STUDIA INFORMATICA 2009
Volume 30 Number 3A (85)

Krzysztof DOBOSZ, Tomasz WESOŁOWSKI
Politechnika Śląska, Instytut Informatyki

ZASTOSOWANIE NARZĘDZI DO ZACIEMNIANIA KODU BAJTOWEGO W PROCESIE OPTYMALIZACJI PAMIĘCIOWEJ

Streszczenie. Artykuł przedstawia podstawowe metody zaciemniania kodu bajtowego przygotowywanego dla maszyny wirtualnej KVM. Szczególna uwaga zostaje zwrócona na metodę leksykograficzną. Następnie przetestowano kilka darmowych narzędzi do zaciemniania. Zauważono, że większość z nich powoduje zmniejszanie rozmiaru zaciemnianych aplikacji.

Słowa kluczowe: zaciemnianie kodu, optymalizacja pamięciowa, midlet

APPLYING TOOLS FOR BYTECODE OBFUSCATION IN THE PROCESS OF THE MEMORY OPTIMIZATION

Summary. The article presents essential methods of obfuscation the bytecode prepared for the KVM virtual machine. The special attention is paid to the lexicographical method. Next a few free tools for obfuscation were tested. Authors noticed that the most of them caused reducing the size of obfuscated applications.

Keywords: code obfuscation, memory optimization, midlet

1. Wstęp

Podczas opracowywania aplikacji dla urządzeń mobilnych wyposażonych w platformę maszyny wirtualnej KVM autorzy częstokroć stają przed koniecznością zmniejszenia rozmiaru kodu bajtowego. Ma to na celu dostosowanie aplikacji do wymogów danego urządzenia. Najbardziej oczywistym sposobem jest tu wykorzystanie odpowiednich narzędzi optymalizacyjnych. Zauważono jednak, że niektóre programy dedykowane zaciemnianiu kodu aplikacji również powodują zmniejszanie jej rozmiaru. Przebadano więc, które ze znanych narzędzi

oraz jakie metody zaciemniania mogą przynieść zysk w postaci zmniejszenia rozmiaru wynikowego kodu bajtowego aplikacji.

Jedną z platform uruchomieniowych instalowanych na urządzeniach mobilnych przez ich producentów jest Java ME [1] oparta na maszynie wirtualnej KVM [2]. Obecnie najpopularniejszymi urządzeniami mobilnymi są telefony komórkowe w konfiguracji CLDC [3] z profilem MIDP [4]. Aplikacje opracowywane dla takich urządzeń nazywane są midletami. Wśród najważniejszych problemów stojących obecnie przed autorami midletów znajduje się m.in. optymalizacja pamięciowa [5, 6, 7] oraz zapewnienie bezpieczeństwa przez odpowiednią ochronę kodu maszynowego przygotowanego dla maszyny wirtualnej [8, 9]. Optymalizacja pamięciowa midletów ma bardzo duże znaczenie, gdyż wiele telefonów komórkowych dysponuje wciąż pamięcią operacyjną o stosunkowo niewielkim rozmiarze, co przenosi się na znaczne trudności z uruchamianiem dużych midletów. Natomiast w celu utrudnienia analizy kodu maszynowego, zwanego dalej kodem bajtowym, przeprowadza się proces jego zaciemniania. Do zaciemnienia kodu można użyć jednego z kilku najbardziej popularnych narzędzi, rozpowszechnianych w sieci Internet [10]. Należy jednak być świadomym faktu, że dla każdego z nich istnieją już bardzo sprawnie działające dekompilatory dość dobrze radzące sobie z różnymi zaciemnieniami i zawiłościami skomplikowanego kodu bajtowego.

Nic nie stoi też na przeszkodzie, aby samemu opracować własną, oryginalną metodę zaciemniania. Wiązać się to jednak będzie ze sporym nakładem czasowym przeznaczonym na przygotowanie algorytmu, implementację i przeprowadzenie testów. Warto więc zastanowić się najpierw, czy dodanie oryginalnego zabezpieczenia jest aż tak ważnym elementem całego projektu. Ostatecznie należy wziąć pod uwagę fakt, że mocno zdeterminowanego włamywacza i tak nie powstrzyma to przed analizą naszego programu.

Żeby nie zniechęcać potencjalnych twórców midletów do zaciemniania, warto zwrócić uwagę na fakt, że wiele najbardziej popularnych narzędzi do zaciemniania kodu niejako przy okazji dokonuje jego częściowej optymalizacji pamięciowej. Zmniejszenie rozmiaru kodu bajtowego aplikacji ma często istotny wpływ na możliwość jej uruchamiania na urządzeniach słabszych sprzętowo. Dlatego też na problem ten zwrócili uwagę autorzy niniejszego artykułu, stawiając pytanie, czy narzędzia dedykowane zaciemnianiu kodu, wykorzystując swoje standardowe konfiguracje, są w stanie efektywnie wpłynąć na zmniejszenie rozmiaru aplikacji. I choć porównania narzędzi do zaciemniania były już wykonywane [10], to nigdy dotąd pod kątem optymalizacji pamięciowej.

2. Metody zaciemniania kodu

Kody bajtowe programów przygotowanych dla maszyny wirtualnej Javy (JVM lub KVM) w stosunkowo łatwy sposób poddają się dekompilacji. Podczas analizy kodu bajtowego zapisanego w postaci asemblerowej łatwo można dostrzec jego prostotę. Składa się na nią m.in.: przechowywanie wszystkich identyfikatorów zdefiniowanych przez programistę w tekście źródłowym, brak rozkazów skoków poza obręb metod, brak skoków z adresowaniem bezwzględnym, brak możliwości samomodyfikacji, wykorzystanie stosu do przekazywania parametrów itp. [2]. Aby więc skomplikować zadanie osobom, które chcą poznać budowę klasy, stosuje się narzędzia do zaciemniania kodu bajtowego. Rozróżniamy trzy podstawowe metody modyfikowania kodu bajtowego programu utrudniające jego analizę: zaciemnianie leksykalne, zaciemnianie przepływu sterowania oraz zaciemnianie struktury programu i danych [8, 11]. W kolejnych podrozdziałach zostaną one skrótowo przedstawione oraz ocenione pod kątem możliwości wykorzystania w procesie optymalizacji pamięciowej.

2.1. Zaciemnianie przepływu sterowania

Analiza przepływu sterowania w programie może doprowadzić do poznania działania zaimplementowanego algorytmu. Aby utrudnić to zadanie, narzędzia do zaciemniania kodu bajtowego stosują różne metody. Najpopularniejsze to:

- modyfikacja warunków logicznych,
- dodawanie fałszywych rozgałęzień,
- umieszczanie w kodzie „pułapek” niepodlegających prostej dekompilacji.

Dwa pierwsze sposoby są stosunkowo proste i można je samemu wykonywać na poziomie tekstu źródłowego zapisanego w języku Java. Trzeci ze sposobów daje najlepszy efekt w utrudnianiu analizy programu, wymaga jednak gruntownej znajomości asemblera maszyny wirtualnej oraz zasady działania kompilatora języka Java.

Bez poddawania wymienionych metod analizie, można wysnuć wnioski, że wynikiem ich działania będzie jedynie rozbudowa kodu bajtowego, co czyni je nieinteresującymi z punktu widzenia optymalizacji pamięciowej.

2.2. Zaciemnianie budowy programu i struktury danych

Odrębnym zagadnieniem w dziedzinie zaciemniania zapisu programu jest modyfikacja kompozycji samego programu oraz wykorzystywanych przez niego struktur danych. W przypadku modyfikowania budowy programu można brać pod uwagę m.in. modyfikację hierarchii dziedziczenia klas i klonowanie metod. Prawdopodobnie ze względu na duży stopień

komplikacji takich operacji badane narzędzia do zaciemniania nie wykonywały takich modyfikacji. W zamian za to zdarzało się usuwanie kodu bajtowego nieużywanych klas i metod!

W przypadku zaciemniania struktur danych należy wziąć pod uwagę przede wszystkim dekompozycję zmiennych oraz restrukturyzację tablic. Nie dostrzeżono jednakże, aby którekolwiek z badanych narzędzi techniki te stosowało. Odnotowano tylko, że usuwane są informacje o nieużywanych atrybutach klas zdefiniowanych w kodzie bajtowym.

Z uwagi na powyższe spostrzeżenia należy stwierdzić, że automatycznie stosowane metody zaciemniania budowy programu i struktur danych najczęściej prowadzą do optymalizacji pamięciowej. Ma to jednak miejsce tylko wówczas, gdy programista nieużywanego kodu i danych sam nie usunął na etapie konserwacji aplikacji.

2.3. Zaciemnianie leksykalne

W procesie kompilacji tekstu źródłowego aplikacji zapisanej w języku Java do postaci kodu bajtowego, zapamiętaniu ulegają wszystkie identyfikatory zdefiniowane przez programistę. Jest to bardzo użyteczne dla mechanizmu dynamicznego ładowania kodu bajtowego w czasie działania aplikacji, jednakże jednocześnie jest to znaczne ułatwienie dla osób starających się przeanalizować działanie programu. Niech przykładowa klasa wygląda tak jak poniżej.

```
class Result
{
   private String name;
   private int score;

   public Result( String name, int score )
   {
      this.name = name;
      this.score = score;
      calculateExtraPoints();
   }

   private void calculateExtraPoints()
       {
          score *= 0.1;
       }

   public String getName()
       {
       return name;
       }

   public int getScore()
       {
       return score;
       }
}
```

Metoda zaciemniania leksykalnego polega na modyfikacji identyfikatorów pojawiających się w kodzie bajtowym na mniej czytelne. Nowe identyfikatory mogą być generowane losowo. Lepszym jednak sposobem jest wykorzystanie jako identyfikatorów słów kluczowych

języka Java – oczywiście tylko wtedy, gdy programujemy z użyciem tego języka [12]. Kompilator odrzuci takie identyfikatory, więc automatyczna ponowna rekompilacja będzie niemożliwa. Przykładowa klasa po zastosowaniu zaciemniania leksykalnego i dezasemblacji może przyjąć postać taką jak poniżej.

```
class try
{
    private String do;
    private int if;

    public try(String s, int i)
    {
        do = s;
        if = i;
        for();
    }

    private void for()
    {
        if *= 0.1;
    }

    public String do()
    {
        return do;
    }

    public int if()
    {
        return if;
    }
}
```

Należy tutaj zwrócić uwagę, że z punktu widzenia optymalizacji pamięciowej najlepsza będzie metoda wykorzystująca jak najkrótsze identyfikatory. Mogą one być generowane losowo lub nadawane kolejno w porządku leksykograficznym.

```
class a
{
    private String a;
    private int b;
        public a(String s, int i)
    {
        a = s;
        b = i;
        c();
    }

    private void c()
    {
        b *= 0.1;
    }

    public final String a()
    {
        return a;
    }

    public final int b()
    {
        return b;
    }
}
```

Dla porównania należy podać, że wygenerowany kod bajtowy oryginalnej klasy miał rozmiar 552 bajty, po zaciemnieniu słowami kluczowymi języka – 503 bajty, a po wykorzystaniu jednoznakowych identyfikatorów – 491 bajtów, co daje około 11% zysku względem rozmiaru oryginału.

Wykorzystując metodę leksykalną, należy zwrócić uwagę, czy w zaciemnianej aplikacji wykorzystywane są mechanizmy refleksji, polegające m.in. na tworzeniu obiektów z wykorzystaniem identyfikatorów klas podanych w formie obiektów klasy String. W takim przypadku należy zachować ostrożność, gdyż automatyczna zmiana wszystkich identyfikatorów w programie może doprowadzić do sytuacji, w której po zaciemnieniu aplikacja przestanie się uruchamiać. Problem ten nie dotyczy platformy Java ME, gdyż wykorzystywana w niej maszyna wirtualna KVM nie realizuje mechanizmu refleksji.

3. Przegląd narzędzi

Do wykonania testów wybrano kilka najpopularniejszych, darmowych narzędzi do zaciemniania kodu:

- Proguard v4.0.1
- bb_mug v1.5
- flmObf v1.0.0
- Javaguard v1.0beta4
- Jmangle v1.0 for jdk1.1
- Marvinobfuscator v1.2b
- VasObfuLite

Niekomercyjne narzędzia do zaciemniania w niemal wszystkich przypadkach nie są już rozwijane – zawieszono prace nad projektami, stąd testowane wersje pochodzą sprzed kilku lat. Może to być powodem niezbyt dobrego działania tych programów. Dodatkowym mankamentem jest konieczność przygotowywania skryptów koniecznych do przeprowadzenia procesu zaciemniania. Brak jest jakiegokolwiek standardu ujednolicającego proces zaciemniania. Tylko jeden z testowanych programów posiadał interfejs graficzny, pozwalający na wizualne ustawienie wszystkich opcji – rozwiązanie takie, jako bardziej intuicyjne, z pewnością przyda się użytkownikom początkującym lub takim, którym zaciemnianie kodu potrzebne jest sporadycznie. Tryb wsadowy i sterowanie zaciemnianiem za pomocą skryptów z kolei jest rozwiązaniem idealnym dla programistów często poddających swój kod zaciemnianiu, pod warunkiem iż skrypty pozwalają całkowicie zautomatyzować ten proces. Niestety, stworzenie skryptów dla niektórych z testowanych programów było zajęciem żmudnym, a co najgorsze skrypty te nie umożliwiają wykonania operacji zaciemniania automatycznie, np. po

każdym przebudowaniu projektu. Nie pojawiły się żadne zastrzeżenia co do wydajności, a operacja zaciemniania różnych midletów zawsze przebiegała w akceptowalnym czasie.

Poważnym problemem okazał się fakt, że większość z testowanych narzędzi nie wspiera operacji wykonywanych na midletach. Wskutek tego testowane midlety często nie uruchamiały się po zaciemnieniu kodu. Konieczne było wówczas użycie do weryfikacji klas zewnętrznego narzędzia – w tym przypadku był to Proguard – jednak nawet takie rozwiązanie, choć polepszało sytuację, nie eliminowało całkowicie problemów z uruchomieniem midletów. Najczęstszym problemem była zmiana nazwy klasy głównej, przez co emulator nie mógł odnaleźć punktu startowego midletu – nie pomogły różne zmiany w plikach manifestu i deskryptora. W przypadku programu VasObfuLite, który operuje bezpośrednio na plikach źródłowych, na skutek zaciemnienia kodu cztery z sześciu midletów nie skompilowały się z powodu błędów składniowych. Poza tym operowanie plikami źródłowymi czy plikami zawierającymi kod bajtowy zamiast archiwami JAR jest bardziej praco- i czasochłonne.

4. Wyniki badań

Do roli materiału testowego wybrano kilka powszechnie dostępnych midletów demonstracyjnych dostarczanych wraz ze środowiskiem Sun Wireless Toolkit [13]. Należą do nich:

a) AdvancedMultimediaSupplements.jar
b) Demo3D.jar
c) HelloWorld.jar
d) i18nDemo.jar
e) NetworkDemo.jar
f) Tumbleweed.jar

W celu przygotowania wymienionych programów do testów, sprawdzono je najpierw we wspomnianym wcześniej środowisku uruchomieniowym. Wszystkie programy działały bez zarzutów. Aby prawidłowo ustalić procentową zmianę rozmiaru kodu bajtowego, usunięto z plików JAR zasoby (łącznie z plikiem manifestu). Różnica pomiędzy rozmiarami plików JAR z zasobami i bez nich jest zbliżona – pewne różnice pojawiają się chociażby na skutek kompresji. Procentowa zmiana rozmiaru liczona jest jako stosunek różnicy rozmiaru oryginalnego archiwum i archiwum po zaciemnieniu do rozmiaru oryginalnego archiwum bez zasobów (tylko pliki klas). Tabela 1 przedstawia rozmiary materiału testowego.

Tabela 1

Rozmiar testowanych danych

Aplikacja	oryginalny [B]	bez zasobów [B]
a	771 512	53 001
b	95 107	18 910
c	1 152	734
d	17 348	8 643
e	12 735	12 176
f	13 220	9 254

Tabela 2 przedstawia wyniki optymalizacji z wykorzystaniem wymienionych narzędzi. Rozmiar aplikacji (z zasobami) po zaciemnianiu podany jest w bajtach. Brak danych w tabeli oznacza, że narzędzie nie wygenerowało zaciemnionego kodu bajtowego.

Tabela 2

Rozmiar aplikacji po zaciemnianiu

Aplikacja	Proguard	bb_mug	JavaGuard	MarvinObf	VasObfuLite
a	763 249	brak	766 655	776 140	-
b	90 864	93 276	91 707	94 976	-
c	1 038	1 088	1 217	1 469	1 152
d	16 087	17 024	17 222	18 855	17 162
e	11 441	11 909	12 124	15 000	-
f	11 208	11 887	11 912	13 020	-

W tabeli 3 umieszczono procentowy zysk osiągnięty względem oryginalnego rozmiaru aplikacji. W pozycjach oznaczonych wartością ujemną uzyskano zwiększenie rozmiaru kodu bajtowego, czyli efekt odwrotny do oczekiwanego. W pozycjach bez zamieszczonych wartości nie uzyskano wyniku na skutek błędnego działania narzędzia zaciemniającego kod.

Tabela 3

Procentowe zmniejszenie rozmiaru całej aplikacji

Aplikacja	Proguard	bb_mug	JavaGuard	MarvinObf	VasObfuLite
a	1,07 %	brak	0,63 %	- 0,60 %	-
b	4,46 %	1,93 %	3,57 %	0,14 %	-
c	9,90 %	5,56 %	5,64 %	- 27,52 %	0,00 %
d	7,27 %	1,87 %	0,73 %	- 8,69 %	1,07 %
e	10,16 %	6,49 %	4,80 %	- 17,79 %	-
f	15,22 %	10,08 %	9,89 %	1,51 %	-

W tabeli 4 również przedstawiono procentowy zysk jednak względem oryginalnego rozmiaru aplikacji pozbawionej plików z zasobami.

Tabela 4

Procentowe zmniejszenie kodu aplikacji

Aplikacja	Proguard	bb_mug	JavaGuard	MarvinObf	VasObfuLite
a	15,59 %	brak	9,16 %	- 8,73 %	-
b	22,44 %	9,68 %	17,98 %	0,69 %	-
c	15,53 %	8,72 %	8,86 %	- 43,19 %	0,00
d	14,59 %	3,75 %	1,46 %	- 17,44 %	2,15
e	10,63 %	6,78 %	5,02 %	- 18,60 %	-
f	21,74 %	14,40 %	14,13 %	2,16 %	-

5. Uwagi końcowe

Podsumowując przeprowadzone badania, można stwierdzić, że narzędzia do zaciemniania w większości przypadków zmniejszają rozmiar kodu testowanych midletów. Zmiana ta dochodzi maksymalnie do 20%, co może czynić niektóre z zastosowanych narzędzi pożytecznymi podczas optymalizacji. Największe różnice w rozmiarze plików midletu można zauważyć wtedy, gdy podczas implementacji kodu tego midletu używano długich identyfikatorów klas oraz ich składowych. Podczas testów najlepszy efekt osiągnięto za pomocą aplikacji Proguard. Dodatkowe badania wykazały, iż rozmiar midletu można zmniejszyć jeszcze bardziej, wykorzystując niektóre bardziej zaawansowane funkcje tego narzędzia.

Optymalizacja rozmiaru kodu bajtowego nie ma jednak wielkiego znaczenia, jeżeli midlet zawiera zasoby o dużych rozmiarach, co jest charakterystyczne dla aplikacji multimedialnych. Zachodzi wtedy potrzeba osobnego optymalizowania zasobów, takich jak pliki graficzne, dźwiękowe czy sekwencje wideo. Należy jednak zauważyć, że niskie wyniki niektórych narzędzi w procesie optymalizacji wcale nie dyskredytują ich w roli, dla jakiej zostały opracowane, czyli zaciemniania kodu bajtowego.

Podstawowym problemem napotkanym podczas badań był brak wsparcia dla midletów ze strony większości testowanych narzędzi. Pojedynczy midlet składa się z plików z kodem bajtowym, plików zasobów oraz tekstowego tzw. pliku manifestu, spakowanych razem w jednym archiwum w formacie jar. Poza tym nazwy plików z kodem bajtowym klas muszą być zgodne z ich identyfikatorami. Testowane narzędzia podczas zaciemniania metodą leksykograficzną nie zawsze radziły sobie z zachowaniem odpowiednich powiązań w obrębie pliku archiwum. Dlatego też konieczne było wykorzystanie dodatkowych narzędzi do weryfikacji poprawności budowy midletów.

BIBLIOGRAFIA

1. The Java ME Platform. Witryna: http://java.sun.com/javame/ (czerwiec 2008).
2. The K virtual machine. Witryna: http://java.sun.com/products/cldc/wp/ (czerwiec 2008).
3. Connected Limited Device Configuration. Witryna: http://java.sun.com/products/cldc/ (czerwiec 2008).
4. Mobile Information Device Profile. Witryna: http://java.sun.com/products/midp/ (czerwiec 2008).
5. Hardwick J.: Java optimization. Witryna: http://www.cs.cmu.edu/~jch/java/optimization.html (maj 2008).
6. Shirazi J.: Java Performance Tuning. Witryna: http://www.javaperformancetuning.com (maj 2008).
7. Wojtyczka M.: Tworzenie i optymalizacja gier na urządzenia mobilne w technologii J2ME. Praca dyplomowa magisterska, Politechnika Śląska, Gliwice 2006.
8. D. Low: Java Control Flow Obfuscation. MSc thesis, University of Auckland, 1998. Witryna: http://www.cs.auckland.ac.nz/~cthombor/Pubs/dlowthesis.pdf (czerwiec2008).
9. C. Collberg, C. Thomborson, D. Low: Obfuscation techniques for enhancing software security. U.S. Patent 6668325. Witryna: http://www.freepatentsonline.com/6668325.html (lipiec 2008).
10. H. Lai: A comparative survey of Java obfuscators available on the Internet. 415.780 Project Report, Computer Science Department, University of Auckland, 2001.
11. M. Karnick, J. MacBride, S. McGinnis, Y. Tang, and R. Ramachandran: A Comparative Study of Java Obfuscators. IASTED International Conference on Software Engineering and Applications, Phoenix, Arizona 2005, November 14–16, s. 82÷86.
12. Dobosz K. (red.): Laboratorium programowania w języku Java. Wydawnictwo Politechniki Śląskiej, Gliwice 2004.
13. Sun Wireless Toolkit for CLDC. Witryna. http://java.sun.com/products/sjwtoolkit/ (czerwiec 2008).

Recenzent: Dr inż. Maciej Bargielski

Wpłynęło do Redakcji 16 stycznia 2009 r.

Abstract

The article presents the thought of applying tools for bytecode obfuscation in the process of memory optimization. At the beginning (chapter 1) the paper introduces to the main problems of preparing applications for Java ME Platform: obfuscation and optimization. Next (chapter 2) essential methods of obfuscation the bytecode prepared for the KVM virtual machine are presented and their effect in reduction of size of applications. The special attention is paid to the lexicographical method. Simple example of code written in Java language shows the idea of this method. Reduction of the size of obfuscated applications equals about 11% is noticed.

In the chapter 3, several free tools for obfuscation testing were chosen and reviewed. Unfortunately the most of them do not support Java ME Platform. Then the generated obfuscated bytecode should be verified.

The chapter 4 presents result of testing. Several midlets form Sun WTK examples was obfuscated and results are placed in tables. Table 1 includes original size of tested data, table 2 – size of obfuscated application, table 3 – percentage reducing the size of the entire applications, and table 4 – percentage reducing the applications bytecode. The best result was over 21% of bytecode reduction. But in several other cases reduction finished with wrong obfuscated bytecode. The last chapter contains conclusions.

Adresy

Krzysztof DOBOSZ: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, krzysztof.dobosz@polsl.pl
Tomasz WESOŁOWSKI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, tomasz.wesolowski@polsl.pl