

## A collage of various programming language logos and icons. The central focus is a large yellow circle containing a blue hexagon with a white 'C' and a blue triangle with two white plus signs. Surrounding this are several other logos: PHP (top left), Python (bottom left), Ruby (bottom right), and Java (bottom right). There are also various icons like a magnifying glass, a plus sign, a minus sign, a percentage sign, a building, a phone, and a document.

LG 전자  
김주완



# Who am I

주중엔

평범한 developer

주말엔

한때 cycle rider

종종 swimmer

가끔 rock climber



# 오늘은

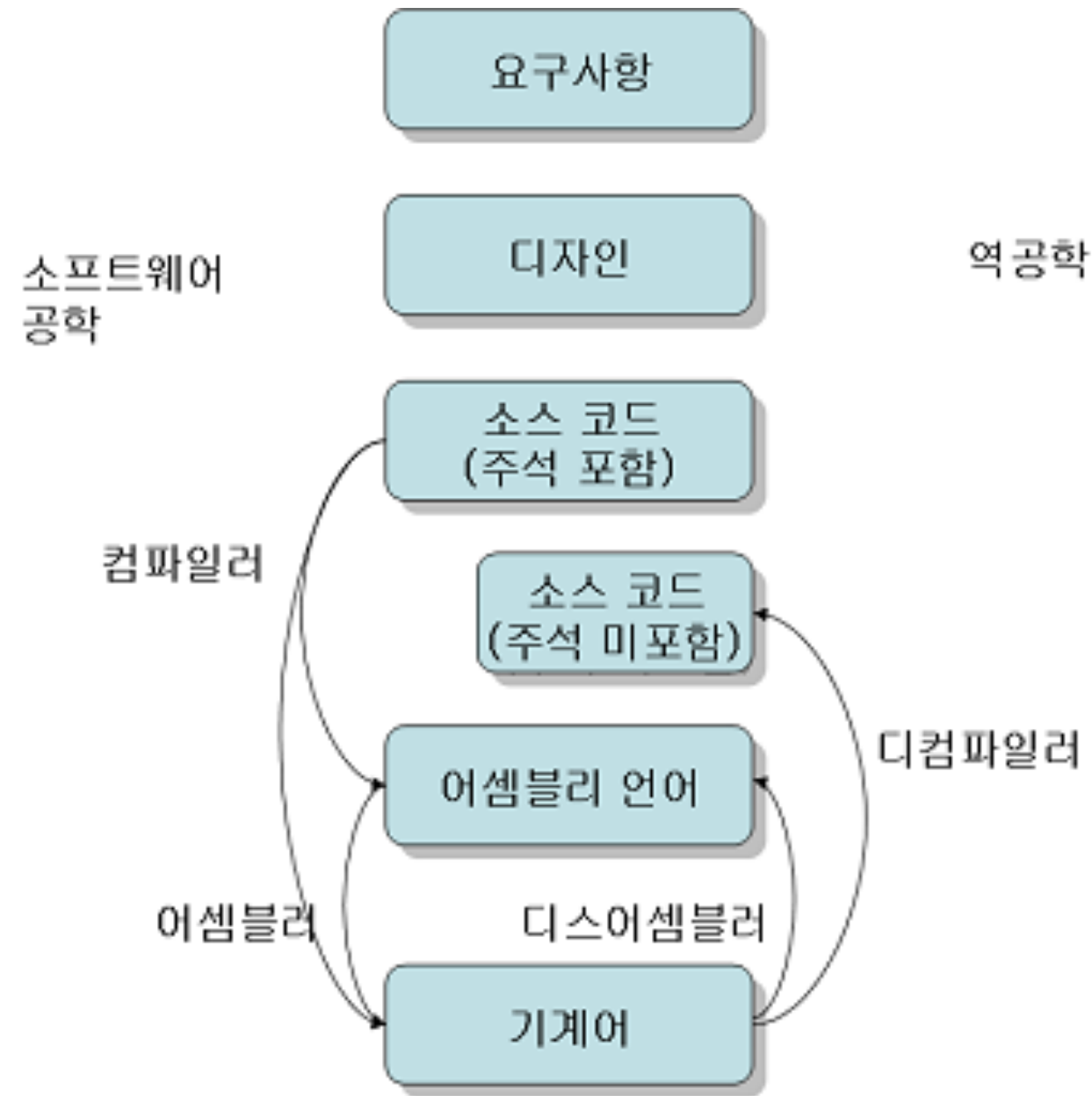
- 난독화?
- C++ Template metaprogramming
- 난독화 library 분석을 통해 어떻게 구현하는지  
ADVObfuscator(<https://github.com/andrivet/ADVobfuscator>)

# Disclaimer

- 이렇게 사람 많은데서 발표 처음이에요
- 저는 C++ 전문가는 아닙니다.
- 보안 엔지니어로서 난독화 기법이 왜 필요한지,  
C++ Template metaprogramming 을 이용해  
난독화 기능이 구현 가능하다는 점을 소개 하려 합니다.



# Reverse engineering



원본 소스코드

```
int foo1(int a) {  
    return a+1;  
}
```

디스어셈블링 된 어셈블리

```
.text:0000BC84 ; int __fastcall foo1(int a)  
.text:0000BC84 _24foo1i  
.text:0000BC84  
.text:0000BC84 a = R0  
.text:0000BC84 01 30 ADDS a, #1  
.text:0000BC86 70 47 BX LR  
.text:0000BC86 ; End of function foo1(int)
```

디컴파일 된 소스코드
















```
int __fastcall foo1(int a)  
{  
    return a + 1;  
}
```

디컴파일러의 성능 =  
얼마나 원 소스코드와 유사하게 디컴파일 되는가

# Decompile

한번쯤 들어봤을  
가전용 Android App

Native library  
디컴파일 결과

Function name	
	RootChecker::RootChecker(_JNIEnv *,_j...
	RootChecker::IsRooted(void)
	RootChecker::~~RootChecker()
	EnvironmentChecker::IsDebugMode(void)
	EnvironmentChecker::IsInEmulator(void)
	TamperingCheckerServer::TamperingCh...
	RootChecker::Init(void)
	RootChecker::IsRootedSuExistCheckJav...
	RootChecker::IsRootedRootManagement...
	RootChecker::IsRootedRunSuCommand...
	RootChecker::CheckBinaryByName(std:...
	EnvironmentChecker::Init(void)
	TamperingChecker::GetAPKCertFingerPr...
	TamperingChecker::~~TamperingChecker()
	TamperingCheckerStandalone::~~Tampe...

함수명이 다 보임 !

```
int __fastcall RootChecker::IsRooted(RootChecker *this)
{
    RootChecker *v1; // r4@1
    int v2; // r0@1
    int v3; // r1@1
    int result; // r0@1
    int v5; // r0@2
    int v6; // r5@2
    int v7; // r0@2
    int v8; // r6@2
    int v9; // r0@2

    v1 = this;
    LOBYTE(v2) = RootChecker::Init(this);
    v3 = v2;
    result = 1;
    if ( v3 == 1 )
    {
        LOBYTE(v5) = RootChecker::IsRootedSuExistCheckJava(v1);
        v6 = v5;
        LOBYTE(v7) = RootChecker::IsRootedRootManagementAppCheck(v1);
        v8 = v7 | v6;
        LOBYTE(v9) = RootChecker::IsRootedRunSuCommandCheck(v1);
        result = v9 | v8;
    }
    return result;
}
```

소스코드와 거의 유사하게 보임 !

# Problem

## 최근 해킹 사례

- 가전 기기 원격 해킹  
안드로이드 App. 리버싱 & 분석  
서버 로그인 취약점 공격  
인가되지 않는 **사용자 기기 원격 제어**
- 사내 보안 App. 권한 우회  
MDM(Mobile Device Management) 앱 리버싱 & 분석  
(카메라/마이크 등) 사내에서 사용 금지 → **권한 우회하여 카메라/마이크 사용**

## 시사점

취약점 줄여나가며 / 지속적으로 개선 가능한 형태로 SW 만드는게 최선  
동시에 **분석을 어렵게** 하여 공격 시도를 줄이는 방법도 필요

# Solution ?

- 난독화 (Obfuscation)
  - 소프트웨어를 보호하기 위한 기법 중 하나
  - not a bullet-proof solution

“Deliberate act of creating source or machine code difficult for humans to understand”

–WIKIPEDIA, APRIL 2014



# Type of obfuscator

- **Source code obfuscator**
  - 컴파일 전에, 소스 코드를 난독화
  - source language 에 의존적
- **Binary code obfuscator**
  - 컴파일 이후, 바이너리 코드를 난독화
  - target machine에 의존적

# Source code obfuscator

- **Direct source code obfuscation**
  - 주로 프로그래머에 의해 수작업으로 난독화, 유지 보수 및 관리 X
- **Pre-processing obfuscator**
  - 컴파일 전에 전처리를 통해 난독화
- **Abstract syntax tree(AST) or Intermediate representation(IR) obfuscator**
  - Intermediate representation 형태로 변환하여 난독화, ex) o-llvm
- **Bytecode obfuscator**
  - (Java, .NET의 경우)컴파일러에 의해 generation된 bytecode 난독화
  - source code obfuscator와 binary obfuscator의 중간 형태

# (Pre-processing obfuscator) 어떻게 구현 ?

- 결국 난독화는 원래 프로그램의 semantic 은 유지하면서 코드를 변경하는 일
- macro? No!  
가능할수도 있지만 별로 상상하고 싶지 않음 ...
- **C++ Template metaprogramming !**

# metaprogramming?

Metaprogramming is writing  
programs that generate or  
manipulate code

# Template

- enable generic programming
- provide type safety

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

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

## Template Specialization

```
// 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<typename bool>
struct Vector {
    void set(int position, bool b);
    bool get(position);
    // ...
};
```

## Variadic templates

```
template <typename... T>
class tuple {
public:
    constexpr tuple();
    explicit tuple(const T&...);
    [...]
};

tuple<int, string, double> values1{123, "test", 3.14};
auto values2 = make_tuple(123, "test", 3.14);

cout << get<0>(values1);
cout << get<0>(values2);
```



# C++ Template metaprogramming

- Template을 활용하여, 컴파일 타임에 동작하는 프로그램을 만드는 프로그래밍
- 컴파일 시점에 많은 것을 결정하여, 실행 시점의 계산을 줄여줌
- Turing-complete

# Example #1

```
// 일반적인 factorial 함수
unsigned int factorial (unsigned int n) {
    return n == 0? 1: n * factorial(n-1);
}
```

```
// 템플릿 클래스로 factorial 함수를 정의
template <unsigned int n>
struct factorial {
    enum { value = n * factorial<n-1>::value };
}; // 재귀적으로 class를 생성하며 factorial의 값을 구함
```

```
template<>
struct factorial<0> {
    enum { value = 1 };
} // 재귀 함수의 종결점인 0 값에 대한 기본값
```

# Example #1

```
int val = factorial<4>::value; // 이러한 호출에 대해
//-----
struct factorial<4> {
    enum { value = n * factorial<3>::value }; // value = 24로 계산됨
};
struct factorial<3> {
    enum { value = n * factorial<2>::value }; // value = 6로 계산됨
};
struct factorial<2> {
    enum { value = n * factorial<1>::value }; // value = 2로 계산됨
};
struct factorial<1> {
    enum { value = n * factorial<0>::value }; // value = 1로 계산됨
};
struct factorial<0> {
    enum { value = 1 };
} // 재귀 함수의 종결점인 0 값에 대한 기본값
//-----
int val = 24; // 컴파일 완료 시점에 값이 치환됨
```

This program runs in **constant time**!

# Example #2

```
template <int length>
Vector<length>& Vector<Length>::operator+=(const Vector<Length>& rhs)
{
    for (int i=0 ; i<length ; ++i)
        value[i] += rhs.value[i];
    return *this;
}
```

```
// 컴파일 타임에 template으로 loop의 횟수를 알고 있기에, 컴파일러가 loop unrolling 수행
template <>
Vector<2>& vector<2>::operator+=(const Vector<2>& rhs)
{
    value[0] += rhs.value[0];
    value[1] += rhs.value[1];
    return *this;
}
```

# C++ Template metaprogramming

- **장점**

- 늘어나는 컴파일 시간, 좀 더 빨라지는 실행 시간
- generic programming

- **단점**

- 가독성이 떨어지고, 디버깅 어려움
- 컴파일 에러 발생 시, 원인 찾기 어려움



# 난독화 라이브러리 구현하기

- 문자열 난독화 (String obfuscation)
- 제어 흐름 난독화 (Control flow obfuscation)

# ADVObfuscator

- <https://github.com/andrivet/ADVobfuscator>

- **BSD-3 license**

- **Prerequisites**

C++ 11/14 지원하는 컴파일 환경

Boost library (MSM)

- **Compatibility**

ADVobfuscator has been tested with:

- Xcode (LLVM) 8.1.0 under Mac OS X 10.12
- GCC 7.2.0 under Debian 10 (buster - testing)
- Visual Studio 2017 (15.3.3) under Windows 10
- Boost 1.65.0

# String obfuscation

- 문자열을 알아보기 힘든 방식 (인코딩/암호화)으로 바꾸는 것
- 대상
  - string literal
  - error message / log information /
  - function & class name / URL / etc ...
- 라이브러리 구현 목표
  - developer-friendly syntax
  - only C++ without external tool
  - compile time obfuscation, deobfuscation at runtime
  - obfuscation/deobfuscation cost minimize
  - each string, obfuscate differently
  - each compilation, obfuscate differently

# Goal

## 난독화 미적용

```
void foo3() {  
    LOGD("special string: %s", "secret string");  
}
```

## 디컴파일 결과

```
int sub_9A08()  
{  
    return j_j__android_log_print(3, "test", "special string: %s", "secret string");  
}
```

## 난독화 적용

```
void foo2() {  
    LOGD(OBFUSCATED("special string: %s"), OBFUSCATED("secret string"));  
}
```

## 디컴파일 결과

```
v31 = 12;  
v32 = 86;  
v33 = 77;  
v34 = 75;  
v35 = 28;  
v0 = 0;  
v36 = 0;  
do  
{  
    *(&v18 + v0) ^= (_BYTE)v17 + (_BYTE)v0;  
    ++v0;  
}  
while ( v0 != 18 );
```

```
v1 = 0;  
v36 = 0;  
v3 = 117;  
v4 = 103;  
v5 = 101;  
v6 = 116;  
v7 = 103;  
v8 = 118;  
v9 = 34;  
v10 = 117;  
v11 = 118;  
v12 = 116;  
v13 = 107;  
v14 = 112;  
v15 = 105;  
v16 = 0;  
do  
    *(&v3 + v1++) -= 2;  
while ( v1 != 13 );  
_android_log_print(3, "test", &v18, &v3);
```

알아보기 힘들 !

# 1st implementation - Approach

- 가장 단순하게
- 고정 길이 문자열
- XOR 이용 encoding
- XOR 의 key 값은 0x55로 hardcoding



# 1st implementation

```
template<int... I> // I is a list of indexes 0, 1, 2, ...
struct MetaString1
{
    // Constructor. Evaluated at compile time
    constexpr MetaString1(const char* str)
        : buffer_{encrypt(str[I])...} { }          // buffer_{encrypt(str[0]), encrypt(str[1]), encrypt(str[2]), ...}

    // Runtime decryption. Most of the time, inlined
    inline const char* decrypt()
    {
        for(size_t i = 0; i < sizeof...(I); ++i)
            buffer_[i] = decrypt(buffer_[i]);
        buffer_[sizeof...(I)] = 0;
        return buffer_;
    }

private:
    // Encrypt / decrypt a character of the original string
    constexpr char encrypt(char c) const { return c ^ 0x55; }
    constexpr char decrypt(char c) const { return encrypt(c); }

private:
    // Buffer to store the encrypted string + terminating null byte
    char buffer_[sizeof...(I) + 1];
};
```

```
#define OBFUSCATED1(str) (MetaString1<0, 1, 2, 3, 4, 5>(str).decrypt())
```

```
cout << OBFUSCATED1("Britney Spears") << endl;
```

**“Britne” 까지만 obfuscate  
나머진 truncate**

# 2nd implementation

- In 1st implementation `(MetaString1<0, 1, 2, 3, 4, 5>(str).decrypt())`

- 문자열 길이에 따른 index list 생성

- C++ 11

```
template<int... I>
struct Indexes { using type = Indexes<I..., sizeof...(I)>; };

template<int N>
struct Make_Indexes { using type = typename Make_Indexes<N-1>::type::type; };
// Indexes<0, 1, 2, 3, ..., N>

template<>
struct Make_Indexes<0> { using type = Indexes<>; };
```

- C++ 14의 std:index\_sequence 이용 가능

```
#define OBFUSCATED2(str) (MetaString2<Make_Indexes<sizeof(str) - 1>::type>(str).decrypt())
```

```
cout << OBFUSCATED2("Katy Perry") << endl;    전체 문자열 obfuscate
```

# 3rd implementation

- Problem in 2nd implementation
  - hard-coded key
- Randomization of encryption key
- New template parameter for Key

```
template<int... I, int K>  
struct MetaString3<Indexes<I...>, K>
```

난수 생성기 이용 random 값 사용

# Generating (pseudo-) random numbers

- Need compile time random number, not runtime
- Linear congruential engine 사용하여 난수 생성
  - Macros (C & C++) 이용 seed & key 생성
    - **\_\_TIME\_\_**: compilation time (standard)
    - **\_\_COUNTER\_\_**: incremented each time it is used  
(표준은 아니지만 대부분의 compiler 에서 지원)

# 3rd implementation

- Different **seed** for each compilation

- `__TIME__`

```
constexpr char time[] = __TIME__; // __TIME__ has the following format: hh:mm:ss in 24-hour time
// Convert time string (hh:mm:ss) into a number
constexpr int DigitToInt(char c) { return c - '0'; }
const int seed = DigitToInt(time[7]) +
    DigitToInt(time[6]) * 10 +
    DigitToInt(time[4]) * 60 +
    DigitToInt(time[3]) * 600 +
    DigitToInt(time[1]) * 3600 +
    DigitToInt(time[0]) * 36000;
```

- Different **key** for each string

- `__COUNTER__`

```
template<int... I, int K>
struct MetaString3<Indexes<I...>, K>
```

```
#define OBFUSCATED3(str) (MetaString3<Make_Indexes<sizeof(str) - 1>::type,
                                MetaRandomChar3<__COUNTER__>::value>(str).decrypt())
```

Random key 생성



# 4th implementation

- Randomization of encryption algorithm

```
template<int N, char Key, typename Indexes>  
struct MetaString4;
```

```
template<char K, int... I>  
struct MetaString4<0, K, Indexes<I...>>    encrypt = c XOR random key
```

```
template<char K, int... I>  
struct MetaString4<1, K, Indexes<I...>>    encrypt = c XOR (random key + c)
```

```
template<char K, int... I>  
struct MetaString4<2, K, Indexes<I...>>    encrypt = c + random value
```

```
#define DEF_OBFUSCATED4(str) MetaString4<MetaRandom<__COUNTER__, 3>::value, ...
```

```
DEF_OBFUSCATED4("test"); // encrypt = c XOR 0x55  
DEF_OBFUSCATED4("test"); // encrypt = c XOR 0x61  
DEF_OBFUSCATED4("test"); // encrypt = c XOR (0x37 + c)  
DEF_OBFUSCATED4("test"); // encrypt = c + 13
```

같아 보이지만,  
모두 다른 MetaString  
다른 key  
다른 난독화 방식 사용

# 문자열 난독화 미적용

## 소스코드

```
void foo3() {  
    LOGD("special string: %s", "secret string");  
}
```

## 난독화 미적용 분석 결과

```
.text:0000BDC8  
.text:0000BDC8  
.text:0000BDC8 03 49  
.text:0000BDCA 04 A2  
.text:0000BDCC 08 A3  
.text:0000BDCE 03 20  
.text:0000BDD0 79 44  
.text:0000BDD2 3E F0 19 BD  
.text:0000BDD2
```

```
; void foo3()  
_Z4foo3v  
    LDR        R1, =(aTest_0 - 0xBDD4) ; CODE XREF: Java_com_lgt  
    ADR        R2, aSpecialStringS ; "special string: %s"  
    ADR        R3, aSecretString ; "secret string"  
    MOVS       R0, #3  
    ADD        R1, PC ; "test"  
    B.W        j_j__android_log_print  
; End of function foo3(void)
```

주요 문자열 확인 가능 !

# 문자열 난독화 적용

```
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
.text:0000BC88
```

```
; void Foo2()
_Z4foo2v
```

```
var_44= -0x44
var_40= -0x40
var_3F= -0x3F
var_3E= -0x3E
var_3D= -0x3D
var_3C= -0x3C
var_3B= -0x3B
var_3A= -0x3A
var_39= -0x39
var_38= -0x38
var_37= -0x37
var_36= -0x36
var_35= -0x35
var_34= -0x34
```

```
-----
.text:0000BCD2 8D F8 1F 20
.text:0000BCD6 8D F8 20 00
.text:0000BCDA 8D F8 21 30
.text:0000BCDE 8D F8 22 80
.text:0000BCE2 8D F8 23 10
.text:0000BCE6 6F 21
.text:0000BCE8 8D F8 24 10
.text:0000BCEC 68 21
.text:0000BCEE 8D F8 25 10
.text:0000BCF2 3B 21
.text:0000BCF4 8D F8 26 10
.text:0000BCF8 26 21
.text:0000BCFA 8D F8 27 20
.text:0000BCFE 8D F8 28 10
.text:0000BD02 8D F8 29 00
.text:0000BD06 06 A8
```

```
-----
STRB.W      R2, [SP,#0x48+var_29]
STRB.W      R0, [SP,#0x48+var_28]
STRB.W      R3, [SP,#0x48+var_27]
STRB.W      R8, [SP,#0x48+var_26]
STRB.W      R1, [SP,#0x48+var_25]
MOVS        R1, #0x6F
STRB.W      R1, [SP,#0x48+var_24]
MOVS        R1, #0x68
STRB.W      R1, [SP,#0x48+var_23]
MOVS        R1, #0x3B
STRB.W      R1, [SP,#0x48+var_22]
MOVS        R1, #0x26
STRB.W      R2, [SP,#0x48+var_21]
STRB.W      R1, [SP,#0x48+var_20]
STRB.W      R0, [SP,#0x48+var_1F]
ADD         R0, SP, #0x48+var_30 ; this
```

```
.text:0000BD94
.text:0000BD94
.text:0000BD94
.text:0000BD94
.text:0000BD94 00 21
.text:0000BD96
.text:0000BD96
.text:0000BD96
.text:0000BD96 12 29
.text:0000BD98 08 BF
.text:0000BD9A 70 47
.text:0000BD9C 42 5C
.text:0000BD9E FF 32
.text:0000BDA0 42 54
.text:0000BDA2 01 31
.text:0000BDA4 F7 E7
.text:0000BDA4
```

```
; const unsigned __int8 *__fastcall andrivet::ADUobfusca
_ZN8andrivet13ADUobfuscator10MetaStringILi2ELc65ENS0_7Ir
; CODE XREF: foc
; andrivet::ADUc
```

```
this = R0
MOVS        R1, #0
```

```
loc_BD96    ; CODE XREF: anc
i = R1      ; size_t
```

```
CMP         i, #0x12
IT EQ
BXEQ        LR
LDRB        R2, [this,i]
ADDS        R2, #0xFF
STRB        R2, [this,i]
ADDS        i, #1
B           loc_BD96
```

```
; End of function andrivet::ADUobfuscator::MetaString<2,
```

주요 문자열 확인 및 분석 어려움 !

문자열 마다 암호화 key, algorithm 다름  
(컴파일 시 변경 되기에) 바이너리 마다 다름

-> Obfuscated characters

-> Deobfuscated

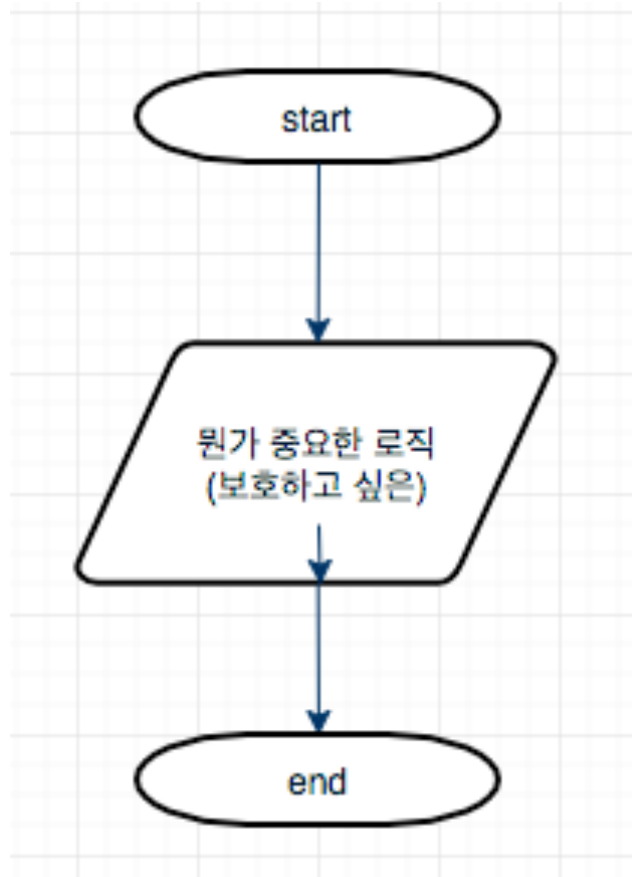
```
const unsigned __int8 *__fastcall
{
    int i; // r1@1
    for ( i = 0; i != 18; ++i )
        --result[i];
    return result;
}
```

# 난독화 라이브러리 구현하기

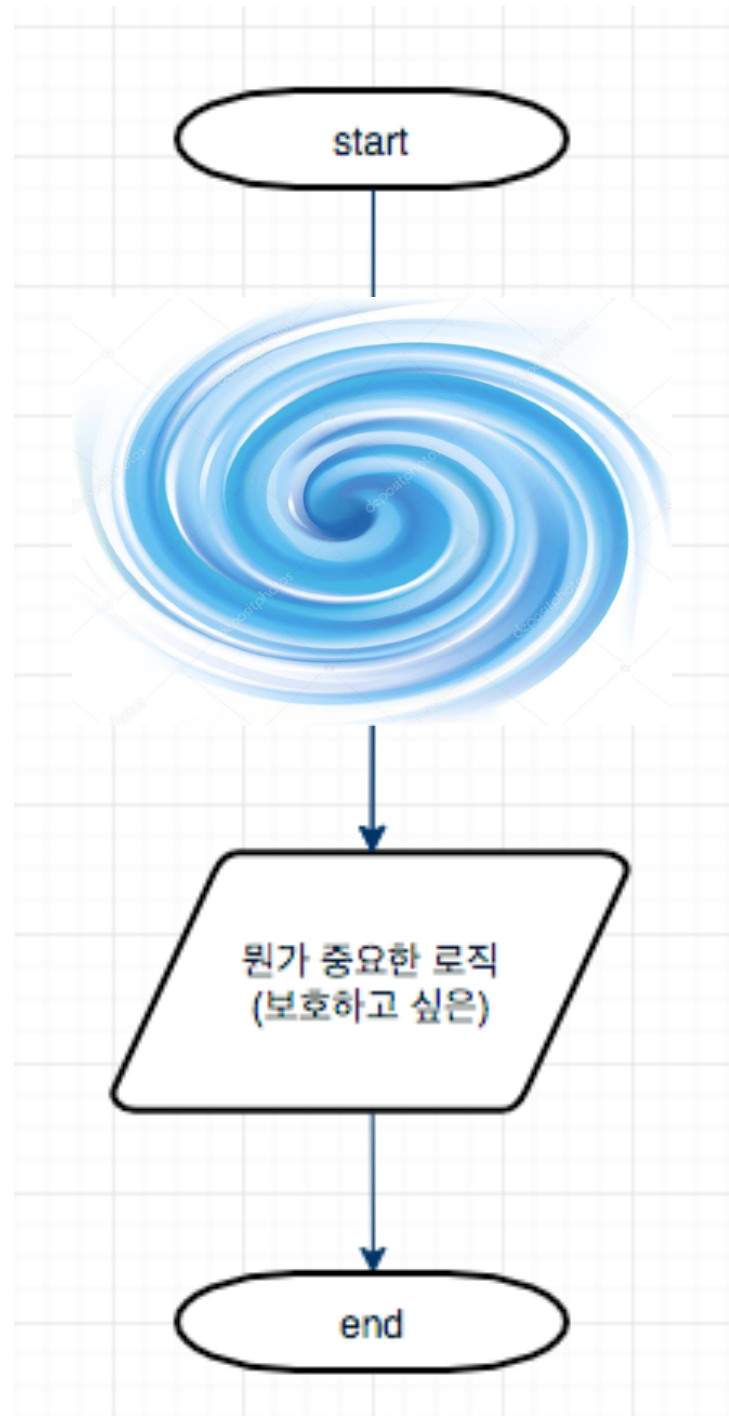
- 문자열 난독화 (String obfuscation)
- 제어 흐름 난독화 (Control flow obfuscation)

# 제어 흐름 난독화

Before

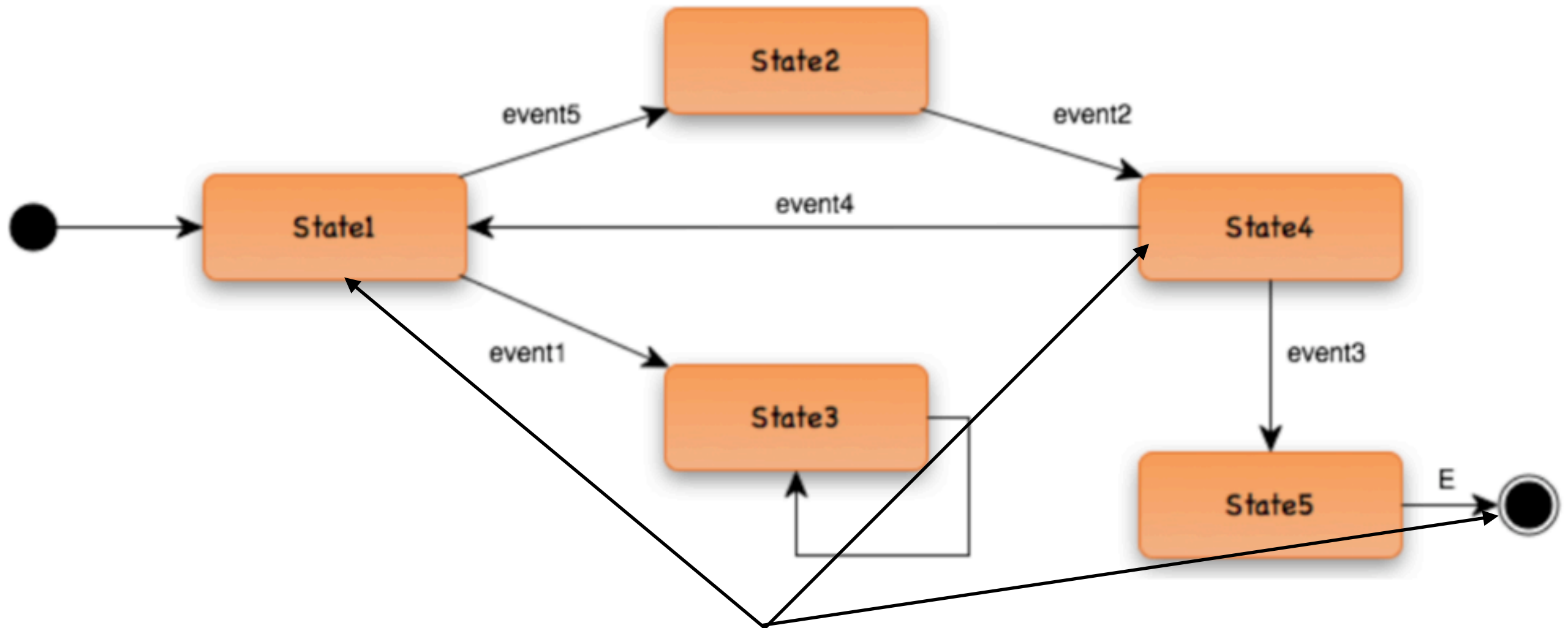


After



- 아무것도 하지 않는 코드 삽입
- 코드를 여기저기 복사하고 옮김
- 인코딩

# Finite State Machine



중간 어딘가에서  
보호하고 싶은 함수를 실행

# Boost Meta State Machine(MSM) library

- Generate code (FSM) at compile-time

```
// --- Transition table
struct transition_table : mpl::vector<
//   Start      Event      Next      Action      Guard
//   +-----+-----+-----+-----+-----+
Row < State1  , event5      , State2
Row < State1  , event1      , State3
//   +-----+-----+-----+-----+-----+
Row < State2  , event2      , State4
//   +-----+-----+-----+-----+-----+
Row < State3  , none        , State3
//   +-----+-----+-----+-----+-----+
Row < State4  , event4      , State1
Row < State4  , event3      , State5
//   +-----+-----+-----+-----+-----+
Row < State5  , E           , Final, CallTarget
//   +-----+-----+-----+-----+-----+
> {};
```

**types**

**template parameter**

```
static inline void run(StateMachine& machine, F f, Args&&... args)
{
    machine.start();
    // Generate a lot of transitions (at least 55, at most 98)
    Unroller<55 + MetaRandom<__COUNTER__, 44>::value>{}([&]()
    {
        machine.process_event(event5{});
        machine.process_event(event2{});
        machine.process_event(event4{});
    });
    machine.process_event(event5{});
    machine.process_event(event2{});
    machine.process_event(event3{});
    machine.process_event(E{f, args...});
}
```

# 제어 흐름 난독화 미적용

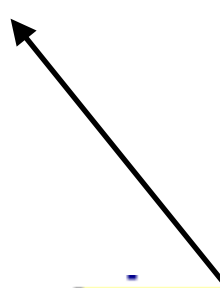
- 원본 소스 코드

```
int a = 3;  
b = foo1(a);  
LOGD("org foo1: %d", b);
```

- 디컴파일 결과

```
int __fastcall sub_9870(int a1)  
{  
    return a1 + 1;  
}
```

```
v4 = sub_9870(3);  
_android_log_print(3, "test", "org foo1: %d", v4, 3);
```





# 제어 흐름 난독화 적용

- 원본 소스 코드

```
int a = 3;  
auto c = OBFUSCATED_CALL_RET(int, foo1, a);  
LOGD("obfuscatred foo1: %d", c);
```

- 디컴파일 결과

```
v9 = (char *)&loc_9ABE + 2;  
sub_9534((int)&v14, (int)&v9, 1);  
v5 = v15;  
sub_68D2((int)&v16);  
_android_log_print(3, "test", "obfuscatred foo1: %d", v5, v8);
```

foo1 주소 + random 값

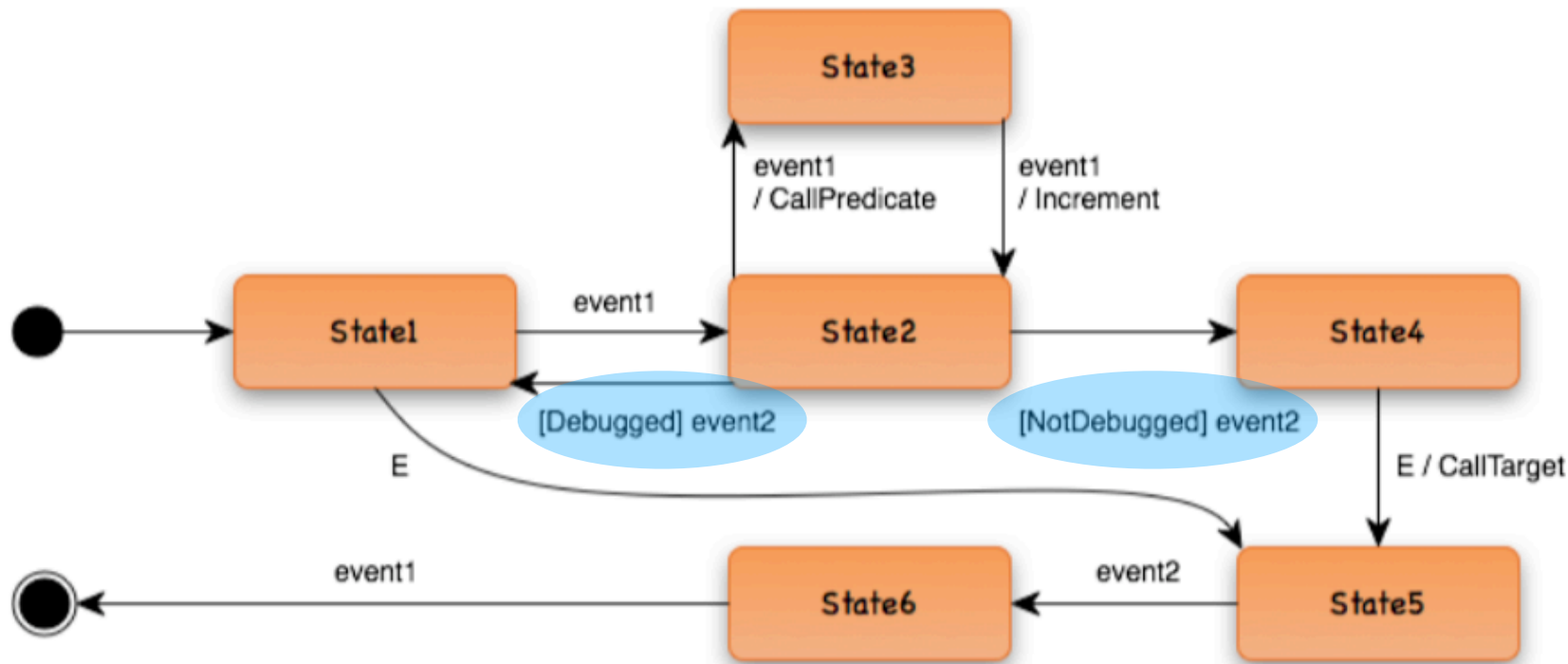
FSM 중, v14 - random 값 = foo1 주소 이용 호출

FSM 으로 진입하면 분석 불가

제어 흐름 확인 어려움 !

FSM을 통해 생성된 코드는 매번 변경됨  
함수 주소값도 변경해서 전달

# 제어 흐름 난독화 - more



```

struct transition_table : mpl::vector<
//   Start   Event   Next   Action   Guard
//   +-----+-----+-----+-----+-----+
Row < State1 , event1 , State2
Row < State1 , E     , State5
//   +-----+-----+-----+-----+-----+
Row < State2 , event1 , State3 , CallPredicate
Row < State2 , event2 , State1 , none      , Debugged
Row < State2 , event2 , State4 , none      , NotDebugged
//   +-----+-----+-----+-----+-----+
Row < State3 , event1 , State2 , Increment
//   +-----+-----+-----+-----+-----+
Row < State4 , E     , State5 , CallTarget
//   +-----+-----+-----+-----+-----+
Row < State5 , event2 , State6
//   +-----+-----+-----+-----+-----+
Row < State6 , event1 , Final
//   +-----+-----+-----+-----+-----+
> {};
  
```

조건 (= debugger / virtual machine 탐지)  
에 따라 분기 하는 FSM

# Demo

**Android Application의 native에 난독화 적용**

# 적용 예시

- 문자열 난독화

주요한 문자열은 암호화 알고리즘 이용

- 제어 흐름 난독화

FSM 구조 변경 복잡도 증가

보호하는 함수를 중간에 호출하도록 수정

조건절을 사용하여 분기 처리

안드로이드 루팅 / 앱 위변조 / 에뮬레이터 여부 등

감사합니다.