Inginerie Inversă
şi
Tehnici de Protecție

Ingineria inversă - generalități

Ce este ingineria inversă ?

Ingineria inversă este procesul de extragere a cunoştinţelor sau a elementelor de design din orice obiect făcut de mâna omului. Conceptul a fost abordat cu mult înaintea calculatoarelor sau a tehnologiei moderne şi probabil datează din perioada revoluției industriale. *Ingineria inversă* este similară cu cercetarea ştiinţifică numai că, în cazul procesului de *inginerie inversă* obiectul investigat este făcut de mâna omului, iar în cercetarea ştiinţifică este vorba despre un fenomen natural.

Ingineria inversă este de obicei un proces derulat pentru a obține informațiile lipsă, ideile şi filosofia de design atunci când asemenea informații lipsesc. În unele cazuri informațiile sunt deținute de cineva care nu este dispus să le distribuie. În alte cazuri informațiile au fost pierdute sau distruse.

În mod tradițional *ingineria inversă* constă în studiul unor obiecte create de om şi "disecția" lor pentru a le descoperi secretele designului. Asemenea secrete au fost apoi folosite pentru a face produse similare sau mai bune. În multe industrii *ingineria inversă* implică examinarea produsului sub un microscop sau analizarea componentelor sale pentru a descoperi rolul fiecareia.

Tehnologia software este una din cele mai complexe tehnologii existente în zilele noastre. *Ingineria inversă* în acest caz este ingineria deconstruirii programelor şi constă în deschiderea „cutiei" unui program pentru a privi înăuntru. Ca şi ingineria programării, *ingineria inversă* a programelor este un proces pur virtual ce necesită numai un procesor şi mintea umană. Ingineria inversă a programelor necesită întelegerea profundă a calculatoarelor şi a ingineriei programării şi combină o serie de elemente, cum ar fi: spargerea de cod, programare, analiză logică, inspirația rezolvării unui puzzle. Procesul este folosit de o largă varietate de oameni pentru o largă varietate de scopuri. În majoritatea industriilor, *ingineria inversă* este folosită în scopul de a dezvolta produse competitive. În cazul industriei software, programarea este un proces atât de complex încât în multe cazuri *ingineria inversă* pentru

scopuri competitive este considerată a fi un proces prea sofisticat şi nu se justifică din punct de vedere financiar.

În general, există două categorii de aplicații ale *ingineriei inverse* a programelor: una are legatură cu securitatea sistemelor software, iar cealaltă cu ingineria programării. Ingineria inversă a programelor este foarte cunoscută printre spărgătorii de sistem care o folosesc pentru a analiza şi eventual a depăşi diferite scheme de protecție.

Internetul a schimbat complet industria calculatoarelor şi aspectele legate de securitatea proceselor asistate de calculator. Software-ul maliţios, cum ar fi viruşii, se răspândeşte foarte repede într-o lume în care milioane de persoane sunt conectate la Internet şi folosesc e-mail-ul zilnic. Acum 10-15 ani, un virus ar fi trebuit în mod normal copiat pe o dischetă şi acea dischetă introdusă într-un alt calculator pentru ca virusul să se răspândească. Procesul de infectare era destul de lent, iar procedeul de apărare era mult mai simplu de realizat, deoarece canalele de infecție erau numai câteva şi cereau intervenția umană pentru ca programul să se răspândească. In zilele noastre, datorită Internet-ului viruşii moderni se pot răspândi automat către milioane de calculatoare, fără nici o intervenție umană. Astfel, ingineria inversă poate fi folosită pentru a localiza vulnerabilitățile sistemelor de operare. Aceste "puncte slabe" pot fi utilizate apoi pentru a depăşi sistemul de apărare, permițând infectarea. Dincolo de infectare, responsabilii atacului folosesc uneori tehnicile de deconstruire pentru a localiza vulnerabilitățile programelor, care permit unui program maliţios să aibă acces la informațiile sensibile sau chiar să obțină control total asupra sistemului.

De cealaltă parte a lanțului, cercetătorii de programe anti-virus diseacă şi analizează fiecare nou program maliţios. Ei folosesc tehnici de deconstruire pentru a urmări fiecare pas pe care programul îl urmează şi evaluează daunele pe care le-ar putea cauza, rata preconizată de infectare, modalitățile prin care ar putea fi îndepărtat din sistem şi cum infecția ar putea fi evitată.

Deconstrucția algoritmilor criptagrafici

Algoritmii criptografici pot fi împărțiți în două grupe: algoritmi limitați şi algoritmi bazati pe o cheie. Algoritmii limitați sunt de felul celor în care o literă dintr-un text este

mutată cu cateva litere mai sus sau mai jos. Secretul îl reprezintă algoritmul însuşi. Dar odată ce algoritmul este expus, nu mai este sigur. Algoritmii limitați sunt caracterizați de o securitate foarte slabă deoarece deconstruirea lor conduce la aflarea algoritmului şi deci a secretului de criptare. Deoarece algoritmul este secretul, deconstruirea poate fi privită ca o modalitate de a descifra algoritmul.

În algoritmii bazați pe o cheie, secretul este cheia, adică o valoare numerică, folosită de către algoritm pentru a cifra sau descifra mesajul. Astfel, utilizatorii codifică mesajele folosind chei care sunt păstrate secrete. Destinatarul va descifra mesajul utilizând o altă cheie, numită cheie publică, primită de la expeditor. Algoritmii folosiți sunt de obicei cunoscuți şi deci, deconstruirea lor nu mai are sens. Pentru a descifra un mesaj codat cu un cifru bazat pe cheie, este suficient:

- să obținem cheia, sau
- să fie încercate toate combinațiile posibile până obținem cheia, sau
- să descoperim un defect în algoritm care poate fi folosit pentru a extrage cheia sau mesajul original.

Exista situații în care deconstruirea implementările secrete ale cifrelor bazate pe cheie, poate conduce la aflarea acesteia.

Managementul Drepturilor Digitale

În zilele noastre, majoritatea informațiilor cu drept de autor au o formă digitală. Muzică, filme, chiar şi cărți sunt acum disponibile digital. Această tendință furnizează uriaşe beneficii consumatorului şi uriaşe complicații autorilor şi distribuitorilor. Astfel, consumatorii beneficiază de materiale de o calitate ridicată, uşor accesibile şi simplu de administrat. Furnizorii beneficiază de o distribuție de înaltă calitate la preț redus şi de o mai bună gestiune. Informația digitală este însă extrem de fluidă. Este foarte uşor de mânuit şi poate fi uşor de duplicat. Această fluiditate înseamnă că odată ce materialele care au drept de autor ajung la consumatori, pot fi mutate sau duplicate, încât pirateria a devenit un lucru obişnuit. În mod tradițional, companiile software au luat măsuri împotriva pirateriei introducând tehnologii de protejare a produsului. Acestea sunt părți software adiționale, furnizate odată cu produsul

protejat, care încearcă să prevină sau să oprească utilizatorii să copieze programul. Aceste tehnologii sunt numite tehnologiile managementului drepturilor digitale (MDD). Astfel, conținutul protejat este activ sau "inteligent" şi poate decide dacă este accesibil sau nu. MDD este un element pasiv care de obicei este citit de un alt program, făcând-o mult mai dificil de a o controla sau de a restricționa folosirea. Din acest motiv pentru a controla o tehnologie MDD trebuie să înțelegem cum funcționează. Folosind tehnici inginerie inversă, hackerul poate învăța secretele tehnologiei şi poate descoperi cele mai simple modificări care pot fi făcute programului pentru a anula protecția.

Programele Open-Source

Unul din punctele forte a software-ului open-source îl constitue faptul că a fost analizat şi dezvoltat de mii de ingineri software imparțiali. Prin urmare, orice problemă semnalată este rezolvată foarte rapid. Problemele de securitate pot fi descoperite din timp, de cele mai multe ori înainte ca programele malițioase să le folosească în avantajul lor.

Pentru un software brevetat, a cărui cod de sursă nu este accesibil, deconstructia sa reprezintă soluția pentru căutarea de vulnerabilități în securitate. Ingineria inversă nu poate să facă un software brevetat la fel de accesibil precum un software-ul open-source, dar pot fi analizate diverse segmente de cod şi pot fi evaluate riscurile care pot să apară, din punctul de vedere al securității.

Deconstrucția în dezvoltarea de software

Deconstrucția poate fi foarte utilă în dezvoltarea de software. De exemplu, dezvoltatorii de software pot folosi tehnici de deconstrucție pentru a descoperi cum funcționează un software nedocumentat sau slab documentat. În alte situații, deconstructia poate fi folosită pentru a determina codul bibliotecilor folosite de programe sau chiar ale unui sistem de operare. Prin inginerie inversă este posibil să se obțină informații valoroase despre produsele concurenței cu scopul de a îmbunătăți propria tehnologie.

Software de nivel-scăzut

Calculatoarele şi software-ul sunt construite strat peste strat. Pe cel mai de jos strat se găseşte microprocesorul, care conține milioane de tranzistoare microscopice, pulsând la viteze foarte mari. În zilele noastre, majoritatea celor care dezvoltă software folosesc limbaje de nivel-înalt, care sunt uşor de învățat şi de utilizat. Cu ajutorul lor, ei crează programe care, în cea mai mare parte sunt controlabile printr-o interfață grafică (GUI). Prin această interfață, utilizatorul trimite comenzi programului, programul trimite comenzi sistemului de operare, iar sistemul de operare trimite comenzi procesorului sau altor dispozitive electronice conectate cu procesorul. De exemplu, comenzile care crează o fereastră, încarcă o pagină Web, sau arată o imagine, au în spate, la nivel inferior, mii sau chiar milioane de comenzi.

Software-ul de nivel-scăzut (cunoscut şi sub denumirea de software de sistem) este un nume generic pentru infrastructura care asigură funcționarea corectă a programelor. Această categorie cuprinde compilatoarele, depanatoarele, sistemele de operare, limbajele de programare de nivelul-scăzut şi limbajul de ansamblare. Software-ul de nivel-scăzut constitue interfața dintre limbajul procesorului (codul maşină) şi limbajele de nivel înalt utilizate de programatori pentru a scrie programele. Sistemele de operare constitue interfața prin care sistemul electronic este gestionat şi prin care este simplificată interacțiunea utilizatorului cu monitorul, mouse-ul, tastatura şi alte dispositive hardware. Nu cu mulți ani în urmă, programatorii trebuiau să lucreze la acest nivel-scăzut, aceasta fiind singura posibilitate pentru a scrie software. În zilele noastre, sistemele de operare moderne şi uneltele moderne existente simplifică mult procesul de dezvoltare a programelor şi izolează programatorii de detaliile nivelului scăzut.

Principala condiție pentru a fi un bun „deconstructor” constă însă într-o solidă înțelegere a software-ului de nivel-scăzut şi a programării la acest nivel. Aceasta deoarece aspectele de nivel-scazut ale unui program sunt adesea singurele lucruri cu care poate opera un „decostructor”. Detaliile nivelului-înalt sunt aproape întotdeauna eliminate înainte ca programul software să fie furnizat clientului.

Limbajul de Asamblare

Limbajul de asamblare este limbajul de pe cel mai scăzut nivel al lanțului software. Aceasta îl face cel mai potrivit pentru ingineria inversă. Orice operație executată de un program are un corespondent în limbajul de asamblare.

Limbajul de asamblare depinde de arhitectura procesorului şi deci, fiecare platformă are propriul său limbaj de asamblare.

Codul pe care îl recunoaşte şi pe care îl poate executa un microprocesor se numeşte cod maşină. Acesta este un cod binar, adică este o susccesiune de biți ce conține o listă de instrucțiuni pe care CPU trebuie să le execute. Codul maşină este alcătuit doar din 0 şi 1 şi prin urmare, chiar dacă îl vizualizăm cu ajutorul unui program, nu putem înțelege prea multe. Ca să-l putem înțelege este nevoie ca acest cod să fie descifrat şi interpretat într-un limbaj pe care să-l putem citi. Acesta este limbajul de asamblare. Limbajul de asamblare este o reprezentare textuală simplă a acelor. Acesta foloseşte cuvinte ca MOV(mişcare), ADD(adunare) sau ECHG(schimbare) pentru a reprezenta diverse comenzi ale codului maşină. Fiecare comandă a limbajului de asamblare este reprezentată de un număr, numit codul operației, sau *opcod*. Codul obiect este o secvență de opcod-uri şi alte numere folosite pentru a realiza operații. CPU citeşte codul obiect din memorie, îl decodează, şi execută comenzile date de opcod-uri.

Dacă este scris un program în limbaj de asamblare, acesta poate fi transformat în cod maşină prin intermediul unui *asamblor*. Procesul invers, de transformare a unui cod maşină în limbaj de asamblare, este realizat prin intermediul unui *dezasamblor*. Dezasambloarele reprezintă cele mai importante instrumente utilizate în ingineria inversă.

Compilatoare

Compilatoarele sunt programe speciale, direct asociate cu un anumit limbaj de nivel înalt, care încarcă în memorie codul sursă al unui program scris în limbajul respectiv, îl interpretează, îl verifică din punct de vedere sintactic, verifică legăturile şi generează un cod maşină (sau un cod de octeți) asociat codului sursă respectiv. Codul maşină generat va depinde de arhitectura procesorului pe care va fi executat şi de sistemul de operare prin

intermediul căruia a fost creat. Deci un cod maşină este dependent de platformă. Un program compilat la cod maşină pentru sistemul de operare Windows, nu poate fi executat pe Linux, de exemplu.

Sisteme de operare

Un sistem de operare este un ansamblu de programe ce administrează computerul din punct de vedere hardware-ul şi al aplicațiilor. Un sistem de operare se ocupă de multe sarcini diferite şi poate fi privit ca un coordonator între diferitele elemente ale calculatorului. Sistemele de operare sunt un element esențial pentru funcționarea computerelor, fără de care acestea nu pot funcționa. Cum am văzut mai sus, codul maşină asociat unui program este produs de un compilator care este executat pe un sistem de operare. Prin urmare, o condiție esențială pentru orice deconstructor este buna cunoaştere a sistemului de operare.

Procesul de deconstrucție

Procesul de inginerie inversă a unui program se desfăşoară de obicei în doua faze distincte. Prima fază constă în analiza la o scară largă a programului, a modului de funcționare, a funcțiilor acestuia, analiză care conduce la determinarea structurii generale a programului şi la localizarea zonelor de interes. Această fază se numeşte *deconstrucția la nivel de sistem*. În faza a doua se trece la deconstrucția prin mijloace specifice a zonelor de interes, fază numită *deconstrucția la nivel de cod*.

Deconstrucția la nivel de sistem presupune folosirea serviciilor oferite de program pentru a obține informații. Se execută programul, se identifică anumite mesaje pe care acesta le afişează, sunt urmărite şi analizate datele de intrare şi de ieşire, etc. Majoriatea informațiilor sunt furnizate de către sistemul de operare.

Deconstrucția la nivel de cod este un process complex ce necesită multă experiență, o bună cunoaştere a tehnicilor de inversiune, a calculatorului şi sistemului de operare, a limbajului în care a fost scris programul şi a limbajului de asamblare. În această fază se analizează codul la nivel inferior, prin intermediul limbajului de asamblare şi se caută detalii

privind funcționarea acestuia. Multe dintre aceste detalii sunt generate automat de către compilator şi nu sunt cunoscute nici de autorii programului.

Uneltele folosite în ingineria inversă

Deconstrucția la nivel de sistem necesită o varietate de instrumente pentru diverse scopuri cum ar fi: detecție, monitorizare, analiză, etc. Majoritatea dintre acestea afişează informații despre aplicație şi despre mediul de execuție, furnizate de sistemul de operare. Cum aproape toată comunicarea între program şi mediul exterior se desfăşoară prin sistemul de operare, acesta poate fi monitorizat şi interogat pentru a obține aceste informații. Instrumentele de monitorizare a sistemului pot monitoriza activitatea de rețea, accesul la fişiere, accesul la regiştri, etc.

Instrumentele pentru deconstrucția la nivel de cod sunt: *dezasambloarele*, *depanatoarele* şi *decompilatoarele*.

Dezasambloarele sunt programe care citesc un program executabil binar şi generează o serie de fişiere text ce conțin codul în limbaj de asamblare asociat codului maşină, pentru întregul program sau doar pentru o parte a acestuia. Acesta este un proces relativ simplu considerând că limbajul de asamblare este o rescriere a codului binar. Întrucât codul binar este dependent de platformă, dezasambloarele vor depinde şi ele de platformă. Un dezasamblor de bună calitate este un element esențial în procesul de inginerie inversă.

Depanatoarele sunt programe care permit dezvoltatorilor de software să observe programul lor în timpul execuției. Cele mai importante trăsături ale unui depanator sunt posibilitatea de a plasa puncte de întrerupere şi posibilitatea de a urmări codul. Punctele de întrerupere permit programatorilor să selecteze o anumită funcție sau o linie de cod, oriunde în codul programului, iar depanatorul va opri execuția programului la acel punct şi va afişa starea programului. Mai departe programatorul poate comanda continuarea normală a execuției sau poate comanda ca execuția să fie desfăşurată instrucțiune cu instrucțiune. În acest caz se poate urmări starea programului la fiecare pas, prin valorile ce intervin în calcule, prin încărcătura variabilelor, etc. Aceasta permite programatorilor să vadă exact fluxul

programului şi să detecteze erorile logice (erorile de sintaxă sunt detectate de către compilator, în timpul compilării).

Deoarece programatorii au acces la codul sursă al programului lor, depanatorul prezintă programul prin codul sursă şi permite plasarea de puncte de întrerupere în acesta şi urmărirea codului sursă linie cu linie, chiar dacă, de fapt depanatorul lucrează cu codul maşină asociat.

Pentru un deconstructor, depanatoarele sunt aproape la fel de importante ca şi pentru un programator, dar din raţiuni diferite. În primul rând, deconstructorii folosesc depanatoarele în modul dezasamblare. În modul dezasamblare, un depanator foloseşte un dezasamblor care să dezasambleze codul obiect în timpul execuţiei programului. Deconstructorii pot citi codul dezasamblat şi pot urmări activitatea CPU în timpul execuţiei fiecărei instrucţiuni. La fel ca în cazul depanării cu cod sursă, deconstructorii pot plasa puncte de întrerupere în locuri de interes ale codului dezasamblat şi apoi pot urmări starea programului la acele puncte.

Prin urmare, pentru procesul de inginerie inversă este nevoie de un bun depanator cu un bun dezasamblor asociat.

Decompilatoarele reprezintă următorul nivel după dezasambloare. Un decompilator ia un program executabil binar şi încearcă să obţină codul sursă al acestuia în limbajul de nivel înalt în care a fost scris. Ideea pe care se bazează este cea de inversarea a procesului de compilare, adică un proces de inginerie inversă. Decompilarea este completă dacă în urma recompilării codului sursă obţinut se obţine un program binar executabil cu aceleaşi caracteristici cu ale programului iniţial. Decompilarea este însă un proces foarte complex şi foarte dificil de realizat. Astfel, pe majoritatea platformelor actuale, obţinerea codului sursă din cod binar este deocamdată imposibilă. Aceasta deoarece, în majoritatea limbajelor de nivel înalt, există multe elemente importante care sunt omise în timpul compilării şi care sunt imposibil de recuperat.

Cu toate acestea, există numeroase încercări de realizare de decompilatoare, pentru diverse limbaje şi diverse platforme, există grupuri de cercetare în această direcţie cu numeroase publicaţii, teze de doctorat, etc. Amintesc aici pe Dr. Cristina Cifuentes, de la Sun Microsystems Laboratories, a cărei teză de doctorat abordează tocmai acest subiect:

C Cifuentes, *Reverse Compilation Techniques*, PhD thesis, Faculty of Information Technology, Queensland University of Technology, July 1994.

Etica ingineriei inverse

Dezbaterea privind legalitatea ingineriei inverse continuă şi astăzi, după mulți ani de dezbateri. De obicei se discută despre impactul pe care îl are ingineria inversă asupra societății şi bineînțeles, acest impact depinde în mare măsură de scopul în care este folosită ingineria inversă. În continuare vom da argumente pro şi contra utilizării ingineriei inverse.

Interoperabilitatea

Dezvoltarea a două programe distincte care să comunice între ele nu este o sarcină uşoară, chiar şi atunci când ambele programe sunt dezvoltate în cadrul aceleaşi firme. Legăturile software sunt atât de complexe şi programele sunt atât de sensibile încât aceste lucruri rar funcționează corect la prima încercare. Cu atât mai dificilă este problema când un producător software doreşte să dezvolte un program care să comunice cu o componentă dezvoltată de altă companie. Acesta are nevoie de foarte multe informații, care de multe ori nu sunt furnizate de cealaltă companie. Această problemă este întâmpinată frecvent de producătorii de hardware, care trebuie să scrie drivere pentru diverse sisteme de operare.

O platformă software este orice sistem software sau mecanism hardware ale cărui programe pot ajunge în top. De exemplu, atât Microsoft Windows cât şi Sony Playstation sunt platforme software. Pentru un dezvoltator de platforme software, decizia dacă să publice sau nu detaliile legăturilor cu platforma sa este una critică. Pe de o parte, expunerea legăturilor software permite altui producător să fie capabil să dezvolte software care poate ajunge în vârful platformei. Aceasta ar putea conduce vânzările către o platformă ascendentă, dar vânzătorul ar putea, de asemenea, oferi propriul lui software. Publicarea legăturilor software ar putea, de asemenea, crea o nouă competiție pentru propriile aplicații ale vânzătorului.

Prin urmare, în general este dificil de a obține aceste informații şi atunci acestea ar putea fi obținute prin inginerie inversă. Cum acest lucru este interzis de către firma proprietară, procesul este ilegal.

Competiția

Când este folosită pentru interoperabilitate, de ingineria inversă beneficiază societatea pentru că simplifică (sau permite) dezvoltarea de tehnologii noi şi îmbunătățite. Când ingineria inversă este folosită în dezvoltarea produselor de competiție, situația este mai complicată. Oponenții ingineriei inverse, de obicei, pretind că aceasta împiedică inovația întrucât dezvoltatorii software ezită între a investi în cercetare şi dezvoltare sau a dezvolta noile tehnologii inspirați de unele existente, descifrate prin inginerie inversă.

Prin inginerie inversă pot fi furați algoritmii, segmente de cod, porțiuni de interfață, etc, care pot fi folosite în propriile aplicații. Aceasta este o violare clară ale legilor copyright-ului şi este uşor de dovedit. Un exemplu mult mai complicat este aplicarea unui proces de decompilare programului, modificarea codurilor sursă obținute pentru a ascunde urmele şi recompilarea lor aşa încât să se obțină un program binar executabil cu o funcționalitatea identică cu originalul. Acesta evident este obținut dintr-un cod diferit şi este mult mai dificil de dovedit că a fost furat. Procedeul poate fi aplicat doar unei părți de program ceea ce face şi mai dificil de dovedit.

Exemple: ingineria inversă a programelor Java

Limbajul Java a fost lansat oficial în anul 1995, de către firma Sun Microsystems şi este limbajul cu cea mai rapidă evoluție din istoria limbajelor de programare. La ora actuală constitue una dintre cele mai utilizate platforme de programare, fapt datorat în primul rând portabilității programelor scrise în Java. Deoarece este foarte mult utilizat în Web, securitatea limbajului este foarte importantă. Este importantă pentru web-designeri, pentru administratorii de sistem, pentru comerțul electronic, pentru tranzacțiile bancare, etc.

Pentru a înțelege punctul slab al limbajului, trebuie să cunoaştem caracteristicile acestuia.

Majoritatea limbajelor de programare existente sunt de două categorii: limbaje interpretate sau limbaje compilate. Limbajele interpretate sunt uşor de învățat şi de folosit. Un astfel de limbaj, foarte răspândit şi utilizat, este limbajul BASIC. Numele acestuia indică tocmai faptul că este uşor de folosit. După ce a fost scris un program în BASIC, acesta poate fi rulat imediat. Interpretorul BASIC citeşte instrucțiunile linie cu linie, le interpretează şi le execută. Deoarece interpretarea codului sursă are loc în timpul rulării, se pierde din viteza de execuție. Din acest motiv programatorii preferă limbajele compilate deşi acestea sunt mai greu de utilizat. După ce a fost scris un program într-un limbaj compilat, este folosit un compilator pentru a-l transforma în cod maşină. Astfel, în timpul rulării este executat doar codul maşină, aceasta conducînd la o creştere a vitezei de execuție. Dezavantajul constă în faptul că programul respectiv poate fi executat numai pe platforma pentru care a fost compilat. Deci creşterea vitezei are loc în detrimentul portabilității. Cele mai cunoscute limbaje de programare compilate sunt COBOL, FORTRAN, Pascal, C şi C++. Ultimile două sunt şi cele mai folosite.

Limbajul C a apărut la începutul anilor ’70 şi a atras atenția programatorilor încă de la început. Pentru limbajele cu care intra în competiție la acel moment, COBOL şi FORTRAN, limbajul C era mult mai avansat. Astfel, limbajul C a fost cel mai folosit limbaj de programare între anii ‘70 şi ‘80. La începutul anilor ’80 calculatoarele personale încep să pătrundă pe piață ceea ce a condus la o creştere foarte mare a cererii de software. Pentru reducerea costurilor şi a timpilor de dezvoltare a proiectelor, firmele producătoare îşi puneau tot mai des problema

refolosirii codului. Codul dezvoltat în C este însă dificil de refolosit. De multe ori este mai uşor şi consumăm mai puțin timp dacă scriem totul de la început decât să refolosim diverse coduri deja scrise. Soluția a fost oferită de o echipă de la firma AT&T Bell Labs care a propus o extensie a limbajului C, numită C++. Părintele noului limbaj este Bjarne Stroustrup. Limbajul C++ permite crearea unor module de cod, numite obiecte, folosite pentru executarea unor funcții specifice. Aceste obiecte pot fi create şi utilizate în orice program C++, astfel realizându-se refolosirea codului. Limbajul C++ a constituit o evoluție logică a limbajului C şi de la apariție până la ora actuală a rămas cel mai folosit limbaj de programare.

Deşi este foarte popular, limbajul C++ are şi puncte slabe, acestea fiind moştenite de la limbajul C. Principalele deficiențe ale limbajului C++ sunt legate de folosirea pointerilor şi gestionarea memoriei. O singură linie de cod care accesează o locație greşită de memorie poate duce la blocarea aplicației si chiar a sistemului.

Java este pentru programarea anilor ’90 ce a fost C++ pentru programarea anilor ’80. Java este un limbaj standardizat care se bazează pe cel mai folosit limbaj al momentului, adică C++. Spre deosebire de C++, care este un limbaj procedural şi orientat pe obiecte, Java este un limbaj complet orientat pe obiecte. Problemele limbajelor C şi C++ au fost eliminate în noul limbaj. În Java nu există pointeri, iar gestionarea memoriei se face automat. Java este un limbaj cu execuție multifilară, adică interpretorul Java poate să ruleze mai multe fire de execuție în paralel.

Limbajul Java este un limbaj compilat şi interpretat. Prin compilarea unui program scris într-un limbaj de programare compilat, codul sursă este transformat în cod maşină. În urma compilării unui program Java, codul sursă este transformat într-un limbaj de nivel mediu, numit cod de octeți. Codul de octeți este diferit de codul maşină. Codul maşină este reprezentat printr-o succesiune de 0 şi de 1, pe când codul de octeți se aseamănă mai mult cu limbajul de asamblare. Codul maşină poate fi executat direct de procesor, dar codul de octeți trebuie interpretat înainte de a fi executat. Acest lucru cade în seama mediului de execuție Java, interpretorul Java interpretează codul de octeți şi îl execută. Întârzierea produsă în timpul rulării de această interpretare este destul de mică, executarea programului fiind aproape la fel de rapidă ca şi cea a unui program compilat la cod maşină. Problemele de viteză au fost rezolvate de apariția interpretoarelor optimizate, JIT (Just In Time), interpretoare introduse în majoritatea browserelor actuale. De asemenea, firma Sun

Microsystems a produs microprocesoare optimizate Java în care codul Java rulează mult mai repede decât pe procesoarele obişnuite.

Diferența majoră dintre codul maşină şi cel de octeți este următoarea: codul maşină poate fi rulat numai pe platforma (arhitectură de procesor + sistemul de operare) pentru care a fost compilat, pe când codul de octeți poate fi rulat pe orice platformă pe care se găseşte mediul de execuție Java, aceasta asigurînd neutralitatea arhitecturală a limbajului Java. Orice compilator, inclusiv cel Java, compilează codul sursă pentru o anumită platformă. Diferența constă în faptul că în limbajul Java această platformă este una virtuală, creată în memoria calculatorului. Prin urmare, mediul de execuție Java crează în memorie un "calculator" imaginar, numit Maşina Virtuală Java, pentru care este compilat şi pe care este rulat codul de octeți. În acest proces, interpretorul Java joacă rolul de intermediar între Maşina Virtuală şi maşina reală. Datorită maşinii virtuale, limbajul Java este neutru din punct de vedere arhitectural, ceea ce înseamnă că programele Java pot fi rulate pe orice platformă pe care este instalat mediul de execuție Java.

Limbajul Java este portabil, ceea ce înseamnă că putem folosi componentele Java pe platforme eterogene.

Limbajul Java este distribuit, adică putem folosi atât obiecte memorate local, cât şi obiecte stocate în rețea.

Java este compatibil cu operarea în rețea, adică poate fi utilizat în rețele complexe şi acceptă toate protocoalele de rețea obişnite.

Java este un limbaj sigur. Deoarece Java nu foloseşte pointeri iar memoria este alocată numai în timpul încărcării obiectelor, accesul la stiva sistemului şi la zona de memorie liberă este oprit. De asemenea, Java asigură securitatea sistemului în timpul execuției, prin verificarea codului de octeți de către interpretor. Astfel, înaintea executării unui program, interpretorul Java verifică validitatea codului semnalînd următoarele: conversiile ilegale de date, valori şi parametri incorecți, modificarea claselor sau folosirea incorectă a acestora, etc.

Codul de octeți al claselor Java respectă un format unic numit *formatul class*. Acesta poate fi dezasamblat şi, datorită faptului că execuția sa este esențial bazată pe stivă, poate fi chiar decompilat. Acesta este prețul portabilității programelor Java şi implicit al succesului acestui limbaj.

De la apariția limbajului s-au scris multe decompilatoare Java, dar cel mai performat este JAD (Java Decompiler) scris de Pavel Kouznetsov. Ca să înțelegem cum lucrează un astfel de decompilator, vom face acest proces pas cu pas, "manual".

Să luăm la întâmplare un program java, de exemplu un applet descărcat din Web. Fie acesta *snow.class*, referit de pagina web http://thefrontier.tripod.com/winterwonderland.htm. Parametrii acestui applet îi aflăm din codul HTML al documentului (eticheta <applet>) şi tot de aici aflăm numele clasei primare a appletului, în cazul nostru *snow*. Eticheta applet este:

```
<applet code="snow.class" align="baseline" width="480" height="320">
      <param name="grph" value="images/churchwinter.jpg">
      <param name="snows" value="700">
      <param name="threadsleep" value="50">
</applet>
```

Acum, în bara de adrese a browser-ului scriem http://thefrontier.tripod.com/snow.class şi vom salva pe HDD fişierul *snow.class*.

Mai departe vom folosi programul JCD (Java Class Disassembler), scris de autor, pentru a dezasambla codul de octeți al clasei *snow*. Programul JCD este Open Sources şi poate fi descărcat de la adresa http://www.math.uaic.ro/~drusu/files/jcd.zip. Termenul "dezasamblat" este impropriu folosit aici, dar îl vom utiliza întrucât limbajul maşini virtuale Java seamănă destul de mult cu limbajul de asamblare.

Cum ştim, un fişier clasă are numele identic cu cel al fişierului sursă care l-a generat, deci vom crea un fişier text cu numele *snow* şi extensia *java*. În acesta vom scrie codul sursă obținut în urma decompilării. Utilizând JCD obținem următorul cod dezasamblat:

```
//////////////////////////////////////////////////////////
    This code was generated with JAVA CLASS DISASSEMBLER
    Copyright © Danut Rusu, 2001
//////////////////////////////////////////////////////////


ClassFile:   snow.class

  ---------------------------------------------------------------------

00000000        Signature                               CAFE BABE
00000004        Minor Version                           3
00000006        Major Version                           45

CONSTANT  POOL
```

```
00000008        Constant Pool Count             197 (198 - 1)
0000000a        CONSTANT_Class                  Entry (1) (94)
0000000d        CONSTANT_Methodref              Entry (2) Class (1) Name/Type (95)
00000012        CONSTANT_Fieldref               Entry (3) Class (53) Name/Type (96)
00000017        CONSTANT_Methodref              Entry (4) Class (97) Name/Type (98)
0000001c        CONSTANT_Fieldref               Entry (5) Class (53) Name/Type (99)
00000021        CONSTANT_Fieldref               Entry (6) Class (100) Name/Type (101)
00000026        CONSTANT_Fieldref               Entry (7) Class (100) Name/Type (102)
0000002b        CONSTANT_Methodref              Entry (8) Class (97) Name/Type (103)
00000030        CONSTANT_Fieldref               Entry (9) Class (53) Name/Type (104)
00000035        CONSTANT_Methodref              Entry (10) Class (29) Name/Type (105)
0000003a        CONSTANT_Fieldref               Entry (11) Class (53) Name/Type (106)
0000003f        CONSTANT_String                 Entry (12) (71)
00000042        CONSTANT_Methodref              Entry (13) Class (54) Name/Type (107)
00000047        CONSTANT_Fieldref               Entry (14) Class (53) Name/Type (108)
0000004c        CONSTANT_Methodref              Entry (15) Class (109) Name/Type (110)
00000051        CONSTANT_Methodref              Entry (16) Class (109) Name/Type (111)
00000056        CONSTANT_String                 Entry (17) (112)
00000059        CONSTANT_Fieldref               Entry (18) Class (53) Name/Type (113)
0000005e        CONSTANT_Fieldref               Entry (19) Class (53) Name/Type (114)
00000063        CONSTANT_Fieldref               Entry (20) Class (53) Name/Type (115)
00000068        CONSTANT_Methodref              Entry (21) Class (1) Name/Type (116)
0000006d        CONSTANT_String                 Entry (22) (117)
00000070        CONSTANT_String                 Entry (23) (118)
00000073        CONSTANT_String                 Entry (24) (119)
00000076        CONSTANT_String                 Entry (25) (120)
00000079        CONSTANT_Class                  Entry (26) (121)
0000007c        CONSTANT_Methodref              Entry (27) Class (26) Name/Type (122)
00000081        CONSTANT_Fieldref               Entry (28) Class (53) Name/Type (123)
00000086        CONSTANT_Class                  Entry (29) (124)
00000089        CONSTANT_Fieldref               Entry (30) Class (53) Name/Type (125)
0000008e        CONSTANT_Methodref              Entry (31) Class (54) Name/Type (126)
00000093        CONSTANT_Methodref              Entry (32) Class (54) Name/Type (127)
00000098        CONSTANT_Methodref              Entry (33) Class (26) Name/Type (128)
0000009d        CONSTANT_Methodref              Entry (34) Class (26) Name/Type (129)
000000a2        CONSTANT_Fieldref               Entry (35) Class (53) Name/Type (130)
000000a7        CONSTANT_Class                  Entry (36) (131)
000000aa        CONSTANT_Fieldref               Entry (37) Class (53) Name/Type (132)
000000af        CONSTANT_Class                  Entry (38) (133)
000000b2        CONSTANT_Methodref              Entry (39) Class (38) Name/Type (134)
000000b7        CONSTANT_Methodref              Entry (40) Class (38) Name/Type (135)
000000bc        CONSTANT_Methodref              Entry (41) Class (38) Name/Type (136)
000000c1        CONSTANT_Methodref              Entry (42) Class (97) Name/Type (137)
000000c6        CONSTANT_Fieldref               Entry (43) Class (138) Name/Type (139)
000000cb        CONSTANT_Methodref              Entry (44) Class (140) Name/Type (141)
000000d0        CONSTANT_Methodref              Entry (45) Class (140) Name/Type (142)
000000d5        CONSTANT_Methodref              Entry (46) Class (140) Name/Type (143)
000000da        CONSTANT_Methodref              Entry (47) Class (53) Name/Type (144)
000000df        CONSTANT_Methodref              Entry (48) Class (53) Name/Type (145)
000000e4        CONSTANT_Fieldref               Entry (49) Class (138) Name/Type (146)
000000e9        CONSTANT_Fieldref               Entry (50) Class (53) Name/Type (147)
000000ee        CONSTANT_Methodref              Entry (51) Class (148) Name/Type (149)
000000f3        CONSTANT_Methodref              Entry (52) Class (54) Name/Type (95)
000000f8        CONSTANT_Class                  Entry (53) (150)
000000fb        CONSTANT_Class                  Entry (54) (151)
000000fe        CONSTANT_Class                  Entry (55) (152)
00000101        CONSTANT_Utf8                   Entry (56) mainThread
0000010e        CONSTANT_Utf8                   Entry (57) Ljava/lang/Thread;
00000123        CONSTANT_Utf8                   Entry (58) offScrn
0000012d        CONSTANT_Utf8                   Entry (59) Ljava/awt/Image;
00000140        CONSTANT_Utf8                   Entry (60) offGrph
0000014a        CONSTANT_Utf8                   Entry (61) Ljava/awt/Graphics;
00000160        CONSTANT_Utf8                   Entry (62) rand
00000167        CONSTANT_Utf8                   Entry (63) Ljava/util/Random;
0000017c        CONSTANT_Utf8                   Entry (64) stopFlag
00000187        CONSTANT_Utf8                   Entry (65) I
0000018b        CONSTANT_Utf8                   Entry (66) stopTime
```

```
00000196        CONSTANT_Utf8                   Entry (67) J
0000019a        CONSTANT_Utf8                   Entry (68) snowX
000001a2        CONSTANT_Utf8                   Entry (69) [I
000001a7        CONSTANT_Utf8                   Entry (70) snowY
000001af        CONSTANT_Utf8                   Entry (71) snows
000001b7        CONSTANT_Utf8                   Entry (72) wind
000001be        CONSTANT_Utf8                   Entry (73) threadSleep
000001cc        CONSTANT_Utf8                   Entry (74) dim
000001d2        CONSTANT_Utf8                   Entry (75) Ljava/awt/Dimension;
000001e9        CONSTANT_Utf8                   Entry (76) gAlc
000001f0        CONSTANT_Utf8                   Entry (77) [Ljava/awt/Image;
00000204        CONSTANT_Utf8                   Entry (78) mt
00000209        CONSTANT_Utf8                   Entry (79) Ljava/awt/MediaTracker;
00000223        CONSTANT_Utf8                   Entry (80) init
0000022a        CONSTANT_Utf8                   Entry (81) ()V
00000230        CONSTANT_Utf8                   Entry (82) Code
00000237        CONSTANT_Utf8                   Entry (83) LineNumberTable
00000249        CONSTANT_Utf8                   Entry (84) start
00000251        CONSTANT_Utf8                   Entry (85) stop
00000258        CONSTANT_Utf8                   Entry (86) run
0000025e        CONSTANT_Utf8                   Entry (87) paint
00000266        CONSTANT_Utf8                   Entry (88) (Ljava/awt/Graphics;)V
0000027f        CONSTANT_Utf8                   Entry (89) update
00000288        CONSTANT_Utf8                   Entry (90) drawBackSnow
00000297        CONSTANT_Utf8                   Entry (91) <init>
000002a0        CONSTANT_Utf8                   Entry (92) SourceFile
000002ad        CONSTANT_Utf8                   Entry (93) snow.java
000002b9        CONSTANT_Utf8                   Entry (94) java/util/Random
000002cc        CONSTANT_NameAndType            Entry (95) Name (91) Type (81)
000002d1        CONSTANT_NameAndType            Entry (96) Name (62) Type (63)
000002d6        CONSTANT_Class                  Entry (97) (153)
000002d9        CONSTANT_NameAndType            Entry (98) Name (154) Type (155)
000002de        CONSTANT_NameAndType            Entry (99) Name (74) Type (75)
000002e3        CONSTANT_Class                  Entry (100) (156)
000002e6        CONSTANT_NameAndType            Entry (101) Name (157) Type (65)
000002eb        CONSTANT_NameAndType            Entry (102) Name (158) Type (65)
000002f0        CONSTANT_NameAndType            Entry (103) Name (159) Type (160)
000002f5        CONSTANT_NameAndType            Entry (104) Name (58) Type (59)
000002fa        CONSTANT_NameAndType            Entry (105) Name (161) Type (162)
000002ff        CONSTANT_NameAndType            Entry (106) Name (60) Type (61)
00000304        CONSTANT_NameAndType            Entry (107) Name (163) Type (164)
00000309        CONSTANT_NameAndType            Entry (108) Name (71) Type (65)
0000030e        CONSTANT_Class                  Entry (109) (165)
00000311        CONSTANT_NameAndType            Entry (110) Name (166) Type (167)
00000316        CONSTANT_NameAndType            Entry (111) Name (168) Type (169)
0000031b        CONSTANT_Utf8                   Entry (112) threadsleep
00000329        CONSTANT_NameAndType            Entry (113) Name (73) Type (65)
0000032e        CONSTANT_NameAndType            Entry (114) Name (68) Type (69)
00000333        CONSTANT_NameAndType            Entry (115) Name (70) Type (69)
00000338        CONSTANT_NameAndType            Entry (116) Name (170) Type (169)
0000033d        CONSTANT_Utf8                   Entry (117) graphic
00000347        CONSTANT_Utf8                   Entry (118) graph
0000034f        CONSTANT_Utf8                   Entry (119) grph
00000356        CONSTANT_Utf8                   Entry (120) tento1.gif
00000363        CONSTANT_Utf8                   Entry (121) java/awt/MediaTracker
0000037b        CONSTANT_NameAndType            Entry (122) Name (91) Type (171)
00000380        CONSTANT_NameAndType            Entry (123) Name (78) Type (79)
00000385        CONSTANT_Utf8                   Entry (124) java/awt/Image
00000396        CONSTANT_NameAndType            Entry (125) Name (76) Type (77)
0000039b        CONSTANT_NameAndType            Entry (126) Name (172) Type (173)
000003a0        CONSTANT_NameAndType            Entry (127) Name (174) Type (175)
000003a5        CONSTANT_NameAndType            Entry (128) Name (176) Type (177)
000003aa        CONSTANT_NameAndType            Entry (129) Name (178) Type (179)
000003af        CONSTANT_NameAndType            Entry (130) Name (64) Type (65)
000003b4        CONSTANT_Utf8                   Entry (131) java/lang/InterruptedException
000003d5        CONSTANT_NameAndType            Entry (132) Name (56) Type (57)
000003da        CONSTANT_Utf8                   Entry (133) java/lang/Thread
```

```
000003ed        CONSTANT_NameAndType                  Entry (134) Name (91) Type (180)
000003f2        CONSTANT_NameAndType                  Entry (135) Name (84) Type (81)
000003f7        CONSTANT_NameAndType                  Entry (136) Name (181) Type (182)
000003fc        CONSTANT_NameAndType                  Entry (137) Name (183) Type (81)
00000401        CONSTANT_Class                        Entry (138) (184)
00000404        CONSTANT_NameAndType                  Entry (139) Name (185) Type (186)
00000409        CONSTANT_Class                        Entry (140) (187)
0000040c        CONSTANT_NameAndType                  Entry (141) Name (188) Type (189)
00000411        CONSTANT_NameAndType                  Entry (142) Name (190) Type (191)
00000416        CONSTANT_NameAndType                  Entry (143) Name (192) Type (193)
0000041b        CONSTANT_NameAndType                  Entry (144) Name (90) Type (81)
00000420        CONSTANT_NameAndType                  Entry (145) Name (87) Type (88)
00000425        CONSTANT_NameAndType                  Entry (146) Name (194) Type (186)
0000042a        CONSTANT_NameAndType                  Entry (147) Name (72) Type (65)
0000042f        CONSTANT_Class                        Entry (148) (195)
00000432        CONSTANT_NameAndType                  Entry (149) Name (196) Type (197)
00000437        CONSTANT_Utf8                         Entry (150) snow
0000043e        CONSTANT_Utf8                         Entry (151) java/applet/Applet
00000453        CONSTANT_Utf8                         Entry (152) java/lang/Runnable
00000468        CONSTANT_Utf8                         Entry (153) java/awt/Component
0000047d        CONSTANT_Utf8                         Entry (154) size
00000484        CONSTANT_Utf8                         Entry (155) ()Ljava/awt/Dimension;
0000049d        CONSTANT_Utf8                         Entry (156) java/awt/Dimension
000004b2        CONSTANT_Utf8                         Entry (157) width
000004ba        CONSTANT_Utf8                         Entry (158) height
000004c3        CONSTANT_Utf8                         Entry (159) createImage
000004d1        CONSTANT_Utf8                         Entry (160) (II)Ljava/awt/Image;
000004e8        CONSTANT_Utf8                         Entry (161) getGraphics
000004f6        CONSTANT_Utf8                         Entry (162) ()Ljava/awt/Graphics;
0000050e        CONSTANT_Utf8                         Entry (163) getParameter
0000051d        CONSTANT_Utf8                    Entry (164) (Ljava/lang/String;)Ljava/lang/String;
00000546        CONSTANT_Utf8                         Entry (165) java/lang/Integer
0000055a        CONSTANT_Utf8                         Entry (166) valueOf
00000564        CONSTANT_Utf8                    Entry (167) (Ljava/lang/String;)Ljava/lang/Integer;
0000058e        CONSTANT_Utf8                         Entry (168) intValue
00000599        CONSTANT_Utf8                         Entry (169) ()I
0000059f        CONSTANT_Utf8                         Entry (170) nextInt
000005a9        CONSTANT_Utf8                         Entry (171) (Ljava/awt/Component;)V
000005c3        CONSTANT_Utf8                         Entry (172) getDocumentBase
000005d5        CONSTANT_Utf8                         Entry (173) ()Ljava/net/URL;
000005e8        CONSTANT_Utf8                         Entry (174) getImage
000005f3        CONSTANT_Utf8         Entry (175) (Ljava/net/URL;Ljava/lang/String;)Ljava/awt/Image;
00000628        CONSTANT_Utf8                         Entry (176) addImage
00000633        CONSTANT_Utf8                         Entry (177) (Ljava/awt/Image;I)V
0000064a        CONSTANT_Utf8                         Entry (178) waitForID
00000656        CONSTANT_Utf8                         Entry (179) (I)V
0000065d        CONSTANT_Utf8                         Entry (180) (Ljava/lang/Runnable;)V
00000677        CONSTANT_Utf8                         Entry (181) sleep
0000067f        CONSTANT_Utf8                         Entry (182) (J)V
00000686        CONSTANT_Utf8                         Entry (183) repaint
00000690        CONSTANT_Utf8                         Entry (184) java/awt/Color
000006a1        CONSTANT_Utf8                         Entry (185) black
000006a9        CONSTANT_Utf8                         Entry (186) Ljava/awt/Color;
000006bc        CONSTANT_Utf8                         Entry (187) java/awt/Graphics
000006d0        CONSTANT_Utf8                         Entry (188) setColor
000006db        CONSTANT_Utf8                         Entry (189) (Ljava/awt/Color;)V
000006f1        CONSTANT_Utf8                         Entry (190) fillRect
000006fc        CONSTANT_Utf8                         Entry (191) (IIII)V
00000706        CONSTANT_Utf8                         Entry (192) drawImage
00000712        CONSTANT_Utf8      Entry (193) (Ljava/awt/Image;IILjava/awt/image/ImageObserver;)Z
00000748        CONSTANT_Utf8                         Entry (194) white
00000750        CONSTANT_Utf8                         Entry (195) java/lang/Math
00000761        CONSTANT_Utf8                         Entry (196) abs
00000767        CONSTANT_Utf8                         Entry (197) (I)I
```

Codul de mai sus reprezintă Tabela Constantelor clasei *snow*. Din punctul de vedere al decompilatorului, din acest cod ne interesează în primul rând intrările CONSTANT_Utf8 al căror conținut reprezintă calea relativă către o clasă. Acestea sunt clasele referite şi utilizate în program, deci trebuiesc importate. Identificăm următoarele clase:

```
java/lang/Thread
java/awt/Image
java/awt/Graphics
java/util/Random;
java/awt/Dimension
java/awt/MediaTracker
java/lang/InterruptedException
java/applet/Applet
java/lang/Runnable
java/awt/Component
java/lang/String
java/lang/Integer
java/awt/Color
java/awt/image/ImageObserver
java/lang/Math
```

Cum pachetul *java.lang* este importat implicit în orice clasă Java, toate clasele din acest pachet nu ne interesează. Din pachetul *java.awt* sunt utilizate mai multe clase, deci vom importa întreg pachetul. Prin urmare puntem scrie în *snow.java*:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.image.ImageObserver;
import java.util.Random;
```

```
CLASS  DECLARATION

0000076e        Access Flags                    ACC_PUBLIC
00000770        This Class                      snow
00000772        Super Class                     java/applet/Applet
```

Din acest cod aflăm modificatorii clasei şi superclasa, deci putem scrie:

```
public class snow extends Applet
```

```
INTERFACES

00000774        Interfaces Count                    1

Interface #1

00000776        Interface                           java/lang/Runnable
```

Tabela interfețelor ne dă informații despre interfețele implementate de această clasă. În acest caz este o singură interfață şi anume *java.lang.Runnable*. Prin urmare, declarația de clasă devine:

```
public class snow extends Applet implements Runnable{
```

```
FIELDS

00000778        Fields Count                        14

Field #1

0000077a        Access Flags
0000077c        Name                                mainThread
0000077e        Type                                Ljava/lang/Thread;
00000780        Attributes Count                    0

Field #2

00000782        Access Flags
00000784        Name                                offScrn
00000786        Type                                Ljava/awt/Image;
00000788        Attributes Count                    0

Field #3

0000078a        Access Flags
0000078c        Name                                offGrph
0000078e        Type                                Ljava/awt/Graphics;
00000790        Attributes Count                    0

Field #4

00000792        Access Flags
00000794        Name                                rand
00000796        Type                                Ljava/util/Random;
00000798        Attributes Count                    0

Field #5

0000079a        Access Flags
0000079c        Name                                stopFlag
0000079e        Type                                I
000007a0        Attributes Count                    0
```

```
Field #6

000007a2        Access Flags
000007a4        Name                                    stopTime
000007a6        Type                                    J
000007a8        Attributes Count                        0

Field #7

000007aa        Access Flags
000007ac        Name                                    snowX
000007ae        Type                                    [I
000007b0        Attributes Count                        0

Field #8

000007b2        Access Flags
000007b4        Name                                    snowY
000007b6        Type                                    [I
000007b8        Attributes Count                        0

Field #9

000007ba        Access Flags
000007bc        Name                                    snows
000007be        Type                                    I
000007c0        Attributes Count                        0

Field #10

000007c2        Access Flags
000007c4        Name                                    wind
000007c6        Type                                    I
000007c8        Attributes Count                        0

Field #11

000007ca        Access Flags
000007cc        Name                                    threadSleep
000007ce        Type                                    I
000007d0        Attributes Count                        0

Field #12

000007d2        Access Flags
000007d4        Name                                    dim
000007d6        Type                                    Ljava/awt/Dimension;
000007d8        Attributes Count                        0

Field #13

000007da        Access Flags
000007dc        Name                                    gAlc
000007de        Type                                    [Ljava/awt/Image;
000007e0        Attributes Count                        0

Field #14

000007e2        Access Flags
000007e4        Name                                    mt
000007e6        Type                                    Ljava/awt/MediaTracker;
000007e8        Attributes Count                        0
```

Codul de mai sus reprezintă *Tabela câmpurilor* care ne dă informații despre atributele acestei clase.

Valoarea lui *Fields_Count* este 14, deci sunt declarate 14 atribute. Primul atribut se numeşte *mainThread*, este de tipul *java.lang.Thread* şi nu are modificatori expliciți. Aceste informații se transpun în cod sursă prin instrucțiunea:

```
Thread mainThread;
```

În mod similar obținem următoarele declarații:

```
Image offScrn;
Graphics offGrph;
Random rand;
int stopFlag;
long stopTime;
int[] snowX;
int[] snowy;
int snows;
int wind;
int threadSleep;
Dimension dim;
Image[] gAlc;
MediaTracker mt;
```

Urmează zona metodelor şi partea cea mai importantă a clasei. Valoarea câmpului *Methods_Count* indică numărul de metode din această clasă, în acest caz 8.

```
METHODS
000007ea        Methods Count                       8
```

Fiecare metodă începe cu header-ul acesteia care indică modificatorii metodei (de acces şi de mod), o referința în tabela constantelor unde găsim numele metodei şi semnătura metodei (tipul parametrilor şi tipul returnat), numărul de atribute al metodei. Urmează apoi lista de atribute. De exemplu, prima metodă este publică, se numeşte *init*, are lista de parametri vidă şi returnează *void*.
Deci putem scrie în fişierul sursă următoarea declarație:

```
public void init(){

}
```

Codul fiecărui atribut începe cu o referință la tabela constantelor unde se găseşte numele atributului. Urmează apoi numărul de octeți ai atributului. Cel mai important atribut al unei metode este atributul *Code*. Acesta conține pe lângă informațiile de mai sus câmpurile *Max_Stack*, ce indică dimensiunea maximă a stivei de operanzi în timpul execuției metodei, *Max_Locals*, numărul maxim de variabile locale utilizate în codul metodei, *Code_Count*, numărul de octeți ai codului metodei şi codul propriu-zis al metodei. În cazul metodei *init* atributul *Code* este stocat pe 553+6 octeți, stiva atinge dimensiunea maximă 5, are 4 variabile locale, iar codul propriu-zis ocupă 369 de octeți.

```
Method #1

000007ec        Access Flags                    ACC_PUBLIC
000007ee        Name                            init
000007f0        Type                            ()V
000007f2        Attributes Count                1
000007f4        Attribute Name                  Code
000007f6        Bytes Count                     553
000007fa        Max Stack                       5
000007fc        Max Locals                      4
000007fe        Code Count                      369
```

Urmează acum partea cea mai dificilă şi anume decompilarea codului metodei. Pentru a reuşi acest lucru trebuie să cunoaştem setul de instrucțiuni cu care operează Maşina Virtuală Java (alcătuit din 201 instrucțiuni). Ideea de bază constă în simularea execuției codului respectiv. Vom utiliza o stivă abstractă în care vom încărca în loc de valori, identificatori, operatori,…, deci cod sursă.

```
0       aload_0
1       new  java/util/Random
4       dup
5       invokespecial  java/util/Random/<init>  ()V
8       putfield  snow/rand  Ljava/util/Random;
```

Instrucțiunea *aload_0* încarcă în stivă valoarea variabilei locale 0. Pentru orice metodă de instanță, variabila locală 0 va stoca întotdeauna referința la obiectul curent, reprezentată în limbajul Java prin cuvântul rezervat *this*. Noi vom scrie în stivă *this*. Instrucțiunea *new* crează în zona de memorie liberă a JVM un nou obiect de tipul specificat şi va introduce în stivă referința la zona respectivă. Noi vom scrie în stivă *new Random*. Instrucțiunea *dup* dublează vârful stivei, deci din nou în vârful stivei avem *new Random*.

Instrucțiunea *invokespecial* apelează un constructor. Ea trebuie să găsească în vârful stivei referința la un obiect nou creat pe care o extrage şi o foloseşte. În cazul nostru este vorba de referința *new Random*. La nivelul codului de octeți toți constructorii au numele <init>, nume atribuit de către compilator în timpul compilării. Semnătura acestui constructor este ()V, adică nu are parametri şi returnează void. Toți constructorii au tipul void în mod implicit. După execuția acestui constructor, rămâne în stivă referința la un obiect inițializat, pe care o vom nota prin *new Random()*. Instrucțiunea *putfield* încarcă o valoare într-un atribut. Ea extrage din stivă primul operand (găseşte *new Random()*), apoi referința la un obiect ce conține atributul (găseşte *this*), identifică atributul printr-o referință pe 2 octeți la tabela constantelor, octeți ce urmează după codul instrucțiunii (în acest caz, *rand* de tipul *Random*) şi încarcă în atributul respectiv valoarea extrasă din stivă. Instrucțiunea *putfield* face parte dintre acele instrucțiuni JVM care în timpul decompilării generează cod sursă. Prin urmare putem scrie:

```
Random rand = new Random();
```

prima instrucțiune a metodei *init*.

Evoluția stivei în timpul execuției instrucțiunilor de mai sus este:

		new Random		
	new Random	*new Random*	*new Random()*	
this	*this*	*this*	*this*	

```
11        aload_0
12        aload_0
13        invokevirtual  java/awt/Component/size  ()Ljava/awt/Dimension;
16        putfield  snow/dim  Ljava/awt/Dimension;
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus este:

	this	*this.size()*	
this	*this*	*this*	

Instrucțiunea *invokevirtual* apelează o metodă de instanță. Metoda se numeşte *size()* şi este căutată în obiectul curent, prin urmare *invokevirtual* foloseşte o referință *this* din stivă pentru a identifica metoda. Metoda nu se găseşte însă în obiectul curent, este căutată în obiectul superclasei (clasa *Applet*) şi apoi în superclasa acesteia, clasa *Component*. Semnătura metodei este

()Ljava/awt/Dimension;, adică nu are parametri şi returnează o referință la un obiect *Dimension*. Prin urmare vom încărca în stiva noastră abstractă acest obiect, reprezentat prin codul *this.size()*. Instrucțiunea *putfield* generează cod sursă, deci obținem:

```
this.dim = this.size();
```

instrucțiune echivalentă cu

```
dim = size();
```

```
19        aload_0
20        aload_0
21        aload_0
22        getfield  snow/dim  Ljava/awt/Dimension;
25        getfield  java/awt/Dimension/width  I
28        aload_0
29        getfield  snow/dim  Ljava/awt/Dimension;
32        getfield  java/awt/Dimension/height  I
35        invokevirtual  java/awt/Component/createImage  (II)Ljava/awt/Image;
38        putfield  snow/offScrn  Ljava/awt/Image;
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus este:

		this	*this.dim*	*this.dim.width*
	this	*this*	*this*	*this*
this	*this*	*this*	*this*	*this*

this	*this.dim*	*this.dim.height*
this.dim.width	*this.dim.width*	*this.dim.width*
this	*this*	*this*
this	*this*	*this*

this.createImage(this.dim.width, this.dim.height)	
this	

Instrucțiunea *getfield* încarcă în stivă valoarea unui atribut; în cazul decompilării vom încărca în stivă numele atributului. De exemplu, instrucțiunea *getfield* de la offset-ul 25 trebuie să găsească în stivă referința la un obiect de tip *Dimension*, referință folosită pentru a obține atributul *width* din acest obiect. Instrucțiunea *invokevirtual* de la offset-ul 35 apelează metoda *createImage* din obiectul *this*, identificată la nivel superior, în clasa *Component*, metodă care are 2 parametri de tipul *int*. Deci

instrucțiunea trebuie să găsească în vârful stivei o valoare de tip *int* (al doilea parametru) pe care o extrage, apoi o valoare de tip *int* (primul parametru), apoi o referință la obiectul ce conține metoda (în cazul nostru *this*) pe care o extrage. Metoda crează un nou obiect de tip *Image*, iar referința acestuia este introdusă în stivă. Instrucțiunea *putfield* generează cod sursă şi obținem:

```
this.offScrn  =  this.createImage(this.dim.width, this.dim.height);
```

instrucțiune echivalentă cu

```
offScrn  =  createImage(dim.width, dim.height);
```

```
41        aload_0
42        aload_0
43        getfield  snow/offScrn  Ljava/awt/Image;
46        invokevirtual  java/awt/Image/getGraphics  ()Ljava/awt/Graphics;
49        putfield  snow/offGrph  Ljava/awt/Graphics;
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus este:

this	*this* *this*	*this.offScrn* *this*	*this.offScrn.getGraphics()* *this*	

Şi din nou putfield generează cod sursă:

```
offGrph = offScrn.getGraphics();
```

```
52        aload_0
53        ldc  "snows"
55        invokevirtual java/applet/Applet/getParameter  (Ljava/lang/String;)Ljava/lang/String;
58        astore_1
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus este:

this	*"snows"* *this*	*this.getParameter("snows")*	

Instrucțiunea *astore_1* extrage vârful stivei şi îl stochează în variabila locală 1. Ne uităm acum la sfârşitul codului acestui atribut, la lista de atribute ale atributului *Code*. Observăm că are un singur atribut şi anume *LineNumberTable*. Cum atributul *LocalVariableTable* nu se găseşte în această listă, nu putem şti cum a fost denumită de programator această variabilă şi orice altă variabilă locală. Tipul variabilei va fi tipul datei încărcate în ea, iar numele îl vom da noi în concordanță cu acest tip. De

exemplu, pentru tipul *int* putem folosi ca identificator i, i1, i2, …, pentru tipul *String* putem folosi s, s1, s2, …, etc. Deoarece variabila locală 1 este folosită pentru prima dată, aici trebuie să avem o declarație. Prin urmare instrucțiunile *..store* pot genera cod sursă. Prefixul *a* al instrucțiunii ne spune că este încărcată o valoare de tip referință. Deci, ținând seama că metoda *getParameter()*, din clasa Applet, returnează un *String*, putem scrie instrucțiunea:

```
String s = getParameter("snows");
```

```
59        aload_1
60        ifnonnull  72
63        aload_0
64        bipush  100
66        putfield  snow/snows  I
69        goto  83
```

Instrucțiunea *aload_1* încarcă în stivă variabila locală 1, deci, din punctul de vedere al decompilatorului, identificatorul s. Instrucțiunea *ifnonnull* extrage din stivă referința la un obiect, în cazul nostru *s*, şi o compară cu referința *null*. În cazul în care nu coincid, execuția codului continuă de la offset-ul 72. Prin urmare, instrucțiunile *ifnull* şi *ifnonnull* generează cod sursă. Pentru ca decompilarea să continue de la offset-ul 63 şi nu de la 72, vom scrie testul invers. Deci avem structura:

```
if(s==null){

}
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus este:

		100	
	this	*this*	
s	*s*	*s*	

Instrucțiunea *bipush 100* încarcă în stivă valoarea întreagă de tipul byte 100 care apoi va fi încărcată în atributul *snows* (şi deci obținem cod sursă). Instrucțiunea *goto 83* determină ca execuția codului să contnue de la offset-ul 83. În concluzie, avem o structură de tipul *if-else*. Prin urmare avem codul:

```
if(s==null){
      snows = 100;
}else{

}
```

Pe varianta "else" se execută codul:

```
72        aload_0
73        aload_1
74        invokestatic  java/lang/Integer/valueOf  (Ljava/lang/String;)Ljava/lang/Integer;
77        invokevirtual  java/lang/Integer/intValue  ()I
80        putfield  snow/snows  I
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus este:

this	*s* *this*	*Integer.valueOf(s)* *this*	*Integer.valueOf(s).intValue()* *this*	

Instrucțiunea *invokestatic* apelează o metodă statică. Referențierea acesteia se face prin clasa ce o conține şi nu printr-un anumit obiect. Prin urmare, este executată metoda statică *valueOf* din clasa *java.lang.Integer*, metodă ce extrage din vârful stivei un parametru de tip *String* (în cazul nostru *s*) şi returnează în stivă referința la un obiect *Integer*. Este invocată apoi metoda *intValue()* a acestui obiect. Rezultatul este un *int*, pe care în stivă noi îl reprezentăm prin *Integer.valueOf(s).intValue()*. În consecință, obținem codul sursă:

```
if(s==null){
      snows = 100;
}else{
      snows = Integer.vaueOf(s).intValue();
}
```

Facem aici un mic comentariu privind corectitudinea acestui cod.

Ne uităm mai întâi la sfârşitul codului propriu-zis, la *Tabela Excepțiilor*. Observăm că aceasta are un singur element, dat de excepția *InterruptedException*. Ori, în codul de mai sus, instrucțiunea `Integer.vaueOf(s).intValue()` converteşte stringul *s* la o valoare de tip *int*. Cum stringul s este preluat din documentul HTML ce lansează applet-ul, prin metoda *getParameter*, valoarea sa poate fi orice şi poate să nu reprezinte un număr. Prin urmare, poate fi lansată o excepție de tipul *NumberFormatException* care nu este tratată. Codul corect este:

```
if(s==null){
      snows = 100;
}else{
      try{snows = Integer.vaueOf(s).intValue();}
      catch(NumberFormatException){...}
}
```

```
83        aload_0
84        getfield  snow/snows  I
87        sipush  500
```

```
90        if_icmple  100
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus este:

this	*this.snows*	*500* *this.snows*	

Instrucțiunea *if_icmple 100* extrage din vârful stivei o valoare întreagă, v2 (500, în cazul nostru), apoi o altă valoare întreagă, v1 (*this.snows*, în cazul nostru), efectuează testul v1<=v2, iar în caz afirmativ execuția continuă de la offset-ul 100. Prin urmare generează cod sursă. Pentru ca decompilarea să continue cu offset-ul 93, vom face testul invers, v1>v2.

```
93        aload_0
94        sipush  500
97        putfield  snow/snows  I
```

this	*500* *this*	

Prin urmare avem

```
if (snows > 500) snows = 500;
```

```
100       aload_0
101       ldc  "threadsleep"
103       invokevirtual java/applet/Applet/getParameter  (Ljava/lang/String;)Ljava/lang/String;
106       astore_1
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus este:

this	*“threadsleep”* *this*	*this.getParameter(“threadsleep”)*	

```
s = getParameter(“threadsleep”);
```

```
107       aload_1
108       ifnonnull  120
```

s	

Vom face testul invers pentru ca execuția să continue de la offset-ul 112. Existența lui *goto* pe poziția 117 ne indică o structură *if-else*. Deci avem:

```
if (s==null){}else{}
```

```
111       aload_0
112       bipush  80
114       putfield  snow/threadSleep  I
117       goto  131
```

this	*80* *this*	

```
if (s==null){
      threadSleep = 80;
}else{}
```

```
120       aload_0
121       aload_1
122       invokestatic  java/lang/Integer/valueOf  (Ljava/lang/String;)Ljava/lang/Integer;
125       invokevirtual  java/lang/Integer/intValue  ()I
128       putfield  snow/threadSleep  I
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus este:

this	*s* *this*	*Integer.valueOf(s)* *this*	*Integer.valueOf(s).intValue()* *this*	

Și codul obținut:

```
if (s==null){
      threadSleep = 80;
}else{
      threadSleep = Integer.vaueOf(s).intValue();
}
```

```
131       aload_0
132       getfield  snow/threadSleep  I
135       bipush  10
137       if_icmpge  149
```

this	*this.threadSleep*	*10* *this.threadSleep*	

Instrucțiunea *if_icmpge 149* extrage din vârful stivei o valoare întreagă, v2 (10, în cazul nostru), apoi o altă valoare întreagă, v1 (*this.threadSleep*, în cazul nostru), efectuează testul v1>=v2, iar în caz afirmativ execuția continuă de la offset-ul 149. Prin urmare generează cod sursă. Pentru ca decompilarea să continue cu offset-ul 140, vom face testul invers, v1<v2.

```
140        aload_0
141        bipush  10
143        putfield  snow/threadSleep  I
146        goto  166
```

this	*10* *this*	

Existența lui *goto* indică o structură *if-else*. Prin urmare avem

```
if (threadSleep < 10) threadSleep = 10;
else
```

```
149        aload_0
150        getfield  snow/threadSleep  I
153        sipush  1000
156        if_icmple  166
```

this	*this.threadSleep*	*1000* *this.threadSleep*	

```
if (threadSleep > 1000)
```

```
159        aload_0
160        sipush  1000
163        putfield  snow/threadSleep  I
```

this	*1000* *this*	

```
if (threadSleep > 1000) threadSleep = 1000;
```

Deci codul final este:

```
if (threadSleep < 10) threadSleep = 10;
else if (threadSleep > 1000) threadSleep = 1000;
```

```
166       aload_0
167       aload_0
168       getfield  snow/snows  I
171       newarray  int
173       putfield  snow/snowX  [I
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus este:

this	*this* *this*	*this.snows* *this*	*new int[this.snows]* *this*	

Instrucțiunea *newarray* crează o matrice 1-dimensională de tipul specificat (*int* în cazul nostru) , având lungimea dată de o valoare întreagă extrasă din vârful stivei şi introduce în stivă referința la noua matrice. Prin urmare obținem codul:

```
snowX = new int[snows];
```

```
176       aload_0
177       aload_0
178       getfield  snow/snows  I
181       newarray  int
183       putfield  snow/snowY  [I
```

Analog ca mai sus:

```
snowY = new int[snows];
```

```
186       iconst_0
187       istore_2
188       goto  260
```

Instrucțiunea *iconst_0* încarcă în stivă constanta întreagă 0, care apoi este extrasă de *istore_2* şi stocată în variabila locală 2. Cum această variabilă este utilizată pentru prima dată, avem o declarație. Tipul ei este *int*. Prin urmare putem scrie

```
int i = 0;
```

Instrucțiunea goto 260 de la offset-ul 188 ne trimite la offset-ul 260. Cum nu este în cadrul unei structuri condiționale, ca până acum, trebuie să analizăm codul de la 260:

```
260       iload_2
261       aload_0
262       getfield  snow/snows  I
265       if_icmplt  191
```

Observăm că avem următoarea execuție:

	this	*this.snows*	
i	*i*	*i*	

Instrucțiunea *if_icmplt 191* efectuează testul *i < snows* şi în caz afirmativ execuția va continua de la offset-ul 191. Această întoarcere sugerează existența unei structuri de ciclare. Datorită lui *goto* de la 188, este vorba de o structură *while* sau o structură *for*. Poate fi utilizată oricare dintre acestea. Deoarece este implicată o variabilă întreagă, incrementată cu 1 pe poziția 257, înainte de test, preferăm structura *for*. Prin urmare, putem scrie:

```
for(int i = 0; i < snows; i++){
}
```

Trecem mai departe la blocul de cod al acestei structuri.

```
191       aload_0
192       getfield  snow/snowX  [I
195       iload_2
196       aload_0
197       getfield  snow/rand  Ljava/util/Random;
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus:

			this	*this.rand*
		i	*i*	*i*
this	*this.snowX*	*this.snowX*	*this.snowX*	*this.snowX*

```
200       invokevirtual  java/util/Random/nextInt  ()I
203       aload_0
204       getfield  snow/dim  Ljava/awt/Dimension;
207       getfield  java/awt/Dimension/width  I
210       iconst_2
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus:

this.rand.nextInt() *i* *this.snowX*	*this* *this.rand.nextInt()* *i* *this.snowX*	*this.dim* *this.rand.nextInt()* *i* *this.snowX*	*this.dim.width* *this.rand.nextInt()* *i* *this.snowX*	*2* *this.dim.width* *this.rand.nextInt()* *i* *this.snowX*

```
211         idiv
212         irem
```

this.dim.width / 2 *this.rand.nextInt()* *i* *this.snowX*	*this.rand.nextInt() % (this.dim.width / 2)* *i* *this.snowX*

```
213         aload_0
214         getfield  snow/dim  Ljava/awt/Dimension;
```

this *this.rand.nextInt() % (this.dim.width / 2)* *i* *this.snowX*	*this.dim* *this.rand.nextInt() % (this.dim.width / 2)* *i* *this.snowX*

```
217         getfield  java/awt/Dimension/width  I
220         iconst_2
```

this.dim.width *this.rand.nextInt() % (this.dim.width / 2)* *i* *this.snowX*	*2* *this.dim.width* *this.rand.nextInt() % (this.dim.width / 2)* *i* *this.snowX*

```
221       idiv
```

this.dim.width / 2
this.rand.nextInt() % (this.dim.width / 2)
i
this.snowX

```
222       iadd
223       iastore
```

this.rand.nextInt() % (this.dim.width / 2) + this.dim.width / 2	
i	
this.snowX	

Instrucțiunea *iastore* extrage din vârful stivei o valoare de tipul *int* (în cazul acesta *this.rand.nextInt() % (this.dim.width / 2) + this.dim.width / 2*), apoi un index întreg (*i*) şi apoi referința la o matrice 1-dimensională de tipul *int* (în cazul nostru *this.snowX*). Instrucțiunea încarcă pe poziția *i* a matricei valoarea extrasă din stivă, deci furnizează cod sursă:

```
this.snowX[i] = this.rand.nextInt()%(this.dim.width/2)+this.dim.width/2;
```

instrucțiune echivalentă cu

```
snowX[i] = rand.nextInt()%(dim.width/2)+dim.width/2;
```

Cum am văzut mai sus, au fost utilizate instrucțiuni JVM pentru operații aritmetice: *idiv* (pentru împărțire – operanzi de tip *int*), *irem* (pentru modulo – operanzi de tip *int*) şi *iadd* (pentru adunare – operanzi de tip *int*). De exemplu, instrucțiunea *iadd* extrage din vârful stivei o valoare de tipul *int*, v2, apoi o altă valoare de tipul *int*, v1, efectuează adunarea v1+v2 şi introduce rezultatul în stivă.

```
224       aload_0
225       getfield  snow/snowY  [I
228       iload_2
229       aload_0
230       getfield  snow/rand  Ljava/util/Random;
233       invokevirtual  java/util/Random/nextInt  ()I
236       aload_0
237       getfield  snow/dim  Ljava/awt/Dimension;
240       getfield  java/awt/Dimension/height  I
243       iconst_2
244       idiv
245       irem
246       aload_0
```

```
247       getfield  snow/dim  Ljava/awt/Dimension;
250       getfield  java/awt/Dimension/height  I
253       iconst_2
254       idiv
255       iadd
256       iastore
```

Observăm că acest cod este asemănător cu cel analizat mai sus şi furnizează următorul cod sursă:

```
snowY[i] = rand.nextInt()%(dim.height/2)+dim.height/2;
```

```
257       iinc  2  1
```

Această instrucțiune incrementează variabila locală 2 (adică pe *i*) cu 1 şi a fost cuprinsă în declarația lui *for*. Prin urmare, codul instrucțiunii *for* este:

```
for(int i = 0; i < snows; i++){
      snowX[i] = rand.nextInt()%(dim.width/2)+dim.width/2;
      snowY[i] = rand.nextInt()%(dim.height/2)+dim.height/2;
}
```

```
268       aload_0
269       ldc  "graphic"
271       invokevirtual java/applet/Applet/getParameter(Ljava/lang/String;)Ljava/lang/String;
274       astore_1
```

this	*“graphic”* *this*	*this.getParameter(“graphic”)*	

```
s = getParameter(“graphic”);
```

```
275       aload_1
276       ifnonnull  304
```

```
if(s==null){}
```

```
279       aload_0
280       ldc  "graph"
282       invokevirtual java/applet/Applet/getParameter(Ljava/lang/String;)Ljava/lang/String;
285       astore_1
```

```
if(s==null){
      s = getParameter(“graph”);
}
```

```
286        aload_1
287        ifnonnull  304
```

```
if(s==null){
      s = getParameter(“graph”);
      if(s==null){}
}
```

```
290        aload_0
291        ldc  "grph"
293        invokevirtual java/applet/Applet/getParameter(Ljava/lang/String;)Ljava/lang/String;
296        astore_1
```

```
if(s==null){
      s = getParameter(“graph”);
      if(s==null){
            s = getParameter(“grph”);
      }
}
```

```
297        aload_1
298        ifnonnull  304
```

```
if(s==null){
      s = getParameter(“graph”);
      if(s==null){
            s = getParameter(“grph”);
            if(s==null){}
      }
}
```

```
301        ldc  "tento1.gif"
303        astore_1
```

```
if(s==null){
      s = getParameter(“graph”);
      if(s==null){
            s = getParameter(“grph”);
```

```
                if(s==null){
                        s = “tento1.gif”;
                }
        }
}
```

```
304         aload_0
305         new  java/awt/MediaTracker
308         dup
309         aload_0
```

Evoluția stivei în timpul execuției instrucțiunilor de mai sus:

			this
		new MediaTracker	*new MediaTracker*
	new MediaTracker	*new MediaTracker*	*new MediaTracker*
this	*this*	*this*	*this*

```
310         invokespecial  java/awt/MediaTracker/<init>(Ljava/awt/Component;)V
313         putfield  snow/mt  Ljava/awt/MediaTracker;
```

new MediaTracker(this)	
this	

```
this.mt = new MediaTracker(this);
```

sau echivalent cu

```
mt = new MediaTracker(this);
```

```
316         aload_0
317         iconst_1
318         anewarray  java/awt/Image
321         putfield  snow/gAlc  [Ljava/awt/Image;
```

	1	*new Image[1]*	
this	*this*	*this*	

```
gAlc = new Image[1];
```

```
324        aload_0
325        getfield  snow/gAlc  [Ljava/awt/Image;
328        iconst_0
329        aload_0
330        aload_0
```

				this
			this	*this*
		0	*0*	*0*
this	*this.gAlc*	*this.gAlc*	*this.gAlc*	*this.gAlc*

```
331        invokevirtual  java/applet/Applet/getDocumentBase  ()Ljava/net/URL;
334        aload_1
335  invokevirtual java/applet/Applet/getImage(Ljava/net/URL;Ljava/lang/String;)Ljava/awt/Image;
338        aastore
```

	s	
this.getDocumentBase()	*this.getDocumentBase()*	
this	*this*	*this.getImage(this.getDocumentBase(),s)*
0	*0*	*0*
this.gAlc	*this.gAlc*	*this.gAlc*

```
gAlc[0] = this.getImage(this.getDocumentBase(),s);
```

```
339        aload_0
340        getfield  snow/mt  Ljava/awt/MediaTracker;
343        aload_0
344        getfield  snow/gAlc  [Ljava/awt/Image;
347        iconst_0
348        aaload
```

			0	
	this	*this.gAlc*	*this.gAlc*	*this.gAlc[0]*
this	*this.mt*	*this.mt*	*this.mt*	*this.mt*

```
349        iconst_0
350        invokevirtual  java/awt/MediaTracker/addImage  (Ljava/awt/Image;I)V
```

0 *this.gAlc[0]* *this.mt*	*this.mt.addImage(this.gAlc[0],0)*	

```
mt.addImage(this.gAlc[0],0);
```

```
353        aload_0
354        getfield  snow/mt  Ljava/awt/MediaTracker;
357        iconst_0
358        invokevirtual  java/awt/MediaTracker/waitForID  (I)V
361        aload_0
362        iconst_0
363        putfield  snow/stopFlag  I
366        return
```

this	*this.mt*	*0* *this.mt*	*this.mt.waitForID(0)*	

```
mt.waitForID(0);
```

this	*0* *this*	

```
stopFlag = 0;
```

```
00000973        Exception Table Count                 1
00000975        Start PC                              353
00000977        End PC                                367
00000979        Handler PC                            367
0000097b        Catch Type                            java/lang/InterruptedException
0000097d        Code's Attributes Count               1
0000097f        Attribute Name                        LineNumberTable
00000981        Bytes Count                           158
00000985        LineNumberTable Count                 39
00000987        Start PC                              0
00000989        Line Number                           24
0000098b        Start PC                              11
0000098d        Line Number                           25
0000098f        Start PC                              19
00000991        Line Number                           26
```

Din tabela excepțiilor vedem că între adresele 353 şi 367 poate fi lansată excepția *InterruptedException*. Acest lucru se reflectă în codul sursă în felul următor:

```
try{
      mt.waitForID(0);
}
catch(InterruptedException e){}
```

În concluzie, codul sursă decompilat până acum este următorul:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.image.ImageObserver;
import java.util.Random;

public class snow extends Applet implements Runnable{
      Image offScrn;
      Graphics offGrph;
      Random rand;
      int stopFlag;
      long stopTime;
      int[] snowX;
      int[] snowy;
      int snows;
      int wind;
      int threadSleep;
      Dimension dim;
      Image[] gAlc;
      MediaTracker mt;

public void init(){
      Random rand = new Random();
      dim = size();
      offScrn  =  createImage(dim.width, dim.height);
      offGrph = offScrn.getGraphics();
```

```
String s = getParameter(“snows”);
if(s==null){
      snows = 100;
}else{
      snows = Integer.vaueOf(s).intValue();
}
if (snows > 500) snows = 500;
s = getParameter(“threadsleep”);
if (s==null){
      threadSleep = 80;
}else{
      threadSleep = Integer.vaueOf(s).intValue();
}
snowX = new int[snows];
snowY = new int[snows];
for(int i = 0; i < snows; i++){
      snowX[i] = rand.nextInt()%(dim.width/2)+dim.width/2;
      snowY[i] = rand.nextInt()%(dim.height/2)+dim.height/2;
}
s = getParameter(“graphic”);
if(s==null){
      s = getParameter(“graph”);
      if(s==null){
            s = getParameter(“grph”);
            if(s==null) s = “tento1.gif”;
      }
}
mt = new MediaTracker(this);
gAlc = new Image[1];
gAlc[0] = this.getImage(this.getDocumentBase(),s);
mt.addImage(this.gAlc[0],0);
try{
      mt.waitForID(0);
}
catch(InterruptedException e){}
stopFlag = 0;
```

```
}
}
```

Codul dezasamblat este mult mai mare, dar informațiile nefolositoare au fost şterse. Lăsăm ca temă cititorului decompilarea celorlalte 7 metode din această clasă. Exemplul expus mai sus este unul simplu, iar decompilarea s-a desfăşurat într-o manieră secvențială. Există însă situații care presupun o analiză semantică a codului de octeți. La nivelul limbajului Java, valorile de adevăr *true* şi *false* nu pot fi substituite prin 1 şi 0 ca în alte limbaje, însă, la nivelul codului de octeți, *true* şi *false* sunt reprezentate de numerele 1 şi 0 de tipul *int*. Acest lucru generează dificultăți în decompilare, unde tipul variabilei trebuie dedus printr-o analiza semantică şi în loc de *if(i==1)* să putem scrie *if(b)*, unde *b* este o variabilă booleană. Propunem cititorului să scrie o clasă simplă, cu o singură metodă care să conțină o instrucțiune de forma if((a1 && a2 && … && an) || (b1 && b2 && … && bn) || …. || (z1 && z2 && … && zn)), unde a1, a2, …, b1, b2,…, etc, sunt toate variabile booleane. Să compileze clasa, iar apoi să decompileze codul de octeți obținut.

Cum am văzut, procesul de decompilare nu este simplu, dar este posibil de realizat. Toate aceste operații, făcute "manual" mai sus, sunt realizate instantaneu de către un decompilator. Creatorul decompilatorului trebuie să țină seama însă de toate situațiile ce pot să apară într-un cod de octeți. Decompilarea este "completă" atunci când, codul sursă obținut poate fi recompilat fără modificări, obținându-se o clasă care funcționează identic cu cea originală. De la acest punct se poate trece mai departe la studiul codului decompilat, se poate identifica uşor funcționalitatea anumitor segmente de cod prin comentare, recompilare şi execuție.

Protecția programelor împotriva ingineriei inverse

Cum am văzut în cursurile precedente, una dintre formele de atac asupra programelor Java (şi cea mai importantă în ce priveşte consecinţele) o constitue decompilarea. De exemplu, prin decompilare pot fi plasate coduri ilegale în programele publicate în Web, acestea fiind apoi recompilate şi substituite cu cele originale.

Din nefericire, nu există metode sigure de protecţie împotriva decompilării. Trebuie însă menționat faptul că procesul de decompilare este foarte dificil de realizat şi prima protecţie este aceasta. În cazul programelor Java, datorită utilizării codului de octeți, decompilarea este posibilă. Este un proces complex, dar mult mai uşor de realizat comparativ cu decompilarea codului maşină generat de un compilator C++, de exemplu.

O metodă de protecţie împotriva decompilării o constitue "mascarea codului", termenul consacrat fiind "obfuscation". Această metodă constă în transformarea codului de octeți prin redenumirea identificatorilor pachetelor, claselor, atributelor şi metodelor, înlăturarea informațiilor de depanare, criptarea valorilor literale de tip String, rearanjarea intrărilor din tabela constantelor, etc. Toate acestea au drept scop "bruiajul" procesului de decompilare; decompilatoarele sunt "păcălite" şi nu pot furniza un cod sursă coerent şi corect, ceea ce face ca rezultatul decompilării să fie inutilizabil. Codul de octeți rezultat în urma procesului de mascare trebuie să fie corect din punct de vedere sintactic şi semantic, să poată fi executat şi să aibă o execuție identică cu a codului original.

În continuare prezentăm câteva metode de masacre "agresivă" a codului claselor, constând în modificări ale codului de octeți al metodelor. Ca şi în exemplul de decompilare, vom folosi programul JCD (Java Class Disassembler) pentru a dezasambla codul claselor şi identificarea poziției în codul de octeți unde vom face anumite modificări. Spre deosebire de alte dezasambloare Java, JCD furnizează posibilitatea urmăririi în paralel a codului de octeți şi a codului dezasamblat. În cazul exemplelor noastre, modificările pot

fi realizate efectiv cu un editor HEX, iar clasele rezultate pot fi testate prin decompilare cu JAD. Acesta este cel mai rapid şi performant decompilator Java existent la ora actuală. Un alt decompilator Java este programul JCD (Java Class Decompiler), un program scris în limbajul HPVee de către autor.

Pentru o tratare unitară, metodele prezentate mai jos vor fi incluse într-un algoritm.

Mascare de nivel inferior

Prin mascare de nivel inferior înțelegem modificarea fişierelor clasă prin redenumirea identificatorilor, înlăturarea informației de depanare şi criptarea valorilor literale de tip String.

Fişierele clasă au în general o cantitate mare de informație care nu foloseşte în timpul execuției, ci doar în procesul de depanare a programului respectiv. Aceste informații sunt conținute în atributele *LineNumberTable* şi *LocalVariableTable*, ambele fiind atribute ale atributului *Code* (atributul principal al fiecărei metode). Atributul *LocalVariableTable* furnizează numele, tipul şi domeniul fiecărei variabile locale, definită în cadrul unei metode şi toate aceste informații pot fi utilizate în procesul de decompilare pentru a obține un cod sursă cât mai apropiat de cel original. Înlăturarea acestor informații nu afectează validitatea codului de octeți, iar clasele obținute au dimensiuni mai mici. Acest lucru poate fi realizat cu următorul algoritm:

```
for(int k = methods_count; k >= 1; k--){
   goto method[k].Code_ Attribute;
   int length = Code_Attribute.Bytes_Count;
   length -= 8 + Code_Count + 2;
   length -= Exception_Table_Length*8 + 2;
   goto Code's_Attributes;
   delete length bytes;
   Code's_Attributes_Count = 0;
   Code_Attribute.Bytes_Count - = length;
}
```

Următorul pas constă în redenumirea identificatorilor. În acest scop, tabela constantelor fiecărei clase (Constant_Pool) este scanată şi sunt identificate intrările de tipul CONSTANT_Fieldref şi CONSTANT_Methodref, apoi intrările

CONSTANT_NameAndType referite de acestea şi apoi intrarile CONSTANT_Utf8 referite ce cele din urmă. Algoritmul redenumeşte stringurile ce reprezintă identificatori după o regulă de forma: a, b,…, aa, ab, …, ba, bb,… Pot fi folosite de asemenea şi alte combinații, cum ar fi: aA, aB, etc.

Prin intermediul unui editor HEX oricine poate modifica valorile literale de tip String dintr-o clasă Java. Mai mult, modificând în constanta respectivă valoarea câmpului "length", se poate schimba şi lungimea stringului respectiv, lucru ce nu poate fi realizat în cazul codului maşină, unde un string identificat poate fi schimbat doar cu un alt string de aceeaşi lungime. Din aceste considerente, criptarea acestor stringuri este imperativă. Realizarea acestor modificări, la nivelul unei clase compilate, nu este trivială. Evident că, în urma criptării stringurile nu mai pot fi identificate şi deci nu pot fi modificate. Pentru a funcționa însă corect clasa respectivă, stringurile criptate trebuiesc decriptate în timpul rulării. Prin urmare, clasa respectivă trebuie să conțină metoda de decriptare. Ori inserția unei metode noi în codul compilat este un lucru dificil întrucât presupune foarte multe modificări la nivelul tabelei constantelor şi nu numai. Pentru a minimiza numărul de instrucțiuni inserate, metoda de decriptare se va insera ca ultima metodă, altfel numărul de modificări este foarte mare. Evident că, orice referire la o constantă Utf8, trebuie substituită cu o combinație de instrucțiuni ce include apelul metodei de decriptare (care primeşte ca parametru constanta Utf8 a cărei valoare a fost criptată). Din aceste considerente, algoritmul de inserție şi criptare trebuie să respecte următorii paşi:

1. Este creat un string aleatoriu cu o lungime minimă precizată (pot fi utilizate şi caractere speciale sau caractere Unicode). Acest string va fi cheia folosită de metodele de criptare şi decriptare.
2. Se redenumeşte metoda de decriptare în concordanță cu algoritmul de redenumire a identificatorilor. Să presupunem ca funcția de decriptare este numită *dd*.
3. Se caută în tabela constantelor intrările de tipul *CONSTANT_String* şi se criptează conținutul intrărilor *CONSTANT_Utf8* adresate de acestea.
4. Sunt efectuate următoarele schimbări în codul de octeți:
   - Se incrementează corespunzător câmpul *Constant_Pool_Count*,

- Se scriu în tabela constantelor toate mesajele criptate şi se incrementează / decrementează *CONSTANT_Utf8.length* dacă este necesar,
- Se adaugă la tabela constantelor intrările suplimentare cu numele şi semnătura metodei de decriptare, numele şi semnăturile metodelor invocate de aceasta, valorile literale folosite,
- Atributul Code al fiecărei metode care conține un mesaj criptat trebuie să suporte următoarele modificări: *Bytes_Count* şi *Code_Count* sunt incrementate cu 3*N (unde N este numărul de mesaje criptate ale metodei), *Max_Stack* este incrementat cu 1 şi, pentru fiecare mesaj criptat, o instrucțiune *invokestatic* (care apelează metoda *dd*, iar parametrul ei este un index la tabela constantelor, la o intrare de tip *CONSTANT_Methodref* inserată de asemenea) este inserată dupa codul instrucțiunii *ldc*, care încarcă în stivă referința la mesajul criptat,
- Se adaugă codul de octeți al metodei *dd*, după ultima metodă şi înainte de atributele clasei,
- Se incrementează câmpul *Methods_Count* cu 1.

Să presupunem în continuare că stringul aleatoriu este:

```
String key = ".\rT\3752>l:h\u1212";
```

şi presupunem de asemenea că metoda de criptare este:

```
String encrypt (String s) {
  char[] ac = s.toCharArray();
  char[] ac1 =key.toCharArray();
  for(int k = 0; k < ac.length; k++)
    ac[k] ^= ac1[k % ac1.length];
  return new String(ac);
}
```

De exemplu, dacă aplicăm metoda *encrypt* stringului “Java is a portable language”, obținem următorul mesaj criptat:

```
dl\"\234\022W\037\032\t\u1232^b&\211S\\\000_H\u127eOc3\210SY\t
```

Acum, dacă aplicăm aceeaşi metodă stringului criptat, obținem stringul inițial. Prin urmare funcția *encrypt* este idempotentă. Acesta este doar un exemplu. Nu este obligatoriu ca funcția de criptare să coincidă cu cea de decriptare.

Schimbăm acum numele metodei cu *dd* şi trebuie să inserăm codul următor în codul de octeți al clasei:

```
  static String dd (String s) {
    char[] ac = s.toCharArray();
    char[] ac1 = null;
    ac1 =".\rT\3752>l:h\u1212".toCharArray();
    for(int k = 0; k < ac.length; k++)
      ac[k] ^= ac1[k % ac1.length];
    return new String(ac);
  }
```

Vor fi efectuate următoarele operații:

```
Let be n = Constant_Pool_Count and u, v, w, x, y are indexes at
Constant_Pool such that:
CONSTANT_Class   Entry (u) (..)
(reference at this class)
CONSTANT_Utf8    Entry (v) <init>
CONSTANT_Utf8    Entry (w) Code
CONSTANT_Class   Entry (x) (y)
CONSTANT_Utf8    Entry (y) java/lang/String
We look for u, v, w, x into Constant_Pool.
  if (x == 0) {x = n; y = x + 1; n++; flag = true;}
Add at Constant_Pool the following entries:

if(flag) {
Constant_Pool_Count += 15;
CONSTANT_Class   Entry (x) (x+1)
bytes: 07 (short)(x+1)
(reference at String Class)

CONSTANT_Utf8    Entry (x+1) java/lang/String
bytes:     01  0010  6A61  7661  2F6C  616E  672F  5374  7269  6E67
(fully qualified form of String Class)

} else Constant_Pool_Count += 13;

CONSTANT_Methodref    Entry (n) Class (u) Name/Type (n+1)
bytes: 0A (short)u (short)(n+1)
(reference at dd method from this Class)

CONSTANT_NameAndType   Entry (n+1) Name (n+2) Type (n+3)
bytes: 0C (short)(n+2) (short)(n+3)
(name and type of dd method)

CONSTANT_Utf8    Entry (n+2) dd
bytes: 0100  0264  64
(decrypting method's name)

CONSTANT_Utf8    Entry (n+3) (Ljava/lang/String;)Ljava/lang/String;
```

```
bytes:     01  0026  284C  6A61  7661  2F6C  616E  672F  5374  7269  6E67
3B29  4C6A  6176  612F  6C61  6E67  2F53  7472  696E  673B  0100  0A53
6F75  7263  6546  696C  65
(decrypting method's descriptor)

CONSTANT_Methodref          Entry (n+4) Class (x) Name/Type (n+5)
bytes: 0A (short)x (short)(n+5)
(reference at toCharArray from String Class)

CONSTANT_NameAndType   Entry (n+5) Name (n+6) Type (n+7)
bytes: 0C (short)(n+6) (short)(n+7)
(name and type of toCharArray method)

CONSTANT_Utf8    Entry (n+6) toCharArray
bytes: 01  000B  746F  4368  6172  4172  7261  79
(toCharArray name)

CONSTANT_Utf8    Entry (n+7) ()[C
bytes: 01  0004  2829  5B43
(toCharArray descriptor)

CONSTANT_String  Entry (n+8) (n+9)
bytes: 08  (short)(n+9)
(reference at the encrypting key)

CONSTANT_Utf8 Entry (n+9) .\rT\3752>l:h\u1212
bytes: 01  000D  2E0D  54C3  BD32  3E6C  3A68   E188  92
(encrypting key)

CONSTANT_Methodref        Entry (n+10) Class (x) Name/Type (n+11)
bytes: 0A (short)x (short)(n+11)
(reference at the String(char[]) constructor)

CONSTANT_NameAndType   Entry (n+11) Name (b) Type (n+12)
bytes: 0C (short)v (short)(n+12)
(name and type of String(char[]) constructor)

CONSTANT_Utf8    Entry (n+12) ([C)V
bytes: 0100  0528  5B43  2956
(descriptor of String(char[]) constructor)

  for(int k = 1; k <= methods_count; k++){
    goto method[k].Code_ Attribute;
    for each ldc instruction which refers a
    CONSTANT_String Entry do
      Write after ldc code:  B8 (short)n;
  }
```

Codul metodei *dd*, inserată după ultima metodă, este:

```
0008     (short)(n+2)     (short)(n+3)     0001 (short)w 0000     003D
0006    0004    0000    0031    2AB6    (short)(n+4)    4C01    4D12
(byte)(n+8)    B6    (short)(n+4)    4D03    3EA7    0013    2B1D    5C34
```

```
2C1D     2CBE     7034     8292     5584      0301     1D2B     BEA1     FFED
BB    (short)x    59    2BB7    (short)(n+10)    B000    0000    00
```

Deşi criptarea valorilor literale de tip *String* este o măsură de protecție împotriva editoarelor HEX, ea nu oferă prea multă protecție împotriva procesului de decompilare. Aceasta deoarece metoda de decriptare trebuie să se găsească în codul mascat şi atunci, prin decompilare putem afla codul acestei metode, putem să o folosim într-un alt program pentru a scana sursa decompiată, identificarea stringurilor criptate, decriptarea acestora, substituirea stringurilor criptate cu cele decriptate, eliminarea metodei de decriptare. Un astfel de program este *JDecode*, scris de către autor. Mai mult, există decompilatoare (de exemplu JAD) care fac acest lucru automat, în timpul procesului de decompilare. În concluzie, metodele expuse sunt un impediment doar pentru cei care nu ştiu prea multe despre structura formatului class şi decompilarea acestuia. Cineva suficient de motivat poate trece uşor peste acestea. Din aceste motive sunt necesare alte metode de mascare, mai eficiente.

Mascare de nivel superior

Prin mascarea de nivel superior înțelegem modificarea codului metodelor prin inserția de “cod de bruiaj” sau prin înlocuirea instrucțiunilor cu altele, echivalente din punct de vedere semantic, dar care schimbă structurile standard, confuzionând decompilatoarele. Algoritmul prin care se poate realiza aceasta are următoarea structură generală:

```
for(int k = 1; k <= methods_count; k++){
   int i = 0, j = 0, tag = 0;
   int maxs = method[k].Max_Stack;
   int maxl = method[k].Max_Locals;
   boolean flag, flag1, flag2;
   int count = method[k].Code_Count;
   int[] lengths = new int[count];
   int offset;
   while(i < count){
      tag = Byte[i];
      lengths[j] = getInstructionLength(tag);
      i += lengths[j]; j++;
   }
   while(i >= 0){
      i - = lengths[j]; j--;
```

```
tag = Byte[i];
if instruction[j] is a control flow's
instruction and branchoffset > 0,
increments this branchoffset with the
count of the inserted bytes between i and
i+branchoffset;
// goto instruction
if(tag == 167 && branchoffset > 0){
  //this block replaces goto instruction
  //with a sequence of the type:
  //    iload_(maxl-1)
  //    ifeq (branchoffset+..)
  if(!flag1){flag1 = true; maxs++;
  maxl++;}
  branchoffset += the count of the
  inserted bytes between i+3 and
  i+3+branchoffset;
  int n = flag2?maxl-2:maxl-1, h = 1;
  if(n > 4){
    inserts two bytes at i+3; count+=2;
    Byte[i] = 21;  // iload n
    Byte[i+1] = n; h = 2;
  }else{
    inserts one byte at i+3; count++;
    Byte[i] = 26+n;  // iload_n
  }
  Byte[i+h] = 153;  // ifeq branchoffset
  write branchoffset into Byte[i+h+1]
  and Byte[i+h+2];
  continue;
}
// if instructions
if(tag is between 153 and 166 || tag == 198 || tag == 199){
  if(branchoffset > 0){
  //this block inserts a sequence of the
  //following type before the current
  //instruction:
  //    iload_(maxl-1)
  //    ifne (with a branchoffset > 0)
  if(!flag){flag = true; offset = i;
  continue;}
  if(!flag1){flag1 = true; maxs++;
  maxl++;}
  int n = flag2?maxl-2:maxl-1, h = 1;
  branchoffset = offset - i;
  if(n > 4){
    inserts 5 bytes at i; count += 5;
    Byte[i] = 21;     // iload n
    Byte[i+1] = n;
    h = 2; offset = i-5;
  }else{
    inserts 4 bytes at i; count += 4;
    Byte[i] = 26+n;  // iload_n
```

```
          offset = i-4;
        }
        Byte[i+h] = 154;  // ifne branchoffset
        write branchoffset into Byte[i+h+1]
        and Byte[i+h+2];
        continue;
        }else{
        //this block is similar with above
        //blocks and inserts a sequence of the
        //following type after the current
        //instruction:
        //    iload_(maxl-1)
        //    ifne offset
        //where branchoffset < offset < -8
        }
      }
      // tableswitch and lookupswitch
      if(tag == 170 || tag == 171){
        if(!flag1){flag1 = true; maxs++;
        maxl++;}
        if(!flag2){flag2 = true; maxs++;
        maxl++;}
        //this block is similar with the above
        //blocks
        //it inserts before the current
        //instruction, between istore and iload,
        //a sequence of the type:
        //    iload_(maxl-2)
        //    iload_(maxl-1)
        //    ifne (branchoffset=7)
        //    ifne (branchoffset at an instruction
        //placed after lookupswitch)
        //inserts between iload and
        //lookupswitch a sequence of the type:
        //    iload_(maxl-1)
        //    ifne (branchoffset at an instruction
        //placed after lookupswitch)
      }
    }
    for each instruction with a negative
    branchoffset, writes the actual branchoffse;
    Max_Stack = maxs; Max_Locals = maxl;
    Bytes_Count += count - Code_Count
    Code_Count = count;
  }
```

Pentru exemplificare vom efectua câteva teste concrete. Acestea sunt foarte simple, dar cum vom vedea, sunt suficiente pentru a "bruia" decompilatorul JAD. Pot fi efectuate foarte multe modificări de acest gen, totul depinzând de imaginaţia celor care scriu programele de mascare. Mai mult, pot fi introduse atribute noi în codul de octeţi, atribute cunoscute doar de programul de mascare şi care să conţină cheia de de-mascare.
Toate codurile următoare au fost dezasamblate şi mascate cu ajutorul programului JCD şi decompilate apoi cu JAD.

for/while Obfuscation

Considerăm metoda:

```
static void a(){
   for(int i=0; i< 100;i++)
          System.out.println(i);
}
```

Prin mascare obţinem următorul cod:

```
0     iconst_0
1     istore_1
2     iconst_0
3     istore_0
4     iload_1              // goto substitute
5     ifeq 18
8     getstatic java/lang/System/out Ljava/io/PrintStream;
11    iload_0
12    invokevirtual java/io/PrintStream/println (I)V
15    iinc 0 1
18    iload_0
19    bipush 100
21    if_icmplt 8
24    iload_1              // code of confusion
25    ifne 15
28    return
```

Instrucţiunile inserate şi înlocuite sunt scrise cu bold-italics.
În urma decompilării s-a obţinut:

```
static void a()
  {
      int i;
      int j;
      j = a;
      i = 0;
```

```
        if(j == 0) goto _L2; else goto _L1
_L1:
        System.out.println(i);
_L4:
        i++;
_L2:
        if(i < 100)
            continue; /* Loop/switch isn't completed */
        if(j == 0)
            return;
        if(true) goto _L4; else goto _L3
_L3:
        if(true) goto _L1; else goto _L5
_L5:
    }
```

if Obfuscation

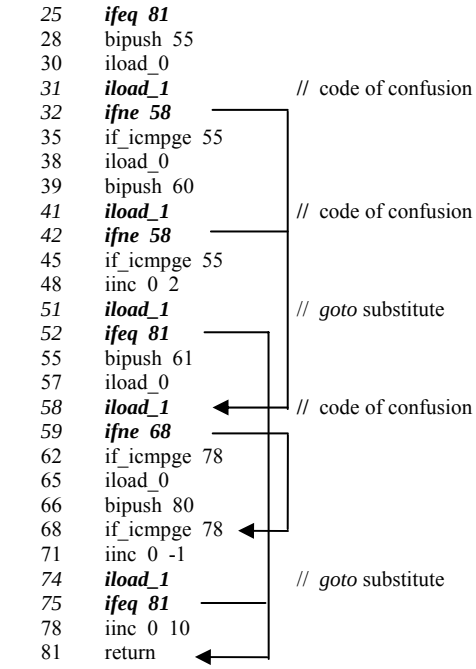
Prin mascarea codului metodei

```
static void a(){
  int i = (int)(Math.random()*100);
  if(i<50) i++;
  else if(55<i && i<60) i+=2;
  else if(61<i && i<80) i--;
  else i+=10;
}
```

obținem următorul cod de octeți:

0000011c	Access Flags	ACC_STATIC
0000011e	Name	a
00000120	Type	()V
00000122	Attributes Count	1
00000124	Attribute Name	Code
00000126	Bytes Count	73+23
0000012a	Max Stack	4+1
0000012c	Max Locals	1+1
0000012e	Code Count	61+23

```
 0    iconst_0
 1    istore_1
 2    invokestatic  java/lang/Math/random  ()D
 5    ldc2_w  100.0D
 8    dmul
 9    d2i
10    istore_0
11    iload_0
12    bipush  50
14    iload_1              //  code of confusion
15    ifne  31
18    if_icmpge  28
21    iinc  0  1
24    iload_1              //  goto substitute
```



00000186	Exception Table Count	0
00000188	Code's Attributes Count	0

În urma decompilării cu decompilatorul JAD obţinem:

```
static void a()
    {
        int i;
        int j;
        j = a;
        i = (int)(Math.random() * 100D);
        i;
        50;
        if(j != 0) goto _L2; else goto _L1
_L1:
        JVM INSTR icmpge 30;
           goto _L3 _L4
_L3:
        break MISSING_BLOCK_LABEL_23;
_L4:
        break MISSING_BLOCK_LABEL_30;
        i++;
        if(j == 0)
            break MISSING_BLOCK_LABEL_83;
        55;
        i;
_L2:
        if(j != 0) goto _L6; else goto _L5
_L5:
        JVM INSTR icmpge 57;
           goto _L7 _L8
```

```
_L7:
        i;
        60;
        if(j != 0) goto _L6; else goto _L9
_L9:
        JVM INSTR icmpge 57;
           goto _L10 _L8
_L10:
        i += 2;
        if(j == 0)
            break MISSING_BLOCK_LABEL_83;
_L8:
        61;
        i;
_L6:
        if(j != 0) goto _L12; else goto _L11
_L11:
        JVM INSTR icmpge 80;
           goto _L13 _L14
_L13:
        break MISSING_BLOCK_LABEL_67;
_L14:
        break MISSING_BLOCK_LABEL_80;
        i;
        80;
_L12:
        JVM INSTR icmpge 80;
           goto _L15 _L16
_L15:
        break MISSING_BLOCK_LABEL_73;
_L16:
        break MISSING_BLOCK_LABEL_80;
        i--;
        if(j == 0)
            break MISSING_BLOCK_LABEL_83;
        i += 10;
    }
```

switch Obfuscation

Considerăm metoda:

```
  static void b(){
    int i = (int)(Math.random()*100);
    switch(i){
      case 1: i += 2; break;
      case 3: i--; break;
      default: i += 10;
    }
  }
```

Prin mascare obținem următorul cod de octeți:

```
00000132    Access Flags                ACC_STATIC
00000134    Name                        a
00000136    Type                        ()V
00000138    Attributes Count            1
0000013a    Attribute Name              Code
0000013c    Bytes Count                 90
00000140    Max Stack                   6
00000142    Max Locals                  3
00000144    Code Count                  78

    0    iconst_0
    1    istore_2
    2    iconst_0
    3    istore_1
    4    invokestatic  java/lang/Math/random  ()D
    7     ldc2_w  100.0D
   10    dmul
   11    d2i
   12    istore_0
   13    iload_1                  //  code of confusion
   14    iload_2
   15    ifne  22
   18    ifne  59
   21    iload_0
   22    iload_2
   23    ifne  56
   26    lookupswitch
   28    Default = 70
   32    Pairs Count = 2
   36    Key = 1,  Offset = 52
   44    Key = 3,  Offset = 63
   52    iinc  0  2
   55    iload_1                  //  goto substitute
   56    ifeq  73
   59    iload_2                  //  code of confusion
   60    ifeq  73
   63    iinc  0  -1
   66    iload_1                  //  goto substitute
   67    ifeq  73
   70    iinc  0  10
   73    return

00000196    Exception Table Count       0
00000198    Code's Attributes Count     0
```

În urma decompilării obținem:

```
static void a()
    {
        int i;
        int j;
        int k;
        k = b;
        j = a;
        i = (int)(Math.random() * 100D);
        j;
        if(k != 0) goto _L2; else goto _L1
_L1:
        if(j != 0) goto _L4; else goto _L3
```

```
_L3:
        i;
_L2:
        if(k != 0) goto _L6; else goto _L5
_L5:
        JVM INSTR lookupswitch 2: default 74
    //                   1: 56
    //                   3: 67;
           goto _L7 _L8 _L9
_L8:
        i += 2;
        j;
_L6:
        JVM INSTR ifeq 77;
           goto _L4 _L10
_L10:
        break MISSING_BLOCK_LABEL_77;
_L4:
        if(k == 0)
            break MISSING_BLOCK_LABEL_77;
_L9:
        i--;
        if(j == 0)
            break MISSING_BLOCK_LABEL_77;
_L7:
        i += 10;
    }
```

În modificările de mai sus instrucțiunea *goto* a fost înlocuită cu instrucțiunea *if*. Combinații mai complicate pot fi construite dacă pentru salturi utilizăm instrucțiunile *tableswitch* şi *lookupswitch*, care pot "bruia" şi mai tare decompilatoarele.

Cum am văzut în toate aceste exemple, rezultatul decompilării diferă mult de codul sursă original şi este dificil de citit şi de înțeles. La un cod de octeți mare, aceste modificări pot genera surse decompilate inutilizabile.

Bibliografie

[1] James Gosling, Bill Joy, and Guy Steele, *The Java™ Language Specification*, Addison Wesley Longman, Inc. 1996.

[2] Pavel Kouznetsov, *JAD*, http://kpdus.tripod.com/jad.html

[3] Tim Lindholm, and Frank Yellin, *The Java™ Virtual Machine Specification*, Second Edition, Sun Microsystems, Inc. 1999.

[4] Godfrey Nolan, *Decompiling Java*, The McGraw - Hill Companies, Inc. 1998.

[5] Gregory Wroblewski, "*General Method of Program Code Obfuscation*", Proceedings of the International Conference on Software Engineering Research and Practice (SERP) 2002, Las Vegas, USA, June 2002, pp. 153-159.