РЕФЕРАТ

Актуальність теми. Захист коду програми від декомпіляції завжди була, є і буде актуальною. В першу чергу це відноситься до комерційних продуктів, які, як правило, представляють фірмову таємницю для конкурентів. Абсолютного захисту від декомпіляції НЕ існує, але на сьогоднішній день у розробників програмного забезпечення є безліч програмних продуктів, які називаються обфуськатор (від англійського "obfuscate - ставити в глухий кут, заплутувати"), які здатні ускладнити розуміння декомпілювати коду.

Об'єктом дослідження є засоби захисту інтелектуальної власності на базі алгоритму обфускації для вихідного C/C++ коду на архітектурі XScale.

Предметом дослідження є способи захисту інтелектуальної власності розробленої за допомогою мов C/C++ на архітектурі XScale.

Методи дослідження. В роботі використовуються методи оптимізації, методи системного аналізу, теорії графів, а також методів моделювання.

Мета роботи полягає у дослідженні і підвищенні ефективності методу захисту висхідного програмного коду на основі алгоритму обфускації на базі архітектури Xscale від зворотної інженерії.

Для досягнення поставленої мети в роботі вирішуються наступні задачі.

1. Дослідження існуючих методів реалізації алгоритму обфускації.

2. Дослідження різних варіантів імплементації алгоритму обфускації на мові програмування C/C++.

3. Порівняння роботи досліджених імплемплементацій на різноманітних висхідних кодах.

Наукова новизна одержаних результатів полягає в тому, що у роботі була підвищена ефективність протидії реверсінжинірінгу задля програм написаних на C/C++ на архітектурі XScale, а саме:

1. Запобігання сприйняття зловмисником алгоритмів програми;
2. Маскування робочого коду за конструкціями, що використовуються;
3. Невеликий вплив на швидкодію програми.

Практична цінність зводиться до підвищення ефективності захисту

оригінальних алгоритмів програмного забезпечення для збереження комерційної таємниці підприємства власника програмного забезпечення мінімалізуючи вплив на роботу оригінальної програми.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на:

- «Використання алгоритмів обфускації для захисту блоків інтелектуальної власності» - Scientific Journal Applied Aspects of Information Technology, Vol. 4, №2;
- «Використання методу захисту білої скриньки для інформації» - Scientific Journal Applied Aspects of Information Technology, Vol. 4, №2;

Структура та обсяг роботи.

Магістерська дисертація складається з вступу, трьох розділів, висновків та додатків.

У вступі надано загальну характеристику програмного коду, проблематику розробки алгоритму обфускації на архітектурі XScale, сформульовано мету дослідження, показано практичну цінність роботи.

У першому розділі надано детальне обґрунтування актуальності напрямку досліджень, виконано оцінку поточного стану в даній сфері, представлено теоретичний огляд виділення ключових фраз з неструктурованих текстів.

У другому розділі розроблено та описано аглоритм обфускації для захисту від реверсінжинірінгу інтелектуальнох власності створеної за допомогою мов програмування C/C++ на архітектурі XScale.

У третьому розділі проведено апробацію.

У висновках проаналізовано отримані результати роботи.

Ключові слова: алгоритм обфускації, алгоритм Колберга, архітектура XScale , мова програмування C/C++.

ABSTRACT

Actuality of the theme. Protecting program code from decompilation has always been, is and will be relevant. This primarily applies to commercial products, which are usually a trademark for competitors. There is no absolute protection against decompilation, but today software developers have many software products called obfuscators (from the English "obfuscate - to put in a dead end, confuse"), which can complicate the understanding of decompiled code.

The object of research is the means of intellectual property protection based on the obfuscation algorithm for the source C / C ++ code on the XScale architecture.

The subject of research is ways to protect intellectual property developed using C / C ++ languages on the XScale architecture.

Research methods. The paper uses optimization methods, systems analysis methods, graph theory, as well as modeling methods.

The scientific novelty of the obtained results is that the efficiency of counteraction to reverse engineering for programs written in C / C ++ on XScale architecture is increased.

The practical value comes down to counteracting reverse engineering and protecting against it programs on the XScale architecture written in C / C ++ programming languages.

Approbation of work. The main provisions and results of the work were presented and discussed at:

• "Using obfuscation algorithms to protect intellectual property blocks" - Scientific Journal Applied Aspects of Information Technology, Vol. 4, №2;

• "Using the method of protecting the white box for information" - Scientific Journal Applied Aspects of Information Technology, Vol. 4, №2;

Structure and scope of work.

The master's dissertation consists of an introduction, three sections, conclusions and appendices.

The introduction provides a general description of the program code, the problems of developing an obfuscation algorithm on the XScale architecture, formulates the purpose of the study, shows the practical value of the work.

The first section provides a detailed justification of the relevance of the research direction, an assessment of the current situation in this area, a theoretical overview of the selection of key phrases from unstructured texts.

The second section develops and describes the agglomeration of obfuscation to protect against reverse engineering of intellectual property created using C / C ++ programming languages on the XScale architecture.

In the third section the approbation is carried out.

The results of the work are analyzed in the conclusions.

Keywords: obfuscation algorithm, Colberg algorithm, XScale architecture, C / C ++ programming language.

3 РЕАЛІЗАЦІЯ РОЗРОБЛЕНОГО АЛГОРИТМУ

3.1 Загальна структура розробленої програми

Для демонстрації роботи розглядаємого алгоритму розроблено метод його реалізації, а також створені тести для перевірки роботи імплементації алгоритму. Особливістю розробленої програми є використання у ній алгоритмів обфускації для архітектурних XScale.

Програма складається з наступних модулів:

- Сам обфускатор, який складається:
  - реалізації функціональної стейт машини, яка генерує випадкові кодові події для алгоритму обфускації;
  - кастомної реалізації рандому, для більш непередбачуванної для зловмисника логіки генерації випадковості;
  - реалізації алгоритму генерації і роботи з чергами, для кодування/декодування строкових літералів згідно алгоритму обфускації;
  - реалізації алгоритму кодування/декодування строкових літералів за допомогою кастомного рандому і реалізованих способів роботи з чергами;
- Приклад використання програми, де вказано як використовувати реалізацію кастомного алгоритму рандому з користувацьким зерном, використання стейт машини – задання їй реальних дій та заглушок, переходів від одних подій до інших і задання самих подій, а також приклад використання реалізації алгоритму кодування/декодування строкових літералів, використання різних операцій в якості кодувального ключа і вивід результатів у закодовану таблицю літералів;
- Реалізація упакування розробленого методу в хмарне рішення за допомогою Docker для різних платформ розробки. Також

розроблений приклад використання запакованного рішення з допомогою мови програмування Python, і скрипт який пакує розроблений алгоритм в один файл для зручності пакування та підключення;

- Реалізація юніт тестів для кастомної реалізації алгоритму рандому, кодування/декодування строкових літералів, реалізованих алгоритмів роботи з чергами, і реалізації функіональної стейт машини та роботи з нею.

Структура проєкту складається з таких папок:

- obfuscator – папка, де знаходится імплементація основних алгоритмів роботи с кодуванням/декодуванням строкових літералів, функціональною стейт машиною, реалізації кастомного рандому та черг дій;
  - fsm.hpp – імплементація функціональної стейт машини та методів роботи з нею;
  - random.hpp – реалізація алгоритму кастомного рандому для більш складного для зловмиснику вихідного результату;
  - sequence.hpp – імплементація черги кодових подій та інтерфейсу роботи з ними;
  - string.hpp – реалізація алгоритмів кодування/декодування строкових літералів на основі різних бітових операцій;
- sample – папка, що мінасстить приклади роботи з реалізованними алгоритмами роботи с кодуванням/декодуванням строкових літералів, функціональною стейт машиною та реалізації кастомного рандому;
  - random.cpp – приклад використання алгоритму рандому;
  - state_machine.cpp – приклад використання стейт машини;
  - string_obfs.cpp – приклад використання кодування/декодування строкових літералів;

- script – папка, що містить у собі скрипти для збирання імплементованної бібліотеки в докер під різні платформи, а також тест на її працездатність після зборки;
    - merge.py – скрипт для докеру що збирає імплементації алгоритмів в різних файлах і генерує один зручний для підключення файл;
    - string_obfs_tester.py – скрипт для тестувння зібранної бібліотеки попереднім скриптом;
- test – папка, де знаходяться різноманітні тести розробленних алгоритмів роботи с кодуванням/декодуванням строкових літералів, функціональною стейт машиною, реалізації кастомного рандому та черг дій;
    - fsm.cpp – імплементація тестів для функціональної стейт машини;
    - random.cpp – імплементація тестів для алгоритму кастомного рандому;
    - sequence.cpp – імплементація тестів для черг кодових подій та інтерфейсів роботи з ними;
    - string.cpp – імплементація тестів для реалізації алгоритмів кодування/декодування строкових літералів;

Розглянемо ці елементи розробленої програми, їх особливості реалізації та їх функції у всьому проекті, детальніше у наступних пунктах даного розділу.

3.2 Функціональна стейт машина

Функціональна стейт машина – фрагмент програми, що реалізує стейт машину для створення послідовності кодових подій і створює дерево переходів від одної події до іншої на певних умовах та за певним порядком, заданим користувачем системи або рандомом, передаючи або ні результати таких подій до наступного вузла (стейту). Метою такої стейт машини є перетасувати логічні кодові події що містять реальну бізнес логіку програми з заглушками, які не мають сенсу з точки зору результати програми, проте вони роблять процес зворотньої інженерії більш складним, оскільки в готовому бінарному файлі становится набагато важче знайти частини коду що містять реальний алгоритмічний зміст. Дана реалізація дозволяє сформувати стейти і список кодових подій на етапі компіляції програми, за рахунок чого майже не збільшується швидкодія алгоритму. Розглянемо більш детально частини реалізації стейт машини нижче.

- Реалізація сторення наступного кроку в стейт машині. В ній задається подія по якій відбувається переход до цього стану, дія яка при цьому відбувается і сам стан який присвоюється стейт машиною.

```
template <typename Event, typename State, void(*Action)() = FreeAction>
    struct Next {
        using event = Event;
        using state = State;
        constexpr static void(*action)() = Action;
    };
```

- Стейдж – це набір даних стейта, яка включає сам стейт, дію та ссилки на наступні стейти.

```
template <typename State, typename... Nexts>
struct Stage {
```

```
    using state = State;
    template <typename NextInfo, typename Event>
    using act = std::conditional_t<
        std::is_same_v<typename NextInfo::event, Event>, NextInfo, Pass>;
    template <typename Event>
    using next = typename First<None, act<Nexts, Event>...>::type;
};
```

- Реалізації наступниї стейджів – пустих, та тих що містять іменну подію

```
template <typename Stage_, typename Event>
struct next_stage {
    using type = typename Stage_::template next<Event>;
};

template <typename Event>
struct next_stage<None, Event> {
    using type = None;
};
```

- Реалізація «збуджувача» стейту і механізму повертання даних

```
template <typename State>
struct action_invoker {
    static auto action() {
        State::action();
        return typename State::state{};
    }
};

template <>
struct action_invoker<None> {
    static auto action() {
```

```
        return None{};
    }
};
```

- Реалізація самої стейт машини. Приймає набір стейджів, за допомогою фільтру працює механізм знаходження наступного стейту. Метод run дозволяє запустит стейт машину.

```
template <typename... Specs>
    struct StateMachine {
        template <typename State, typename StageT>
        using filter = std::conditional_t<
            std::is_same_v<typename StageT::state, State>, StageT, Pass>;

        template <typename State>
        using find = typename First<None, filter<State, Specs>...>::type;

        template <typename State, typename Event>
        using next_t = typename next_stage<find<State>, Event>::type;

        template <typename State, typename Event>
        static auto run(State state, Event event) {
            using next_state = next_t<std::decay_t<State>,
std::decay_t<Event>>;
            return action_invoker<next_state>::action();
        }
    };
```

3.3 Алгоритм кастомного рандому

Алгоритм кастомного рандому базується як і звичайний алгоритм, на

зерні локального часу машини, проте задля більш складної зворотньої інженерії реалізація використовує логічну операцію XOR з певними побітовими зсувами і арифметичними операціями. Це забезпечує більш складну реверсивну інженерію послідовності стейтів або декодування строкових літералів, оскільки без знання локального часу коли сбиралася программа, або без чисел що використовивулися для зсувів майже неможливо знайти те число яке використовувалося для частин алгоритму обфускації. Нижче наведено фрагмент реалізації алгоритму:

```
template <size_t Seed, size_t Idx>
    struct Xorshiftplus {
        using prev = Xorshiftplus<Seed, Idx - 1>;

        constexpr static size_t update() {
            constexpr size_t x = prev::state0 ^ (prev::state0 << 23);
            constexpr size_t y = prev::state1;
            return x ^ y ^ (x >> 17) ^ (y >> 26);
        }

        constexpr static size_t state0 = prev::state1;
        constexpr static size_t state1 = update();

        constexpr static size_t value = state0 + state1;
    };

    template <size_t Seed>
    struct Xorshiftplus<Seed, 0> {
        constexpr static size_t state0 = Seed;
        constexpr static size_t state1 = Seed << 1;
        constexpr static size_t value = state0 + state1;
```

```
};
```

3.4 Черги

В алгоритмі обфускації використовуються черги, і для послідовності кодових подій, і для кодування строкових літералів. Вони дозволяють за допомогою деяких визначенних елементарних умов створити послідовність будь яких програмних типів, як і символів для кодування строкових літералів, так і для кодових подій або стейтів для стейт машини. Нижче наведено фрагменти з їх реалізації.

- Реалізація типу елемента черги що може приймати як і прості POD типи, так і складні типи, як наприклад стейт з стейт машини.

```
template <typename T, T Val>
  struct TypeVal {
    using value_type = T;
    constexpr static T value = Val;
  };
```

- Реалізація черг, з різними спеціалізаціями шаблонів в залежності від кількості заданих елементів. Має в собі умовний індекс для оперування чергою в рамках більш високорівневого коду.

```
template <typename T, T Val, T... Others>
  struct Sequence {
    using value = TypeVal<T, Val>;
    using next = Sequence<T, Others...>;

    constexpr static std::size_t size = 1 + sizeof...(Others);

    template <std::size_t Idx>
    using index = std::conditional_t<Idx == 0, value, typename
```

```
next::template index<Idx - 1>>;
};

template <typename T, T Val>
struct Sequence<T, Val> {
    using value = TypeVal<T, Val>;

    constexpr static std::size_t size = 1;

    template <std::size_t Idx>
    using index = std::conditional_t<Idx == 0, value, Nothing>;
};
```

- Реалізація черг, де зберігається лише тип логічних даних і їх кількість. Зроблено для оптимізації та для того, щоб не передавати зловмиснику надлишкових даних.

```
template <typename T, typename... Ts>
struct TypeSeq {
    using type = T;
    using next = TypeSeq<Ts...>;

    constexpr static std::size_t size = 1 + sizeof...(Ts);

    template <std::size_t Idx>
    using index = std::conditional_t<Idx == 0, type, typename
next::template index<Idx - 1>>;
};

template <typename T>
struct TypeSeq<T> {
```

```
    using type = T;

    constexpr static std::size_t size = 1;

    template <std::size_t Idx>
    using index = std::conditional_t<Idx == 0, type, Nothing>;
  };
```

- Реалізація інтерфейсу для роботи з чергою. В ній міститься взяття індексу черги для роботи з нею на більш високому кодовому рівні і її розмір.

```
template <typename... T>
  struct SeqPack {
    constexpr static std::size_t size = MinVal<T::size...>::value;

    template <std::size_t Idx, typename U>
    using getter = typename U::template index<Idx>;

    template <std::size_t Idx>
    using index = TypeSeq<getter<Idx, T>...>;
  };
```

3.5 Кодування/декодування строкових літералів

Кодування/декодування строкових літералів – важлива частина обфускації, при зберіганні імен аттрибутів або статичних логічних данних в незакодованому варіанті призводить до легкого зчитування цих данних з бінарного файлу програми, і внаслідок цього, до легкої заміни даних або зчитування захищенної інформації. Тому для алгоритму обфускації, який в першу чергу націленний на захист захищенної інформації використовується кодування таких даних. Нижче приведені частини реалізації алгоритму

кодування/декодування строкових літералів.

- Реалізація строкового літералу для роботи з ним.

```
template <std::size_t size, Encoder encoder, Decoder decoder>
    class String {
    public:
        template <std::size_t... Idx>
        constexpr String(char const* str,
                         std::index_sequence<Idx...>):
            str{ encoder(str[Idx])... } {
            // Do Nothing
        }

        inline char const* decode() const {
            for (char& chr : str) {
                chr = decoder(chr);
            }
            return str;
        }

    private:
        mutable char str[size];
    };
```

- Реалізація створення строкового закодованого літералу на основі рандому.

```
template <typename Table, std::size_t size>
    constexpr auto make_string(char const(&str)[size]) {
        using pair = typename Table::template
index<MAKE_RAND_VAL(Table::size)>;
        constexpr Encoder encoder = pair::template index<0>::value;
```

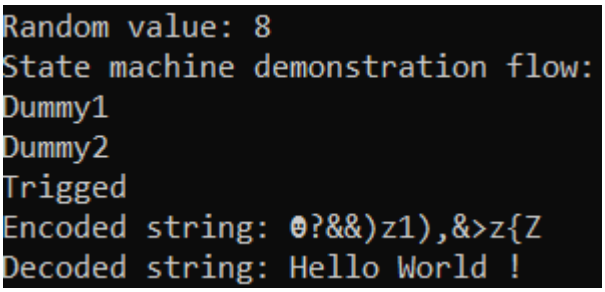
```
    constexpr Decoder decoder = pair::template index<1>::value;

    return make_string<encoder, decoder>(str);
}
```

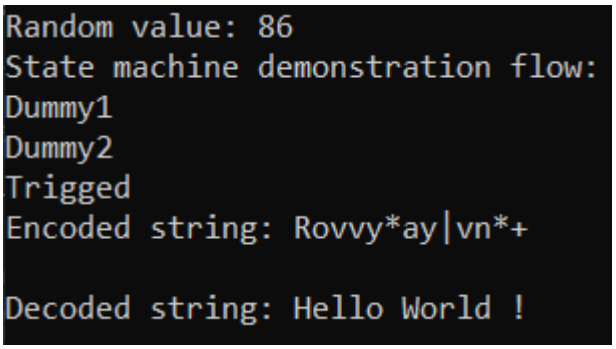
3.6 Таблиця результатів

- Вплив на розмір бінарного файлу та швидкості виконання програми.

	Оригінал	Закодовані дані	Зміненний порядок кодових подій	Закодованні дані + зміненний порядок
Розмір файла з кодом (KB)	9659	15611	13352	17332
Розмір об’єктного файла(KB)	9181	13228	13200	15416
Час виконання (s)	6.611	7.666	6.677	7.711

- Результати роботи підсистем, що зображені нижче на рисунках 7 та 8. В них вказаний вивід випадкого числа, порядок виконання стейт машини та закодований/декодований строковий літерал.





Рисунки 7,8 - Результати роботи підсистем.

ВИСНОВКИ РОЗДІЛУ 3

Як можна побачити на результуючій таблиці, попри оптимізації і те що алгоритм накладається на код в основному під час сборки, він впливає на швидкодію алгоритму. Проте, в середньому збільшуючи швидкість алгоритма на 1/6 – 1/7 відносно часу роботи алгоритму (результати вказані для достатньо великих реальних комерційних алгоритмів), обфускація надійно захищає алгоритм від зворотньої інженерії. Збільшення розміру коду і об’єктного файлу водночас і має недолік для систем що мають обмеженний розмір для зберігання, і перевагу в плані зворотньої інженерії, оскільки значно збільшує обсяг матеріалу для аналізу для зловмисника.