 D8 31 15"_obf_bytes,
    "04 C7 23 C3 18 96 05 9A 07 12 80 E2 EB 27 B2 75"_obf_bytes,
    "09 83 2C 1A 1B 6E 5A A0 52 3B D6 B3 29 E3 2F 84"_obf_bytes,
    "53 D1 00 ED 20 FC B1 5B 6A CB BE 39 4A 4C 58 CF"_obf_bytes,
    "D0 EF AA FB 43 4D 33 85 45 F9 02 7F 50 3C 9F A8"_obf_bytes,
    "51 A3 40 8F 92 9D 38 F5 BC B6 DA 21 10 FF F3 D2"_obf_bytes,
    "CD 0C 13 EC 5F 97 44 17 C4 A7 7E 3D 64 5D 19 73"_obf_bytes,
    "60 81 4F DC 22 2A 90 88 46 EE B8 14 DE 5E 0B DB"_obf_bytes,
    "E0 32 3A 0A 49 06 24 5C C2 D3 AC 62 91 95 E4 79"_obf_bytes,
    "E7 C8 37 6D 8D D5 4E A9 6C 56 F4 EA 65 7A AE 08"_obf_bytes,
    "BA 78 25 2E 1C A6 B4 C6 E8 DD 74 1F 4B BD 8B 8A"_obf_bytes,
    "70 3E B5 66 48 03 F6 0E 61 35 57 B9 86 C1 1D 9E"_obf_bytes,
    "E1 F8 98 11 69 D9 8E 94 9B 1E 87 E9 CE 55 28 DF"_obf_bytes,
    "8C A1 89 0D BF E6 42 68 41 99 2D 0F B0 54 BB 16"_obf_bytes};

  // Rijndael Inverse S-Box (obfuscated)
  static constexpr ObfuscatedBytes<16 * 3> inv_sbox[16] = {
    "52 09 6a d5 30 36 a5 38 bf 40 a3 9e 81 f3 d7 fb"_obf_bytes,
    "7c e3 39 82 9b 2f ff 87 34 8e 43 44 c4 de e9 cb"_obf_bytes,
    "54 7b 94 32 a6 c2 23 3d ee 4c 95 0b 42 fa c3 4e"_obf_bytes,
    "08 2e a1 66 28 d9 24 b2 76 5b a2 49 6d 8b d1 25"_obf_bytes,
    "72 f8 f6 64 86 68 98 16 d4 a4 5c cc 5d 65 b6 92"_obf_bytes,
    "6c 70 48 50 fd ed b9 da 5e 15 46 57 a7 8d 9d 84"_obf_bytes,
    "90 d8 ab 00 8c bc d3 0a f7 e4 58 05 b8 b3 45 06"_obf_bytes,
    "d0 2c 1e 8f ca 3f 0f 02 c1 af bd 03 01 13 8a 6b"_obf_bytes,
    "3a 91 11 41 4f 67 dc ea 97 f2 cf ce f0 b4 e6 73"_obf_bytes,
    "96 ac 74 22 e7 ad 35 85 e2 f9 37 e8 1c 75 df 6e"_obf_bytes,
    "47 f1 1a 71 1d 29 c5 89 6f b7 62 0e aa 18 be 1b"_obf_bytes,
    "fc 56 3e 4b c6 d2 79 20 9a db c0 fe 78 cd 5a f4"_obf_bytes,
    "1f dd a8 33 88 07 c7 31 b1 12 10 59 27 80 ec 5f"_obf_bytes,
```

```
    "60 51 7f a9 19 b5 4a 0d 2d e5 7a 9f 93 c9 9c ef"_obf_bytes,
    "a0 e0 3b 4d ae 2a f5 b0 c8 eb bb 3c 83 53 99 61"_obf_bytes,
    "17 2b 04 7e ba 77 d6 26 e1 69 14 63 55 21 0c 7d"_obf_bytes};

// Rijndael round constants (obfuscated)
static constexpr auto rcon = "01 02 04 08 10 20 40 80 1b 36"_obf_bytes;
```

Note: These data are obfuscated. Otherwise, they will be present in the binary in clear and an attacker will be able to easily detect them and infer the algorithm used.

We then need to multiply in the Galois field GF(2^8). This is done with the following function `gmul` (chapter 2.2.1 of the AES proposal):

```C++
/// Multiplication in GF(2^8) of two bytes.
/// \param v0 First argument
/// \param v1 Second argument
/// \return Result of the multiplication of v0 and v1 in GF(2^8)
/// \remark  https://en.wikipedia.org/wiki/Rijndael_MixColumns
[[nodiscard]] constexpr Byte gmul(Byte v0, Byte v1) {
  Byte product = 0; // Initial product value
  // For each bit...
  for(std::size_t i = 0; i < 8; ++i) {
    // If least significant bit is set, add (xor) v0 to product
    if(v1 & 1) product ^= v0;
    // Set high_bit to the x^7 term of v0
    const bool high_bit = v0 & 0x80;
    // Shift v0 to the left to multiply it by x (v0 = v0 * x)
    v0 <<= 1;
    // Turn x^8 into x^4+x^3+x+1
    if(high_bit) v0 ^= 0x1B;
    // Right shift v1
    v1 >>= 1;
  }
  return product;
}
```

Note: Here we are not using `consteval` because this function is called both at compile time (for the encyption) and at runtime (for the decryption). The function is `constexpr` so it can be evaluated at compile time when it is called with constant arguments and at runtime when it is called with non-constant arguments.

The addition in GF(2^8) is just the XOR of the two bytes and is defined by the operator `^` for two `Word` or a `Word` and a `Byte`:

```C++
/// Addition (XOR) in GF(2^8) of two words.
/// \param w0 The left operand.
/// \param w1 The right operand.
/// \return The result of the addition in GF(2^8) of each byte.
[[nodiscard]] constexpr Word operator^(const Word &w0, const Word &w1) {
  return Word{
    static_cast<Byte>(w0[0] ^ w1[0]),
    static_cast<Byte>(w0[1] ^ w1[1]),
    static_cast<Byte>(w0[2] ^ w1[2]),
    static_cast<Byte>(w0[3] ^ w1[3])
  };
}

/// Addition (XOR) in GF(2^8) of a word and a byte.
/// \param w0 The left operand.
/// \param b The right operand.
/// \return The result of the addition in in GF(2^8) of each byte of the
left operand with the right operand.
[[nodiscard]] constexpr Word operator^(const Word &w0, Byte b) {
  return Word{
    static_cast<Byte>(w0[0] ^ b),
    static_cast<Byte>(w0[1] ^ b),
    static_cast<Byte>(w0[2] ^ b),
    static_cast<Byte>(w0[3] ^ b)
  };
}
```

We then define basic operations of AES like SubWord, SubBytes transformations and their inverse (chapter 4.2.1 of the AES proposal):

```C++
/// SubWord Transformation - non-linear byte substitution using
sbox.
/// \param word Word to transform.
/// \return Transformed Word.
[[nodiscard]] constexpr Word sub_word(const Word &word) {
  return Word{
    sbox[high(word[0])][low(word[0])],
    sbox[high(word[1])][low(word[1])],
    sbox[high(word[2])][low(word[2])],
    sbox[high(word[3])][low(word[3])]
  };
```

```
}

/// SubBytes Transformation - non-linear byte substitution using sbox.
/// \param state State to transform.
/// \return Transformed state.
[[nodiscard]] constexpr State sub_bytes(const State &state) {
  return State{
    sub_word(state[0]),
    sub_word(state[1]),
    sub_word(state[2]),
    sub_word(state[3])};
}

/// InvSubWord Transformation -Inverse of SubWord.
/// \param word Word to transform.
/// \return Transformed Word.
[[nodiscard]] constexpr Word inv_sub_word(const Word &word) {
  return Word{
    inv_sbox[high(word[0])][low(word[0])],
    inv_sbox[high(word[1])][low(word[1])],
    inv_sbox[high(word[2])][low(word[2])],
    inv_sbox[high(word[3])][low(word[3])]
  };
}

/// InvSubBytes Transformation - Inverse of SubBytes.
/// \param state State to transform.
/// \return Transformed state.
[[nodiscard]] constexpr State inv_sub_bytes(const State &state) {
  return State{
    inv_sub_word(state[0]),
    inv_sub_word(state[1]),
    inv_sub_word(state[2]),
    inv_sub_word(state[3])};
}
```

The next transformations are ShiftRows and its inverse (chapter 4.2.2 of the AES proposal):

```
/// ShiftRows Transformation - bytes in the last three rows are
cyclically shifted.
```
C++

```
/// \param state State to transform.
/// \return Transformed state.
/// \remark Section 5.1.2
[[nodiscard]] constexpr State shift_rows(const State &state) {
  return State{
    Word{state[0][0], state[1][1], state[2][2], state[3][3]}, // c0
    Word{state[1][0], state[2][1], state[3][2], state[0][3]}, // c1
    Word{state[2][0], state[3][1], state[0][2], state[1][3]}, // c2
    Word{state[3][0], state[0][1], state[1][2], state[2][3]}  // c3
  };
}

/// InvShiftRows Transformation - Inverse of ShiftRows.
/// \param state State to transform.
/// \return Transformed state.
/// \remark Section 5.1.2
[[nodiscard]] constexpr State inv_shift_rows(const State &state) {
  return State{
    Word{state[0][0], state[3][1], state[2][2], state[1][3]}, // c0
    Word{state[1][0], state[0][1], state[3][2], state[2][3]}, // c1
    Word{state[2][0], state[1][1], state[0][2], state[3][3]}, // c2
    Word{state[3][0], state[2][1], state[1][2], state[0][3]}  // c3
  };
}
```

The next transformations are MixColumns and its inverse (chapter 4.2.3 of the AES proposal):

```
/// MixColumns Transformation - Multiply a column by a fixed
polynomial.                                                    C++
/// \param c The column to transform.
/// \return The transformed column.
/// \remark Section 5.1.3
[[nodiscard]] constexpr Word mix_column(const Word &c) {
  // 4.2.3 - The MixColumn transformation
  // c(x) = 3 * x^3 + 1 * x^2 + 1 * x + 2 modulo x^4 + 1
  const auto v0{gmul(c[0], 0x02) ^ gmul(c[1], 0x03) ^      c[2]        ^
  c[3]};
  const auto v1{     c[0]        ^ gmul(c[1], 0x02) ^ gmul(c[2], 0x03) ^
  c[3]};
  const auto v2{     c[0]        ^      c[1]        ^ gmul(c[2], 0x02) ^
  gmul(c[3], 0x03)};
```

```
    const auto v3{gmul(c[0], 0x03) ^ c[1] ^ c[2] ^
    gmul(c[3], 0x02)};
    return Word{
      static_cast<Byte>(v0),
      static_cast<Byte>(v1),
      static_cast<Byte>(v2),
      static_cast<Byte>(v3)};
  }

  /// InvMixColumns Transformation - Inverse of MixColumns.
  /// \param c The column to transform.
  /// \return The transformed column.
  /// \remark Section 5.3.3
  [[nodiscard]] constexpr Word inv_mix_column(const Word &c) {
    // 4.2.3 - The MixColumn transformation
    // c(x) = 3 * x^3 + 1 * x^2 + 1 * x + 2 modulo x^4 + 1
    const auto v0{gmul(c[0], 0x0e) ^ gmul(c[1], 0x0b) ^ gmul(c[2], 0x0d) ^
    gmul(c[3], 0x09)};
    const auto v1{gmul(c[0], 0x09) ^ gmul(c[1], 0x0e) ^ gmul(c[2], 0x0b) ^
    gmul(c[3], 0x0d)};
    const auto v2{gmul(c[0], 0x0d) ^ gmul(c[1], 0x09) ^ gmul(c[2], 0x0e) ^
    gmul(c[3], 0x0b)};
    const auto v3{gmul(c[0], 0x0b) ^ gmul(c[1], 0x0d) ^ gmul(c[2], 0x09) ^
    gmul(c[3], 0x0e)};
    return Word{
      static_cast<Byte>(v0),
      static_cast<Byte>(v1),
      static_cast<Byte>(v2),
      static_cast<Byte>(v3)};
  }
```

We declare two helper functions that take a `State` as input and apply the `mix_column` or `inv_mix_column` transformation to each column of the state:

```
  /// MixColumns Transformation - Multiply columns by a fixed
  polynomial.                                                    C++
  /// \param state The state to transform.
  /// \return The transformed state.
  [[nodiscard]] constexpr State mix_columns(const State &state) {
    return State{mix_column(state[0]), mix_column(state[1]),
    mix_column(state[2]), mix_column(state[3])};
  }
```

```
/// InvMixColumns Transformation - Inverse of MixColumns.
/// \param state The state to transform.
/// \return The transformed state.
[[nodiscard]] constexpr State inv_mix_columns(const State &state) {
  return State{inv_mix_column(state[0]), inv_mix_column(state[1]),
  inv_mix_column(state[2]), inv_mix_column(state[3])};
}
```

The next step is to implement the Round Key Addition (chapter 4.2.4 of the AES proposal):

C++

```
/// AddRoundKey Transformation - Add a Round Key to the State.
/// \param state The current state.
/// \param ekey The round key.
/// \return The transformed state.
/// \remark Section 5.1.4
[[nodiscard]] constexpr State add_round_key(const State &state, const
EKey &ekey, std::size_t round) {
  State new_state;
  for(std::size_t c = 0; c < 4; ++c)
    for(std::size_t r = 0; r < 4; ++r)
      new_state[c][r] = state[c][r] ^ ekey[round * 4 + c][r];
  return new_state;
}
```

The next step is to implement the Key Expansion (chapter 4.3.1 of the AES proposal):

C++

```
/// Key Expansion - Generate a key schedule.
/// \param key The key to be expanded.
/// \return The expanded key.
/// \remark Section 5.2
[[nodiscard]] constexpr EKey key_expansion(const Key &key) {
  EKey ekey;
  const auto nk = n_key / 32;

  // First 4 words: copy of the encryption key
  for(std::size_t i = 0; i < nk; ++i)
    ekey[i] = {key[4 * i], key[4 * i + 1], key[4 * i + 2], key[4 * i +
    3]};

  const auto n_r = n_rounds();
```

```
    for(std::size_t i = nk; i < 4 * (n_r + 1); ++i) {
      Word temp = ekey[i - 1];
      if(i % nk == 0) {
        temp = sub_word(rot_word(temp));
        temp[0] ^= rcon[i / nk - 1];
      }
      else if(nk > 6 and i % nk == 4)
        temp = sub_word(temp);
      ekey[i] = ekey[i - 4] ^ temp;
    }

    return ekey;
  }
```

With all these transformations, we can implement the AES encryption (chapter 4.4 of the AES proposal):

C++

```
/// Encrypt a block (128-bit) with a key.
/// \param block Block to be encrypted with AES.
/// \param key AES key.
/// \return The encrypted block.
[[nodiscard]] constexpr Block encrypt(const Block &block, const Key &key)
{
  using namespace details;

  const auto ekey = key_expansion(key);
  State state = add_round_key(to_state(block), ekey, 0);
  for(std::size_t round = 1; round < n_rounds(); ++round)
    state = add_round_key(mix_columns(shift_rows(sub_bytes(state))), ekey,
    round);
  state = add_round_key(shift_rows(sub_bytes(state)), ekey, n_rounds());
  return to_block(state);
}
```

The decryption is just the inverse of the encryption:

C++

```
/// Decrypt (at runtime) a block of bytes with a key.
/// \param block bytes to be decrypted with AES.
/// \param key AES key.
/// \return The decrypted block.
[[nodiscard]] inline Block decrypt(const Block &block, const Key &key) {
  using namespace details;
```

```
  const auto ekey = key_expansion(key);

  State state = add_round_key(to_state(block), ekey, n_rounds());
  for(std::size_t round = n_rounds() - 1; round >= 1; --round)
    state =
    inv_mix_columns(add_round_key(inv_sub_bytes(inv_shift_rows(state)),
    ekey, round));
  state = add_round_key(inv_sub_bytes(inv_shift_rows(state)), ekey, 0);
  return to_block(state);
}
```

3.6.2 | AES in CTR mode

To encrypt data larger than 128 bits, we can use AES in CTR (Counter) mode. The idea is to encrypt a nonce with AES and XOR the result with the data to be encrypted. The nonce is incremented for each block of data to be encrypted. This way, we can encrypt data of any size with AES.

C++

```
/// Encrypt in-place a string with a key using CTR (Counter) code
(using a nonce)
/// \param block bytes to be encrypted with AES. The number of bytes does
not need to be a multiple of 128.
/// \param key AES key.
/// \param nonce The random nonce to initialize the stream.
template<std::size_t N>
[[nodiscard]] consteval std::array<Byte, N> encrypt_ctr(const
std::array<Byte, N> &block, const Key &key, const Nonce &nonce) {
  Block ctr{
    nonce[0], nonce[1], nonce[2], nonce[3], nonce[4], nonce[5], nonce[6],
    nonce[7],
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
  };

  std::array<Byte, N> encrypted;
  const auto nb_whole_blocks = N / 16;
  const auto nb_bytes_last_block = N % 16;
  for(std::size_t i = 0; i < nb_whole_blocks; ++i) {
    auto encrypted_ctr = encrypt(ctr, key);
    // Combine the cipher and the plain bytes
    for(std::size_t j = 0; j < 16; ++j) encrypted[i * 16 + j] = block[i *
    16 + j] ^ encrypted_ctr[j];
    // Update the counter
```

```
    for(std::size_t j = 0; j < 8; ++j) ctr[8 + j] = static_cast<Byte>((i
    >> j * 8) & 0x00000000000000FF);
  }

  const auto encrypted_ctr = encrypt(ctr, key);
  for(std::size_t j = 0; j < nb_bytes_last_block; ++j)
    encrypted[nb_whole_blocks * 16 + j] = block[nb_whole_blocks * 16 + j]
    ^ encrypted_ctr[j];

  return encrypted;
}
```

The decryption is similar to the encryption but it is called at runtime:

```
/// Decrypt in-place a string with a key using CTR (Counter) code
(using a nonce)                                                    C++
/// \param data bytes to be decrypted with AES. The number of bytes does
not need to be a multiple of 128.
/// \param key AES key.
/// \param nonce The random nonce to initialize the stream.
inline void decrypt_ctr(Byte *data, size_t size, const Key &key, const
Nonce &nonce) {
  Block ctr{
      nonce[0], nonce[1], nonce[2], nonce[3], nonce[4], nonce[5],
      nonce[6], nonce[7],
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
  };

  const auto nb_whole_blocks = size / 16;
  const auto nb_bytes_last_block = size % 16;
  for(std::size_t i = 0; i < nb_whole_blocks; ++i) {
    auto encrypted_ctr = encrypt(ctr, key);
    // Combine the cipher and the plain bytes
    for(std::size_t j = 0; j < 16; ++j) data[i * 16 + j] = data[i * 16 +
    j] ^ encrypted_ctr[j];
    // Update the counter
    for(std::size_t j = 0; j < 8; ++j) ctr[8 + j] = static_cast<Byte>((i
    >> j * 8) & 0x00000000000000FF);
  }

  const auto encrypted_ctr = encrypt(ctr, key);
  for(std::size_t j = 0; j < nb_bytes_last_block; ++j)
```

```
        data[nb_whole_blocks * 16 + j] = data[nb_whole_blocks * 16 + j] ^
        encrypted_ctr[j];
    }
```

3.6.3 | Encryption of strings with AES

File: `aes_string.h`.

A key and a nonce are generated at compile time and used to encrypt the string with AES in CTR mode in the constructor of the `Aestring` class:

```cpp
/// A compile-time string encrypted with AES-CTR.
template<std::size_t N>
struct AesString {
  /// Construct a compile-time string encrypted with AES-CTR.
  /// \lparam str Array of characters to be encrypted at compile-time.
  /// \remark A key and a nonce are generated on the fly.
  consteval AesString(const char (&str)[N]) noexcept
  : key_{generate_random_block<16>(generate_sum(str, 0))},
    nonce_{generate_random_block<8>(generate_sum(str, 16))} {
    // Compile-time copy of the data
    std::copy(str, str + N, data_.begin());
    // Compile-time encryption
    auto encrypted = encrypt_ctr(data_, key_, nonce_);
    // Compile-time copy of the encrypted data
    std::copy(encrypted.begin(), encrypted.end(), data_.begin());
  }

  ...

  /// Encrypted or decrypted data.
  std::array<Byte, N> data_{};
  /// Is the data encrypted (default) or decrypted (i.e. used)?
  bool encrypted_ = true;
  /// The nonce used to chain blocks (CTR).
  Nonce nonce_{};
  /// The key used to encrypt the data.
  Key key_{};

  ...
};
```

When the string is destroyed, the data are erased from memory:

```cpp
    /// Destruct the string by first erasing its content.
    /// \remark The erasing may be omitted by the compiler.
    constexpr ~AesString() noexcept { erase(); }

    ...

    /// Erase the information stored by the string (data, key and nonce)
    constexpr void erase() noexcept {
      if (encrypted_) return;
      std::fill(data_.begin(), data_.end(), 0);
      std::fill(key_.begin(), key_.end(), 0);
      std::fill(nonce_.begin(), nonce_.end(), 0);
    }
```

The remaining of the implementation is very similar to the implementation of `ObfuscatedString`. The decryption is performed either by an implicit cast operator to `const char*` that modifies the instance or by an explicit call to `decrypt()` that does not modify the instance:

```cpp
    /// Implicit conversion to a pointer to (const) characters, like
    a regular string.
    operator const char *() noexcept {
      constexpr auto random = call::generate_random(__LINE__);
      const ObfuscatedMethodCall call{random, &AesString::decrypt_inplace};
      call(random, this);
      return reinterpret_cast<const char *>(data_.data());
    }

    /// Decrypt the encrypted string.
    [[nodiscard]] constexpr std::string decrypt() const {
      std::array<std::uint8_t, N> buffer;
      std::copy(data_.begin(), data_.end(), buffer.begin());
      if(encrypted_) decrypt_ctr(buffer.begin(), N, key_, nonce_);
      std::string str;
      str.resize(N - 1);
      std::copy(buffer.begin(), buffer.end() - 1, str.begin());
      return str;
    }

    /// Run-time decryption
    void decrypt_inplace() noexcept {
```

```
    if(!encrypted_) return;
    decrypt_ctr(reinterpret_cast<Byte*>(data_.data()), N, key_, nonce_);
    encrypted_ = false;
  }
```

We also define a user-defined literal to create an `AesString` from a string literal:

```C++
/// User-defined literal "_aes"
template<AesString str>
consteval auto operator""_aes() { return str; }
```

This way, we can encrypt a string at compile time with AES-CTR like this:

```C++
void aes_encryption_strings() {
  std::cout << "This is a string containing a secret that has to be hidden
  with AES"_aes << "\n";
}
```

3.6.4 | Limitations

Currently, only relatively small strings can be encrypted with AES-CTR with all compilers. When using long strings, the compilation may fail with an obscure error message such as (Clang):

```Shell
error: no matching literal operator for call to 'operator""_aes'
with arguments of types 'const char *' and 'unsigned long', and no
matching literal operator template
```

or (MSVC):

```Shell
error C2672: 'andrivet::advobfuscator::operator ""_aes': no
matching overloaded function found
expression did not evaluate to a constant
```

The limit is around 104 bytes when compiling with Clang, 121 bytes with MSVC. There is apparently **no limit** with GCC 15. It is able to deal with strings such as:

```C++
auto s1 = R"(-----BEGIN CERTIFICATE-----
MIICUTCCAfugAwIBAgIBADANBgkqhkiG9w0BAQQFADBXMQswCQYDVQQGEwJDTjEL
MAkGA1UECBMCUE4xCzAJBgNVBAcTAkNOMQswCQYDVQQKEwJPTjELMAkGA1UECxMC
VU4xFDASBgNVBAMTC0hlcm9uZyBZYW5nMB4XDTA1MDcxNTIxMTk0N1oXDTA1MDgx
NDIxMTk0N1owVzELMAkGA1UEBhMCQ04xCzAJBgNVBAgTAlBOMQswCQYDVQQHEwJD
TjELMAkGA1UEChMCT04xCzAJBgNVBAsTAlVOMRQwEgYDVQQDEwtIZXJvbmcgWWFu
ZzBcMA0GCSqGSIb3DQEBAQUAA0sAMEgCQQCp5hnG7ogBhtlynpOS21cBewKE/B7j
V14qeyslnr26xZUsSVko36ZnhiaO/zbMOoRcKK9vEcgMtcLFuQTWDl3RAgMBAAGj
```

```
gbEwga4wHQYDVR0OBBYEFFXI70krXeQDxZgbaCQoR4jUDncEMH8GA1UdIwR4MHaA
FFXI70krXeQDxZgbaCQoR4jUDncEoVukWTBXMQswCQYDVQQGEwJDTjELMAkGA1UE
CBMCUE4xCzAJBgNVBAcTAkNOMQswCQYDVQQKEwJPTjELMAkGA1UECxMCVU4xFDAS
BgNVBAMTC0hlcm9uZyBZYW5nggEAMAwGA1UdEwQFMAMBAf8wDQYJKoZIhvcNAQEE
BQADQQA/ugzBrjjK9jcWnDVfGHlk3icNRq0oV7Ri32z/+HQX67aRfgZu7KWdI+Ju
Wm7DCfrPNGVwFWUQOmsPue9rZBgO
-----END CERTIFICATE-----)"_aes;
```

My guess is that the limit is related to the maximum size of a template parameter pack (the string literal is passed as a template parameter pack to the user-defined literal operator) but I have not been able to find any documentation about this limit. As far as I know, there is no requirement in the C++ standard about this limit and it is up to the compiler to decide it. So it is not really a bug in Clang and MSVC but rather a limitation of their implementation. I have not been able to find any workaround for this issue.

3.7 | Reverse engineering

We will compare two similar programs, one with clear strings and one with obfuscated strings, to see the difference in the binary and how it can be reverse engineered. The first program is a simple CrackMe that asks the user for a password and checks if it is correct.

C++

```
#include <iostream>

int main() {
  std::string guess;
  std::cout << "Guess me if you can: ";
  if(std::getline(std::cin, guess); guess == "Can you spot this secret
  inside the binary?")
    std::cout << "Congratulations\n";
  else
    std::cout << "Nope\n";
}
```

The second program is the same but with all the strings obfuscated with ADVObfuscator:

C++

```
#include <iostream>
#include <advobfuscator/string.h>

using namespace andrivet::advobfuscator;
```

```cpp
int main() {
  std::string guess;
  std::cout << "Guess me if you can: "_obf;
  if(std::getline(std::cin, guess); guess == "Can you spot this secret
  inside the binary?"_obf.decode())
    std::cout << "Congratulations\n"_obf;
  else
    std::cout << "Nope\n"_obf;
}
```

Important: Be sure to compile both programs in release mode with optimizations enabled and without debug symbols. Otherwise, the strings may be present in clear in the binary and the reverse engineering will be trivial. It is also better to strip the binary to remove all the symbols that may help the reverse engineering. For example, with CMake:

```shell
cmake -DCMAKE_BUILD_TYPE=Release -S . -B build
cmake --build build
strip build/guessme
```

Using Binary Ninja, it is trivial to decompile the first program and see the string "Can you spot this secret inside the binary?" in clear in the decompiled code:

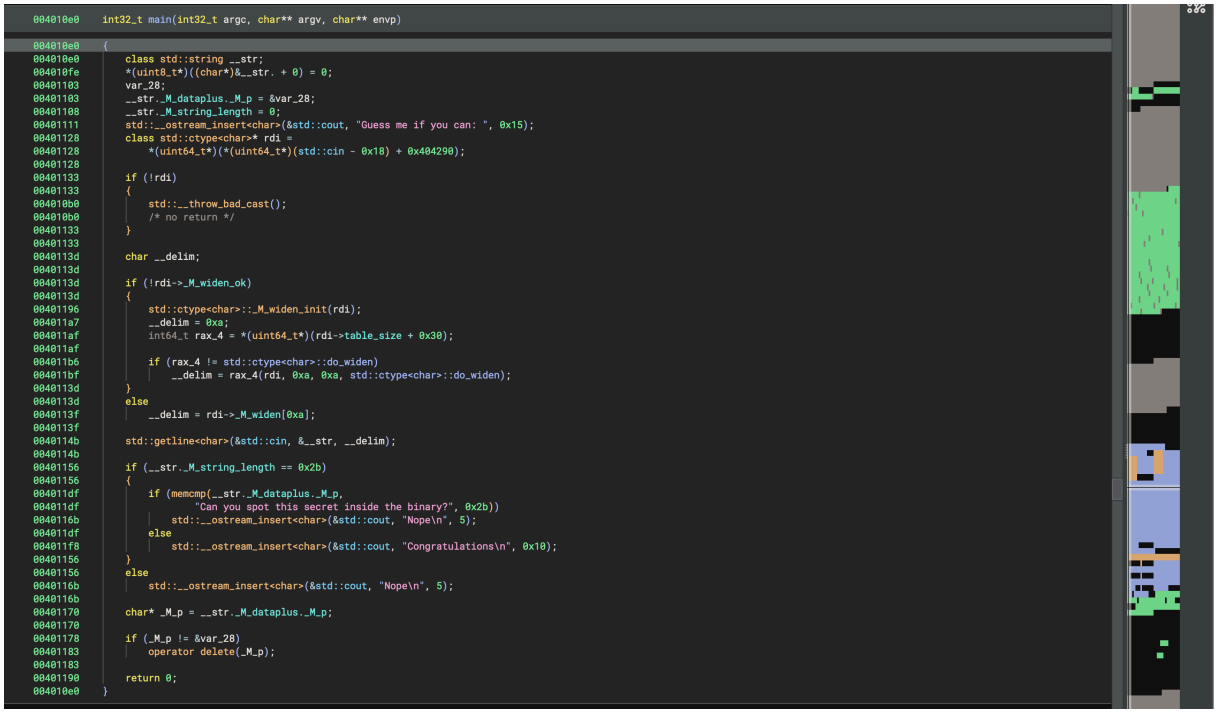

Figure 3 : Decompilation of the first program.

The string is in clear in the binary and it is easy to find it and see where it is used:

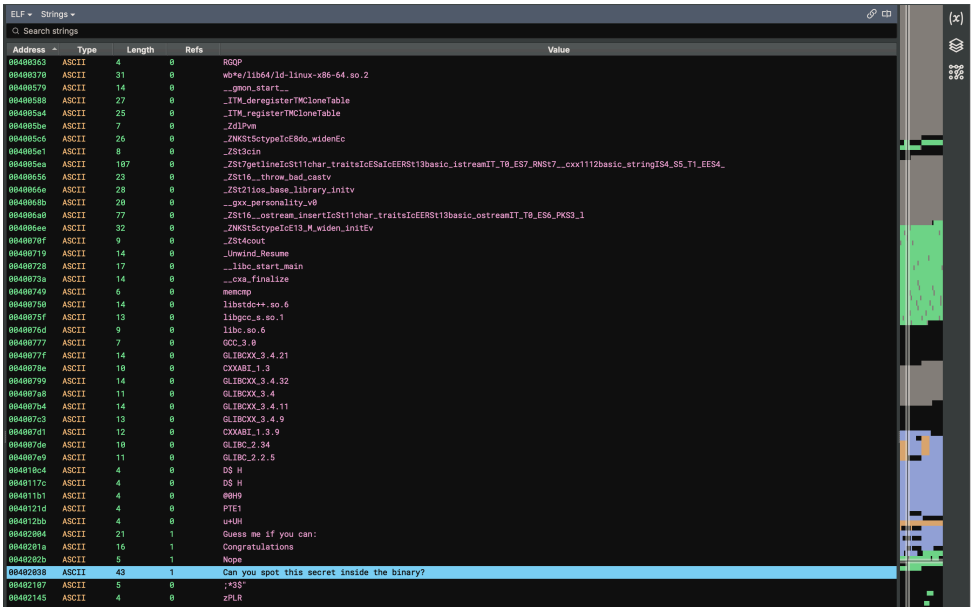

Figure 4 : Strings in the first program.

The decompilation of the second program is much more difficult to understand. The strings are not present in clear in the binary, they are obfuscated:

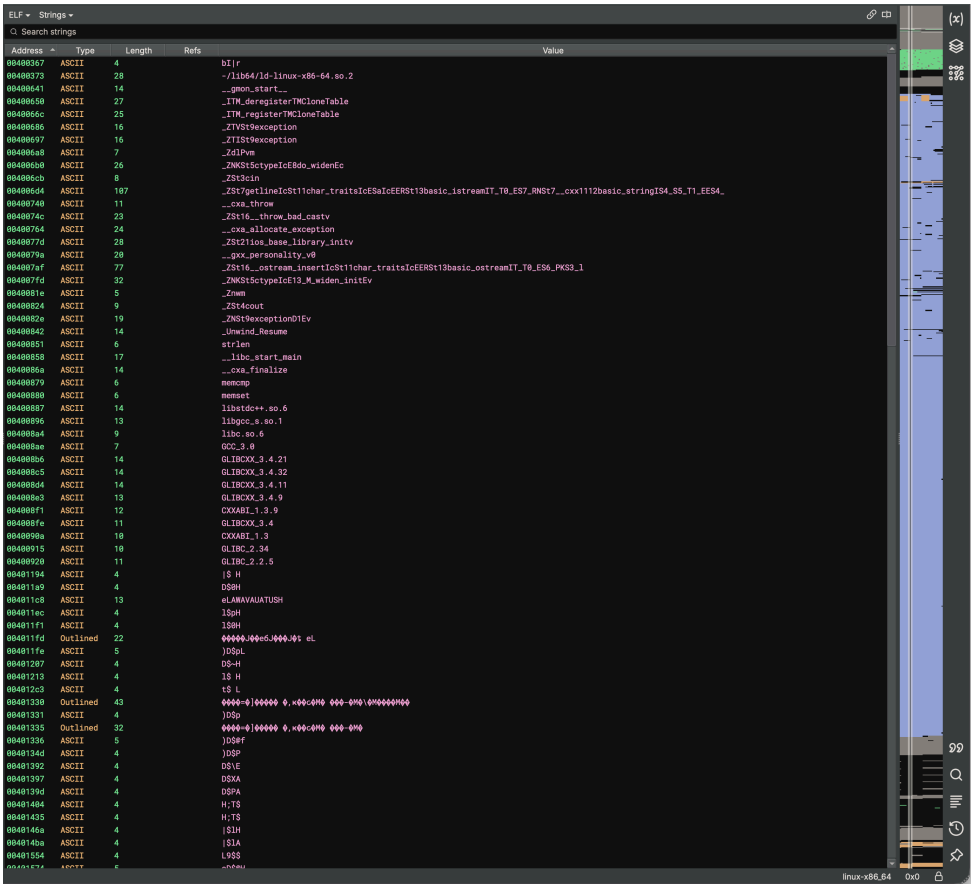

Figure 5 : Strings in the second program.

The start of the code is similar to the first program:

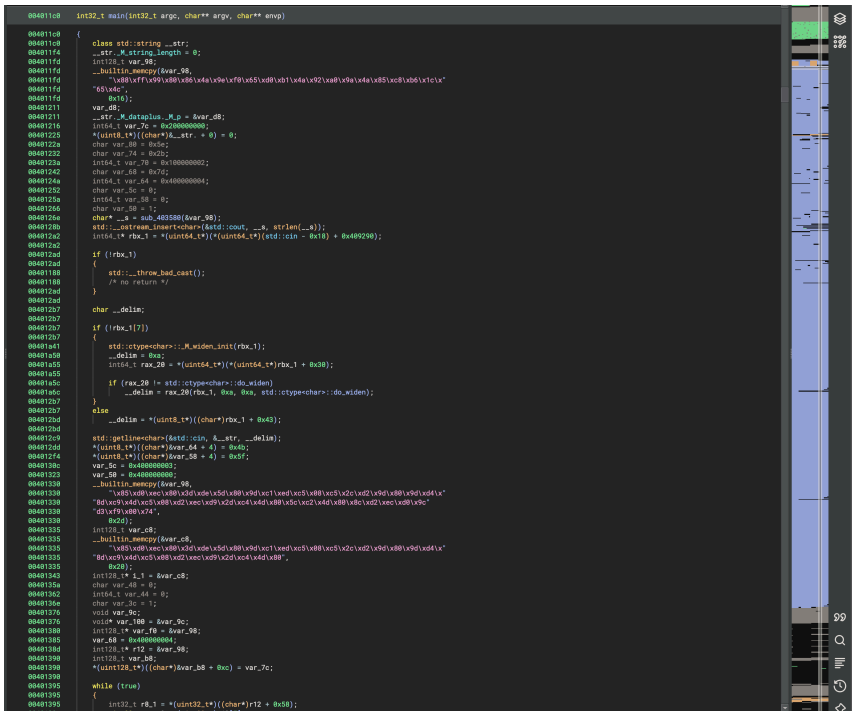

Figure 6 : Decompilation of the second program.

Then there is a big `while` loop that is not present in the first program. This loop is generated by the finite state machine:

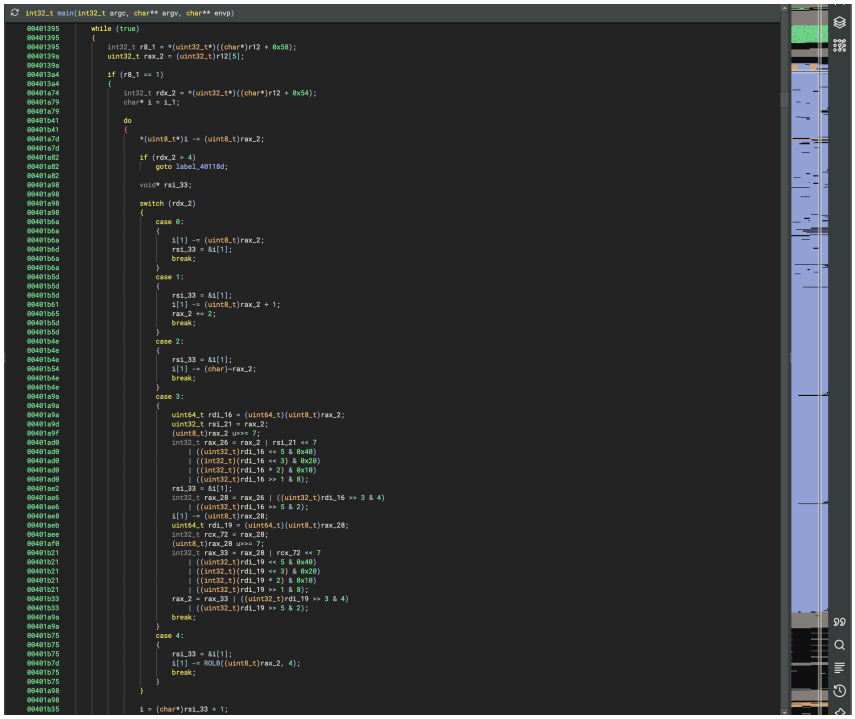

Figure 7 : Decompilation of the second program with the finite state machine.

The loops continue. At some point, the decoding of the string “Can you spot this secret inside the binary?” is performed but it is not clear where:

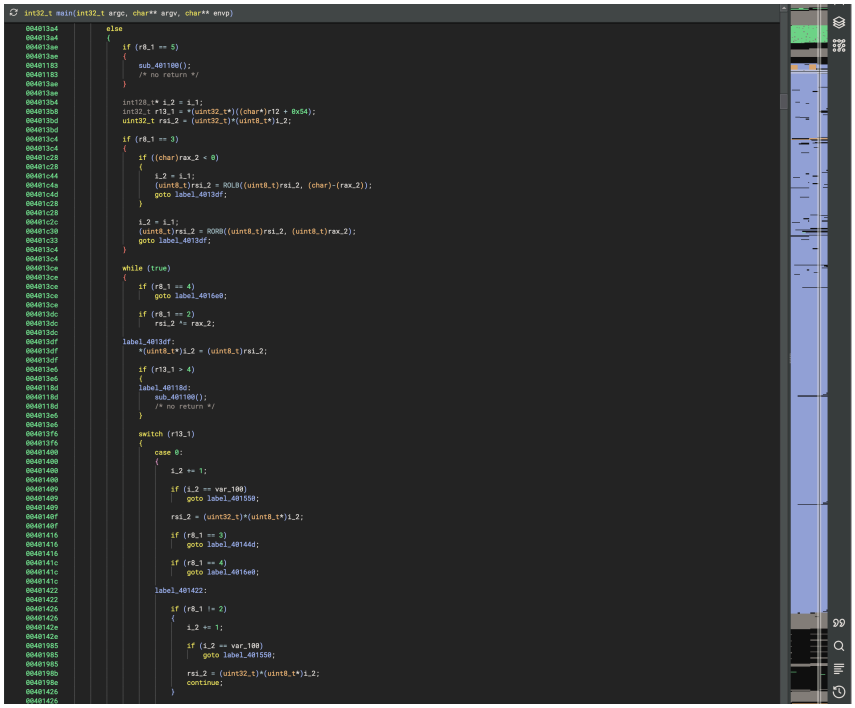

Figure 8 : Decompilation of the second program with more loops.

The end of the function is more recognizable:

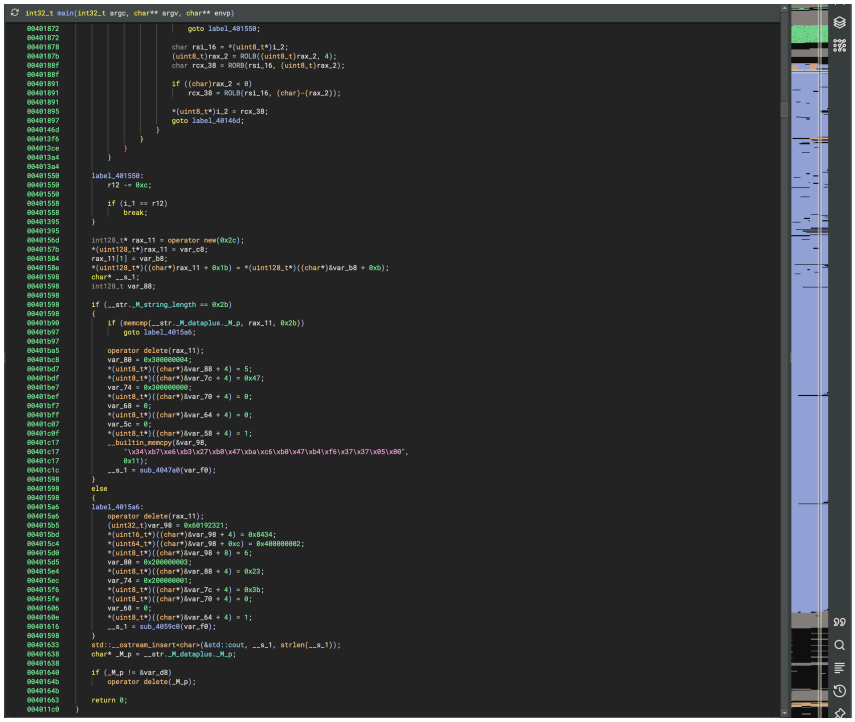

Figure 9 : Decompilation of the second program with the end of the function.

It is however far from trivial to understand the code and to find the string “Can you spot this secret inside the binary?” in the decompiled code using static analysis. The only thing that is not obfuscated is the length of the string.

In this part of the code, we can see this code:

```asm
00401c17            __builtin_memcpy(&var_98,
00401c17
"\x34\xb7\xe6\xb3\x27\xb0\x47\xba\xc6\xb0\x47\xb4\xf6\x37\x37\x05\x00",
00401c17                0x11);
00401c1c            __s_1 = sub_4047a0(var_f0);
```

`var_f0` is an alias to `&var_98`. The string “\x34\xb7…” is the obfuscated version of the string “Congratulations\n” and `sub_4047a0` is the function that decodes it.

The decompilation of this function is the following:

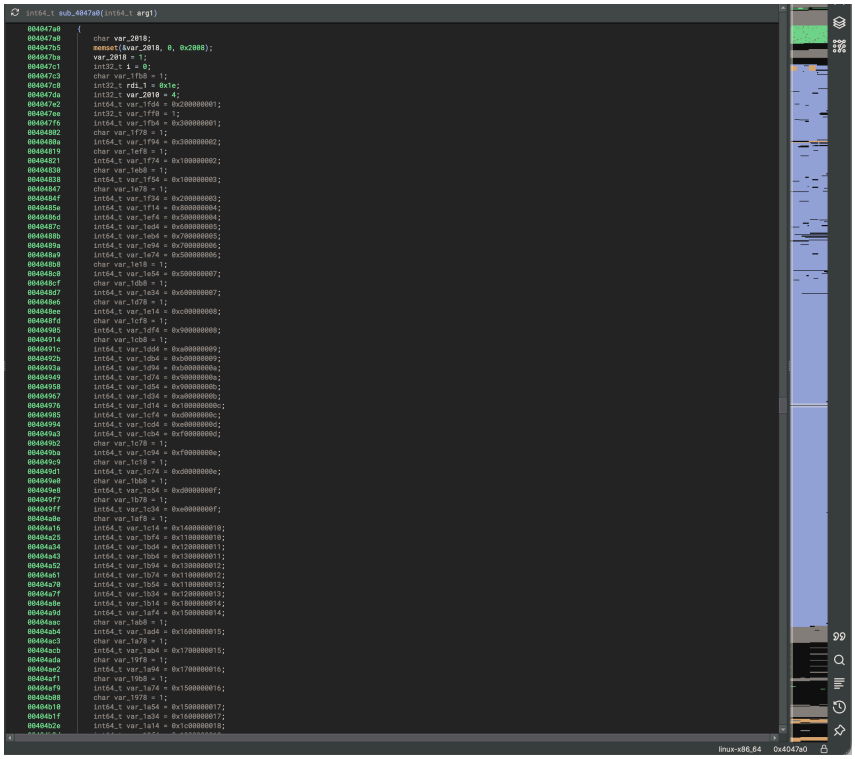

Figure 10 : Decompilation of sub_4047a0.

The disassembly does not give much more information about the decoding process:

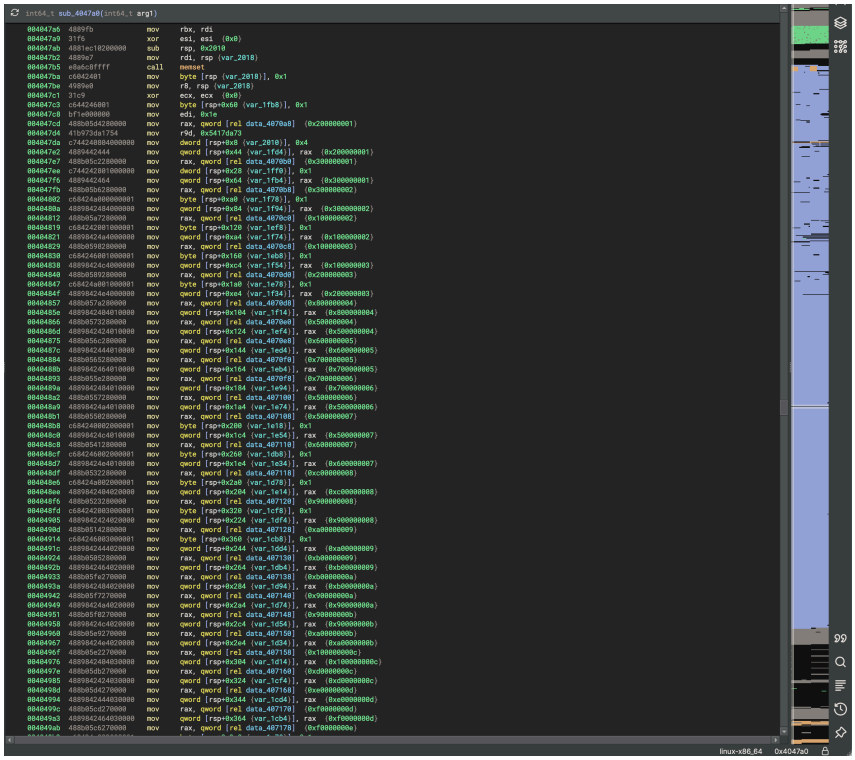

Figure 11 : Disassembly of sub_4047a0.

4 | Conclusion

This new version of ADVObfuscator is much more powerful than the previous ones. Its interface is more user-friendly and avoid using macros. The obfuscation is more powerful and the reverse engineering is much more difficult. The implementation of AES-CTR is a nice addition to the library and allows to encrypt larger strings with a strong encryption algorithm. The main limitation of this implementation is that it can only encrypt relatively small strings (around 100 bytes) due to limitations of some compilers. As of today, only GCC is able to correctly compile the code.

It is maybe possible to modify the AES implementation to be compatible with more compilers but it is probably not worth the effort. This is a gray area of C++ and compilers do not help much to understand the limitations of their implementation. I am also not convinced that, when compiled, strings encrypted with AES-CTR would be more difficult to reverse engineer than strings only obfuscated. AES looks more secure on the surface but at the end, the code has the decryption key. It is not provided by a secure mean. It is thus probably much worth the effort to enhance the obfuscation and in particular the obfuscation of function calls. As of today, the address of the function is not obfuscated (only the call itself is) and thus the reverse engineering tools are able to compute cross-references.

Another interesting direction to explore is to port (or rewrite) this library to the Rust programming language. Rust has a powerful macro system that provides some metaprogramming capabilities and it is also a popular language for systems programming. It would be interesting to see how the obfuscation techniques can be implemented in Rust and how they compare to the C++ version from a reverse engineering point of view.

5 | Appendix

5.1 | Installation

ADVobfuscator is a header-only C++ library that integrates cleanly with CMake. There are several possibilities to install and use it:

- Manual download
- Install & use via `find_package`
- Add as a Git submodule / subdirectory
- Use with `FetchContent`

5.1.1 | Manual download

If you don't use CMake or prefer to copy files manually:

- Click the green `Code` button on GitHub, then click on `Download ZIP` or download only the `include/` folder
- Copy the `include/advobfuscator/` directory into your own project's `include/` folder.
- Include it in your code:

```cpp
#include "advobfuscator/obfuscate.h"
```

- Make sure your compiler includes the path:

```shell
g++ -Iinclude myapp.cpp
```

- If you are using CMake, here is an example of `CMakeLists.txt`:

```cmake
cmake_minimum_required(VERSION 3.14)
project(myproject LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(myapp src/main.cpp)

# Add the path to the manually downloaded headers
target_include_directories(myapp PRIVATE ${CMAKE_SOURCE_DIR}/include)
```

5.1.2 | Install & Use via `find_package`

- Clone and Install the Library

```shell
git clone https://github.com/yourusername/advobfuscator.git
```

```
cd advobfuscator
cmake -B build -DCMAKE_INSTALL_PREFIX=/your/install/prefix
cmake --build build --target install
```

- Replace `/your/install/prefix` with the desired install location (e.g., `/usr/local` or a custom path).
- Link from Your CMake Project

```cmake
cmake_minimum_required(VERSION 3.14)
project(myproject)

# Add path to CMAKE_PREFIX_PATH if not system-installed
list(APPEND CMAKE_PREFIX_PATH "/your/install/prefix")

find_package(advobfuscator REQUIRED)

add_executable(myapp main.cpp)
target_link_libraries(myapp PRIVATE advobfuscator::advobfuscator)
```

5.1.3 | Add as a Git Submodule / Subdirectory

- Add the Library to Your Project

```shell
git submodule add https://github.com/andrivet/advobfuscator.git
external/advobfuscator
```

- Link from Your CMake Project

```cmake
add_subdirectory(external/advobfuscator)

add_executable(myapp main.cpp)
target_link_libraries(myapp PRIVATE advobfuscator::advobfuscator)
```

5.1.4 | Use with `FetchContent`

- If you want CMake (3.14 or higher) to automatically fetch and integrate ADVObfuscator:

```cmake
include(FetchContent)

FetchContent_Declare(
    advobfuscator
    GIT_REPOSITORY https://github.com/andrivet/advobfuscator.git
```

```
    GIT_TAG        v2.0  # Or use a branch or commit hash
)

FetchContent_MakeAvailable(advobfuscator)

add_executable(myapp main.cpp)
target_link_libraries(myapp PRIVATE advobfuscator::advobfuscator)
```

5.2 | Source Code

All the code of the library is available on GitHub: https://github.com/andrivet/advobfuscator.

5.3 | Usage

5.3.1 | Obfuscation of strings

Strings can be obfuscated using ADVobfuscator UDL (user-defined literal) `_obf`:

```
#include <advobfuscator/string.h>

std::cout << "abc"_obf << '\n';
```

The string is obfuscated at compile time. The UDL constructs (at compile-time) an instance of `ObfuscatedString`. At run-time, there is an implicit cast operator to `const char*` so the deobfuscated string can be converted. This code is thus (almost) equivalent to:

```
std::cout << ObfuscatedString{"abc"}.decode() << '\n';
```

It is also possible to use std::format:

```
#include <advobfuscator/format.h>

std::cout << std::format("{}\n", "abc"_obf);
```

Instances of obfuscated strings can be manipulated like any object. The implicit cast operator to const `char*` does modify the instance however (to decode the string). If the instance is immutable, you have to call explicitly decode() that returns a std::string and does not modify the instance:

```
static constexpr auto s4 = "An immutable compile-time string"_obf;
std::cout << s4.decode() << '\n';
```

5.3.2 | Obfuscation of data

Blocks of data (`uint8_t`) can be obfuscated at compile-time using `_obf_bytes`:

```cpp
#include <advobfuscator/bytes.h>

static constexpr auto data = "01 02 04 08 10 20 40 80 1b 36"_obf_bytes;
```

The format has to follow these rules:

- Each byte is represented by two hexadecimal digits.
- These hexadecimal digits can be in lower or upper case.
- Bytes have to be separated by space.

At compile-time, an instance of ObfuscatedBytes is created. This class provides a subscript operator that decodes, at run-time, the obfuscated data:

```cpp
auto d = data[0];  // d is an uin8_t
```

It is also possible to decode the whole data with `decode()`:

```cpp
auto decoded = data.decode(); // decoded is an std::array<uint8_t, N>
```

There is also a `data()` member function that decodes the data in-place:

```cpp
auto data = "01 02 04 08 10 20 40 80 1b 36"_obf_bytes;
auto decoded = data.data(); // decoded is an const std::uint8_t
```

5.3.3 | Encryption of strings with AES

In this version, it is also possible to encrypt the strings at compile-time using AES. The usage is however limited because of limitation of compilers (compile-time AES is quite complex for them). In practice, you can also encrypt strings that are not too long with the `_aes` UDL. The behavior is similar to obfuscated strings:

```cpp
#include <advobfuscator/aes_string.h>

std::cout << "This is a string containing a secret that has to be hidden
with AES"_aes << "\n";
```

Note: The S-box and other well-known data used by AES are obfuscated.

5.4 | Compilers support

ADVobfuscator has been tested with:

Compiler	Version	OS	CPU	Obfuscation	AES
Apple Clang	17.0.0	macOS 15	AArch64	YES	limited
Clang	21.1.8	Debian 14	x86_64	YES	limited
Clang	19.1.7	Debian 13	x86_64	YES	limited
Clang	18.1.8	Debian 13	x86_64	YES	limited
Clang	17.0.6	Debian 13	x86_64	YES	limited
GCC	15.1.0	macOS 15	x86_64	YES	YES
GCC	14.2.0	macOS 15	AArch64	YES	YES
GCC	14.2.0	Debian 13	x86_64	YES	YES
GCC	13.3.0	macOS 15	AArch64	NO	NO
Visual Studio 2022	17.14.13	Windows 11	AArch64	YES	limited
Visual Studio 2026	18.0.339	Windows 11	AArch64	YES	limited

5.5 | A brief introduction to metaprogramming

5.5.1 | Templates

Originally, templates were designed to enable generic programming and provide type safety. A classical example is the design of a class representing a stack of objects. Without templates, the stack will contain a set of generic pointers without type information (i.e. of `void*`). As a consequence, it is possible to mix incompatible types and it is required to cast (explicitly or implicitly) pointers to appropriate types. The compiler is not able to enforce consistency. This is delegated to the programmer. With templates, the situation is different: it is possible to declare and use a stack of a given type and the compiler will enforce it and produce a compilation error in case of a mismatch:

```C++
template<typename T> struct Stack
{
    void push(T* object);
    T* pop();
};

Stack<Singer> stack;
stack.push(new Apple());    // compilation error
```

Contrary to other languages like Java, such templates do retain the types of objects they are manipulating. Each instance of a template generates code for the actual types used. As a consequence, the compiler has more latitude to optimize generated code by taking into account the exact context. Moreover, and thanks to a mechanism called specialization, this kind of optimization is also accessible to the programmer. For example, it is possible to declare a generic Vector template for objects and another

version specialized for boolean. The two templates share a common interface but can use a completely different internal representation.

```cpp
// Generic Vector for any type T
template<typename T>
struct Vector
{
    void set(int position, const T& object);
    const T& get(position);
    // ...
};

// template specialization for boolean
template<>
struct Stack<bool>
{
    void set(int position, bool b);
    bool get(position);
    // ...
};
```

5.5.2 | Variadic templates

There are several situations where it is necessary to manipulate a list of types. It is the case for example when defining a tuple, a list of values of various types. Until C++11, the number of types (and thus of values) were arbitrarily limited by the implementation. It is not the case anymore with the C++11 and later: they are able to manipulate a list of types with variadic templates. For example, tuple can be defined by the following code:

```cpp
template <typename... T>
class tuple {
public:
    tuple();
    explicit tuple(const T&... args);
    // ...
};
```

A tuple is created and used this way:

```cpp
tuple<int, string, double> values{123, “test”, 3.14};
cout << get<0>(values);
```

Or, by using `make_tuple` helper:

```cpp
auto values = make_tuple(123, "test", 3.14);
cout << get<0>(values);
```

It is important to note that `make_tuple` and `get` are evaluated at compile time, not at runtime. They are compile-time entities.

5.5.3 | Constexpr and consteval

The `constexpr` keyword was introduced in C++11 to allow the evaluation of functions at compile time. A `constexpr` function can be evaluated at compile time if all its arguments are known at compile time and if its definition is available. If these conditions are not met, the function can still be evaluated at runtime. For example:

```cpp
constexpr int factorial(int n) {
    return n <= 1 ? 1 : (n * factorial(n-1));
}

auto f5 = factorial(5); // evaluated at compile time, f5 is a constant expression
```

The `consteval` keyword was introduced in C++20 to indicate that a function **must** be evaluated at compile time. A `consteval` function cannot be evaluated at runtime. If it is called with arguments that are not known at compile time, the program will fail to compile.

The new version of the library makes extensive use of `consteval` functions to perform obfuscation at compile time. This allows to generate obfuscated data and code that is not present in the source code and that is not present in the binary in release builds. In case, the obfuscation cannot be performed at compile time (for example because of the use of a non-`consteval` function), the compiler will throw an error. This ensures that the obfuscation is always performed at compile time and that the original data is not present in the binary without relying on undefined behavior or on the optimization capabilities of the compiler.

These two keywords were specifically added to the language for metaprogramming. They implies both `const` and `inline`.

5.5.4 | Metaprogramming

It was not the original intent of the designers of C++ but C++ templates are in fact a sub-language. This language is Turing-complete and similar to functional programming. It is evaluated entirely at compile time, not at run time. For example, it is possible to declare the following:

```cpp
template<int N>
```

```
struct Fibonacci { static constexpr int value = Fibonacci<N-1>::value +
Fibonacci<N-2>::value; };

template<>
struct Fibonacci<1> { static constexpr int value = 1; };

template<>
struct Fibonacci<0> { static constexpr int value = 0; }
```

It is an implementation of Fibonacci sequence using recursion (note: it can be implemented differently, this code is designed this way to illustrate our discussion).

The code:

```
Fibonacci<20>::value                                                    C++
```

is entirely computed at compile time and will be replaced by its result (6756). There is no computing and no cost at run time. We use recursion because C++ templates define a functional language: there is no variables, no loops, etc. Every statement is immutable like in Lisp or Haskell. Using this sub-language, we are able to generate code and not only to compute numbers. Templates are able to operate on types and make computation on them, or on other templates. We will use these possibilities to implement obfuscation schemas like encryption of string literals.

The previous version of the library relied on template metaprogramming techniques. The syntax of template metaprogramming is however quite complex and not very intuitive. It is also difficult to maintain and to debug. The new version of the library relies on C++20 features like `consteval` and `constexpr` that allow to perform computations at compile time without relying on template metaprogramming. This makes the code much easier to understand, to maintain and to debug.

5.6 | History

Version	Date	Description
0.1	December 1, 2011	First version, strings literals obfuscation, experimental
1.0	March 1, 2013	Major enhancements, based on work from Samuel Neves, Filipe Araujo [15] and on work from malware author “LeFF”
1.1	June 7, 2014	Enhancements for Hack In Paris 2014. Choose obfuscation algorithm randomly, experiments with finite state machines

Version	Date	Description
1.2	September 26, 2014	Enhancements for Black Hat Europe 2014. Choose finite state machine (FSM) randomly from a set, change FSM behavior depending on a runtime value (debugger detection)
2.0	2025	Complete rewrite with C++20, better obfuscation techniques, more user-friendly interface, compile-time AES encryption

5.7 | Copyright and License of the Library

Bibliography

[1] S. Andrivet, "C++11 metaprogramming applied to software obfuscation - Black Hat Europe 2014," 2014.

[2] B. Stroustrup, *The C++ programming language: C++ 11*, 4. ed., 4. print. Upper Saddle River, NJ: Addison-Wesley, 2015.

[3] N. Pantazopoulos, "Automating Pikabot's String Deobfuscation." 2024.

[4] K. Henson, "TrickBot gang uses template-based metaprogramming in Bazar malware." 2022.

[5] A. Parata, "Deobfuscating C++ ADVobfuscator with Sojobo and the B2R2 binary analysis framework." 2020.

[6] Mandiant, "FLOSS Version 2.0." 2022.

[7] ISO, "C++20 — ISO/IEC 14882:2020." 2020.

[8] Wikipedia, "Obfuscation (software) — Wikipedia, The Free Encyclopedia." 2026.

[9] B. Barak *et al.*, "On the (im)possibility of obfuscating programs," *Journal of the ACM*, vol. 59, no. 2, pp. 1–48, Apr. 2012, doi: 10.1145/2160158.2160159.

[10] J. Cappaert, "Code Obfuscation Techniques for Software Protection," Doctoral dissertation, 2012. [Online]. Available: https://cosicdatabase.esat.kuleuven.be/backend/publications/files/these/199

[11] M. Madou, B. Anckaert, B. Bus, K. De Bosschere, J. Cappaert, and B. Preneel, "On the Effectiveness of Source Code Transformations for Binary Obfuscation.," 2006, pp. 527–533.

[12] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*, Pearson new international edition, second edition. in Pearson custom library. Essex: Pearson, 2014.

[13] W. H. Payne, J. R. Rabung, and T. P. Bogyo, "Coding the Lehmer pseudo-random number generator," *Communications of the ACM*, vol. 12, no. 2, pp. 85–86, Feb. 1969, doi: 10.1145/362848.362860.

[14] S. K. Park and K. W. Miller, "Random number generators: good ones are hard to find," *Communications of the ACM*, vol. 31, no. 10, pp. 1192–1201, Oct. 1988, doi: 10.1145/63039.63042.

[15] S. Neves and F. Araujo, "Binary code obfuscation through C++ template metaprogramming," 2012.

This document was generated with Typst using latexlike-report 1.0.0.