DIPLOMATERVEZÉSI FELADAT

szigorló informatikus mérnök hallgató részére

Kódvisszafejtés .NET Platformon

A .NET Framework már évek óta elérhető, és rengeteg szakember, fejlesztő használja. Minden újonnan bevezetett architektúra vagy futtató környezet esetében rendkívüli fontossággal bír a biztonság kérdésének elemzése. A biztonság számos összetevőn alapul, azonban a keretrendszer esetében elmondható, hogy az általa futtatott állományok, szerelvények biztonsági szintje tekinthető az egyik leghangsúlyosabb kérdésnek. Erre a kérdésre a visszafejthetőség vizsgálatával kaphatunk átfogó választ. Az eddig még kevesek által mélyen érintett témával mindenképp érdemes foglalkozni, hiszen az ilyen motorban rejlő lehetőségek kiaknázására - talán meglepő módon - számos informatikai szegmensben lehet igény.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be a .NET Framework működését
- Elemezze röviden a keretrendszer által használt szerelvényeket
- Térjen ki a visszafejthetőség kérdésére, annak megvalósíthatóságára
- Elemezze milyen lépések szükségesek a visszafejtéshez
- Tárgyalja a visszafejthetőség következményeit, vázolja a benne rejlő lehetőségeket
- Mutasson be egy példaalkalmazást, amely visszafejtésre képes

Konzulens:	**Albert István**
Záróvizsgatárgyak:	**Adatbázisok (Katona Gyula)** **Mesterséges intelligencia (dr. Dobrowiecki Tadeusz)** **Szoftver technikák (Benedek Zoltán, Dr. Charaf Hassan)**

Budapesti Műszaki- és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatika Tanszék

KÓDVISSZAFEJTÉS .NET PLATFORMON

KONZULENS

Albert István

BUDAPEST, 2009

HALLGATÓI NYILATKOZAT

Alulírott , szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök, stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Tudomásul veszem, hogy az elkészült diplomatervben található eredményeket a Budapesti Műszaki- és Gazdaságtudományi Egyetem, a feladatot kiíró egyetemi intézmény saját céljaimra felhasználhatja.

Kelt: Budapest, 2005. 05. 18.

.....

Tartalomjegyzék

Összefoglaló

Már több mint 3 éve megkezdődött a .NET keretrendszer terjedése. Rengeteg fejlesztő tér át a régi, sok vesződséggel járó fejlesztési módszerekről a kényelmes, egyszerű, korszerű módszerre. Az bizonyított, hogy munkájuk hatékonysága valóban jelentősen megnő, azonban nem biztos, hogy mindenki tisztában van ennek veszélyeivel.

Annak ellenére, hogy valaki nagy rutinnal dolgozik a .NET keretrendszerrel, meglehet, nincs tisztában azzal az árral, amit a kényelméért fizetnie kell. Kutatásommal nem a keretrendszert bírálom, csupán szeretnék rámutatni, hogy amíg a felügyeletlen gépi kódot rendkívül nehéz magas szintű nyelvre visszafejteni, addig ez „az új világban", a felügyelt kódokkal korántsem olyan lehetetlen feladat. Dolgozatomban ismertetni fogom a visszafejtés buktatóit, az általam használt visszafejtési algoritmusokat, a visszafejthetőség előnyeit és hátrányait.

A Microsoft (és más cégek) mérnökei felismerve a feltörhetőség problémáját, próbáltak hatékony megoldásokat kifejleszteni a visszafejthetőség megnehezítésére. Dolgozatomban elemzem a módszerek sikerét, gyenge pontjait, kijátszhatóságukat, és természetesen a visszafejtés ellenszereit.

Munkámban beletekintek a keretrendszer végrehajtó motorjának belső működésébe is, a felügyelt kódot tartalmazó futtatható állomány felépítésébe, valamit, ami talán egy fejlesztő számára a legfontosabb: vajon lehetséges-e úgy módosítani egy állományt, hogy annak viselkedése megváltozzon, azonban a változtatást a keretrendszer ne észlelje.

Kutatás folytatásában még rengeteg lehetőség rejlik. Néhány terület, ahol felhasználható lehet egy kódvisszafejtésre képes szoftver: aspektus orientált programozás, önfejlesztő szoftverek, kódzagyváló algoritmusok, mesterséges intelligencia, elosztott tudásbázis építés.

Abstract

That was more than 3 years ago, when the first version of the .NET Framework was published. There are a lot of developers who switched from the old, less comfortable developing methods to the new, faster and more effective ones. Proven that their work is more efficient, but it is not surely true, that everyone knows the dangers of this environment.

Nevertheless the programmer has many years of experience, it is possible, that he/she doesn't realize the price, he/she must pay for his/her extra comfort. The goal of my research is not to judge the Framework. I only wish to point out the major difference between decompilation of the x86 machine code and MSIL managed code. Unfortunately it is much easier to get a high level source code of the managed code. In my paper I am planning to describe the decompilation steps, the algorithms I used, the advantages and drawbacks of the "easy-decompilable" architecture.

The engineers of Microsoft (and other companies) recognized the problem of cracking, so tried to develop a solution to make harder the decompilation process. My aim is to analyze the success, the weak spots, the cheats and the reverse engineering of these solutions.

I am going to study the inner working mechanisms of the Framework, the structure of the managed assemblies. I try to answer the probably most important question: is that possible to modify a compiled assembly in any way to change its behavior bypassing the .NET Framework's security verifications?

There are so many future possibilities in this research. Some of these research fields are aspect oriented programming, self-developing software, obfuscation algorithms, artificial intelligence, shared knowledgebase development.

Irodalomjegyzék

[1] Common Language Infrastructure - ECMA 335 (http://www.ecma-international.org)

[2] Mike Perry - Introduction to Reverse Engineering Software (http://www.acm.uiuc.edu)

[3] Serge Lidin - Inside Microsoft .NET IL Assembler (MSPress, 2002, ISBN 0-7356-1547-0)

[4] Emmanuel Schanzer – Performance Tips and Tricks in .NET Applications (Microsoft, 2001)

[5] Robert Morgan - Building an optimizing compiler (Digital Press, 1998, ISBN 1-5555-8179)

[6] Thomas Lengauer and Robert Endre Tar Jan - A Fast Algorithm for Finding Dominators in a Flowgraph (ACM Press, 1979, ISSN 0164-0925, Stanford University)

[7] Jan Gray – Writing Faster Managed Code: Know What Things Cost (Microsoft, 2003)

[8] Andrew Kennedy and Don Syme – Design and Implementation of Generics for the .NET Common Language Runtime (Microsoft Research, 2001, Cambridge, UK)

[9] Matthew MacDonald - New Language Features in C# 2.0, Part 1 (OnDotNet.com, http://www.ondotnet.com, 2004)

[10] Secure Coding Guidelines for the .NET Framework (MSDN)

[11] .NET Framework Developer's Guide - Assembly Security Considerations (MSDN)

[12] .NET Framework Developer's Guide - Delay Signing an Assembly (MSDN)

[13] .NET Framework Developer's Guide - Strong Name Scenario (MSDN)

[14] .NET Framework Developer's Guide - Strong-Named Assemblies (MSDN)

[15] .NET Framework Developer's Guide - File Signing Tool (Signcode.exe) (MSDN)

[16] Raviraj – Aspect Oriented Programming (CSharpCorner.com, http://www.c-sharpcorner.com/Code/2002/Nov/aop.asp, 2002)

[17] Különböző dokumentumok a CLR-ről (http://www.developer.hu)

1 Bevezetés - a biztonság illúziója

Manapság rengeteget beszélünk a biztonságról. Nap, mint nap hallhatunk az újabb hacker támadásokról vagy a crackerek újabb programtöréseiről, és szinte tényleg nincs olyan értékes szoftver, amihez ne születne törés a kiadása után maximum egy hónappal.

A kérdés csak az: minek tudható ez be?

Már jó ideje elérhető a Microsoft új generációs, szabványos keretrendszere a .NET keretrendszer. Ezzel a fejlesztők jelentősen gyorsabban és kényelmesebben, tehát hatékonyabban tudnak szoftvert fejleszteni. A keretrendszer szabványossága jóvoltából tetszőleges platformon futtatható állományok készíthetők, áthidalva ezzel a napjainkban oly sok gondot jelentő Unix – Windows - MacOS és egyéb operációs rendszerek közötti határokat. A sok előnyt látva rengeteg programozó tér át naponta az új keretrendszerre való fejlesztésre.

Valóban jobb lesz az „új világ”, a .NET keretrendszer világa?

Mennyire gondoltak a feltörhetőségre a készítők?

Mit jelent ez a fejlesztőknek?

És még rengeteg kérdés merülhet fel mindenkiben, aki használni szeretné a Microsoft szabványos keretrendszerét. A válaszok korántsem megnyugtatóak. Egyelőre nincs biztos megoldás a teljes védelemre. Ahogy igaz az, hogy nincs támadhatatlan hálózat, ugyanúgy igaz, hogy nincs támadhatatlan szoftver sem. Egy részleges megoldás létezik, mint ahogy az a hálózatok kellő védelemmel való ellátásánál is: tegyük annyira költségessé a feltörést, hogy ne érje meg belefogni.

De vajon mennyire nagy feladat feltörni egy programot?

És mennyire nagy feladat kilesni a benne rejlő esetlegesen titkos eljárást?

Sajnos a keretrendszer ebből a szempontból nem elég szigorú a felhasználókkal szemben. Bárki hozzáférhet a szerelvényekben lévő IL kódokhoz, ami – amint azt később bemutatom – elegendő ahhoz, hogy az eredeti forrásfájlhoz hasonlót állítsunk elő.

Amíg a piacon lévő egyik legnevesebb x86-os állományok visszafejtésére képes program kifejlesztése közel 10 évet vett igénybe, addig kutatómunkám keretein belül a majdnem másfél éve kezdett visszafejtőm ugyancsak képes az alapvető funkciókat ellátni. Ugyan a keretrendszer készítői nem ebből a célból, de sok olyan funkcionalitást

beleépítettek a Frameworkbe, amely jelentősen megkönnyíti a szerelvények visszafejtését.

Ezeket felhasználva, folyamatosan épül egy olyan feldolgozó könyvtár, amely képes egy assembly (szerelvény) megfejtésére, kinyerve abból minden olyan információt, amire valakinek szüksége lehetne ahhoz, hogy annak működését módosítsa. Fontos azonban leszögeznem, hogy a program nem ártó szándékkal jött létre, nem elérhető nyilvánosan, csupán kutatási célokat szolgál, rámutatva a gyenge pontokra a jövendőbeli hibajavításokban bízva.

A motor jelenleg a „Wriggler” kódnevet viseli.

2 Belső működés

Ebben a fejezetben végignézzük az összes szükséges lépést, ami a „nyers” bináris futtatható állományból számunkra könnyedén olvasható magas szintű programkódot állít elő.

Fontos megemlíteni, hogy minden részfeladatot egy-egy külön modul végez, amelyek szabadon kiterjeszthetők, kicserélhetők.

2.1 Felépítés

A kódvisszafejtő írásakor törekedtünk a világos és tiszta struktúrára, hiszen ahogy több ember részvételével folytatódott a munka, szükségszerűen a lehető legnagyobb mértékben el kellett tudni választani a munkaterületeket egymástól.

Erre a legalkalmasabb módszernek a moduláris felépítés látszott. Lényege, hogy minden részegység egy-egy külön egység, azaz modul, vagyis DLL, amit a fejlesztők külön-külön írnak, leválasztott projectben, egy közös kódkezelő rendszerben. A modulok együttműködése előre pontosan meghatározott interfészeken keresztül történhet. A közös munka megkezdése előtt ez volt a legfontosabb lépés, amit meg kellett tenni.

Az interfészek megtekinthetők a 3-as számú mellékleten.

3 StreamProcessor modul

Feladata a nyers .EXE, .DLL stb. PE állomány feldolgozása. A modul bemenete egy folyam (Stream), ami mutathat egy fájlra vagy memóriaterületre esetleg hálózati elérésre, attól függően, hogy a megnyitásra kerülő objektum hol helyezkedik el.

A megnyitást követően gyakorlatilag bájtról bájtra megtörténik az állomány beolvasása és ezzel párhuzamosan az adatok megfelelő struktúrákba való rendezése.

Különös figyelmet szenteltünk annak, hogy mindig csak a lehető legkevesebb I/O műveletet végezzünk el, hiszen könnyen előfordulhat, hogy az egyik szűk keresztmetszet az állományt közvetítő réteg lesz.

A feldolgozás lépésivel folyamatosan alakítjuk a fizikai struktúrát egy logikailag jól kezelhető struktúrává. A végeredmény egy IInstruction interfészt implemenetáló objektum tömb lesz, amit a kódvisszafejtő modulnak átadva megkezdődhet az utasítások vizsgálata, majd az AST építése.

Ehhez a művelethez sok olyan műveletet kell elvégezni, ami nem kapcsolódik közvetlenül az utasítások bájtkódjához. Az ilyen jellegű műveletekre példa a metaadatok közel teljes feldolgozása, hiszen ezek nélkül nem ismernénk a string-táblában szereplő értékeket, a más szerelvényekre való hivatkozásokat, vagy az egyes metódusok fizikai elhelyezkedését, nevét, scope-ját vagy paraméterlistáját.

3.1.1 Decompiler modul

A kódvisszafejtés végző egység a feldolgozási folyamat során megkapja az IL utasításokat reprezentáló tömböt, valamint több kiegészítő adatot, amelyeket a metaadat táblákból nyert ki a StreamProcessor. Ezen információk együttes használatával feladata egy olyan absztrakt fa építése, amely tökéletesen megfelel a programrész működésének. Ezzel megteremti annak elméleti lehetőségét, hogy az adott programrészt egy kiválasztott nyelven jelenítsük meg.

A fa helyes felépítéséhez a decompiler modulnak teljes körűen emulálnia kell a stack-et. Ennek megvalósítása egy virtuális stack bevezetésével történt, ami az emuláció során nem a tényleges futás során használt objektumokkal, hanem azok AST fában használt megfelelőjével dolgozik. Két hasznos következménye van ennek: az emuláció

során mindig a valós futtatáshoz hasonló állapotú lesz a stack (mélysége mindig megegyezik a futtatás egyes pillanataiban lévővel), valamint az AST szinte automatikusan épül, mivel a stack a fa csomópontjait és leveleit tartalmazza.

3.1.2 Code Generator modul

A kódgenerátor elsődleges feladata az általános AST-ből egy specifikus nyelvű forráskód előállítása. A cél a forráskód XML formába való öntése, amivel a cél az, hogy a forrásunk egyes elemei meta-információt is tartalmazhassanak. Erre példaként szolgálhat az egyes típusok, úgy, mint számok, hivatkozások metódusokra vagy offsetekre. Így megoldhatóvá válhat a Visual Studio-ból jól ismert „színes megjelenítés" vagy implementálható a metódusok egérgomb-nyomásra való követése, vagy éppen az objektumok típusánal vizsgálata a felületen.

Jelenleg a C# kódgenerátor fejlesztése folyik, azonban már képes a szoftver megjeleníteni a teljes metódusnyi kódrészleteket, igaz, még nem képes a grafikus felület minden támogatását kihasználni.

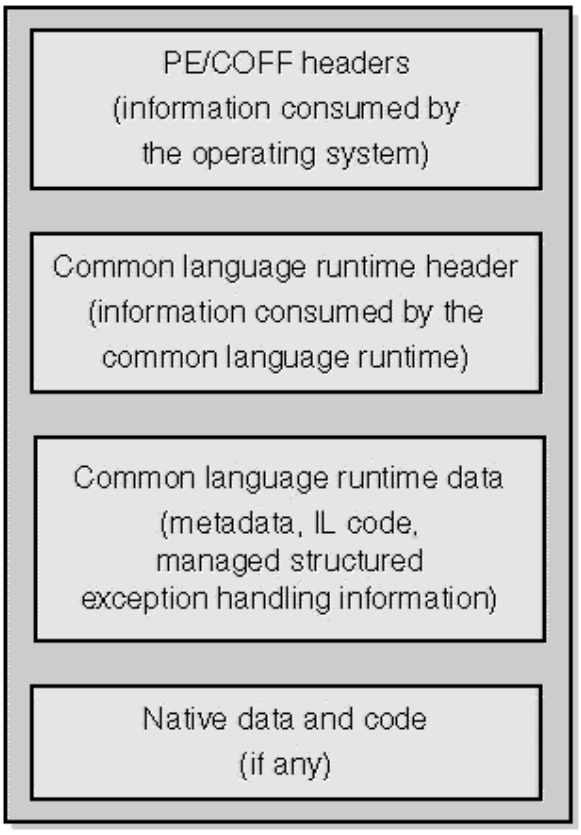
3.1.3 GUI modul

A grafikus felületről érhetjük el a visszafejtő motor funkcióit. Képesek vagyunk kódot generáltatni, nyelvet választani, böngészni a szerelvényeket.

3.2 PE olvasó

A PE, azaz Portable Executable (Hordozható Futtatható Állomány) már a DOS-os időkben is használatos volt, így nem csoda, hogy bizonyos szegmensei már nem használtak, vagy látszólag feleslegesek. Minden DOS és Windows rendszerben ez az a bináris keret, amibe bele kell helyeznünk minden futtatandó állományt. Ezek általában exe, dll, drv, vxd kiterjesztésűek, de ez nem megkötés.

Ennek a fájlformátumnak teljes körű leírása túlmutat ezen a dolgozaton, ezért most csak a felügyelt (.NET-es) PE formátum, számunkra lényeges részeivel fogok foglalkozni.



1. A PE állomány felépítése

A szerelvény egy 64 bájtos MS-Stubbal kezdődik, ennek csupán annyi a szerepe, hogy az operációs rendszer meg tudjon bizonyosodni arról, hogy valóban futtatható állományról van szó, és hogy ő képes-e futtatni.

Ezt egy 4 bájtos aláírás követ, ami minden esetben a „PE „ (0x50 0x45 0x00 0x00) karaktersor.

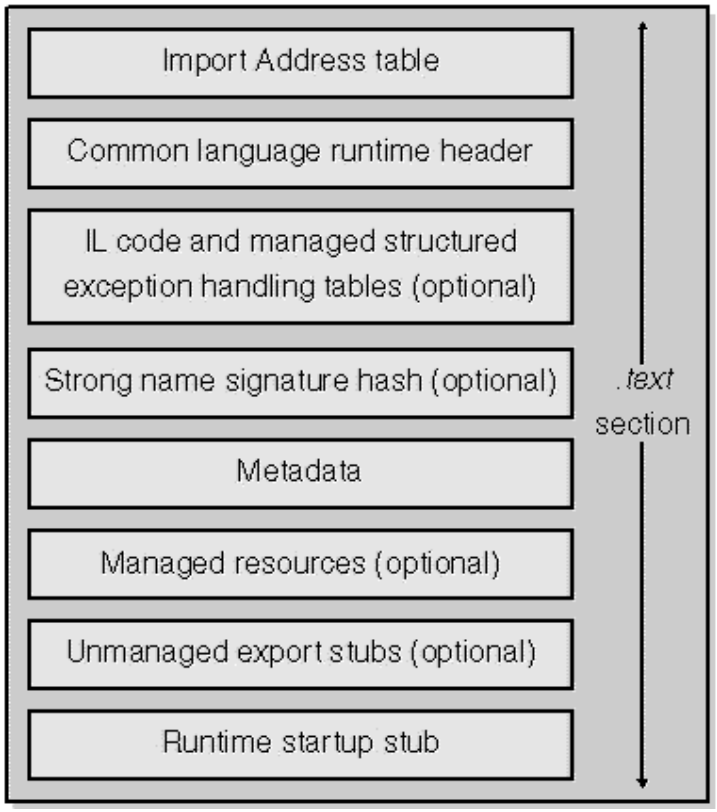
COFF Header: tartalmazza a célprocesszort, az állomány típusát, illetve a benne használt értékek ábrázolási módját.

Commom Language Runtime Header: itt találhatóak a keretrendszer számára fontos információk, például a minimális futtató keretrendszer verziószáma, a metaadatok relatív címe, és ha a szerelvény el van látva erős névvel, akkor annak címe is, de még sok egyéb.

Common Language Runtime Data: ez a rész a számunkra igazán fontos, ezért ezt lentebb részletesebben tárgyaljuk majd.

Native Data and Code: ha a szerelvény tartalmaz natív kódot, akkor az itt található. Ez két esetben fordulhat elő: ha programunk tartalmaz natív kódú részeket,

vagy ha az egész állománynak elkészült egy natív változata is a szerelvényben tárolva. Ha az utóbbival van dolgunk, akkor programunk beindulási ideje jelentősen jobb lehet, ezért a fejlesztők esetenként szándékosan generálnak (az ngen.exe programocskával) egy ilyen, platform-specifikus kódot az állományba.



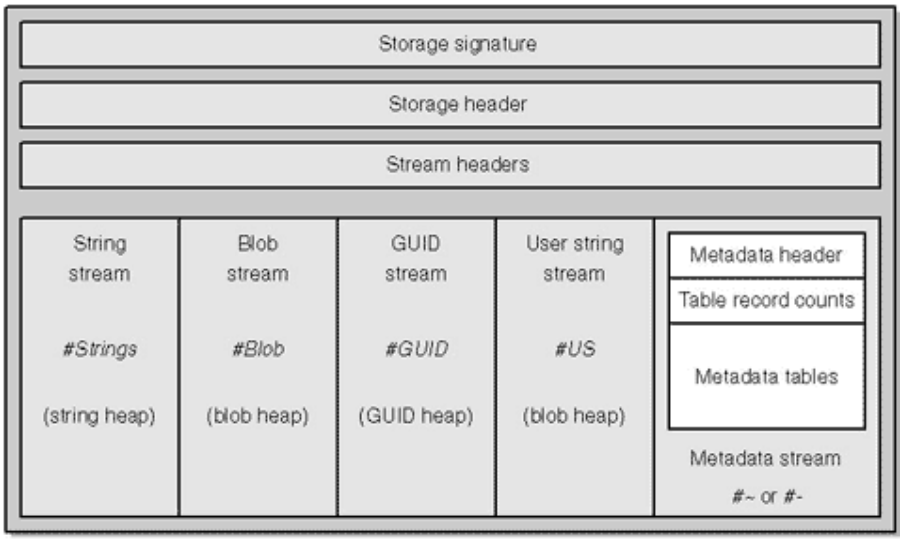
2. Common Language Runtime Data .text szekciója

A .text szekciónak 3 mezőjét fogjuk csak tárgyalni:

- IL code és MSEH: itt található talán a legtöbb számunkra hasznos információ, az IL utasítások bájt kódja és a kivétel kezeléshez szükséges adatok, try, catch finally blokkméretek. Ennek feldolgozása a fő feladat, az utasításokat objektumokká kell tudni alakítani és fordítva.

- String name signature hash: ha a szerelvényünk erős névvel van ellátva, akkor ide fog kerülni annak értéke. Az erős nevekkel és azoknak előnyeiről a későbbiekben még bővebben szó lesz.

- Metadata: meta azaz, a programkódon kívüli adatok, táblázatos formában. Ennek formája a 3. ábrán látható.



3. Metaadatok formája

Metaadatként sokféle információ tárolható, rendkívül rugalmas tárolási rendszer. Itt tárolhatók olyan szövegek, amelyek nem adhatók át egy IL utasításnak operandusként (például egy switch elágazás string típusú tagjainak listája vagy egy objektum típusát leíró szöveg (System.Int32)), de a Win32-es COM objektumokkal való kommunikációhoz használatos egyedi azonosítók is itt vannak elhelyezve, a kultúra specifikus feliratok mellett. Szokás ezekbe a táblázatokba beletenni, hogy ki a szerelvény készítője, milyen verziószámú, és nem utolsó sorban nagyon sok információ itt tárolódik, amire a reflexiónak szüksége van ahhoz, hogy dinamikusan, run-time feltérképezzen egy állományt.

3.3 IL assembly olvasó és író

Jelenleg az IL assembly olvasója külső komponens és írásra nem képes. Ennek fejlesztését már megkezdtük, azonban annak beültetésére még várnunk kell. A jelenleg használt komponens szerkezetileg teljesen be tud illeszkedni a keretrendszer Reflection objektum hierachiájába, csupán két óriási hiányossága van, amelyet reményeink szerint az általunk fejlesztett IL feldolgozó korrigálni fog.

Jelenleg nincs lehetőség a metaadatok táblázatos elrendezése feldolgozására, olyannyira, hogy valójában csak a kivételkezeléssel kapcsolatos információkhoz férhetünk hozzá.

A másik óriási hiányosság, hogy írásra nem képes, és nem is lehet kiterjeszteni a funkcionalitását.

Jelenleg gyakorlatilag annyira képes, hogy a bináris IL kódból létre tudja hozni az annak megfelelő System.Reflection.Instruction objektumokból álló listát. Ez a jelenlegi tesztelésekhez elég, azonban a későbbi, komolyabb tervek megvalósításához kevés lesz.

3.4 IL metaadatok feldolgozása

Minden .NET szerelvény rendelkezik metaadatokkal, a futtató kódhoz lazán kapcsolódó adattömbbel. Ez a .text szekcióban található fizikailag, logikailag pedig adatbázis táblákként lehet őket elképzelni. Ennek a felépítésnek köszönhetően könnyű használni, és rendkívül jól kiterjeszthető. Sajnos ez a funkció az IL beolvasóban el van nagyolva, így csak bizonyos táblák tartamát tudjuk olvasni, és azokat is csak szigorú megkötésekkel. Ez egy olyan pontja a kódvisszafejtőnek, ahol „lefojtódik" a visszafejtési művelet, emiatt nem lehet bizonyos kódrészeket az eredeti magas szintű nyelvre hűen visszaállítani.

Természetesen a fejlesztésben lévő IL olvasó és író már tökéletesen kezelni fogja ezeket a táblázatokat.

3.5 IL kód feldolgozása

Mivel a .NET keretrendszer alacsonyszintű nyelve az IL assembly nagyon hasonlít az x86-os assemblyre sok tekintetben, ezért a feldolgozása is hasonló. A szerelvényben, az IL kód blokkjában, minden, az adott metódusban szereplő, utasításnak megfelelően egy-egy bájtkódot olvashatunk, aminek megfelelőjét egy táblázat alapján állíthatunk össze (1. számú melléklet). A bájtkódok jelenleg (a Framework 1.1-es verziójában) egy vagy két bájt hosszúak lehetnek. Azt, ha van, követi egy, az operandusnak megfelelő bájttömb, majd jön a következő utasítás. Azt, hogy mely metódus pontosan honnan kezdődik, illetve meddig tart, csak a metaadatokból tudhatjuk meg.

3.6 CodeDOM fa felépítése

A CodeDOM, Code Document Object Model, a .NET Framework 1.0 része. Célja, hogy közel bármilyen programot le tudjunk vele írni pusztán a megfelelő objektumok fa-struktúrába való rendezésével. Óriási előnye, hogy ha megvan a megfelelő fánk, akkor tetszőleges nyelvre generálhatunk belőle forráskódot. Pontosabban csak óriási előnye lenne. Ha tökéletesen működne. Sajnos még az 1.1-es

verzióban is voltak hiányosságok, de ezeket a fejlesztés folyamán korrigáltam, így a Wriggler egy kiterjesztett CodeCOM-ot használ, ami már jelentősen robosztusabb ősénél.

Célunk tehát az IL assemblyből egy annak tökéletesen megfelelő CodeCOM fa felépítése.

Sajnos, ahogy egyre bonyolultabb szerkezeteket kezdünk el vizsgálni, láthatjuk, hogy gyakran olyan körülmények adódhatnak, amikor nem egyértelmű a továbbhaladás helyes útja. Tegyük fel, hogy egy elágazásunk érkezik a feldolgozó motor. Honnan tudhatjuk, hogy az most egy „*if*”, egy „*while*”, vagy egy „*for*” klóz? Mit kezdjünk a fordító optimalizálása utáni módosulatokkal? Hogyan fejezzünk ki olyan utasításokat, amelyek az IL-ben szabályosak, de például C#-ban nem ismertek? Mi a helyes döntés egy natív kódbeékelés esetén?

Ezek mind olyan kérdések, amiknek hatékony megoldása rengeteg munkát igényelnek, ha nem teljesen megvalósíthatatlanok.

Most lássuk a motor működését.

3.6.1 A decompiler algoritmus

A kódvisszafejtő algoritmus a program szíve. A megírásában nagy nehézséget jelentett, hogy valójában sehol sem találni értékes forrást az eddig már elkészült decompilerek által használt algoritmusokról. Ennek oka csupán az, hogy semelyik kódvisszafejtőt gyártó csapatnak sem érdeke, hogy valaki belelásson annak működésébe, hiszen ezzel a „másik oldal", a kódzagyválók feladatát könnyítené meg, mivel feltárná előttük a saját gyenge pontjait. Így a következőkben leírt algoritmus teljes mértékben saját elgondolások alapján készült, szem előtt tartva, hogy elsődleges cél a mindent visszafejteni tudás, a sebezhető pontok minimálisra szorítása.

Az algoritmus megírásakor két feldolgozási eljárás közül kellett választanom:

1, folyamatosan végignézem az IL kódot, és megépítem az utamba akadt kifejezéseket, majd ezeket beépítem a fába. Ez nagyon kényelmes és gyors megoldásnak tűnhet, de sajnos nem alkalmazható hatékonyan, mivel ha feltételezzük, hogy vannak feltétel nélküli ugrások (márpedig a Hello Worldön kívül elég nehéz olyan kódot írni, amiben nincs), akkor az első ugrást követően bajba kerülünk, ha nem követjük azt, mivel elvesztjük annak lehetőségét, hogy pontosan tudjuk emulálni, hogy végrehajtáskor mi van a stacken.

2, mindig követem a feltételezett végrehajtási motor útvonalát, emulálom a kiértékelő vermet, visszatérésnél visszatérek, pontosabban befejezem a visszafejtést.

A második elgondolás már közelebb áll az általam jónak talált megoldáshoz. A használt motor a kettő ötvözéséből született.

Minden IL utasítás mellé társítottam egy állapot zászlót, amely a sor feldolgozottsági szintjét, valamit a visszafejtést segítő információ típusát jelzi. A motor elkezdi szekvenciálisan elemezni az utasításokat, majd ha elágazáshoz ér, mindkét irányba elindul, és mindaddig folytatja a munkát, amíg már feldolgozott részhez nem ér. Mivel így a motor végigfut az összes végrehajthatósági ágon, ezért a módszerrel az is kiderülhet, hogy van-e a szerelvényben olyan részt, ami sohasem hívódhat meg. Amikor a feldolgozás után még egyszer ellenőrizzük a zászlók állását, egyértelművé válik, hogy mely utasítással foglalkoztunk és melyekkel nem.

Célszerűen, ha egy egész szerelvényt szeretnénk visszafejteni, akkor minden egyes belépési ponton (entry point) elindulunk és visszafejtünk mindent, majd ha meg valamilyen eljárás kimaradt, akkor azt jelezzük, majd kérés esetén azt is feldolgozzuk.

Természetesen, a dolog nem ennyire egyszerű, egy sornak több mint ötféle állapota lehet. Fel nem dolgozott, feldolgozott, igaz ági elem, hamis ági elem, kihagyott, hibás vagy nem értelmezhető, egyből fába építhető vagy előfeldolgozott. Később ez bővülhet, például a módosítási funkció bevezetésével.

Kihagyott akkor lehet, ha szándékosan nem vettünk róla tudomást. Tipikusan ilyen, ha ugrásnál az ugró utasítás és az ugrás címe közti területet (ideiglenesen) figyelmen kívül hagyjuk.

Hibás vagy nem értelmezhető, ha az utasítás nem ismert vagy abban a környezetben nem alkalmazható. Ez csak akkor fordulhat elő, ha a szerelvény hibás, nem megengedett a stack állapota az adott utasításnál vagy egy speciális – a visszafejtést megnehezítő – program hatására az IL utasítások szokatlan, összezavart sorrendben követik egymást. Az ilyen típusú problémák áthidalásáról a zagyválók fejezetben lesz szó.

Az **előfeldolgozott** elemekről a kódgenerálás fejezetében lesz részletesebben szó, a struktúrált kivételkezelés részletes tárgyalásánál. Alapvetően arról van szó, hogy a feldolgozás menete, a blokkok feldolgozási sorrendje nem kötött, így előfordulhat, hogy valamelyik egység már a fába építésre kész formában van, amikor a hagyományos feldolgozási fázisban elérjük az azon blokk első utasítását. Ezt valahogyan tudatnunk kell a motorral, erre való ez a zászló.

3.6.2 CodeDOM

CodeDOM fogalmak

Mint azt később látni fogjuk, több helyen is szükség lesz valamiféle többlet információ tárolására a CodeDOM objektumokban, a CodeDOM teljes kiterjesztéséhez.

Szerencsére erre van támogatás, ugyanis minden CodeDOM objektum kapott egy UserData nevezetű hashtáblát, amely tetszőlegesen feltölthető.

Az már a fejlesztés kezdetén látszott, hogy nagyon hasznos lenne, ha pontosan nyilván lenne tartva, hogy mely CodeDOM objektum mely IL utasításokból épült fel. Ez lehetővé tenné a kétirányú módosításokat, és sokban segítené a kódvisszafejtő munkáját. Ennek eléréséhez minden CodeDOM objektum létrehozásakor rögzítenünk kell benne a hozzá tartozó utasításokat. Egyszerű esetben, ha primitív elemről van szó, amely nem más CodeDOM objektumok bizonyos kapcsolatából jött létre, akkor nincs gond, a hashtáblába felvesszük adott kulccsal az egyes utasításokat. Az itt használt kulcs az IL<offset>.

Bonyolultabb a helyzet például az összeadásnál vagy egy bináris logikai műveletnél. Ekkor a logikailag összefűzött kifejezésnek megfelelő IL utasítások egyrészről a kifejezésben szereplő két „alkifejezés” minden utasításából másrészről a logikai IL utasításból áll.

Egy másik terület, ahol alkalmaztam a kiterjesztést, a CodeDOM által nem támogatott eljárások támogatásának megoldása. Ilyen többek közt az előjel váltás művelete, amelyre létezik IL utasítás, de CodeDOM megoldás nincs. További hiányosság, hogy argumentumok esetén, az azoknak megfelelő CodeDOM objektum nem tárolja el azok típusát. Ennek pótolása is megtörtént.

Másik alapvető hiányosság, hogy utólag már lehetetlen megtalálni, hogy egy adott kifejezés milyen építőelemekből állt össze. Ehhez minden objektum hashtáblájába beszúrásra került a gyermekeinek objektuma. Ennek szemmel látható előnye, hogy akár XML formátumban is lementhetővé válik a magas szintű nyelvi elemek kapcsolata. Ebből tetszőleges (támogatott) nyelvű forráskód készíthető, úgy hogy a kódgenerátor teljesen függetleníthető a kódvisszafejtőtől. A rugalmasság és a kiterjeszthetőség

szempontjából ez nagyon fontos, mivel nem látható előre, hogy milyen nyelvek támogatására lesz még szükség a jövőben.

A kiterjesztésel megvalósítása nem könnyű feladat a következő okok miatt:

Minden CodeDOM objektum a CodeObject-ből származik. Nekünk minden egyes CodeObjectből származó objektumtípust ki kellene egészíteni a speciális információkat tartalmazó hashtábla kezelésére alkalmas függvényekkel. Ez ráadásul minden objektumtípusnál más és más lehet, amellett, hogy vannak olyan funkciók, amelyeket minden CodeObject-be jó lenne integrálni (pl. IL utasítás hozzárendelés). Annak eldöntését, hogy vajon az éppen vizsgált CodeObject-ből származó objektumban implementálva vannak-e a kiegészítések, futási időben kell tudni ellenőrizni.

A követelmények tehát magasak, én a következő megoldást választottam: készítettem egy interfészt, amelyben definiáltam a minden CodeObjectből származó osztály kiterjesztéseit. Annak ellenőrzése, hogy egy példány ezt implementálja-e futási időben ellenőrizhető. Így sajnos minden egyes származtatott osztályban külön-külön implementálni kellene a fentebb említett függvényeket, ami nem járható út, különösen nem a fejlesztési szakaszban, amikor bármikor változás állhat be a függvények logikájában.

Erre hoztam létre a UserDataFunctions nevezetű statikus osztályt, amely csak statikus függvényekkel implementálja ezeket, majd minden osztályból pontosan ugyanazt hívja meg a kiterjesztett osztály, önmagát is átadva paraméterként. Ha minden osztályt kiterjesztünk, akkor mindegyiknek kell egy olyan osztályt készíteni, amely az eredetiből származik, implementálja a fenti interfészt, de ez még mindig nem elég minden funkció megvalósításához. Jó lenne, ha minden osztály konstruktorában megadhatnánk az aktuális IL utasítást vagy utasításokat, amelyek alapján az adott objektumot elkészítettük.

Ennek talán leghatékonyabb módja, hogy egy (saját fejlesztésű) kódgenerátor segítségével legeneráltatom az összes CodeObject-ből származó osztály kiterjesztett mását, amelyek megváltoztatott paraméterezésű funkcióinak törzsében egy vagy több statikus függvényt hívok meg, amelyet könnyedén változtathatok a kívánt működésnek megfelelően a fejlesztés során.

A CodeDOM kiterjesztése

Ahhoz, hogy mélyebbre áshassunk a motor működésébe, tisztáznunk kell a CodeDOM-ban is használt néhány fogalmat:

Expression: magyarul kifejezés. Ez az alapvető építkezési egység. A fában ez egy csomópontnak képzelhető el. Ezeknek két típusa van: amelyek csak levélként szerepelhetnek és azok, amelyek csak levéllel rendelkező csomópontként.
Minden ilyen osztály a CodeExpression osztályból származik és összesen 24 van az 1.1-es Frameworkben.

Csak levélként:

```
CodeArgumentReferenceExpression
CodeBaseReferenceExpression
CodeParameterDeclarationExpression
CodePrimitiveExpression
CodePropertySetValueReferenceExpression
CodeSnippetExpression
CodeThisReferenceExpression
CodeTypeReferenceExpression
CodeVariableReferenceExpression
```

Csak csomópontként:

```
CodeArrayCreateExpression
CodeArrayIndexerExpression
CodeBinaryOperatorExpression
CodeCastExpression
CodeDelegateCreateExpression
CodeDelegateInvokeExpression
CodeDirectionExpression
CodeEventReferenceExpression
CodeFieldReferenceExpression
CodeIndexerExpression
CodeMethodInvokeExpression
CodeMethodReferenceExpression
CodeObjectCreateExpression
CodePropertyReferenceExpression
CodeTypeOfExpression
```

Statement: magyarul programutasítás blokk. A CodeDOM csak ebből a típusból képes kódot generálni. Tehát minden Expressionből előbb utóbb el kell érni egy

Statement állapotot. A feldolgozás úgy történik, hogy folyamatosan építkezik a motor a stackre Expressionökből, amelyeket az IL utasításokból alakít ki, ezeket a megfelelő módon összefűzi, majd ha egy program utasítási blokk határához ér, akkor kialakít a stacken lévő legutóbbi Expressionből egy Statementet. A felsorolás után egy egyszerű példa tanulmányozása után világossá fog válni a művelet.

```
CodeAssignStatement
CodeAttachEventStatement
CodeCommentStatement
CodeConditionStatement
CodeExpressionStatement
CodeGotoStatement
CodeIterationStatement
CodeLabeledStatement
CodeMethodReturnStatement
CodeRemoveEventStatement
CodeSnippetStatement
CodeThrowExceptionStatement
CodeTryCatchFinallyStatement
CodeVariableDeclarationStatement
```

Példa:

C#

```
int a = 5;
int b = 12;

if ((a == 32) && (b <= 45))
{
      a = 76;
      if (b == 43)
      {
            return;
      }
}
b = 12;
```

IL

```
L_0000: ldc.i4.5
L_0001: stloc.0
L_0002: ldc.i4.s 12
L_0004: stloc.1
L_0005: ldloc.0
L_0006: ldc.i4.s 32
L_0008: bne.un.s L_0019
L_000a: ldloc.1
L_000b: ldc.i4.s 45
L_000d: bgt.s L_0019
L_000f: ldc.i4.s 76
L_0011: stloc.0
L_0012: ldloc.1
L_0013: ldc.i4.s 43
L_0015: bne.un.s L_0019
L_0017: br.s L_001c
L_0019: ldc.i4.s 12
L_001b: stloc.1
L_001c: ret
```

A fenti kódokból a következő CodeDOM objektumok lesznek:

ldc.i4.5 -> `CodePrimitiveExpressionEx(…)`

stloc.0 -> `CodeVariableReferenceExpressionEx(…)`

Mivel ezen két utasítás egy értékadásnak felel meg, ezért az annak megfelelő objektumot is létre kell hoznunk a fentiek átadásával paraméterként: `CodeAssignStatement(…)`

…

Egy elágazáshoz érkeztünk, amelynek feltétel összefüggését az előző utasításokkal feltöltöttük a stackre, így azokat le kell szedni, majd paraméterként átadni a feltétel kialakításához szükséges CodeDom objektumnak:

bne.un.s L_0019 -> `CodeBinaryOperatorExpressionEx(…)`

A feltétel végéhez érve: `CodeConditionStatement(…)`

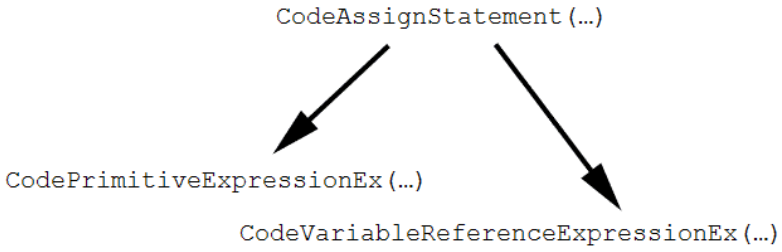
…

Ha a visszatérés pillanatában nem üres a stack, akkor az a visszatérési értéket tartalmazza, amit ugyancsak a megfelelő formába kell hoznunk:

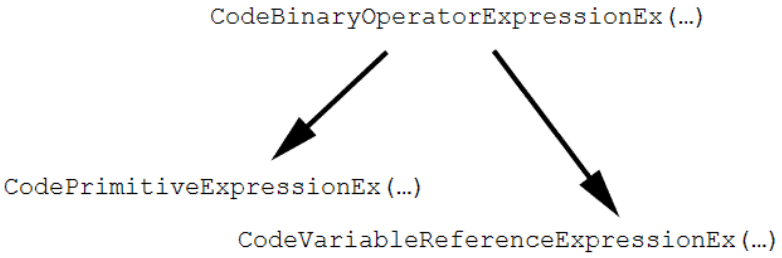
ret -> `CodeMethodReturnStatement(…)`

A fenti IL kódból a következő fa építhető:

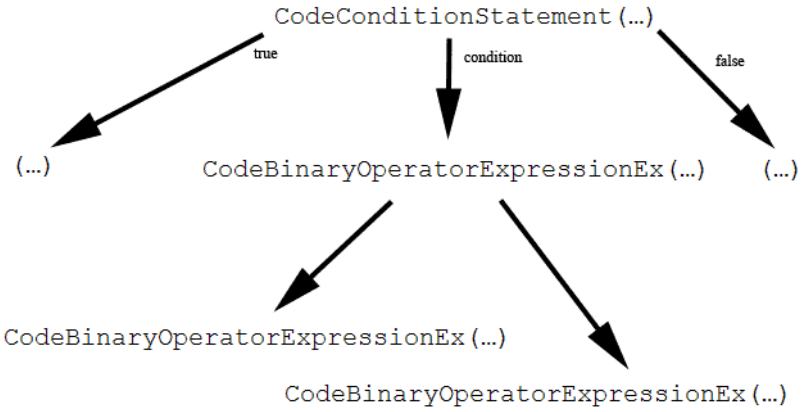
```
int a = 5;
int b = 12;
```




```
(a == 32)
(b <= 45)
```




```
if ((a == 32) && (b <= 45))
```



A példában a már kiterjesztett CodeDom látható, amelynek implementációja a 2. számú mellékletben található.

3.6.3 Gondolkodtató algoritmusok

A következőkben leírt algoritmusok általánosan használhatók, tehát, bármilyen kiértékelési verem alapú nyelven működőképesek. Ha például a motort fel szeretnénk készíteni a JAVA bájtkód visszafejtésére, akkor nem lenne más dolgunk, mint az IL olvasót egy JAVA bájtkód olvasására képes modulra cserélni. Ennek lehetőségét a laza interfész-csatolásokkal biztosítom a későbbi kiterjeszthetőség céljából.

3.6.3.1 *If-then-else* klóz detektálása

IL kódban nincs *if-then-else*, legalábbis könnyen kezelhető formában nincs. Három utasítás típus létezik, amely meghatározza ezeket a kódblokk elválasztó határokat.

- *Feltétel utasítások:* az *if* kezdetén megjelenik egy feltételes elágazást jelentő utasítás, ami természetesen sokféle lehet, például egyenlő (*breq*), nem egyenlő (*brne*), kisebb (*blt*), nagyobb (*bgt*).

Ezen utasítások operandusa az ugrási cím, amelyen a végrehajtás folytatódik a feltétel teljesülésekor. Tehát ezt nevezzük igaz-ágnak. Ha nem teljesül, akkor folytatjuk a végrehajtást az utasítást követően, ez lesz a hamis-ág.

- *Feltétel nélküli ugró utasítások:* csak akkor találkozunk velük, ha *else* blokk is létezik. Ekkor tulajdonképpen az *else* blokk végénél található, ami kiugrik az *if-then-else* utáni részre.

Azonban érdemes odafigyelni arra, hogy egyáltalán nem biztos, hogy van *else* blokk. Ebben az esetben viszont a fentiek szerint az IL kódban lenne egy feltételes utasítás, ami azt a feltételt tartalmazza operandusként, aminek nem teljesülésekor kellene a *then* ágat végrehajtani. Ez az eset IL-ben azonban másként jelenik meg. Az IL kódban a feltételes utasítással mindig a *then*-blokkban szereplő részt ugorjuk át.

Tehát a feltétel a C# vagy egyéb magasabb szintű programnyelven megfogalmazott feltétel **negáltja**.

C# kód:

```
if (int64 > 3) {
      System.Console.WriteLine( "true" );
} else {
      System.Console.WriteLine( "false" );
}
```

IL kód:

```
…
L_000d: ble.s L_001b
L_000f: ldstr "true"
L_0014: call System.Console::Void WriteLine(System.String)
L_0019: br.s L_0025
L_001b: ldstr "false"
L_0020: call System.Console::Void WriteLine(System.String)
L_0025: ret
```

(a zölddel jelölt kódrészlet az igaz, míg a pirossal jelölt a hamis ág)

A *ble.s* IL utasítás jelentése: branch if less or equal. Tehát ugorjunk, ha kisebb vagy egyenlő. A C# kódban ez egy nagyobb feltételként jelenik meg. És valóban a hamis ágra ugrunk, nem pedig az igazra. Tehát az első feltételezéssel ellentétben a feltétel nélküli ugrás (jelen esetben a *br.s*) az igaz ág végén található és a hamis ág utáni utasításra ugrik.

IL assemblyben megtehetjük, hogy tetszőlegesen sokszor hajthatunk végre ugró utasítást a programkódban, így természetesen egy igaz vagy hamis ágon belül is. Ez azt jelenti, hogy a következő példa hibátlan programot eredményez, ám az algoritmusunk jelenlegi formájában nem működőképes.

```
…
L_000d: ble.s L_001b
L_000f: ldstr "true"
L_0014: call System.Console::Void WriteLine(System.String)
L_0019: br.s L_0022
L_001b: ldstr "false"
L_0020: br.s L_0025
L_0022: br.s L_002B
L_0025: call System.Console::Void WriteLine(System.String)
L_002B: ret
```

A Wrigglerben található algoritmus megoldást jelent erre a problémára: elágazásnál a normál feldolgozási menetet megszakítjuk, és mindkét irányba elindul a jelölő metódus, ami megjelöli az általa igaz, illetve hamis ágat felépítő utasításoknak vélteket. A jelölések befejeztével lesznek igaz, hamis, illetve olyan zászlós utasítások, amelyek mindkét jelölővel rendelkeznek. Utóbbiak már kívül esnek az *if-then-else* blokkon, tehát azokkal most nem kell foglalkoznunk.

Most már egyértelmű, hogy mely utasítás hova tartozik, így össze tudjuk ezeket fogni (végrehajtási sorrend szerint), és megkezdődhet az egy vagy két utasítás halmazra a feldolgozó motor elindítása.

Ha nem adódik semmilyen fennakadás, akkor megkapjuk mindkét ágat feldolgozott formában.

Egy részfeladattal azonban még nem foglalkoztunk: az *if* feltételének összeállításával. Itt a fő problémát az jelenti, hogy a feltétel nem feltétlenül csak egy logikai egység, lehet azok logikai műveletekkel való összefűzése. Például:

C# kód:

```
if ((strOut != "Cond1") && (strOut != "Cond2") && (strOut !=
"Cond3")) {
        System.Console.WriteLine( "true" );
}
```

IL kód:

```
L_0000: ldarg.1
L_0001: ldstr "Cond1"
L_0006: call op_Inequality(System.String, System.String)
L_000b: brfalse.s L_0031
L_000d: ldarg.1
L_000e: ldstr "Cond2"
L_0013: call op_Inequality(System.String, System.String)
L_0018: brfalse.s L_0031
L_001a: ldarg.1
L_001b: ldstr "Cond3"
L_0020: call op_Inequality(System.String, System.String)
L_0025: brfalse.s L_0031
L_0027: ldstr "true"
L_002c: call System.Console::Void WriteLine(System.String)
L_0031: ret
```

Első lépésként meg kell határoznunk a feltételt felépítő utasításokat. A kódrészletet tanulmányozva arra a megállapításra juthatnánk, hogy az azonosan a blokk végére mutató ugrási címet tartalmazó feltételes utasítások részei a vizsgált *if* feltételének. Ezzel kapcsolatosan két probléma merülhet fel: nem tudhatjuk, hogy egy adott feltételrészlet, ami feltételes ugrással ér véget, pontosan mely utasítástól kezdődik. A másik gond, az egybeágyazott *if*-ek egy speciális esetében kerül csak elő, amikor nem tudhatjuk, hogy az adott feltételrészlet a gyermek vagy a szülő feltételéhez tartozik.

Az első problémára a fentebb leírt CodeDOM kiterjesztés jelent megoldást. Mivel ennek segítségével meg tudjuk nézni, hogy az egyes CodeExpression-ökhöz mely IL utasítások tartoznak. Ha ez viszont megvan, akkor semmi más dolgunk nincs, mint a feltétel véglegesítésénél megnézzük, hogy a stackről leszedett kifejezéshez pontosan mely IL utasítások tartoznak. Ezeket megjelöljük a „c” (condition) zászlóval.

A második jelenség, sajnos kicsit nehezebben feloldható, de először lássuk pontosan, miről van szó:

C# kód:

```
if ((strOut != "Cond1") && (strOut != "Cond2") && (strOut !=
"Cond3")) {
    System.Console.WriteLine( "true-1" );
    if (strOut != "Cond_2") System.Console.WriteLine( "true-2" )
}
```

IL kód:

```
L_0000: ldarg.1
L_0001: ldstr "Cond1"
L_0006: call op_Inequality(System.String, System.String)
L_000b: brfalse.s L_0048
L_000d: ldarg.1
L_000e: ldstr "Cond2"
L_0013: call op_Inequality(System.String, System.String)
L_0018: brfalse.s L_0048
L_001a: ldarg.1
L_001b: ldstr "Cond3"
L_0020: call op_Inequality(System.String, System.String)
L_0025: brfalse.s L_0048
L_0027: ldstr "true-1"
L_002c: call System.Console::Void WriteLine(System.String)
```

```
L_0031: ldarg.1
L_0032: ldstr "Cond_2"
L_0037: call op_Inequality(System.String, System.String)
L_003c: brfalse.s L_0048
L_003e: ldstr "true-2"
L_0043: call System.Console::Void WriteLine(System.String)

L_0048: ret
```

Látható, hogy a negyedik feltételrészletet az előző algoritmus belerakná az általa egyetlen *if*-ként felismert klóz feltételébe. Ez nem helyes, ugyanis egy új *if*-et kellene létrehoznia és nem az előzőt folytatni. Erre a problémára a megoldás a következő: csak addig dolgozzuk fel az *if* feltételének megfelelő utasításokat, amíg azok sora meg nem szakad.

A feltételeket feldolgozó algoritmus igen sokoldalú, de még viszonylag egyszerű, gyors és átlátható. Röviden a lépések még egyszer:

- feltételes utasításnál az utasítás és operandusának megjelölése „c” zászlóval
- folyamatosan haladva, minden feltételes utasítás és operandusának megjelölése mindaddig, amíg ezek ugrási címe megegyezik az elsővel és a jelölés után nem lesz a „c” zászlóval ellátott utasítások sora megszakítva nem megjelölt utasítással
- ha már nincs több „c”-vel jelölhető feltérképezzük az igaz („t” zászló) és hamis („f” zászló) ágakat alkotó utasításokat
- feltétel elkészítése a „c” utasításokból
- igaz statement elkészítése a „t” utasításokból
- hamis statement elkészítése a „f” utasításokból
- végleges *if-then-else* CodeDOM objektum megalkotása és beszúrása az eddig elkészült fába

3.6.3.2 While vagy for ciklus detektálása

Az *if-then-else* klóz detektálásának algoritmusa nem tűnhet túlzottan egyszerűnek, de az abban bevezetett fogalmak és eljárások ismeretében az iterációk jelentősen könnyebben megérthetők.

Hasonlóan az x86 assemblyhez, az IL esetén is az elágazások és az iterációk megjelenése nagyon hasonlít egymásra. Valójában a ciklus nem más, mint egy ismétlődő feltételes elágazás.

Azonban fel kell figyelnünk egy apró érdekességre, amit a .NET-es fordítónk végez, ha iterációhoz érkezik.

Ha az utca emberét megkérdeznénk, hogy hogyan fordítana le egy egyszerű ciklust IL kódra, szinte biztos, hogy a következő kódrészletet adná válaszul:

```
…
<feltétel ellenőrzés, majd ugrás, ha nem teljesül>
<a ciklus teste>
<ugrás az ellenőrzés előtti utasításra>
<ide ugorjunk, ha már nem teljesül>
…
```

Ezzel szemben a C# fordító a következőképpen viselkedik:

```
…
<ugrás az ellenőrzés előtti utasításra>
<a ciklus teste>
<feltétel ellenőrzés, majd ugrás, ha teljesül>
<ide ugorjunk, ha már nem teljesül>
…
```

Természetesen a decompiler algoritmusának fel kell ismernie mindkét esetet, ugyanis elképzelhető, hogy más magas szintű fordítók máshogy hajtják végre az átalakítást.

A használt algoritmus kísértetiesen hasonlít a fenti *if*-esre, a feltétel előállítása tökéletesen ugyanaz, egyedül a jelölésnél van eltérés, ugyanis akkor tudhatjuk biztosan, hogy iterációról van szó, ha már egy adott („t” vagy „f”) zászlóval megjelölt utasítást még egyszer meg szeretnénk jelölni ugyanazzal.

A ciklus kódblokkjának előállítása az *if then*-blokkjának feleltethető meg.

A sok hasonlóságot az algoritmus is kihasználja, tulajdonképpen egészen a jelölés végeredményéig nem tudja a decompiler, hogy mi lesz a feldolgozott kódrészletből, *if* vagy *for(while)*. Ezt követően sem kell különválasztani a két folyamatot, a blokkok feldolgozását is ugyanaz a kód végzi, végül a CodeDOM objektumok építésénél van némi különbség.

Ez a tulajdonság rendkívül rugalmassá teszi ezt a motor ezen funkcióját. Amint látható, ezek a műveletek gyorsabban is megvalósíthatók lennének, ha nem akarnánk azt, hogy gyakorlatilag minden kódra működjön. Azonban a fejlesztés során elsősorban a **mindent visszafejteni tudás**, a későbbi **kiterjeszthetőség** és az **áttekinthetőség** voltak a legfőbb szempontok.

A ciklusok igazi nehézsége a *while* és a *for* ciklus közti különbség felderítése. A két ciklus közti különbség, hogy a *for* tartalmaz egy inicializálási részt, illetve egy minden iteráció végén végrehajtandó részt a feltétel ellenőrzése mellett. Ilyen megkülönböztetés az IL kódban azonban nincs. Tehát ez természetéből fakadóan egy egyirányú átalakítás. Elméletileg is lehetetlen megállapítani, hogy az eredeti kódban mik voltak pontosan ezek a kódrészek. Ezért egyelőre ezeket a decompiler nem deríti fel, bár a későbbiekben terveim szerint a „tipikus *for*" illetve a *foreach* ciklus felismerhető lesz. Jelenleg minden ciklus while-jellegű, azonban mivel felméréseim szerint a programozók 90%-a szívesebben használ (és lát viszont) *for*-t, mint *while*-t, a visszafordított kódban is *for* szerepel, igaz, üres inicializáló és léptető mezővel.

3.6.3.3 Kivétel kezelési blokkok meghatározása

Kivételkezelés nélkül nincs fejlett futtató környezet. Lássuk, hogyan biztosítja a .NET keretrendszer a blokkok meghatározhatóságát. Minden kivétel osztály a System.Exception osztályból származik, és ezek egy rendszerezett hierarchia szerint viszonyulnak egymáshoz.

Kivételt akkor dob a rendszer, ha a megszokottól eltérő műveletet akarunk végrehajtani.

Vegyük példának a nullával való osztást. Ha nullával osztunk, akkor a DivideByZeroException kivételt fogjuk kapni. Ennek a kivételosztálynak a következő ősei vannak: ArithmeticException, SystemException, Exception.

A kivételek elkapása *try* blokkban történik, kezelésük *catch* blokkban, illetve megadható egy olyan műveletsor, ami csak hiba esetén hajtódik végre, ez lehet *finally* vagy *fault* blokk. A kettő közti különbség, hogy az előbbi mindig végrehajtódik az utóbbi ezzel szemben csak hiba esetén, továbbá a fault blokk nem használatos C# nyelven.

Nézzünk erre egy egyszerű C# példát:

```
try
{
      Console.WriteLine( "try - 1" );
      k=36/k;
      Console.WriteLine( "try - 2" );
}
catch (Exception ex)
{
      Console.WriteLine( "catch" );
}
finally
{
      Console.WriteLine( "finally" );
}
```

Ha k nem nulla a *try* blokkot megelőzően, akkor kivételt nem kapunk, és ekkor a kimenet a következő:

```
try - 1
try - 2
finally
```

Ha azonban a k nulla, akkor kivétel dob a rendszer, mivel nullával próbálunk osztani, tehát a kimenet a következő:

```
try - 1
catch
finally
```

Minket leginkább az ebből készülő IL kód érdekel. Meglepő módon összesen két utasítás van, amely a kivételkezelésre vonatkozik, ezek pedig az *endfinally* és *leave*. Az előbbivel, mint ahogy a neve is mutatja, tudjuk jelezni a *finally* (vagy *fault*) blokk végét. A *leave* a *catch* blokk végén található. És ebből honnan fogja tudni a futtató környezet, hogy pontosan mettől meddig számíthat az egyes kivételekre, és ezeket hogyan kell lekezelnie?

Erre a kérdésre a metaadatok tüzetes vizsgálata után kaphatjuk meg a választ. A .NET assemblyk ugyanis itt tárolják a kivételkezelésre vonatkozó többlet információkat, amelyekre feltétlenül szükségünk van ahhoz, hogy vissza tudjunk állítani bármilyen magasabb szintű nyelvi forráskódot.

Egy metaadat blokk a következő információkat tartalmazza: *try* blokk kezdete, *try* blokk mérete, kezelő blokk típusa, kezelő blokk vége, kezelő blokk mérete.

A kezelő blokk típusa lehet *catch, finally, fault* vagy *filter*. A fenti példának megfelelő metaadatok:

1. *try* kezdete, *try* hossza, *catch* típus, *catch* kezdete, *catch* hossza, kivétel típusa
2. *try* kezdete, *try* hossza, *finally* típus, *finally* kezdete, *finally* hossza

Ellenőrzésképpen nézzük meg, hogy ez már valóban elég információ-e a teljes visszafejtéshez. A *try* blokk offsetjét és hosszát ismerjük, mivel mindkét metaadat tartalmazza ezen értékeket, *catch* blokk megvan az elsőben, *finally* megvan a másodikban. Érdekesség, hogy a *finally* típusnál a *try* blokk mérete akkora, hogy magában foglalja az előtte lévő *catch* blokkot is. Sajnos a blokkok tökéletes elkülönítéséhez elég sokat kell robotolni. És még a buktatókról nem is esett szó. Akkor lássunk egy olyan mintapéldát, amely a lehető összes kellemetlen tulajdonsággal meg van áldva.

C# kód:

```
try
{
try
        {
                k = 36/k;
        }
        finally
        {
                k = 60;
        }

try
        {
                k = k/12;
        }
        catch (System.NullReferenceException)
        {
                WriteLine( "!NullReferenceException!" );
        }
}
catch (System.ArithmeticException ex)
{
```

```
        WriteLine( "!ArithmeticException!"+ex.ToString() );
}
catch
{
        WriteLine( "!Exception!" );
}
finally
{
k = -1;
}
```

IL kód:

```
.maxstack 2
.locals (System.Int32 V_0, System.ArithmeticException V_1)
.try L0002-L0009 finally L0009-L000d
.try L000d-L0014 catch NullReferenceException L0014-L0021
.try L0002-L0023 catch ArithmeticException L0023-L003b
.try L0002-L0023 catch Object L003b-L0048
.try L0002-L004a finally L004a-L004d
L_0000: ldc.i4.0
L_0001: stloc.0
L_0002: ldc.i4.s 36
L_0004: ldloc.0
L_0005: div
L_0006: stloc.0
L_0007: leave.s L_000d
L_0009: ldc.i4.s 60
L_000b: stloc.0
L_000c: endfinally
L_000d: ldloc.0
L_000e: ldc.i4.s 12
L_0010: div
L_0011: stloc.0
L_0012: leave.s L_0021
L_0014: pop
L_0015: ldstr "!NullReferenceException!"
L_001a: call System.Console::Void WriteLine(System.String)
L_001f: leave.s L_0021
L_0021: leave.s L_0048
L_0023: stloc.1
L_0024: ldstr "!ArithmeticException!"
L_0029: ldloc.1
L_002a: callvirt System.Exception::System.String ToString()
L_002f: call System.String::System.String Concat(System.String, System.String)
L_0034: call System.Console::Void WriteLine(System.String)
L_0039: leave.s L_0048
```

```
L_003b: pop
L_003c: ldstr "!Exception!"
L_0041: call System.Console::Void WriteLine(System.String)
L_0046: leave.s L_0048
L_0048: leave.s L_004d
L_004a: ldc.i4.m1
L_004b: stloc.0
L_004c: endfinally
L_004d:                                                                    ret
```

A metaadatok a *.try* előtag után találhatók, összesen 7 darab. Itt már offseteket olvashatunk, mint blokkhatárok, nem kell a kezdő offset és a méret alapján számolgatnunk.

A végső algoritmus sajnos meglehetősen bonyolult lett, összesen 8 lépcsőfokon keresztül juthatunk el az IL kódtól a C# kódig.

Mielőtt azonban belevágnék a fokok leírásába, tudnunk kell még valamit zászlós jelölő rendszerről. Van egy zászló, amiről még nem esett szó, ez pedig a „s", statement, amely arra szolgál, hogy a már feldolgozott és beépíthető blokkokat jelöljük vele. A feldolgozott blokkok kész CodeDOM objektuma bekerül egy a disassembler szintjén lévő globális statement-tárba, majd ha egy „s" zászlós részhez érve szükségünk lenne az ottani utasításokra, akkor azok már korábban elkészített CodeDOM megfelelőjét kikeresi a motor a tárból. Ennek majd az algoritmus utolsó lépésénél lesz jelentősége.

Lássuk a lépéseket részletesen:

1. gyűjtsük össze az azonos *try* blokkal rendelkező *catch* blokkokat: mivel egy *try* blokkhoz attól függően, hogy hányféle kivételt szeretnénk kezelni, több *catch* blokk is tartozhat, ezért ezeket össze kell fogni, mivel a későbbiekben csak együttesen kell őket használnunk.

2. rendezzük a *finally* és *fault* blokkokat a *try* blokkjaik mérete szerint növekvő sorba: ennek lényege a menet közbeni rekurzió elkerülése. Mivel tudjuk, hogy sem *finally* sem *fault* blokkon belül már nem lehet újabb *try-catch-finally* (a CLR szabvány szerint) ezért ezeket kezelhetjük úgy, mint feldolgozási egységeket. A méret szerinti növekvő sorrendezés a feldolgozási sorrendet fogja meghatározni.

3. a *catch* tömböket rendezzük a *try* blokkjuk mérete szerint csökkenő sorba: alapszabályunk, hogy nagyobb *try* blokkal rendelkező *catch* blokk csak nála kisebb *try*

blokkal rendelkező *catch* blokkot tartalmazhat. A következő lépésnél nagyon hasznos lesz, hogy csökkenő sorba rendezzük ezeket.

4. vegyük a legkisebb *finally* vagy *fault* blokkot

5. vegyük azon legnagyobb *catch* blokkot, amely benne van a *finally* vagy *fault* *try* blokkjában: ezen a ponton társíthatjuk a *finally* és *fault* blokkokat a *catch* blokkokkal.

6. rendezzük ismét a besorolt *catch* blokkokat a *catch* blokk offsetjeik szerint: erre azért van szükség, hogy visszanyerjük az eredeti kivétel kezelési sorrendet. Ezen sorrend azért nem lényegtelen, mivel előfordulhat, hogy egy kivétel esetén van kezelő függvényünk az adott kivételre és annak őskivételére is. Ebben az esetben a sorrend dönti el, hogy melyik kivételkezelő *catch* blokk fut lefutni.

7. számoljuk ki a *try* blokkok kezdetét és hosszát: ha van az adott blokkhoz *finally* vagy *fault* blokk, akkor nem kell más tennünk, mint venni az adott *finally* vagy *fault* blokkot és annak *try* blokkjából kivenni az ahhoz tartozó *catch* blokkokat, így megkaphatjuk az azokhoz tartozó *try* blokk méretét. Ha azonban nincs *finally* vagy *fault* blokk, akkor sem kell megijednünk, hiszen az azonos *try* blokkal rendelkező *catch* blokkjaink már „csokorba vannak kötve”, így nincs más dolgunk, mint végignézni a csokrokat és mindegyikhez társítani egy *try* blokkot.

8. dolgozzuk fel az egyes egységeket és építsük meg a végleges CodeDOM fákat, amelyeket majd később be kell építeni a végleges fába: minden blokkhatár kikristályosodott, így feldolgozhatjuk azokat, egymást követve, célszerűen méret szerinti növekvő sorrendben.

A *catch* blokkok feldolgozása kicsit nehezebb két okból: a .NET keretrendszer minden *catch* blokk futtatásának elindítása előtt a stackre helyezi a kivételt megtestesítő objektumot. Annak érdekében, hogy a stack tartalma végig valós tudjon maradni, ezt a motornak is meg kell tennie. A másik nem elhanyagolható dolog, hogy természetesen a *catch* blokkban hivatkozhatunk a kivétel objektumra, tehát ezt, mint helyi változót be kell, hogy vezessük.

Mivel ezek a blokkok nem feltétlenül fedik az egész forráskódot, ezért ezek még nem illeszthető be a végleges fába, csak a statement tárba kerülnek.

Ha ez megvan, akkor a motor az egész metódus feldolgozását megkezdi és amint egy már feldolgozott részhez ér, nem tesz mást, mint kiveszi a már kész részfát a tárból, csatolja a jelenlegihez, majd folytatja a feldolgozást.

3.7 Saját kódgenerátor

A fejlesztés folyamán sokszor felmerült az igény egy saját kódgenerátorra, több okból is. Az automatikus kódgenerátor csak az ismert CodeDom objektumokból tud generálni, elhagyva a kiterjesztéseket, csak tisztán szöveget generálva. Ez azért lehet gond, mivel szeretnénk a felhasználóknak biztosítani olyan lehetőségeket, amelyek már alapkövetelménynek számítanak, mint például a linkek használata, objektumhivatkozások felderítése egy kattintással, színes, sokatmondó HTML alapú megjelenítő ablakok, az egér megfelelő helyen való megállítása esetén cédulák (tooltipek) feltűnése, drag'n'drop és sok más kényelmi funkció. Ezek ugyan javarészt a felhasználói felületet érintik, azonban megfelelő belső struktúra nélkül nem kivitelezhetőek.

Az XML technológia jó tulajdonságai miatt - úgymint a széles körű támogatottság, kiterjeszthetőség és szabványosság – a belső kommunikáció formátumának ezt választottam. Így az IL kódok feldolgozása után a kódvisszafejtő motor XML formátumban „fentebb adja" a kialakított CodeDom fastruktúrát a kódgenerátornak.

Jelenleg a kódgenerátor a Refrection névtérben megtalálható ugyan, de az áttérés már folyamatban van. Mivel az átadott XML-ben minden kiterjesztés átkerül a kódgenerátorhoz, az, hogy a generátor ebből mennyit hasznosít már csak rajta múlik. Ezzel sikerült egy rendkívül lazán kapcsolt moduláris rendszert kialakítani, ami a későbbi továbbfejlesztés szempontjából nagyon fontos.

A generátor legegyszerűbb formája lehet akár egy XSL transzformáció is, de a felhasználói felület integrálásával bármilyen bonyolultságú művelet elvégezhető lehet.

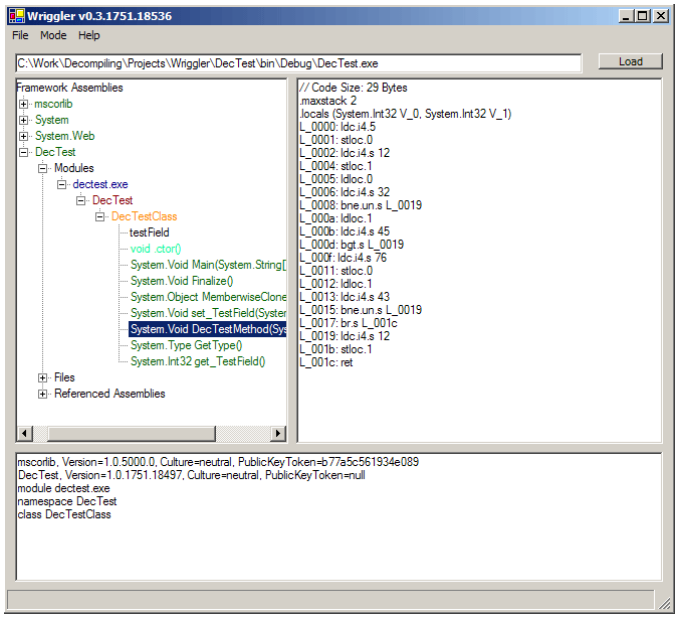
3.8 Megjelenítés

A motor kidolgozásának kezdeti szakaszában kis prioritást kapott a megjelenítés. Jelenleg egy WinForms alkalmazáson keresztül utasíthatjuk a motort a metódusok visszafejtésére, majd az eredményt csak szöveges formában tudjuk megtekinteni.

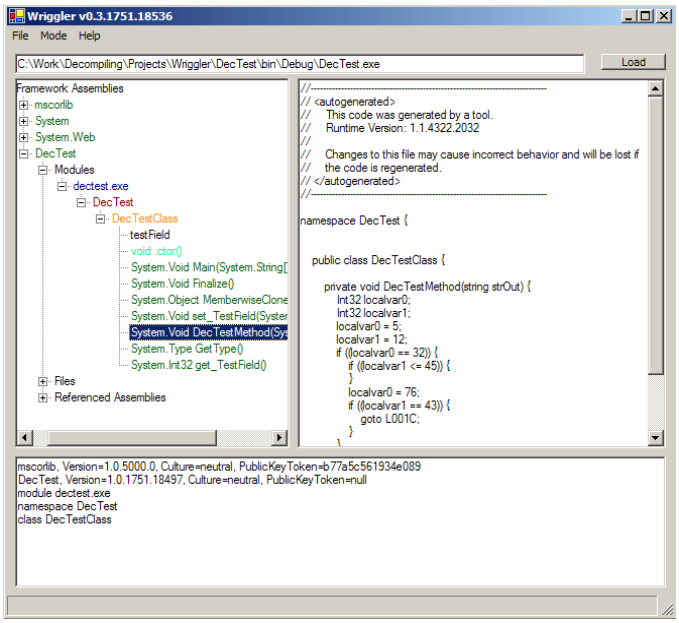
A felhasználói felület bal oldalán láthatjuk a strukturális fát, a szerelvények, modulok, névterek, osztályok listáját megfelelő sorrendezésben.

Jobb oldalon a visszafejtett IL kód vagy C# kód látható, az aktuális metódusra vonatkozóan.

A felület alsó része részlet információkat tartalmaz az éppen kiválasztott modulról, szerelvényről vagy osztályról.



4. A kódvisszafejtő felhasználói interfésze visszafejtés előtt



5. A kódvisszafejtő felhasználói interfésze visszafejtés után

Visszafejtés közben nyomon követhetjük a motor állapotát, lépésről lépésre. Mivel XML formátumban érkezik az információ a feldolgozó egységtől, ezért legegyszerűbben egy DataGrid típusú kontrollal ábrázolható.

offset	display	opcode	operand	operanddata	flags
0	L_0000: ldc.i	27			
1	L_0001: stloc	10			
2	L_0002: ldc.i	31	12	0C	
4	L_0004: stloc	11			
5	L_0005: ldloc	6			
6	L_0006: ldc.i	31	32	20	
8	L_0008: bne.	51	25	0F	
10	L_000a: ldloc	7			
11	L_000b: ldc.i	31	45	2D	
13	L_000d: bgt.s	48	25	0A	
15	L_000f: ldc.i4	31	76	4C	
17	L_0011: stloc	10			
18	L_0012: ldloc	7			
19	L_0013: ldc.i	31	43	2B	
21	L_0015: bne.	51	25	02	
23	L_0017: br.s	43	28	03	
25	L_0019: ldc.i	31	12	0C	
27	L_001b: stloc	11			
28	L_001c: ret	42			

6. A kódvisszafejtő felhasználói interfésze visszafejtés alatt 1

	offset	display	opcode	operand	operanddata	flags
▶	0	L_0000: ldc.i	27			33
	1	L_0001: stloc	10			33
	2	L_0002: ldc.i	31	12	0C	33
	4	L_0004: stloc	11			33
	5	L_0005: ldloc	6			33
	6	L_0006: ldc.i	31	32	20	33
	8	L_0008: bne.	51	25	0F	33
	10	L_000a: ldloc	7			33
	11	L_000b: ldc.i	31	45	2D	33
	13	L_000d: bgt.s	48	25	0A	33
	15	L_000f: ldc.i4	31	76	4C	33
	17	L_0011: stloc	10			33
	18	L_0012: ldloc	7			33
	19	L_0013: ldc.i	31	43	2B	33
	21	L_0015: bne.	51	25	02	33
	23	L_0017: br.s	43	28	03	33
	25	L_0019: ldc.i	31	12	0C	9
	27	L_001b: stloc	11			9
	28	L_001c: ret	42			1
*						

7. A kódvisszafejtő felhasználói interfésze visszafejtés alatt 2

4 Nehezen kivédhető problémák

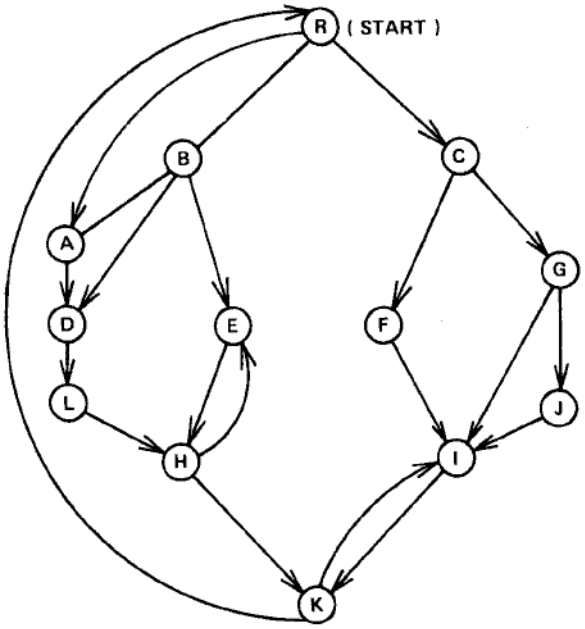
4.1 Kódoptimalizáló algoritmusok

Amikor egy programot lefordítunk fejlesztési (debug) módban, akkor észrevehetjük, hogy az állományunk nagyobb méretű és/vagy több fájt generál a fordító, mint ha a kiadási (release) módban végeznénk el a műveletet. A méretkülönbség természetesen abból adódik, hogy sok, a debugger számára fontos információval egészítjük ki a lefordított programunkat. Ezen kívül azonban még rengeteg dolog másként történik a két üzemmód során. Ami a visszafejtés tekintetében talán a legfontosabb, hogy a kiadási fordításnál egy kódoptimalizálás is végrehajtódik, aminek részletes megismerése segíthet elkerülni a váratlan meglepetéseket, valamit világossá válhat, hogy miért nem lehetséges mindig az eredeti magas szintű kód visszaállítása. Természetesen az optimalizáció jelentős része nem most, fordítási időben, hanem futtatás előtt, a JIT (Just-In-Time compiler) által történik, mivel jelentős teljesítmény-javítást csak az adott platform teljes ismeretében lehet elérni. A következőkben csak a fordítási idejű eljárásokról lesz szó.

A modern kódoptimalizálás szinte egy külön tudományág, így lehetetlenség lenne teljes részletességében tárgyalni az eljárásokat, azonban az alapvető lépéseket érdemes áttekinteni.

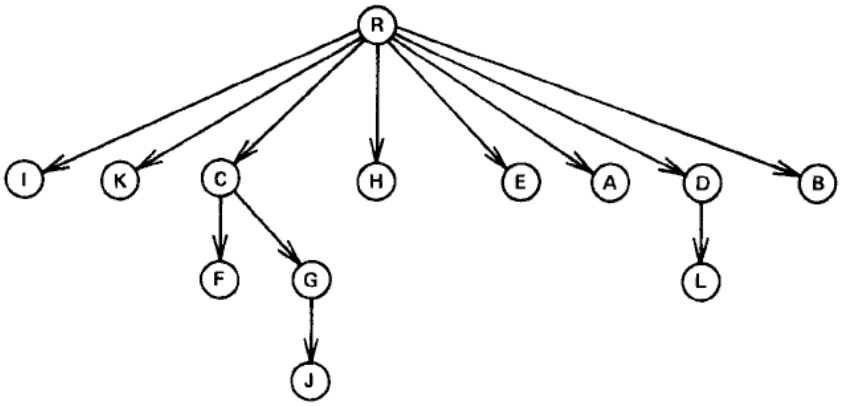
Először miután megkaptuk a kódmintát (a forrásfájlt, vagy a már lefordított bájtkódot, ez most számunkra mindegy), ki kell alakítanunk az arra jellemző folyamat gráfot, amivel a program futását tudjuk nyomon követni. Minden metódusnak külön gráf készül, amelynek csomópontjai az elágazást nem tartalmazó program részletek, amik közötti kapcsolatokat a feltételes vagy feltétel nélküli elágazások határoznak meg.

Az így kapott gráf után máris végrehajthatunk egy optimalizálást: ha esetleg fény derül meg nem hívott kódrészletre, akkor azt nyugodtan kihagyhatjuk a további feldolgozásból.



8. Folyamat gráf

Ha kezünkben a folyamat gráfunk, a további műveletek hatékony elvégzése érdekében ét kell alakítanunk egy SSA (Static Single Assignment) alakú gráfra, amelynek lényege, hogy az uralkodó (dominator) részeket kiszűrje, és azok segítségével feltárhatóak legyenek az optimalizálási lehetőségek az eredeti működést megőrzésével.



9. Static Single Assignment gráf

Az optimalizálások másik fajtája a magas nyelvekre jellemző, ezek közül csak röviden néhány:

- foreach utasítás: ezen utasítás használatával végigléphetünk egy lista minden elemén, és minden elemen egy adott programkódot futtathatunk.

Előnye, hogy a fordítónk döntheti el, hogy egy-egy ciklusban hány elemet „hív le”, illetve, hogy hol helyezkedjen el a ciklusváltozó a memóriában. Nem hinnénk, de jelentős gyorsulást tapasztalhatunk, ha ezeket a döntéseket jól hozza meg (helyettünk) a fordító. Érthető okokból sajnos a foreach ciklusokat nem lehet visszaállítani, csak egy, azzal funkcionalitásában azonos for ciklussal helyettesíteni.

- lista .Length tulajdonsága: a keretrendszerben található listák length tulajdonságát érdemes használni és nem érdemes lokális változóban külön tárolni, mivel ezen is végrehajtható néhány gyorsítás.
- ideiglenes változók: a fordító kiszűrheti a nem használt változókat, vagy több azonos típusú esetén újrafelhasználással megtakaríthatja az újbóli allokáció időveszteségét.
- method inline: bizonyos, általában kis terjedelmű metódusok sokkal hatékonyabbak lehetnek, ha a fordító minden meghívási pontra bemásolja a metódus testét, amivel a metódushívás költségét takaríthatjuk meg. Mivel az ilyen metódust tartalmazó szerelvényben semmilyen információt nem találhatunk, ami arra utalna, hogy egyes kódrészek valaha külön metódusban lettek volna, ezért lehetetlenség azokat metódusokként visszafejteni.

A fenti példákból látható, hogy a program írója is rengeteget segítheti a hatékony kód megszületését azzal, ha hagyja a fordítót hatékonyan dolgozni.

Visszafejtés szempontjából mindkét optimalizációs típust jó ismerni, így fel tudjuk készíteni a nem szokványos megoldások felismerésére, valamit fontos kiemelni, hogy sajnos ezen eljárások következtében lesz olyan kód, amit nem lehet teljesen az eredeti formájában visszaállítani, hiszen a szerelvénybe sem a megírt formában kerül.

4.2 Obfuscatorok, zagyválók

Az obfuscatorok célja pontosan az, hogy a hozzám hasonló emberek dolgát, pontosabban a kódvisszafejtést megnehezítse vagy lehetőség szerint teljesen megakadályozza. Erre a célra, mostanra kialakultak a bevált algoritmusok, nagy újításra sajnos már nincs remény. Azonban ezekre érdemes egy pár szót szentelni, mivel jobb tisztában lenni az ellenség gyenge pontjaival.

Még mielőtt azonban belevágnék a technikai tárgyalásba, fontos tudnunk, hogy minden ilyen jellegű eljárás valahol visszarúg, vagy a programunk rugalmasságából vagy sebességéből veszítünk, sőt drasztikus beavatkozás esetén akár a platform-függetlenséget is elveszíthetjük. Ezt mindenképpen érdemes elkerülni, ugyanis így a keretrendszer egyik legfontosabb tulajdonságától sikerülne megfosztanunk.

Nagyon fontos szem előtt tartanunk azt a tényt, hogy ezek a zagyválók nem módosíthatnak semmilyen publikus, külső forrásból elérhető metódusnevet, vagy adatot, mivel akkor az a módosítás után a külső forrás számára nem lenne elérhető. Ezzel látszólag jelentősen korlátozzuk a zagyváló lehetőségeit, aminek a kódvisszafejthetőségre gyakorolt hatásáról hamarosan, a V-Spot-okról szóló fejezetben tudhatunk meg többet.

4.2.1 Metaadat zagyválás

Egy .NET-es szerelvényben (ahogy klasszikus futtatható állományokban is) a futtatható kód mellett rengeteg, ahhoz kapcsolódó információt tárolhatunk egy PE-ben, egy fájlban. Ahogy azt fentebb a szabványos exe felépítésének tárgyalásakor láthattuk, bizonyos szegmensem az állományon belül pontosan ezen adatok fenntartására vannak kitalálva.

Ezek a blokkok bármilyen adatot tartalmazhatnak, azonban a gyakorlatban leggyakrabban előforduló felhasználási területek a következők: az állomány verziószáma, az igényelt minimális keretrendszer verzió, gyorsító billentyűk, struktúrált kivételkezelés try-catch-finally blokkhatárainak meghatározása, fordító programok lenyomatai, esetleg fordítási direktívái, debug (hibajavítási) információk tárolása, várható memória igény felmérés adatai, ikonok, képek, a dialógus ablakok elrendezési beállításai, legördülő menük tartalma, viselkedésüket meghatározó attribútumok, az

osztályok runtime viselkedését módosító .net attribútumok (például a web szolgáltatásoknál használt [WebMethod] vagy a zászló típusú enumerációknál használt [Flags]), biztonsági és hozzáférési szabályok és még rengeteg létfontosságú adat.

A metaadatok zagyválásánál célunk, hogy a visszafejtő személy a lehető legkevesebb információt tudja kinyerni a szerelvényünkben található meta-információkból, anélkül, hogy a működésben bármilyen változást okoznánk.

Ha megnézzük a fentebb felsorolt alkalmazási területeket, akkor látható, hogy csak minimális lehetőségünk marad, a gyakorlatban is csak egy metaadat táblát szokás összezavarni, ez pedig a lokalizálásnál használt szöveg tábla. Itt minden azonosító mellé egy a helyi beállításokhoz megfelelő szöveg társul.

Természetesen a szöveget nem lehet összekeverni, ezért maradnak az azonosítók. Az azonosítók a fejlesztés folyamata közben célszerűen emberi nyelven is értelmezhető rövidítések. A programkódban ezekre hivatkozunk, majd a végrehajtó motor (CLR) az azonosító helyére az éppen megfelelő nyelvű szöveget helyettesíti. Ha az azonosító emberi nyelven nem értelmezhető, mondjuk egy sorszám, akkor ezzel valóban zavarosabb lesz a visszafejtett kód.

Ezzel azonban több gond is van. Mint az ismeretes, a .NET keretrendszerben komoly támogatást élvez a lokalizálás. Satellite assemblyket (lokalizációs szerelvényeket) hozhatunk létre, amelyek nem tartalmaznak futtatható kódot, csak metaadatot, pontosan az előbb említett azonosító-szöveg táblát. Így lehetővé válik egy szoftver nyelvi csomagjának külön terjesztése, amivel nemcsak adatforgalmat spórolunk meg, hanem fenntartjuk annak lehetőségét, hogy a kiadás után programunkat újabb nyelvekre is átírjuk az eredeti program módosítása nélkül. És pontosan itt a bökkenő.

Tegyük fel, hogy van egy szoftver, amit elkészítettünk 4 nyelven, 3 satellite assembly segítségével. Használjuk a zagyválót. Az azonosítók mind az ős állományban mind a 3 nyelvi szerelvényben helyesen megváltoznak. Azonban ha egy újabb honosítást szeretnénk publikálni, sajnos azzal kell szembenéznünk, hogy sorszámozott, nehezen átlátható azonosítókhoz kell különböző szövegeket rendelnünk. A dolog nem kivitelezhetetlen, de jelentősen megnövelheti a fordítók munkaidejét, mivel a visszakeresés jóval több időt vehet igénybe.

Összességében ez nem egy hatékony módszer, ezért csak akkor használjuk, ha nincs honosítási támogatásunk.

Kijátszására nincs automatizálható módszer.

4.2.2 String védelem

A legegyszerűbb „programtörést” már sokan végrehajtottuk: akár notepaddel megnyitva egy tetszőleges programot, kiolvashatók a programban felbukkanó feliratok, üzenetek. Ha ezeket az állományban módosítjuk, a legközelebbi indításkor, ha ismét felbukkan a szöveg, akkor a módosított állapotban fog megjelenni.

Egy cracker kezében ez nagyon erős fegyver lehet, ha tudja használni. Képzeljük el, hogy pénzt szeretnénk kérni a felhasználóinktól, amihez regisztrációra akarjuk késztetni őket. A programunkhoz kitalálunk egy jelszót, amit ha helyesen ütünk be, akkor a programunk teljes funkcionalitását használni tudja vevőnk, ellenkező esetben nem. A kódot a felhasználóhoz csak a vételár kézhez kapása után áruljuk el.

A rendszer működéséhez szükség lesz a programunkon belül egy olyan űrlapra, ahol a felszabadító kódot be tudja ütni a kedves ügyfél. Hibás kód beütése esetén visszajelzésként közölnünk kell, hogy sajnos nem jó a jelszó. És pontosan ezen a ponton tud a rafinált cracker egy kapaszkodót találni. Nincs más dolga, mint hogy megnézi mi az üzenet, kikeresi annak helyét az állományban, megfelelő kódvisszafejtővel készít egy listát az arra stringre való utalásokról, majd szépen végigböngészi, hogy vajon melyik az a metódus, ahol a jelszó ellenőrzése történik. Egy apró módosítással a hibás jelszó is elfogadhatóvá válik.

Az érezhető, hogy itt a legnagyobb gond, hogy sajnos a szövegek könnyen olvasható formában vannak jelen a futtaható állományban. Célunk tehát ezek kódolása. Ha viszont kódolunk, akkor minden fordításkor az összes szöveget kriptografálni, majd minden egyes használat előtt dekriptografálni kell. Ez bizonyos esetben jelentős számolási igényű is lehet, azonban valóban elfogadható védelmet nyújt. A zagyválók általában támogatják ezt a fajta védekezést, aminek hatására két metódussal bővül és lassul ugyan a programunk, de mégis általános esetben érdemes használni. Lokalizálásnál ugyan jelent plusz munkát, de kevésbé jelentős, mint a metaadat zagyválásnál.

Ez a két módszer egymás mellett is kitűnően használható, sőt ha megoldható, javasolt.

Kijátszására elképzelhető lenne automatikus módszer, minden stringet visszakódolni a belefordított dekódoló metódussal, majd minden dekódoló metódushívást kivenni az állományból.

4.2.3 Control Flow megkavarása

Elérkeztünk a talán legfontosabb és legbonyolultabb zagyválási módszerhez. A dolog lényege, hogy változtassuk olyan szokatlan módon meg az IL utasítások sorrendjét vagy magukat az utasításokat is, hogy a működés helyessége megtartása mellett olyan minták alakuljanak ki, amelyeket a kódvisszafejtők nem ismernek.

Gyakori eset például, hogy a decompilerek nem az általam használt robosztus módszerrel ismerik fel az if-then-else klózt, hanem a kódvisszafejtő készítője megfigyeli több fordító által készített kódot, és arra ír egy-egy felismerő algoritmust, majd megadott eljárással kikeresi az igaz, hamis ágakat, stb.. Az ilyen módszerrel azonban el lehet bánni, mivel ha a zagyváló például egy egyszerű ugró utasítással áthelyezi az ágak törzsét más helyre, akkor ha a kódvisszafejtő nincs kellően felkészítve, nem fogja tudni kezelni a váratlan helyzetet.

A lehetőségek sokaságának szemléltetéséhez elég abba belegondolnunk, hogy IL kóddal hány módszer létezhet egy változó nullázására (ld, xor, pointer mozgatás, stb.).

Az ilyen jellegű módosításokra nagyon nehéz felkészülni, mivel egyetlen megkötést jelentő dokumentum, amire támaszkodhatunk a CLR (Common Language Runtime) szabvány. Pontosan ezért, célom, hogy a stack emuláció tökéletesítésével a visszafejtés folyamán bármely lépést követően a virtuális stack állapota tökéletesen egyezzen meg azzal, amit futtatás közben tapasztalhatunk, ha belenézünk a CLR belső működésébe. Mivel jelenleg a motor virtuális stackje CodeDom objektumokkal működik, és azok bizonyos esetekben nem hordoznak magukban elegendő információt, ezért a CodeDom kiterjesztése különösen nagy szerepet kap.

Azonban, ha minden kiterjesztés elkészül, és az emuláció is tökéletes lesz, akkor is lehetnek olyan gondok, hogy bizonyos IL utasítássorok nem kifejezhetők C# kóddal. Az ilyen jellegű felismerésekre nincs más megoldás, mint a C# kódba megjegyzésként való IL kód beültetés. Ha valaki az ilyen forráskódot ismét fordítani szeretné, akkor kénytelen lesz az egészet IL kódra visszafejteni, majd ott módosítani, és fordítani. Ez a nehézség nem a motor gyengeségéből, hanem az IL kód természetéből adódik. Megnyugtató lehet azonban a tudat, hogy ilyen kóddal rendkívül nehéz találkozni, ugyanis ezek a kódok biztosan nem az elterjedt .NET keretrendszer által támogatott magas szintű nyelveken készültek.

4.2.4 Átnevezés

Ahogy azt a string-táblák összezavarásánál is láthattuk, bizonyos esetben hatásos lehet, ha a fejlesztő által értelmesen elnevezett egységeket átnevezzük megtévesztő, vagy legalább nehezebben követhető megnevezésekre, természetesen ismét figyelve arra, hogy a funkcionalitás ne változzon.

Általános irányelv ilyen esetben, hogy minden változónevet, metódusnevet, osztálynevet, interfész megnevezést és névteret nevezzünk át, ami nem publikus, vagy csak ebből az adott szerelvényből elérhető.

Megoldható az is, hogy a névtér hierarchia teljesen eltűnjön, a névtér nevek úgyszintén megszűnjenek, az osztályok pedig pár betűs értelmetlen betűsorozatok legyenek, úgy, hogy minden metódusok ugyancsak pár betűs értelmetlen betűsor. A kitartó cracker azonban még ilyenkor sem esik pánikba, ugyanis vannak dolgok, amiket nem lehet megváltoztatni, átnevezni. Ilyenek például az olyan osztályok ősosztályainak megnevezései, amik a szerelvényen kívül helyezkednek el. Tehát minden, a Framework által nyújtott és a programunkban felhasznált alaposztály neve tisztán kiolvasható lesz, még zagyválás után is. Még rosszabb helyzet az általunk meghívott külső metódusok esetében, ugyanis akkor minden elnevezésnek maradnia kell; a névtér, osztály és metódusnévnek helyesen kell szerepelnie. Ezek a legjobban használható támpontok egy kódtörő számára, és egyelőre nincs is rá esély, hogy ezeket valamilyen módon kikerüljük.

Mi csupán annyit tudunk tenni, ha elsődleges célunk a visszafejthetőség megakadályozása, hogy a lehető legkevesebb külső hívást indítunk programunkból (főleg a kritikus részeken), illetve a jól tagoltságból, könnyű frissíthetőségből és karbantartásból áldozva, megpróbáljuk szoftverünket minél kevesebb szerelvényben tálalni.

Az ilyen „kapaszkodókat” az irodalom V-Spotként emlegeti, ami a Vulnerable Spot, sebezhetőségi pont rövidítése.

4.2.5 V-Spotok eltüntetése

Az egyik obfuscator készítő cég honlapján olvashatjuk, hogy ők rájöttek, hogyan lehet az ilyen jellegű problémákat megoldani. Természetesen a részletekbe nem avatnak be bennünket, mivel ők maguk is hangsúlyozzák, hogy ilyet még semelyik hasonló

termékben nem láthattunk, ők lennének a megváltók. A technológiájuk sajnos még béta tesztelés alatt van, és semmilyen példaprogramot nem bocsátanak rendelkezésünkre.

Azonban érdemes kicsit elidőzni a probléma felett, ugyanis, mint ahogy azt az előzőekben láthattuk, ennek megszüntetése jelentősen közelebb vihetne minket a megtörhetetlen szoftver megvalósításához.

Tehát még egyszer, a cél a következő: a szerelvényünkben ne legyen hivatkozás a jól ismert, keretrendszer által nyújtott metódusokra, sőt lehetőleg a szerelvényekre se, amellett tartsuk meg az eredeti funkcionalitást. Egyetlen dolog, amin „ronthatunk" a teljesítmény. A megoldást én egy rafinált megoldásban látom:

Dolgozzuk fel a szerelvényünket/szerelvényeinket, szedjük csokorba az összes olyan hivatkozást, ami támadható lehet.

Mindet nevezzük át, egy véletlenszerűen generált zavaros karaktersorra.

Készítsünk egy újabb szerelvényt, amelybe az összes zavaros karaktersornak megfelelő névvel egy-egy metódust helyezünk.

Minden ilyen metódusnak a törzsébe egyetlen hívást tegyünk, és annak visszatérési értékével térjünk vissza.

Az eredeti szerelvény(ek)ben lévő minden keretrendszer részeként említhető referenciát helyezzünk át az új (délibáb) szerelvényünkbe.

És íme a végeredmény: egyetlen szerelvénnyel bővült a szoftverünk, és minden támadható pont (V-Spot) ebben az egy szerelvényben található. A rosszindulatú felhasználónak jelentősen több idejébe fog kerülni bármely pont helyeinek megtalálása, hacsak nem használ valami automatizált módszert, azonban ilyen szoftver jelenleg nem elérhető.

Ahogy azt a fejezet elején is említettem, ez egy saját gondolat, egyáltalán nem biztos, hogy a fenti cég ezt a kérdést így oldotta vagy más ezt így oldaná meg.

4.3 Erős névvel ellátott szerelvények és az „Authenticode” eljárás

Miután lefordítunk egy szerelvényt, és az már készen áll a publikálásra, fontos megbizonyosodnunk arról, hogy azt nem fogja tudni senki megváltoztatni, vagy ha esetleg mégis, akkor arra a futtató környezet még idejében rájön.

Jól bevált módszer az érzékeny adatok hash lenyomattal való ellátása, amit a hálózati technológiák is előszeretettel használnak, mivel számítása gyors és igen nagy valószínűséggel megállapítható, ha a kérdéses információ tömb megsérült. A dolog lényege, hogy miután elkészült a bájt folyam (esetünkben a lefordított szerelvény), kiszámolunk egy fix hosszúságú értéket, ami erre a bájt tömbre jellemző. A jó hash függvény jellemzője, hogy gyorsan számítható, kis Hamming-távolságú folyamokra nagy Hamming-távolságú lenyomati értéket ad vissza és kellően hosszú ahhoz, hogy „Brute Force” (nyers erő) típusú támadás ellen megvédjen, azaz találgatással rendkívül nehéz legyen két azonos lenyomatú folyamot találni. Lássuk, milyen támogatást nyújt a .NET Framework 1.1-es verziója a szerelvények lenyomattal való ellátásához.

Az ECMA-335-ös szabványa alapján a keretrendszer két hashelési algoritmust támogat az MD5-öt és az SHA-1-et. Ezek közismertek, működésükre nem térnék ki, azonban mindkettő megfelel a fenti feltételeknek, azaz jól használható hash függvények. Ha szeretnénk a szerelvényünk megfelelő mezőjét egy értékkel kitölteni, akkor először a Framework SDK egyik programját, nevezetesen a Strong Name (sn.exe) programocskát kell használnunk, hogy generáljunk egy publikus – privát kulcs párost, amelyek 1024 bitesek jelenleg az 1.1-es verzióban. Miután ez megvan, nincs más dolgunk, mint jelezni a fordítónak, hogy ezen kulcsok alapján lássa el erős névvel a szerelvényt, ami magában hordozza a hash lenyomat tárolását is. Jelenleg (a keretrendszer 1.1-es verziójában) nincs lehetőségünk választani, a fordító mindenképpen SHA-1 lenyomatot (160 bit) készít.

Felmerülhet bennünk a kérdés: mit tehetünk, ha szeretnénk erős névvel ellátni a szerelvényünket, de fordítás után még szükségünk lenne annak módosíthatóságára? Pontosan ez a gond merül fel egy kódzagyváló esetében is. Ekkor a „Delay Singing” (halasztott aláírás) nevű módszert kell alkalmaznunk. Ekkor az történik, hogy a fordító nem generálja az erős nevet, azonban a publikus kulcsot belefordítja a szerelvénybe. Így

miután lefordítottuk, majd módosítottuk programunkat, sn.exe megfelelő (-R) paraméterezésével véglegesen elláthatjuk erős névvel. Arra azonban ügyelni kell, hogy ha szeretnénk a fordítás után, de még aláírás előtt használni a szerelvényt, meg kell mondanunk a keretrendszernek, hogy a mi programunkat hash lenyomatát ne ellenőrizze, mivel ellenkező esetben (jogosan) hibát fog jelezni, és megakadályozza a futtatást. Ezt a „–Vr „ argumentummal tehetjük meg.

Az erős névvel való ellátásnak csupán két hátulütője van: többlet adminisztrációt igényel, illetve nem lehet csak erős névvel ellátott szerelvényekre hivatkozni.

Ezzel megtettünk mindent, amivel a saját kódunk módosíthatóságát megnehezíthettük, így azonban még mindig nem lehet a felhasználónk teljes biztonságban, ugyanis hiába tudja azt, hogy az állományt nem módosították a készítése óta, ha nem lehet abban biztos, hogy a kód tőlünk, a fejlesztő cégtől származik. Erre megoldás az „Authenticode” technológia.

Az óriási különbség az Authenticode-ot használó és nem használó szerelvények között, hogy míg az utóbbiak tetszőleges generált kulcsot használhatnak az aláírásuk generálásához, addig az előbbiek mögött egy hierarchikus szervezet csoport áll, amely a megbízható szervezeteknek, cégeknek biztosít egyedi kulcs párost, amivel garantálható, hogy ha a szerelvény a megnevezett forrásból származik. Így a felhasználó, aki üzemeltetni kívánja a szoftvert, eldöntheti, hogy mely cégekben bízik meg, melyekben nem. Másik előnye, hogy a cég bizonyítványa a PE fejlécben helyezkedik el, így még futtatás előtt, sőt a letöltés megkezdése után közvetlenül megállítható a folyamat, ha a forrás nem megbízható.

4.4 Biztonsági megfontolások, tanácsok

Először is, lehetőleg szokjunk rá a jól optimalizálható programozásra. Kódunk így gyorsabb és biztonságosabb lehet, csupán azért, mert bizonyos rossz szokásainkat jókra cseréljük.

Másodszor, ha tehetjük, minden nyilvánosan elérhető kódot írjunk alá erős névvel. Semmilyen hátrányunk nem származik belőle, és kód ellenőrizhetetlen módosíthatóságát jelentősen megnehezítjük.

Harmadszor, használjunk zagyválókat. Mint láthattuk már jól működő programokat készítettek, amikkel (részleges) védelmet tudunk biztosítani programunk

számára. Nagyon fontos tudnunk, hogy létezik olyan gyártó is, aki ingyenesen biztosít bizonyos funkciókat a fentiek közül.

5 A .NET Framework 2.0

Ezen dokumentum megírása idején már elérhető a fejlesztők számára a keretrendszer következő generációjának, a Whidbeynek a béta 2-es verziója. Az új keretrendszerrel együtt fog napvilágot látni a C# 2.0 és néhány változás a CLR-t is érinti, amire a kódvisszafejtőnek is fel kell készülnie. Olyan funkció, ami jelentősen befolyásolhatja a visszafejtés folyamatát a paraméterezett polimorfizmus és a névnélküli metódusok bevezetése.

5.1 Generics

Mindannyian emlékezhetünk a C++-ban használt template-ekre. Amikor megjelent a .NET Framework, akkor nagyon sokan nagy hiányosságának tartották, hogy ahhoz hasonló szerkezet építésére nem volt lehetőség. Valóban, rendkívül hasznos volt C++-ban, hogy konkrét típus meghatározása nélkül képesek voltunk rendkívül hatékony szerkezeteket építeni, majd futási időben meghatározni, hogy milyen típusú objektumokból szeretnénk felépíteni az adott logikai szerkezetet.

Most, hogy a beleláthatunk a keretrendszer következő generációjába, láthatjuk, hogy egy nagyon hasonló funkcionalitást megvalósító szolgáltatással bővült a Framework, aminek neve Generics (vagy paraméterezett polimorfizmus). Ezzel készíthetünk típus nélküli szerkezeteket, későbbi típus meghatározással, mindezt különösebb overhead nélkül.

Egy ilyen szerkezetre tipikus példa egy láncolt lista, egy bináris fa vagy tetszőleges gráf.

Nézzük meg, milyen különbség lesz az IL kódban az objektum alapú és a genereics megközelítés között.

Object-based stack

```
.class Stack {
  .field private class System.Object[] store
  .field private int32 size
  .method public void .ctor() {
    ldarg.0
    call void System.Object::.ctor()
    ldarg.0
    ldc.i4 10
    newarr System.Object
    stfld class System.Object[] Stack::store
    ldarg.0
    ldc.i4 0
    stfld int32 Stack::size
    ret
  }
  .method public void Push(class System.Object x) {
    .maxstack 4
    .locals (class System.Object[], int32)
        :
    ldarg.0
    ldfld class System.Object[] Stack::store
    ldarg.0
    dup
    ldfld int32 Stack::size
    dup
    stloc.1
    ldc.i4 1
    add
    stfld int32 Stack::size
    ldloc.1
    ldarg.1
    stelem.ref
    ret
  }
  .method public class System.Object Pop() {
    .maxstack 4
    ldarg.0
    ldfld class System.Object[] Stack::store
    ldarg.0
    dup
    ldfld int32 Stack::size
    ldc.i4 1
    sub
    dup
    stfld int32 Stack::size
    ldelem.ref
    ret
  }
  .method public static void Main() {
    .entrypoint
    .maxstack 3
    .locals (class Stack)
    newobj void Stack::.ctor()
    stloc.0
    ldloc.0
    ldc.i4 17
    box System.Int32
    call instance void Stack::Push(class System.Object)
    ldloc.0
    call instance class System.Object Stack::Pop()
    unbox System.Int32
    ldind.i4
    ldc.i4 17
    ceq
    call void System.Console::WriteLine(bool)
    ret
} }
```

Generic stack

```
.class Stack<T> {
  .field private !0[] store
  .field private int32 size
  .method public void .ctor() {
    ldarg.0
    call void System.Object::.ctor()
    ldarg.0
    ldc.i4 10
    newarr !0
    stfld !0[] Stack<!0>::store
    ldarg.0
    ldc.i4 0
    stfld int32 Stack<!0>::size
    ret
  }
  .method public void Push(!0 x) {
    .maxstack  4
    .locals (!0[], int32)
        :
    ldarg.0
    ldfld !0[] Stack<!0>::store
    ldarg.0
    dup
    ldfld int32 Stack<!0>::size
    dup
    stloc.1
    ldc.i4 1
    add
    stfld int32 Stack<!0>::size
    ldloc.1
    ldarg.1
    stelem.any !0
    ret
  }
  .method public !0 Pop() {
    .maxstack 4
    ldarg.0
    ldfld !0[] Stack<!0>::store
    ldarg.0
    dup
    ldfld int32 Stack<!0>::size
    ldc.i4 1
    sub
    dup
    stfld int32 Stack<!0>::size
    ldelem.any !0
    ret
  }
  .method public static void Main() {
    .entrypoint
    .maxstack 3
    .locals (class Stack<int32>)
    newobj void Stack<int32>::.ctor()
    stloc.0
    ldloc.0
    ldc.i4 17

    call instance void Stack<int32>::Push(!0)
    ldloc.0
    call instance !0 Stack<int32>::Pop()


    ldc.i4 17
    ceq
    call void System.Console::WriteLine(bool)
    ret
} }
```

A jobb oldali oszlopban aláhúzással jelöltem az eltéréseket. Ahogy az látható, a keretrendszer szerencsére sok alapjában véve is polimorfikus utasítást tartalmazott (pl. ldarg, starg), ahol nem változott az utasítás az operandus típusától függően.

Az utasítások egy másik csoportja operandusként kapta meg egy másik operandus típusát (pl. ldfld, stfld), ezeknél azonban figyelni kell majd, hiszen, mivel nem tudjuk fordításkor, hogy milyen típusú lesz az adott argumentum, ezt helyettesítenünk kell valamivel, amivel jelezni tudjuk a CLR-nek, hogy ide a megfelelő

típust be kell helyettesíteni. Erre az IL-ben a „!0” konstanst kell használni. Ahol az osztályváltozót kell használnunk, ott azt az Osztálynév<!0> módon tudjuk helyettesíteni.

Léteznek azonban olyan utasítok is, amelyeknél az utasítás maga determinálja, hogy milyen típusú kell, hogy legyen az operandus (pl. ldelem.i4, stelem.i4). Ezek például tömbök használatánál kerülnek alkalmazásra. Erre az esetre kellett bevezetni két új utasítást: az ldelem.any és az stelem.any IL utasításokat, amikkel már nem csak referencia típusok tölthetők. Ennek hatására megspórolható a külön box és unbox utasítás.

A keretrendszer működését tekintve, amikor egy polimorfizmusra épülő osztállyal találkozik, akkor annak vázát betölti a memóriába, majd lockolja azt, mindaddig, amíg kérés nem érkezik a példányosítására. Ahogy érkezik egy kérés, ellenőrzi, hogy érkezett-e már ezelőtt ilyen paraméterrel régebben példányosítási kérelem, ha nem, akkor a JIT-nek átadja az osztályhoz tartozó IL kódot és metaadatokat a típust azonosító paraméterrel együtt. A JIT elkészíti a példányt, majd visszaadja annak referenciáját. Érdekesség azonban, hogy ha referencia típusú változóval kérelmezünk egy újabb osztályt, akkor az előzőt csak lemásolja, nem generál újabb osztályt. Ennek magyarázata az, hogy mivel minden referencia típus natív megfelelője egy mutató, így teljesen felesleges lenne újból és újból generálni egy-egy olyan osztályt, amelynek ugyancsak mutató a polimorfikus változó típusa. Értelemszerűen érték típusoknál nem tehetjük ezt meg, hanem kénytelenek vagyunk mindig egy-egy újabb osztályt generálni.

A fentieket látva a visszafejtő motoron a kisebb módosítások mellett főleg kiterjesztésre lesz majd szükség.

5.2 Anonymous methods

Eddig a .NET-es nyelvekben egyértelműen elkülönültek a metódusok egymástól, azonban ez mostantól nem feltétlenül lesz így. A C# 2.0-ban lehetőségünk van metódust építeni egy metódusban. Ennek célja természetesen nem a bonyolítás, hanem a kód jobb átláthatósága. Az ilyen beültetett metódust hívjuk névnélküli vagy anonymous metódusnak. Azonban figyeljünk arra, hogy csak tényleg akkor használjuk, ha meglehetősen rövid a metódus teste.

Lássunk egy példát:

```
public partial class Form1 : Form
{
  public Form1()
  {
    InitializeComponent();

    button1.Click += delegate(object sender, EventArgs e)
    {
      // The following code is part of an anonymous method.
      MessageBox.Show("You clicked the button, and " +
        "This is an anonymous method!");
    };
  }
}
```

Tipikusan egy delegate metódus teste csak egy-két hívásból áll, azért ilyen esetekben kitűnően használható. A C# fordító a szituációtól függően többféleképpen kezelheti a névtelen metódust.

A fenti esetben a metódus egy a Form1-en belüli privát metódusként fog megjelenni, így hozzáfér az osztály publikus és privát változóihoz is.

Az érdekesség a dologban az, hogy ráadásként elérhetjük a metódusunk (jelen esetben a konstruktor) lokális változóit is. Ebben az esetben azonban nem lesz megfelelő a fenti konstrukció (miszerint egy privát metódust hozunk létre az osztályon belül), ezért erre más megoldást kellett találni.

A megoldás, hogy készítsünk egy osztályt az eredeti (Form1) osztályon belül, valamint kiegészíti az adott metódus (Form1.ctor) lokális változóival.

Sajnos a módszerből látszik, hogy nagy valószínűséggel a fordítás után már nem lesz egyértelműen meghatározható, hogy vajon névtelen metódusról van-e szó. Egyedül abban az esetben lehetne biztosra menni, ha valamilyen attribútummal jelezné a fordító az anonim metódusokat, azonban erre utaló nyomot a beta 2-es verzióban nem találtam.

6 Aspektus orientált programozás

Napjainkban kellő népszerűségnek örvend az aspektus orientált programozás, azonban a szakmai körökben is sajnos előfordul egy kisebb fogalomzavar, amit tisztáznunk kell. Az aspektus orientált programozás nem egyezik meg az attribútum orientált programozással. Az attribútum orientált programozás csak egy fajta megvalósítása lehet az aspektus orientált programozásnak, sőt, egyesek szerint nem is valódi megoldás, mivel mérsékeltebbek a lehetőségek egy attribútumos megoldás esetében, mint amire az aspektus orientált programozás hivatott. Ez a gyakorlatban annyit jelent, hogy bizonyos osztályokat vagy metódusokat megjelölünk egy adott attribútummal, aminek hatására azok kiegészülnek az annak megfelelő aspektushoz tartozó programrészekkel.

Nem titkolt célom a későbbiekben a kódvisszafejtő ilyen irányú fejlesztése. Így lehetővé válna az, hogy az aspektusok alkalmazási helyeit még fejlesztési időben megjelölnék, majd fordítás után, attól függően, hogy mely aspektusoknak szeretnénk eleget tenni, a már lefordított állományt lehetne úgy módosítani, hogy megfeleljen az egyes aspektusok követelményeinek.

A megvalósítás látszólag nem nehéz, ha már kezünkben van egy írni és olvasni tudó kódvisszafejtő, hiszen nincs más dolgunk, mint a metaadatokból kiolvasni az aspektusokat jelölő attribútumok helyét, feldolgozni az adott kódrészt, megkeresni az aspektus kódrészleteinek helyeit az osztályban vagy metódusban, végrehajtani a módosításokat, majd kiírni az új állományt. Természetesen itt is felmerül az aspektus orientált programozás egyik nehéz kérdése: milyen nyelvvel lehetne hatékonyan leírni azt, hogy pontosan hova kell beszúrnunk azokat a bizonyos kódrészeket. Naplózásnál például egyszerű, mivel mondjuk minden metódus elejére és végére való beszúrás kielégítő lehet, de számtalan ennél jóval bonyolultabb szituáció is elképzelhető, amihez komoly leíró nyelvre lehet szükség.

Kérdés leginkább az, hogy érdemes-e? A piacon már most is vannak hasonló funkciókat megvalósító .NET-es csomagok, bár igaz, ezek főleg a dinamikus implementációt követik, azaz, nem a szerelvény módosításával ültetik bele a megfelelő kódrészleteket a futtatandó kódba, hanem a JIT fordító elindulásakor egészítik ki a szerelvényben található IL kódot. Természetesen ez jelentős overheaddel jár, de

jelentősen egyszerűbb implementálni, mint az általam megcélzott statikus megvalósítást.

A másik járható út, hogy az aspektus orientált kiterjesztést a kódszerkesztőbe építik be, egy Visual Studio plug-in formájában. Erre is létezik már megvalósítás, bár ennek funkcionalitása minimális, nem is igazán nevezhető az aspektus orientált programozás implementálásának.

7 Piaci versenytársak

A talán (méltán) legnagyobb figyelmet kapó kódvisszafejtő a Lutz Röder által készített Reflector, amely ugyancsak képes IL szerelvényekből magas szintű C# forráskódot előállítani. Ezen program méltó vetélytársa lehetne szoftverünknek, abban az esetben, ha célunk csupán egy újabb kódvisszafejtő elkészítése lenne.

Fontos megjegyeznünk, hogy a Reflector nem képes szerelvények írására, módosítására. Ebben kívánunk a kutatásunkkal előnyt nyerni, illetve rendkívül fontos kiemelnünk a végső céljaink közötti eltéréseket. Erre egy külön fejezetben térnék ki a későbbiekben.

Lássuk azonban azt, hogy miben különbözik a két motor működése. Mivel az általam fejlesztett modul folyamatosan épül és kiegészül, lehetetlen dolog lenne egy általános összehasonlítást készíteni. A tanszékünkön fejlesztett szoftver jelenlegi (2005. március 10.) állapotát tükrözve a fontos különbségek a következők:

Nagyobb utasítás lefedettség a Röder-féle motorban: mivel elsődlegesen kutatási jelleggel építem a visszafejtőt, így (a hosszú távú célokkal ellentétben) jelenleg nem a teljes utasítás arzenál támogatása a fontos, hanem, hogy a C# nyelven írt, majd IL kódra fordított eljárások döntő hányadát gond nélkül sikerüljön feldolgozni. Ezzel megengedjük magunknak azt a könnyedséget, hogy nem kell „natív jellegű”, közvetlen memória menedzsmenttel kapcsolatos utasításokkal foglalkoznunk, hiszen olyan IL utasítást egyetlen C# nyelven írt program sem fog generálni.

Felhasználóbarátabb felület: az idő szűke, és a funkcionalitás helyességének előtérbe helyezésével elfogadható hátrány, hogy kevésbé mutatós a felület, azonban minden alapvető funkcionalitás elérhető, visszajelzéseket küldünk a tesztelőnek, és alapvetően egy tesztelő kényelmét szolgáló funkciók elérhetőek. Hibajavítás segítségére is rengeteg apróság került a felületre, úgymint másolható kivétel ablak, verziószám nyilvántartás modulonként, stack-trace.

Írásra való felkészülés: a mi programunkat már a kezdetektől úgy terveztük, hogy később írásra is képes legyen. Minden modul, azok kapcsolódási interfészei, a közöttük zajló információs csatornák implementációja fenntartja annak lehetőségét,

hogy egy, az írásra is képes modullal egészüljön ki. Ebben a pontban jelentős előnnyel bírunk a Reflectorral szemben, hiszen az csakis olvasásra készült.

Webes felület: a versenytársunknak nevezett program csak WinForms felülettel rendelkezik, és látszólag nem is tervezi(k) a webes környezetre való kiterjesztést. Nálunk jelen pillanatban is folynak ez irányú fejlesztések. Ahhoz, hogy víziónk megvalósítható legyen, rendkívül fontos e funkció megfelelő megvalósítása.

A felhasználók összekapcsolása, a közös munka segítése: ugyancsak a mi fejlesztésünk nyeri a versenyt, ha a közös munkáról beszélünk. A jövőbeni terveink közt talán az egyik legfontosabb, hogy nyitunk a fejlesztői társadalom felé, és elsődleges célunk a fejlesztők munkájának olyan irányú könnyítése, hogy a megszerzett tudás valóban közös legyen. Ennek eléréséhez még rengeteg munka áll előttünk, de mivel az igények felmérésével teljes mértékben alátámasztottuk azon megérzésünket, hogy erre rendkívüli szükség lenne, időt és energiát nem sajnálva folytatjuk a fejlesztést célunk eléréséig.

Natív x86-os metódusok felismerése és diszasszemblálása: sok fejfájást okozhat egy kódvisszafejtőnek az, ha nem tisztán csak IL kódokkal kell dolgoznia. Ezt a kódzagyválók nagy százaléka ki is használja. Azonban mi nem szeretnénk ilyen jellegű gyenge pontot hagyni a motorban, így felkészülünk a x86-os assembly utasítások feldolgozására is. Természetes ezek magas szintű visszafejtésére nem vállalkozunk, hiszen egy ilyen művelet tökéletes végrehajtásához egy egész emberöltő sem lenne elegendő.

Az érezhető, hogy szoftverünket szándékosan máshogy pozícionáljuk, mint a piacon jelenleg megjelent programok. Mi nem kódvisszafejteni akarunk elsősorban, hanem a kódvisszafejtés felhasználásával mások munkáját jelentősen megkönnyíteni. Amíg egy kódvisszafejtővel kapcsolatban gyakran felmerülhet a legalitás, és a moralitás kérdése, így a mi programunk esetében ezek nem vitatható kérdések, ugyanis célunk egy hatékonyabb programozási módszertan kidolgozása erős szoftver-támogatással, ami minden .NET Framework-öt használó fejlesztő munkáját megkönnyítené.

8 A vízió

Ahogy a kutatás egyre jelentősebb méreteket öltött és egyre biztosabbá vált a kódvisszafejtő folyamatos fejlődése, szerencsére több segítőkész kollégát sikerült bevonni a fejlesztésbe. Ahogy nő a projekt egyre több és jobb ötlet születik, a továbbhaladási irányt illetően. A kínálkozó lehetőségek közül nem kis feladat kiválasztani azokat, amelyek később a gyakorlatban is használhatók lehetnek, és nem haladja meg csapatunk kitartását. Mostanra kirajzolódni látszik, hogy a pontos cél, amit mindannyian szeretnék.

Nagyon fontos, hogy tisztázzuk, hogy sok szempontból kivételesnek mondható ez a kutatás. Egyrészről egyetemi kereteken belül, egy, gyakorlatban is azonnal hasznosítható szoftvert gyártunk, ami akár kisebb többlet ráfordítással az iparban is teljes körűen megállná a helyét. Másrészről egyetemisták fejlesztik pusztán lelkesedésből. A mai anyagias világban ilyen színvonalú munkát rendkívül kevesen hajlandóak produkálni bármiféle anyagi juttatás nélkül.

A célok megfogalmazásakor elsődlegesen azokat a szempontokat tartottuk szem előtt, amelyek formáló erővel bírhatnak a jövő programozási szokásaira nézve. Ahhoz, hogy ez teljes egészében ismertethető legyen, néhány fogalom tisztázására ki kell térnünk.

XML: eXtensible Markup Language, a W3C konzorcium egyik szabványa. Az SGML nyelvezet egy leegyszerűsített változata, külön a webes dokumentumokhoz kialakítva. Rengeteg olyan tulajdonsága van, ami ideálissá teszi több olyan feladat ellátására is, amire az XML megjelenése előtt nem volt lehetőség elfogadott, szabványos, platformfüggetlen módon.

Web Service: magyarul Web Szolgáltatás. Definíciója szerint egy interneten használatos olyan alkalmazás, amely az XML, SOAP, WSDL és UDDI nyitott szabványokat használja. A gyakorlatban leginkább a Business-2-Business

környezetekben használatos, bár egyre szélesebb körben terjed. Elsősorban a cégek szolgáltatásokat publikálhatnak más cégek vagy felhasználók felé egy szabványos módon.

Smart Client: a Microsoft által kifejlesztett új architektúrális megközelítés, amely segítségével server oldali logikával rendelkező alkalmazásokat készíthetünk, úgy, hogy a kliens gépen fut a tényleges alkalmazás, de minden, a logikai rétegbe (és az alá) irányuló függvényhívás Web Service-en keresztül történik. Ezzel elérhetjük a maximális kódbiztonságot, hiszen a logikai réteg nem kerül a felhasználóhoz, elhárítva ezzel egy igen jelentős veszélyforrást. Emellett még néhány tulajdonság, amivel egy „Okos Kliensnek" rendelkeznie kell:

biztosítja, hogy ha a kliens nem rendelkezik aktuálisan Internet eléréssel, az alkalmazás továbbra is korlátozott funkcionalitással működőképes marad.

képes valós időben, a hálózatról önmaga frissítésére

több platformot, operációs rendszert támogat (alapvetően a .NET Framework és a Web Service szabványok betartásával)

gyakorlatilag bármilyen eszközön kell tudnia futni, amely rendelkezhet Internet eléréssel

WiKi: egy közösségi weboldal, amely lehetőséget biztosít minden látogató számára, hogy annak tartalmához hozzáírjon, módosítson rajta, esetleg töröljön. Már bizonyítottan ez egy rendkívül hatékony módszer a közös tudás hatékony megosztására.

Jövőbeli célunk a Microsoft fejlesztői társadalom támogatása egy forradalmian új, a kollektív fejlesztési tapasztalatokat és tudást felhasználó módszertan kialakításával, hogy egy még jobb, még erőteljesebb fejlesztői platformmal tudjon minél több fejlesztőt magához csábítani.

Felméréseink alapján a fejlesztők jelentős hányada találkozott már olyan problémával munkája során, amikor rendkívüli módon megkönnyítette és hatékonyabbá tette volna munkáját egy, a víziónkban szereplő rendszer. Az általunk fejlesztett, jelenleg is működő szoftver leginkább a fejlesztőeszköz (Microsoft Visual Studio .NET) integrált, intelligens részeként képzelhető el. A Microsoft kreatív és lelkes fejlesztői társadalma tapasztalatait eszközünk eddig elképzelhetetlen módon kamatoztatja: az Interneten található rendezetlen, struktúrálatlan, nehézkesen kereshető és értelmezhető

tudás, végre egy rendezett, akár moderált, automatikusan kereshető és felhasználható információhalmazzá válik.

Elsődleges fejlődési irányvonalak, amelyeket a Microsoft támogatása segítségével hatékonyabban tudnánk elérni: teljes integráció a fejlesztői környezettel, a megtekintés mellett kiemelt szerepet kapna a Debugger kiterjesztése és integrálása. Lehetőség nyílik továbbá privát és megosztott szerelvények publikálására a fejlesztőeszközből is hatékonyan elérehető webes felületre, a megosztott kódok WIKI módon történő annotálására, hibák keresésére, javaslatok felterjesztésére. A jól felismerhető, tipikus működésbeli zavarok, programhibák így kézi beavatkozás nélkül feltárhatóak, és a közösség által összegyűjtött információ automatikusan, pontosan, hosszadalmas keresgélés nélkül felhasználható a fejlesztő által. A pontos annotáció és szoros integráció segítségével statisztika és különösen értékes visszacsatolás készíthető a további verziók minőségének javítására, amit a felhasználók közösség által értékelt megjegyzései egészítenek ki.

A kutatás minőségének ékes bizonyítéka, hogy Magyarország Magyarország legnagyobb műszaki egyetemének évente megrendezett versenyén az erről készült dolgozat a bírák szerint első díjat érdemelt, és emellett a Microsoft elismerése jeleként különdíjjal jutalmazta a dolgozatott és az erről készített előadást.

Ugyanezen verseny kétévente megrendezett országos fordulóján a Microsoft Magyarország ismét különdíjjal fejezte ki elismerését.

Reményeink szerinti közös munkánk eredményképpen képesek lennénk javítani a hibakeresések gyorsaságát és hatékonyságát, lehetőséget biztosítani arra, hogy a közösség még erőteljesebben segíthesse a Microsoft fejlesztőinek munkáját - akár hibajavítási vagy optimalizálási javaslataikkal, valamit óriási lehetőségek rejlenek a szerelvények közös dokumentálhatóságában, aminek megvalósítása elképzelhetetlen lenne egy, az általunk fejlesztett rendszer segítsége nélkül.

Tehát az általunk megálmodott szoftvercsomag továbblép az eddig elkészült kódvisszafejtők eredményein, potenciálisan csökkentve az elégedetlen fejlesztők számát, a szoftverfejlesztés költségeit és a technológiában rejlő kockázatokat. Így lehetőség nyílik újabb fejlesztési módszertanok kialakítására és a keretrendszer, illetve dobozos termék alapú szoftvergyártás megkönnyítésére.

Reményeink szerint ezzel a forradalmi eszközzel fejlesztők millióinak mindennapi munkáját könnyíthetjük meg.

9 Végszó

Ha valaki biztonságos kódot akar írni, ahhoz sok ráfordítás kell. Ez mára sem változott és valószínűleg a későbbiekben sem fog. **A biztonság egy érték**, ha szoftverről beszélünk. Az, hogy egy adott (keret)rendszer vértezetlenül mennyire támadható, csak másodlagos szempont akkor, ha már léteznek megoldások, amivel kellően biztonságossá tehető. Napjainkban mindenkinek, aki programot készít, úgy kell gondolkodnia, mint egy ártó szándékú felhasználó. Ha megszokja ezt a gondolkodásmódot, akkor képes lesz felmérni az adott lehetőségek felhasználásának fontosságát. Ha nem, akkor csak idő kérdése, és az ő szoftvere is áldozattá válik.

1. számú melléklet

Opcode	Name	Parameter(s)	Pop	Push
00	nop	-	-	-
01	break	-	-	-
02	ldarg.0	-	-	*
03	ldarg.1	-	-	*
04	ldarg.2	-	-	*
05	ldarg.3	-	-	*
06	ldloc.0	-	-	*
07	ldloc.1	-	-	*
08	ldloc.2	-	-	*
09	ldloc.3	-	-	*
0A	stloc.0	-	*	-
0B	stloc.1	-	*	-
0C	stloc.2	-	*	-
0D	stloc.3	-	*	-
0E	ldarg.s	uint8	-	*
0F	ldarga.s	uint8	-	&
10	starg.s	uint8	*	-
11	ldloc.s	uint8	-	*
12	ldloca.s	uint8	-	&
13	stloc.s	uint8	*	-
14	ldnull	-	-	&=0
15	ldc.i4.m1ldc.i4.M1	-	-	int32=-1
16	ldc.i4.0	-	-	int32=0
17	ldc.i4.1	-	-	int32=1
18	ldc.i4.2	-	-	int32=2
19	ldc.i4.3	-	-	int32=3
1A	ldc.i4.4	-	-	int32=4

1B	ldc.i4.5	-	-	int32=5
1C	ldc.i4.6	-	-	int32=6
1D	ldc.i4.7	-	-	int32=7
1E	ldc.i4.8	-	-	int32=8
1F	ldc.i4.s	int8	-	int32
20	ldc.i4	int32	-	int32
21	ldc.i8	int64	-	int64
22	ldc.r4	float32	-	Float
23	ldc.r8	float64	-	Float
25	dup	-	*	*,*
26	pop	-	*	-
27	jmp	<Method>	-	-
28	call	<Method>	N arguments	Ret.value
29	calli	<Signature>	N arguments	Ret.value
2A	ret	-	*	-
2B	br.s	int8	-	-
2C	brfalse.sbrnull.sbrzero.s	int8	int32	-
2D	brtrue.sbrinst.s	int8	int32	-
2E	beq.s	int8	*,*	-
2F	bge.s	int8	*,*	-
30	bgt.s	int8	*,*	-
31	ble.s	int8	*,*	-
32	blt.s	int8	*,*	-
33	bne.un.s	int8	*,*	-
34	bge.un.s	int8	*,*	-
35	bgt.un.s	int8	*,*	-
36	ble.un.s	int8	*,*	-
37	blt.un.s	int8	*,*	-
38	br	int32	-	-
39	brfalsebrnullbrzero	int32	int32	-
3A	brtruebrinst	int32	int32	-
3B	beq	int32	*,*	-

3C	bge	int32	*,*	-
3D	bgt	int32	*,*	-
3E	ble	int32	*,*	-
3F	blt	int32	*,*	-
40	bne.un	int32	*,*	-
41	bge.un	int32	*,*	-
42	bgt.un	int32	*,*	-
43	ble.un	int32	*,*	-
44	blt.un	int32	*,*	-
45	switch	(uint32=N) + N(int32)	*,*	-
46	ldind.i1	-	&	int32
47	ldind.u1	-	&	int32
48	ldind.i2	-	&	int32
49	ldind.u2	-	&	int32
4A	ldind.i4	-	&	int32
4B	ldind.u4	-	&	int32
4C	ldind.i8ldind.u8	-	&	int64
4D	ldind.i	-	&	int32
4E	ldind.r4	-	&	Float
4F	ldind.r8	-	&	Float
50	ldind.ref	-	&	&
51	stind.ref	-	&,&	-
52	stind.i1	-	int32,&	-
53	stind.i2	-	int32,&	-
54	stind.i4	-	int32,&	-
55	stind.i8	-	int32,&	-
56	stind.r4	-	Float,&	-
57	stind.r8	-	Float,&	-
58	add	-	*,*	*
59	sub	-	*,*	*
5A	mul	-	*,*	*
5B	div	-	*,*	*

5C	div.un	-	*,*	*
5D	rem	-	*,*	*
5E	rem.un	-	*,*	*
5F	and	-	*,*	*
60	or	-	*,*	*
61	xor	-	*,*	*
62	shl	-	*,*	*
63	shr	-	*,*	*
64	shr.un	-	*,*	*
65	neg	-	*	*
66	not	-	*	*
67	conv.i1	-	*	int32
68	conv.i2	-	*	int32
69	conv.i4	-	*	int32
6A	conv.i8	-	*	int64
6B	conv.r4	-	*	Float
6C	conv.r8	-	*	Float
6D	conv.u4	-	*	int32
6E	conv.u8	-	*	int64
6F	callvirt	<Method>	N arguments	Ret.value
70	cpobj	<Type>	&,&	-
71	ldobj	<Type>	&	*
72	ldstr	<String>	-	o
73	newobj	<Method>	N arguments	o
74	castclass	<Type>	o	o
75	isinst	<Type>	o	int32
76	conv.r.un	-	*	Float
79	unbox	<Type>	o	&
7A	throw	-	o	-
7B	ldfld	<Field>	o/&	*
7C	ldflda	<Field>	o/&	&
7D	stfld	<Field>	o/&,*	-

7E	ldsfld	<Field>	-	*
7F	ldsflda	<Field>	-	&
80	stsfld	<Field>	*	-
81	stobj	<Type>	&,*	-
82	conv.ovf.i1.un	-	*	int32
83	conv.ovf.i2.un	-	*	int32
84	conv.ovf.i4.un	-	*	int32
85	conv.ovf.i8.un	-	*	int64
86	conv.ovf.u1.un	-	*	int32
87	conv.ovf.u2.un	-	*	int32
88	conv.ovf.u4.un	-	*	int32
89	conv.ovf.u8.un	-	*	int64
8A	conv.ovf.i.un	-	*	int32
8B	conv.ovf.u.un	-	*	int64
8C	box	<Type>	*	o
8D	newarr	<Type>	int32	o
8E	ldlen	-	o	int32
8F	ldelema	<Type>	int32,o	&
90	ldelem.i1	-	int32,o	int32
91	ldelem.u1	-	int32,o	int32
92	ldelem.i2	-	int32,o	int32
93	ldelem.u2	-	int32,o	int32
94	ldelem.i4	-	int32,o	int32
95	ldelem.u4	-	int32,o	int32
96	ldelem.i8ldelem.u8	-	int32,o	int64
97	ldelem.i	-	int32,o	int32
98	ldelem.r4	-	int32,o	Float
99	ldelem.r8	-	int32,o	Float
9A	ldelem.ref	-	int32,o	o/&
9B	stelem.i	-	int32,int32,o	-
9C	stelem.i1	-	int32,int32,o	-
9D	stelem.i2	-	int32,int32,o	-

9E	stelem.i4	-	int32,int32,o	-
9F	stelem.i8	-	int64,int32,o	-
A0	stelem.r4	-	Float,int32,o	-
A1	stelem.r8	-	Float,int32,o	-
A2	stelem.ref	-	o/&,int32,o	-
B3	conv.ovf.i1	-	*	int32
B4	conv.ovf.u1	-	*	int32
B5	conv.ovf.i2	-	*	int32
B6	conv.ovf.u2	-	*	int32
B7	conv.ovf.i4	-	*	int32
B8	conv.ovf.u4	-	*	int32
B9	conv.ovf.i8	-	*	int64
BA	conv.ovf.u8	-	*	int64
C2	refanyval	<Type>	*	&
C3	ckfinite	-	*	Float
C6	mkrefany	<Type>	&	*
D0	ldtoken	<Type>/ <Field>/ <Method>	-	&
D1	conv.u2	-	*	int32
D2	conv.u1	-	*	int32
D3	conv.i	-	*	int32
D4	conv.ovf.i	-	*	int32
D5	conv.ovf.u	-	*	int32
D6	add.ovf	-	*,*	*
D7	add.ovf.un	-	*,*	*
D8	mul.ovf	-	*,*	*
D9	mul.ovf.un	-	*,*	*
DA	sub.ovf	-	*,*	*
DB	sub.ovf.un	-	*,*	*
DC	endfinally endfault	-	-	-
DD	leave	int32	-	-
DE	leave.s	int8	-	-

DF	stind.i	-	int32,&	-
E0	conv.u	-	*	int32
FE 00	arglist	-	*	&
FE 01	ceq	-	*,*	int32
FE 02	cgt	-	*,*	int32
FE 03	cgt.un	-	*,*	int32
FE 04	clt	-	*,*	int32
FE 05	clt.un	-	*,*	int32
FE 06	ldftn	<Method>	-	&
FE 07	ldvirtftn	<Method>	o	&
FE 09	ldarg	uint32	-	*
FE 0A	ldarga	uint32	-	&
FE 0B	starg	uint32	*	-
FE 0C	ldloc	uint32	-	*
FE 0D	ldloca	uint32	-	&
FE 0E	stloc	uint32	*	-
FE 0F	localloc	-	int32	&
FE 11	endfilter	-	int32	-
FE 12	unaligned.	uint8	-	-
FE 13	volatile.	-	-	-
FE 14	tail.	-	-	-
FE 15	initobj	<Type>	&	-
FE 17	cpblk	-	int32,&,&	-
FE 18	initblk	-	int32,int32,&	-
FE 1A	rethrow	-	-	-
FE 1C	sizeof	<Type>	-	int32
FE 1D	refanytype	-	*	&

2. számú melléklet

```
namespace Wriggler.CodeDomEx
{
	public interface ICodeDomExtension
	{
		CodeObject GetParent();
		CodeObject[] GetChildren();
		void SetParent( CodeObject co );

		void WriteXml( string filename );

		void AddInstruction( Instruction inst );
		void AddInstruction( Instruction[] insts );
		void AddInstruction( CodeObject co );
		void AddInstruction( CodeObject[] cos );

		Instruction[] GetInstructions();
		int[] GetInstructionOffsets();

		CodeObject Negate();
	}

	public class UserDataFunctions
	{
		public static void AddInstruction( CodeObject co, Instruction inst)
		{
			try
			{
				co.UserData.Add(  "IL"+inst.Offset.ToString("x4"),  inst
);
			}
			// ignore the exception if we want to add an instruction that
is
// already exists in the list
			catch {}
		}
		public static void AddInstruction( CodeObject co, Instruction[] insts
)
		{
			foreach (Instruction inst in insts) AddInstruction( co, inst
);
		}
```

```
		public static void AddInstruction( CodeObject co, CodeObject aco )
		{
			if (!(aco is ICodeDomExtension)) return;
			foreach (Instruction inst in
UserDataFunctions.GetInstructions(aco))
AddInstruction( co, inst );
		}

		public static void AddInstruction( CodeObject co, CodeObject[] acos )
		{
			foreach (CodeObject aco in acos) AddInstruction( co, aco );
		}

		public static Instruction[] GetInstructions( CodeObject co )
		{
			Instruction[] result;
			IList resultc;

			if (!(co is ICodeDomExtension)) return null;

			resultc = new ArrayList();
			foreach (DictionaryEntry de in co.UserData)
			{
				if (de.Key.ToString().Substring(0,2) == "IL")
resultc.Add( de.Value );
			}

			result = new Instruction[resultc.Count];
			int i=0;
			foreach (object o in resultc) result[i++] = (Instruction)o;

			return result;
		}

		public static int[] GetInstructionOffsets( CodeObject co )
		{
			int i=0;
			int[] result;
			Instruction[] insts = GetInstructions( co );

			if (insts == null) return null;

			result = new int[ insts.Length ];

			i = 0;
			foreach (Instruction inst in insts) result[i++] = inst.Offset;
			return result;
		}
```

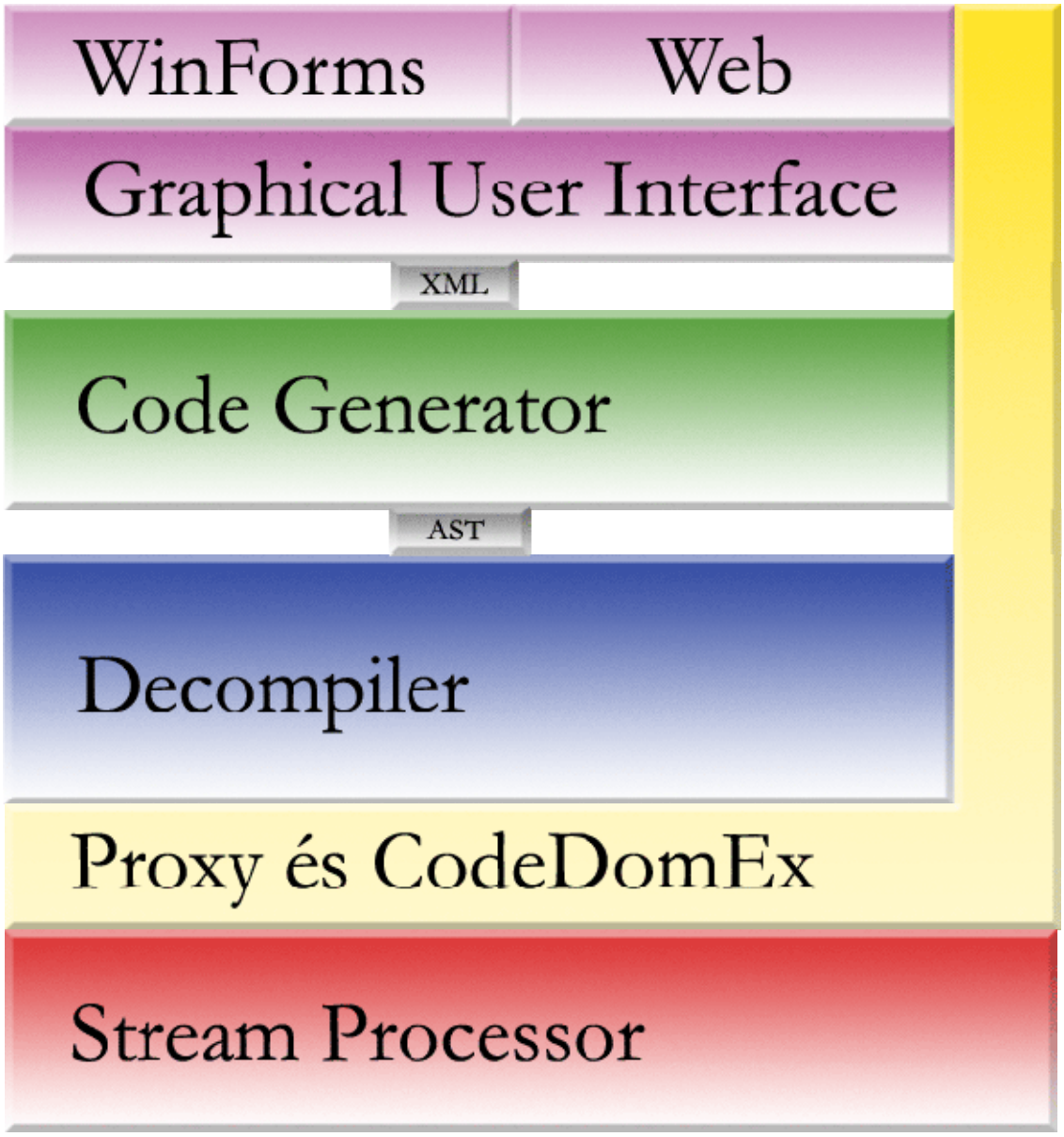
```
        public static CodeObject Negate( CodeObject co )
        {
                // TODO: Implement CodeEx Negate
                return co;
        }
        public static void WriteXml( CodeObject co, string filename )
        {
                // TODO: Implement CodeEx Xml writer
                return;
        }
}

////////// GENERATED ///////////

public class CodeExpressionEx : CodeExpression, ICodeDomExtension
{
        private CodeObject parent;
        private CodeObject[] children;

        public CodeExpressionEx( ) : base()
        {
        }
        public CodeExpressionEx( Instruction inst ) : base()
        {
                UserDataFunctions.AddInstruction( this, inst );
        }

        #region pseudo-static methods
        public CodeObject GetParent()
        {
                return parent;
        }
        public CodeObject[] GetChildren()
        {
                return children;
        }

        public void SetParent( CodeObject co )
        {
                parent = co;
        }

        public void WriteXml( string filename )
        {
                UserDataFunctions.WriteXml( this, filename );
        }
```

```
        public void AddInstruction( Instruction inst )
        {
            UserDataFunctions.AddInstruction( this, inst );
        }

        public void AddInstruction( Instruction[] insts )
        {
            UserDataFunctions.AddInstruction( this, insts );
        }

        public void AddInstruction( CodeObject co )
        {
            UserDataFunctions.AddInstruction( this, co );
        }

        public void AddInstruction( CodeObject[] cos )
        {
            UserDataFunctions.AddInstruction( this, cos );
        }

        public Instruction[] GetInstructions()
        {
            return UserDataFunctions.GetInstructions( this );
        }

        public int[] GetInstructionOffsets()
        {
            return UserDataFunctions.GetInstructionOffsets( this );
        }

        public CodeObject Negate()
        {
            return UserDataFunctions.Negate( this );
        }
        #endregion
    }
    public class CodeArgumentReferenceExpressionEx :
CodeArgumentReferenceExpression,
ICodeDomExtension
    {
        private CodeObject parent;
        private CodeObject[] children;

        public CodeArgumentReferenceExpressionEx( ) : base()
        {
        }
        public CodeArgumentReferenceExpressionEx( Instruction inst ) : base()
        {
```

```
                UserDataFunctions.AddInstruction( this, inst );
        }
        public CodeArgumentReferenceExpressionEx( string parameterName  ) :
base( parameterName )
        {
        }
        public CodeArgumentReferenceExpressionEx( string parameterName ,
Instruction inst ) : base( parameterName )
        {
                UserDataFunctions.AddInstruction( this, inst );
        }



        #region pseudo-static methods
        public CodeObject GetParent()
        {
                return parent;
        }
        public CodeObject[] GetChildren()
        {
                return children;
        }

        public void SetParent( CodeObject co )
        {
                parent = co;
        }

        public void WriteXml( string filename )
        {
                UserDataFunctions.WriteXml( this, filename );
        }

        public void AddInstruction( Instruction inst )
        {
                UserDataFunctions.AddInstruction( this, inst );
        }

        public void AddInstruction( Instruction[] insts )
        {
                UserDataFunctions.AddInstruction( this, insts );
        }

        public void AddInstruction( CodeObject co )
        {
                UserDataFunctions.AddInstruction( this, co );
        }
```

```
        public void AddInstruction( CodeObject[] cos )
        {
            UserDataFunctions.AddInstruction( this, cos );
        }

        public Instruction[] GetInstructions()
        {
            return UserDataFunctions.GetInstructions( this );
        }

        public int[] GetInstructionOffsets()
        {
            return UserDataFunctions.GetInstructionOffsets( this );
        }

        public CodeObject Negate()
        {
            return UserDataFunctions.Negate( this );
        }
        #endregion
    }

(…)
}
```

3. számú melléklet



A kódvisszafejtő moduláris felépítése