STUDIA INFORMATICA 2009
Volume 30 Number 3A (85)

Krzysztof DOBOSZ, Robert SPYRA
Politechnika Śląska, Instytut Informatyki

ZACIEMNIANIE KODU BAJTOWEGO APLIKACJI DLA MASZYNY WIRTUALNEJ KVM

Streszczenie. Artykuł przedstawia podstawowe problemy inżynierii odwrotnej kodu bajtowego aplikacji dla maszyny wirtualnej Javy, koncentrując się na metodach zaciemniania kodu. Następnie opisane zostają trzy nowe technik zaciemniania. Artykuł objaśnia szczegóły ich implementacji w przykładowej aplikacji.

Słowa kluczowe: kod bajtowy, zaciemnianie kodu, Java, maszyna wirtualna, platforma Java ME

OBFUSCATING APPLICATION BYTECODE FOR VIRTUAL MACHINE KVM

Summary. The article presents essential problems of the reverse engineering of the bytecode of the application for the Java virtual machine. It concentrates on methods of code obfuscation. Next it describes three new obfuscation techniques. The article also explains details of their implementation in the demonstration application.

Keywords: bytecode, code obfuscation, Java, virtual machine, Java ME platform

1. Wstęp

We współczesnym świecie coraz większą rolę odgrywa szeroko rozumiana ochrona wartości intelektualnej wyrażonej w postaci programów, bibliotek lub algorytmów. Próby wykradania cennych informacji oraz szczegółów implementacji poszczególnych rozwiązań zmuszają producentów oprogramowania do stosowania różnych metod obrony swoich produktów. Poza obroną prawną, w postaci patentowania rozwiązań oraz dochodzenia praw autorskich, stosowana jest technika programistyczna zwana zaciemnianiem lub obfuskacją ko-

du, która ma na celu utrudnić uzyskanie informacji na temat sposobu implementacji danego programu za pomocą inżynierii odwrotnej.

Celem niniejszego artykułu jest przegląd obecnie używanych technik zaciemniania aplikacji przeznaczonych dla platformy *Java Micro Edition* [1] wykorzystującej maszynę wirtualną KVM w konfiguracji CLDC [2] i profilu MIDP [3] oraz omówienie opracowania i implementacji własnych metod obfuskacji.

2. Inżynieria odwrotna

Inżynieria odwrotna polega na analizie oprogramowania lub jego pojedynczych elementów w celu uzyskania informacji na temat szczegółów jego implementacji, bez dostępu do tekstu źródłowego. Najczęstszymi przyczynami stosowania inżynierii odwrotnej jest utrata dokumentacji oprogramowania bądź jej brak. Równie częstym i bardzo niebezpiecznym zjawiskiem jest próba nielegalnego uzyskania tekstu źródłowego programu lub przynajmniej jego części. To zjawisko jest potęgowane współcześnie przez szybki rozwój języków programowania, które są kompilowane do języków pośrednich, z których w stosunkowo łatwy sposób za pomocą narzędzi do dekompilacji można uzyskać tekst źródłowy lub jego zbliżoną postać. W przypadku języka Java inżynieria odwrotna odnosi się do inżynierii języka pośredniego, zapisanego w plikach z kodem bajtowym (ang. *Java bytecode*), a uruchamianego pod kontrolą maszyny wirtualnej Javy [4, 5].

Proces dekompilacji ma na celu utworzenie tekstu aplikacji jak najbardziej zbliżonego do źródłowego na podstawie kodu bajtowego. Współczesne dekompilatory pozwalają odtworzyć tekst źródłowy niemal identyczny w stosunku do pierwotnego. Jest to możliwe dzięki architekturze plików zawierających kod bajtowy Javy, z których w stosunkowo łatwy sposób można uzyskać całą strukturę klasy wraz ze wszystkimi identyfikatorami użytymi w tekście źródłowym [6]. Lista instrukcji, które są zapisywane w kodzie bajtowym, jest też bardzo łatwa do bezpośredniego przełożenia na instrukcje języka Java, stąd zazwyczaj kod wygenerowany przez dekompilatory może być poddany ponownej kompilacji. Obecnie na rynku jest dostępna duża liczba dekompilatorów, a większość z nich jest darmowa [7, 8].

3. Zaciemnianie kodu bajtowego

Proces zaciemniania kodu bajtowego ma służyć ochronie wartości intelektualnych zawartych w treści programu. Głównym zadaniem aplikacji narzędziowej do zaciemniania kodu jest przeprowadzenie takich transformacji na tekście źródłowym lub na plikach z kodem baj-

towym, aby po zastosowaniu procesu dekompilacji produktu końcowego uzyskany wynik był jak najbardziej nieczytelny oraz aby nie było możliwości jego ponownej kompilacji.

Na rynku obecna jest duża liczba programów, które pozwalają zabezpieczyć oprogramowanie przez zaciemnienie kodu [9, 10, 11, 12, 13]. Pogramy te oferują różną funkcjonalność od podstawowych technik zaciemniania aż do tych najbardziej wyrafinowanych. Techniki zaciemniania opierają się głównie na zmianach przeprowadzanych w plikach z kodem bajtowym. Operacje takie mają przewagę nad zmianami przeprowadzanymi jeszcze na etapie tekstu źródłowego, ponieważ weryfikacja plików z kodem bajtowym podczas ich uruchamiania, nie jest tak restrykcyjna jak tekstu źródłowego podczas kompilacji. Kolejną zaletą modyfikacji w plikach z kodem bajtowym jest to, iż nie jest wymagany tekst źródłowy programu. Proces zaciemniania kodu bajtowego nie powinien wpływać na sposób wykonywania modyfikowanego programu.

3.1. Technika leksykalna

Jest to podstawowa metoda, stosowana niemal przez wszystkie aplikacje do zaciemniania. Polega ona na zmianie pierwotnych identyfikatorów klas, metod, pól oraz pakietów, które podczas procesu kompilacji zostają umieszczone w wynikowym kodzie bajtowym. Identyfikatory najczęściej są skracane do pojedynczych znaków. Rozszerzeniem tego sposobu jest zastosowanie przeciążania identyfikatorów metod. Przeciążanie metod na poziomie kodu bajtowego daje większe możliwości niż na poziomie tekstu źródłowego aplikacji, gdyż nie ma potrzeby przestrzegania tego samego typu zwracanych przez metody wartości. Dodatkowo, można przeciążać identyfikatory atrybutów klas, które różnią się typem. Dzięki temu większość atrybutów i metod zdefiniowanych w klasach ma taką samą nazwę.

Dobre nazewnictwo pomaga w pielęgnacji tekstu i jego przejrzystości, dzięki skracaniu nazw usuwany jest ten bardzo ważny element, który znacznie utrudnia poprawną interpretację tekstu źródłowego dla użytkowników, którzy próbują nielegalnie go wykorzystać.

3.2. Modyfikacja przepływu sterowania

Metoda leksykalna nie wpływa w żaden sposób na przepływ sterowania i dla bardziej doświadczonych użytkowników lub programistów jest jedynie wydłużeniem czasu w poznaniu i odpowiedniej interpretacji zaciemnionego kodu. Dla jeszcze większego utrudnienia stosowane jest zaciemnienie przepływu sterowania. W tym celu do ciała kodu wstawiane są dodatkowe operacje. Należy jednak pamiętać, iż stosowanie tej metody może wpływać na spowolnienie wykonywanego kodu, dlatego zalecanie jest testowanie aplikacji po zastosowaniu tego sposobu zaciemniania.

Do technik związanych ze zmianą przepływu sterowania należy zaliczyć:

- Wstawianie warunków – sposób ten polega na wstawianiu dodatkowych wyrażeń warunkowych do treści metod. Można wyróżnić kilka wariantów tej techniki. Przykładowo, przed ciąg instrukcji możemy wstawić wyrażenie warunkowe, które pozwala ten ciąg pominąć. Jednakże wyrażenie to zawsze będzie logicznie fałszywe, więc skok ten nigdy się nie wykona. Fundamentalne znaczenie ma w tej metodzie warunek, który powinien zostać tak zaimplementowany, aby poznanie jego wartości było jak najtrudniejsze. Przy oczywistych warunkach metoda ta nie ma większego sensu.
- Zrównoleglenie kodu – polega na uruchamianiu dodatkowego wątku z wyrażeniami niemającymi wpływu na poprawne wykonywanie się programu. Modyfikacją tej techniki jest podział pierwotnej sekwencji instrukcji na kilka wątków – w tym przypadku należy pamiętać o odpowiedniej ich synchronizacji.
- Łączenie metod – technika ta polega na połączeniu zawartości kilku metod w jedną. Dzięki temu odwołania do metod, które zostały scalone, zostają zastąpione odwołaniem do nowej wspólnej metody.
- Klonowanie metod – sposób ten polega na tworzeniu wielu wersji jednej i tej samej metody. Dzięki temu zmniejszona jest liczba odwołań do metody, która pozwalała lepiej ją interpretować. Każda metoda działa identycznie, ale jest zaimplementowana w trochę inny sposób, przykładowo: zmieniona jest kolejność instrukcji, w miejscach gdzie jest to możliwe lub zastosowane są techniki zaciemniania struktur danych.
- Zastępowanie metod z bibliotek standardowych – odwołania do metod ze standardowych bibliotek bardzo ułatwiają interpretację zdekompilowanego kodu. Ponadto, identyfikatory tych metod nie zostają poddane zaciemnianiu metodą leksykalną i są w pełni widoczne po dekompilacji. Dlatego mniej skomplikowane metody można zastąpić własną implementacją, która ma taką samą funkcjonalność.

3.3. Przekształcanie struktur danych

Analiza kodu bajtowego aplikacji może być utrudniona, gdy zaciemnieniu ulegnie zmiana struktur danych w niej wykorzystywanych. Można wskazać następujące techniki takiego zaciemniania:

- zaciemnianie tablic – metoda ta zawiera kilka sposobów zmiany struktury tablicy:
  - podział tablicy na większą liczbę wymiarów,
  - scalenie dwóch tabel w jedną,
  - zmiana wymiaru tablicy,
  - zmiana porządku elementów kolekcji,
- reorganizacja klas – polega na dzieleniu klasy na klasę bazową i klasy ją rozszerzające. Często z tym sposobem skojarzone jest zastosowanie dodawania metod, które nie mają

wpływu na przebieg wykonania kodu. Takie działanie powoduje zwiększenie pamięci oraz nieznacząco wpływa na szybkość wykonywanego kodu. Stopień zaciemnienia tekstu źródłowego wzrasta wraz z dodawaniem kolejnych klas. Jednak kolejne stopnie rozszerzenia wpływają zwykle znacząco na wielkość klas,

- zaciemnianie atrybutów klas oraz zmiennych lokalnych – można wskazać kilka odmian tej techniki:
  - przypisywanie wartości – polega na przypisaniu wartości bądź inicjalizacji zmiennej za pośrednictwem specjalnie stworzonej do tego metody. Zazwyczaj metoda ta posiada parametry, które pomagają jeszcze bardziej zniekształcić kod oraz przypisać wiele różnych wartości dla różnych zmiennych tego samego typu. Dodawanie specjalnych metod wiąże się jednak z powiększeniem kodu oraz z możliwością spowolnienia w zależności od złożoności dodanych metod;
  - scalanie zmiennych – odbywa się przez umieszczenie dwóch lub kilku wartości w jednej zmiennej typu „szerszego" niż wymagany, np. dwie zmienne typu *int* zamieniamy na jedną zmienną typu *long;*
  - podział zmiennej – realizowany jest przez podział zmiennej na kilka innych, najczęściej o różnych typach. Dla takiego podziału potrzebne są trzy składniki:
    - metoda *map(v),* która mapuje wartość zmiennej *v* na poszczególne wartości zmiennych *v1…vk*,
    - metoda *unmap(v1,v2, ..vk)*, która mapuje poszczególne wartości zmiennych *v1…vk* na wartość zmiennej *v*,
    - nowe operacje na zmiennych *v1…vk*, które będą równoważne z tymi, które pierwotnie były wykonywane na zmiennej *v*,
  - zwiększanie zasięgu – polega na rozszerzeniu zasięgu zmiennej lokalnej, zazwyczaj odbywa się to na zasadzie zamiany zmiennej lokalnej na atrybut klasy. Pozwala to na wykorzystanie nowego atrybutu klasy dla wielu zmiennych lokalnych;
  - dekompozycja zmiennych – stosuje skomplikowane wyliczenia wartości, która ma zostać przypisana zmiennej.

3.4. Usuwanie dodatkowych informacji z pliku

Niektóre elementy znajdujące się w pliku z kodem bajtowym nie są konieczne dla prawidłowego działania aplikacji. Większość narzędzi do zaciemniania kodu bajtowego usuwa te informacje. Informacjami tymi są:

- tablice zmiennych lokalnych – po ich usunięciu dekompilator nie ma dostępu do nazw zmiennych lokalnych,

- tablice z liniami kodu – wykorzystywane do skojarzenia numeru linii kodu bajtowego z numerem linii w pliku źródłowym,
- informacje na temat nazwy pliku źródłowego,
- informacje o klasie, polu lub metodzie, które są w trakcie wycofywania (ang. *deprecated*).

3.5. Szyfrowanie łańcuchów znakowych

W większości programów w tekście źródłowym można znaleźć zmienne przechowujące stałe łańcuchy znakowe. Po dekompilacji łańcuchy te są bardzo przydatne w poprawnej interpretacji kodu, szczególnie w przypadkach gdy zostały wykorzystane do:

- nazwania różnych elementów GUI,
- objaśnienia błędów,
- dodatkowych informacji podczas obsługi wyjątków.

Aby utrudnić odczytanie tych napisów, narzędzia zaciemniające szyfrują wszystkie łańcuchy znakowe znajdujące się w klasie. Deszyfrowanie następuje już podczas wykonywania się kodu. Specjalna metoda umieszczona w kodzie pozwala na odkodowanie wszystkich łańcuchów znakowych. Metoda ta może wpływać na szybkość wykonywania się aplikacji – im więcej zaszyfrowanych wartości oraz im wyższy stopień złożoności metody deszyfrującej, tym większy wpływ na czas wykonywania kodu.

4. Nowe techniki zaciemniania kodu

W trakcie badań opracowano trzy nowe techniki zaciemniania. Pomimo że częściowo bazują one na metodach powszechnie znanych i wykorzystywanych w innych narzędziach, to wprowadzają do nich nowe elementy skutecznie utrudniające analizę zdekompilowanego kodu bajtowego.

4.1. Zamiana identyfikatorów

Działanie wszystkich znanych technik sprawia, że po procesie dekompilacji zmiany przeprowadzone przez zaciemnienie są w sposób oczywisty widoczne. Osoby zajmujące się uzyskaniem informacji na temat działania programu od razu zaczynają wtedy stosować narzędzia, które pozwolą im uzyskać poprawną interpretację kodu. Podstawowym założeniem nowej techniki było spowodowanie, aby kod po zaciemnieniu nie sprawiał wrażenia zmodyfikowanego.

Technika ta bazuje na metodzie leksykalnej, która jednak jest zmodyfikowana w istotny sposób. Niemal każde narzędzie stosuje metodę leksykalną skracając identyfikatory nazw, ewentualnie podstawiając za nie słowa kluczowe języka Java. W proponowanej technice dokonujemy zamiany identyfikatorów z innymi, które znajdują się w plikach aplikacji. Każdej klasie jest przypisywany identyfikator innej klasy wchodzącej w skład aplikacji. Podobnie jest w przypadku metod, ich parametrów, atrybutów klas oraz zmiennych lokalnych.

W celu przetestowania metody wybrano klasę *PogoRooMIDlet* z aplikacji *Demos3D*, dołączonej do emulatora *Sun Wireless Toolkit*[14]. Tekst źródłowy metody *hopRoo*() z tej klasy jest następujący:

```
private void hopRoo() {
        if (animTime == 0) // OK to start!
         {
            hopCount = 0;
            okToHop = true;
        }
        if (okToHop) {
            switch (keyMoveRoo) {
            case keyForward:
            case keyBackward:
                int oldHopCount = hopCount;
                hopCount = (animTime * 10) / animLength;
                if (hopCount >= MaxHops) {
                    okToHop = false;
                    hopCount = MaxHops - 1;
                }
                turnAngle = (dirRoo * 3.14159f) / 180.0f;
                float h = 0f;
                for (int i = (oldHopCount + 1); i <= hopCount; i++)
                    h += hopSteps[i];
                float x = h * (float)Math.cos(turnAngle);
                float y = h * (float)Math.sin(turnAngle);
                if (keyMoveRoo == keyForward) {
                    tRoo.translate(-x, -y, 0.0f);
                } else {
                    tRoo.translate(x, y, 0.0f);
                }
                break;
            }
        }
    }
```

Następnie przeprowadzono operacje zaciemniania i zdekompilowano za pomocą dekompilatora *Jad* nowo powstały kod bajtowy klasy. Uzyskany tekst źródłowy jest następujący:

```
private void moveRoo()
    {
        if(POGOROO_TRANSFORM_ID == 0)
        {
            CAMERA_GROUP_TRANSFORM_ID = 0;
            okToHop = true;
        }
        if(okToHop)
            switch(MaxHops)
            {
            case 1:
            case 2:
                int i = CAMERA_GROUP_TRANSFORM_ID;
                CAMERA_GROUP_TRANSFORM_ID = (POGOROO_TRANSFORM_ID * 10) /
                                            ROO_BOUNCE_ID;
```

```
                if(CAMERA_GROUP_TRANSFORM_ID >= 10)
                {
                    okToHop = false;
                    CAMERA_GROUP_TRANSFORM_ID = 9;
                }
                patternName = (dirCam * 3.14159F) / 180F;
                float f = 0.0F;
                for(int j = i + 1; j <= CAMERA_GROUP_TRANSFORM_ID; j++)
                    f += posTemp[j];

                float f1 = f * (float)Math.cos(patternName);
                float f2 = f * (float)Math.sin(patternName);
                if(MaxHops == 1)
                    tCams.translate(-f1, -f2, 0.0F);
                else
                    tCams.translate(f1, f2, 0.0F);
                break;
            }
    }
```

Niektóre zmiany wykonane przez mechanizm zaciemniania zostały przedstawione w tabeli 1.

Tabela 1

Przykłady zamian identyfikatorów

Element	Pierwotny identyfikator	Nowy identyfikator
metoda	*hopRoo*	*moveRoo*
atrybut	*animTime*	*POGOROO_TRANSFORM_ID*
atrybut	*hopCount*	*CAMERA_GROUP_TRANSFORM_ID*
atrybut	*keyMoveRoo*	*MaxHops*
atrybut	*turnAngle*	*patternName*
atrybut	*animLength*	*ROO_BOUNCE_ID*
zmienna lokalna	*oldHopCount*	*i*
zmienna lokalna	*i*	*j*
zmienna lokalna	*x*	*f*1
zmienna lokalna	*y*	*f*2
zmienna lokalna	*h*	*f*

W tabeli 1 niektóre z „nowych” identyfikatorów nie występują w definicji modyfikowanej metody, gdyż zostały zaczerpnięte z innej części aplikacji. Natomiast część identyfikatorów została zamieniona na stałe wartości, które reprezentują.

4.2. Dodawanie parametrów metod

Technika dodawania parametrów metod również nie jest stosowana we współczesnych rozwiązaniach. Właściwie użyta może być jednak znacznym utrudnieniem podczas inżynierii odwrotnej. O tym, czy do danej metody może zostać dodany parametr i jakiego typu, rozstrzyga funkcja oceniająca. Działa ona wtedy, gdy sprawdza się częstość występowania typów zmiennych lokalnych. Typ, który najczęściej się pojawia, zostaje wybrany dla dodatkowego parametru metody. W przypadku braku zmiennych lokalnych, bądź braku zmiennych lokalnych o typach obsługiwanych przez technikę dodawania parametrów, metoda ta jest

pomijana. Liczba metod, w których zostanie zastosowana ta technika, jest ograniczona. W przykładowej implementacji wzięto pod uwagę tylko metody prywatne.

Typ dodatkowego parametru metody wybierany jest losowo spośród wykorzystywanych w modyfikowanej aplikacji. Następnie w miejscu wywołania metody jest szukana zmienna lokalna o typie zgodnym z typem dodanego parametru. Jeżeli taka zmienna istnieje i jej użycie jest możliwe w miejscu wywołania metody – zostanie ona użyta. W przypadku nieznalezienia zmiennej lokalnej zostają sprawdzone atrybuty bieżącej klasy. Gdy atrybuty o odpowiednim typie nie są zadeklarowane w danej klasie, metoda jest wywołana z losową wartością o typie zgodnym z dodanym parametrem.

4.3. Wstawianie warunków i operacje na zmiennych lokalnych

Podstawową operacją w tej metodzie jest zmiana identyfikatorów zmiennych lokalnych na tę samą nazwę dla wszystkich zmiennych – jest to możliwe na poziomie kodu bajtowego. Zaproponowano, by było to słowo kluczowe języka Java – *this*. Zamianę taką przeprowadza się wewnątrz wszystkich metod. Następnie dla metod, do których została zastosowana technika dodania parametru (jeżeli będzie to możliwe, bo spełnione będą zasady weryfikacji kodu bajtowego), zostaną wstawione odpowiednie wyrażenia warunkowe. Po dekompilacji będą miały postać:

- *this* == *this*

lub

- *this* != *this*

Należy zauważyć, że dla osoby analizującej tekst źródłowy (dotyczy również kompilatora) identyfikator *this* jest identyczny ze słowem kluczowym *this*. Tak więc tylko od narzędzia do zaciemniania implementującego tę metodę zależy, czym jest pierwszy, a czym drugi argument wyrażenia warunkowego. W zrealizowanej implementacji [16] uzależniono to od typu dodanego parametru metody. W przypadku dodania parametru typu obiektowego pierwszy argument jest jego referencją, drugi – słowem kluczowym. W pozostałych przypadkach oba argumenty są referencjami do tego samego, dodatkowego parametru metody.

W zależności od typu wybranej zmiennej lokalnej (może to być referencja do bieżącego obiektu) wartości tych porównań będą różne. Tak przygotowane wyrażenia warunkowe mogą być używane dla instrukcji:

- warunkowej *if,*
- pętli: *for*, *while* oraz *do while*.

Metoda zaciemniania zaimplementowana w programie przez dodawanie warunków oraz parametrów zostanie zaprezentowana w przykładzie, w którym zaciemnieniu została poddana

klasa *Gameplay* z aplikacji *Tetris* (przykład demonstracyjny emulatora [14]).Tekst źródłowy testowanej metody *sprawdzLinie()* jest następujący:

```
private boolean  sprawdzLinie(){
     boolean linia = true;
     for(int i = WYS - 1; i >=0;i--){
         linia = true;
         for(int j = SZER - 1;j >=0;j--){
            if(kostka.getCell(j,i) == 0){
                linia = false;
                break;
            }
         }
         if(linia){
             if(!dzwiek)
                new Thread(sound).start();
             kasujLinie(i);
             return true;
         }
     }
     return false;
 }
```

Tekst źródłowy metody *sprawdzLinie()* po przeprowadzeniu zaciemniania uzyskany za pomocą dekompilatora *Jad* jest następujący:

```
private boolean t(Object this)
    {
        this = 1;
        if(this == this)
            v();
        do
        {
            int this = 34;
label0:
            do
            {
label1:
                {
                    if(this < 0)
                        break label1;
                    this = 1;
                    int this = 18;
                    if(this == this)
                        break label0;
                    do
                    {
                        if(this < 0)
                            break;
                        if(e.getCell(this, this) == 0)
                        {
                            this = 0;
                            break;
                        }
                        this--;
                    } while(true);
                    if(this != 0)
                    {
                        if(!a3)
                            (new Thread(j2)).start();
                        u(this, this);
                        return true;
                    }
                    this--;
                }
            } while(true);
```

```
            } while(true);
            return false;
        }
```

Należy zwrócić uwagę, że dodanie warunku (w naszej implementacji logicznie zawsze fałszywego)

```
if(this == this)
    v();
```

nie spowoduje wywołania metody v(),
warunek zaś

```
if(this == this)
    break label0;
```

nie spowoduje przerwania pętli. Podczas rekompilacji tekstu źródłowego warunki te zostaną potraktowane przez kompilator jako prawdziwe, co spowoduje zmianę w przepływie sterowania w programie w stosunku do wersji niezaciemnionej. Warto również zauważyć, że identyfikator *this* został również przypisany wszystkim zmiennym lokalnym (pierwotne identyfikatory: *i, j, linia*), przez co są one nie do odróżnienia i utrudnią rekompilację tekstu źródłowego. Pozostałe identyfikatory w zmodyfikowanej wersji metody oznaczają elementy nielokalne (*e, j2, a3*) i zostały wygenerowane standardową technika leksykalną.

5. Implementacja

W celu sprawdzenia skuteczności zaproponowanych technik zaimplementowano aplikację przeznaczoną do zaciemniania kodu bajtowego [15]. Niniejszy rozdział zawiera przegląd ważniejszych klas zdefiniowanych w tym obiektowo zorientowanym narzędziu.

5.1. Klasa *ZarzadzIdentyfikatorami*

Klasa *ZarzadzIdentyfikatorami* odpowiada za generowanie nowych identyfikatorów dla klas i ich atrybutów oraz metod i ich parametrów. Nowe identyfikatory powstają przez mieszanie oryginalnych lub przez przypisywanie skróconych nazw przy wykorzystaniu klasy *GeneratorSkrotow*. Klasa również zajmuje się sprawdzaniem poprawności wygenerowanych identyfikatorów. Sprawdzaniu podlegają również metody publiczne z klas, które rozszerzają oraz implementują klasy lub interfejsy, które nie znajdują się bezpośrednio wśród plików z kodem bajtowym bieżącej aplikacji.

Klasa posiada metody do zwracania nowych identyfikatorów na podstawie oryginalnych.

5.2. Klasa *Archiwum*

Klasa *Archiwum* zarządza całym przebiegiem wykonania procesu zaciemniania. Sterowanie jego przebiegiem znajduje się w metodzie *startZaciemniacz()*. Dla każdego zestawu wszystkie klasy zapamiętywane są w atrybucie *klasy*, który jest listą elementów typu *Klasa*. Po etapie zaciemniania generowana jest dokładnie taka sama liczba klas, jaka została wczytana.

5.3. Klasa *Klasa*

Obiekt typu *Klasa* ma podobną strukturę do pliku z kodem bajtowym. W klasie zawarta jest operacja wczytywania metod oraz atrybutów na podstawie obiektu typu *JavaClass*, dostarczanego przez klasę *Repository*. Ważną funkcjonalnością znajdującą się w tej klasie jest zmiana wszystkich nazw klas oraz sygnatur z typami klas w tablicy stałych. Ponadto, klasa zarządza przebiegiem zmian nazw pól, metod oraz opcjonalnie wstawianiem warunków i parametrów.

5.4. Klasa *Metody*

Klasa zawiera wszystkie metody danej klasy. Umożliwia operację zmiany nazw dla wszystkich metod: tych zdefiniowanych w klasie i wywoływanych na rzecz obiektów klas, które zostały umieszczone w tym samym zestawie. Wszystkie modyfikacje są przeprowadzane zarówno na poziomie tablicy stałych, jak i samych metod. Klasa ta ponadto umożliwia wybór metod prywatnych, które mogą być wywoływane w martwej części kodu – technika wstawiania warunków. Klasa odpowiada również za sterowanie przebiegiem dodawania warunków i parametrów oraz modyfikację odwołań do metod, dla których został dodany parametr.

5.5. Klasa *Metoda*

Klasa reprezentuje metodę znaną ze struktury pliku. Posiada ona m.in. listę instrukcji reprezentowaną przez klasę *ListaInstrukcji* oraz tablicę zmiennych lokalnych. W metodzie dostępne są operacje zamiany identyfikatorów zmiennych lokalnych lub zamiany identyfikatorów na jedną wspólną nazwę. Ponadto, klasa udostępnia metody do wyboru typu, którego będzie parametr do dodania. Steruje bezpośrednio procesem wstawiania warunków, dodawania argumentów oraz modyfikacją odwołań dla zmienionych metod.

6. Uwagi końcowe

Podczas wykonanych prac dokładnie przeanalizowano problem dekompilacji oraz zaciemniania kodu bajtowego. Poznano strukturę budowy plików z kodem bajtowym oraz przetestowano dostępne programy do ich zaciemniania. Na podstawie uzyskanych informacji została opracowana aplikacja, która posiada funkcjonalność niespotykaną w innych narzędziach przeznaczonych do zaciemniania. W aplikacji są dostępne trzy nowe techniki:

- zamiana identyfikatorów klas, metod, pól oraz parametrów spośród znajdujących się w zestawie klas aplikacji,
- dodawanie parametrów dla wybranych metod,
- zmiana nazw parametrów na jedną wspólną.

Możliwa jest dalsza rozbudowa powstałej aplikacji narzędziowej. Może ona polegać zarówno na udoskonaleniu zaimplementowanych technik zaciemniania, jak również na dodaniu nowych. Udoskonaleniu szczególnie mogłaby zostać poddana metoda zmiany nazw spośród już istniejących. Aktualnie podczas operacji zamiany identyfikatorów nie są odróżniane etykiety zmiennych finalnych i niefinalnych, a różnicuje je konwencja nazewnicza stosowana przez programistów. Ciekawa byłaby również możliwość odtworzenia nazw pierwotnych na podstawie pliku z zapisanym odwzorowaniem identyfikatorów. Innym udoskonaleniem mogłoby być opracowanie słownika identyfikatorów z odpowiednikami o przeciwnym znaczeniu, na przykład *otwórz – zamknij*. Po napotkaniu identyfikatora lub przynajmniej części jego nazwy, która znajdowałaby się w słowniku, mógłby on zostać zastąpiony odpowiednią wartością.

BIBLIOGRAFIA

1. The Java ME Platform – dokumentacja techniczna. Witryna: http://java.sun.com/javame/, sierpień 2008.
2. Connected Limited Device Configuration – dokumentacja techniczna. Witryna: http:// java. sun.com/products/cldc/, sierpień 2008.
3. Mobile Information Device Profile – dokumentacja techniczna. Witryna: http://java.sun. com/products/midp/, sierpień 2008.
4. Byte Code Engineering Library. Witryna: http://jakarta.apache.org/bcel/manual.html, wrzesień 2008.
5. Kalinovsky A.: Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering. Ebook, 2004.

6. Byte code viewer. Witryna: http://www.ej-technologies.com/products/jclasslib/overview.html wrzesień 2008.
7. Jad – the fast JAva Decompiler. Witryna: http://www.kpdus.com/jad.html, wrzesień 2008.
8. DJ Java Decompiler. Witryna: http://www.neshkov.com/dj.html, wrzesień 2008.
9. Jshrink: Java Shrinker and Obfuscator. Witryna: http://www.e-t.com/jshrink.html, wrzesień 2008.
10. ProGuard. Witryna: http://proguard.sourceforge.net, wrzesień 2008.
11. RetroGuard for Java Bytecode Obfuscator – Retrologic. Witryna: http://www.retrologic.com wrzesień 2008.
12. Java Obfuscator – Zelix KlassMaster. Witryna: http://www.zelix.com/klassmaster/index.html, wrzesień 2008.
13. DashO – Java Obfuscator, Java Code Protector, Pruner and Watermarker. Witryna: http://www.preemptive.com/dasho-java-obfuscator.html, wrzesień 2008.
14. Sun Wireless Toolkit for CLDC. Witryna. http://java.sun.com/products/sjwtoolkit/, wrzesień 2008.
15. Spyra R.: Analiza efektywności metod zaciemniania i dekompilacji kodu bajtowego aplikacji dla maszyny wirtualnej KVM. Praca dyplomowa magisterska. Politechnika Śląska, Gliwice, grudzień 2008.

Recenzent: Dr inż. Maciej Bargielski

Wpłynęło do Redakcji 24 lutego 2009 r.

Abstract

The article presents essential problems of the reverse engineering of the bytecode of the application for the Java virtual machine. It concentrates on methods of code obfuscation. A structure of the bytecode files was analysed and available programs were tested for obfuscating them. Using obtained information a tool application was implemented, that realized the most popular obfuscation methods.

Next the article describes three new obfuscation techniques. First of them exchanges identifiers: classes, class attributes, methods and its parameters inside bytecode. Second one adds extra method parameter. Third one bases on changing identifiers of variables on one shared. The authors proposal is usage of "this" keyword as common identifier. The article is also explains details of implementation of proposed methods in the demonstration applica-

tion. Several designed classes (elements of object oriented programming) are shortly described.

At the end of article authors presents possible ways of developing of proposed techniques. The method of the exchange of identifiers could store mapping from new names to older names. A possibility of reconstructing original names from a file with written mapping of identifiers would be also interesting. Another improvement could be using the dictionary of words and its counterpart with opposite meaning for example could *open – close*. Identifiers built from dictionary words could be replaced with opposites.

Adresy

Krzysztof DOBOSZ: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, krzysztof.dobosz@polsl.pl
Robert SPYRA: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, rob.spyra@gmail.com