Másolásvédelem szoftver vízjelezés és obfuszkálás segítségével

EBERHARDT GERGELY, NAGY ZOLTÁN
SEARCH-LAB Kft., {gergely.eberhardt, zoltan.nagy}@search-lab.hu
JEGES ERNŐ, HORNÁK ZOLTÁN
BME Méréstechnikai és Információs Rendszerek Tanszék, SEARCH Laboratórium
{jeges, hornak}@mit.bme.hu

Lektorált

Kulcsszavak: szoftver vízjel, obfuszkálás, kódvisszafejtés, megbízható operációs rendszer, mobiltelefonos szoftverek

A szerzői jogok megsértését a szoftverfejlesztő cégek manapság elsősorban jogi úton próbálják orvosolni, mivel a jelenleg létező technológiák nem teszik lehetővé a szoftverek illegális használatának és terjesztésének a megakadályozását. Mindezidáig általános tapasztalat volt, hogy szinte minden piacra dobott másolásvédelmi megoldást rövid időn belül feltörtek. A másolásvédelemi megoldások hiányossága eredményezte néhány a közelmúltban szárnyra kapott üzleti modell bukását is, ami bizonyos értelemben a „dotcom" világ válságához is hozzájárult. Cikkünkben egy olyan megoldást mutatunk be, amely az obfuszkálás és a vízjelezés technikáinak ötvözésével nyújt megfelelő erősségű védelmet a kereskedelmi forgalomba került szoftverek másolása ellen. Az általunk javasolt megoldás elsősorban mobiltelefonon futó alkalmazások – tipikusan játékprogramok – védelmére használható, mivel ezen eszközök esetében a védelmi megoldást az operációs rendszer és a hardver által szavatolt biztonságos működésre alapozhatjuk.

1. Bevezetés

A Business Software Alliance (BSA) adatai szerint az illegális szoftverhasználatból eredő kár éves szinten 30 milliárd dollárra volt becsülhető világviszonylatban 2004-ben [3]. Az Európai Uniót tekintve a használt szoftvereknek csak közel a fele legális, és ezen a helyzeten sem technikai sem jogi úton eddig nem sikerült javítani. Sajnos a köztudatban is az a nézet uralkodik, hogy a szoftverkalózkodást szinte lehetetlen megakadályozni. A mobiltelefonok világában azonban, ahol, tekintettel arra, hogy az azokban futó operációs rendszer sértetlenségében meg lehet bízni, még van arra lehetőség, hogy az illegális szoftvermásolásból eredő máshol tapasztalt károk elkerülhetőek, vagy legalább csökkenthetőek legyenek.

Meggyőződésünk szerint a mobil szoftverek piacán a további növekedés legfontosabb előfeltétele egy erős másolásvédelemi technika megléte, ezért kutatásfejlesztési projektünk ezen szoftverek másolásvédelmét célozta meg. Ezen a területen azonban némileg más peremfeltételekkel kell számolnunk, mint az általában tárgyalt személyi számítógépek esetében.

A megbízható operációs rendszer *(trusted OS)* elengedhetetlen alapja egy megfelelően erős másolásvédelemnek, hiszen amennyiben az operációs rendszer maga is megváltoztatható, akkor bármilyen rá épülő szoftver alapú védelem elvi problémák miatt feltörhető. Elképzelhetőek olyan megoldások is, ahol a fejlesztők egyáltalán nem bíznak meg az operációs rendszerben. Ezen megoldások zöme a ködösítés elvét *(security by obscurity)* kihasználó biztonságra alapoz, azaz egyszerűen elrejti a kód azon részeit, amelyek a szoftver sértetlenségét és érvényességét ellenőrzik. Ezen megoldások kifejlesztői általában azt feltételezik, hogy az elrejtett ellenőrzésnek a visszafejtése vagy a védelem megkerülése kellően sok időt vesz igénybe ahhoz, hogy ne befolyásolja számottevően a bevételek mértékét. Mindannak ellenére, hogy egy ilyen módszer valóban megnöveli a védelem feltöréséhez és megkerüléséhez szükséges időt, a tapasztalatok azt mutatják, hogy a titokban tartott módszeren alapuló védelmeket minden esetben előbb-utóbb feltörték.

A fentiekkel szemben az általunk javasolt szoftver másolásvédelmi megoldás ötvözi a Nyilvános Kulcsú Infrastruktúrát (PKI), az obfuszkálási módszereket valamint a szoftver vízjelezést, miközben egy megbízható, sértetlenségét biztosító operációs rendszert feltételezünk. A pusztán licenszeket és digitális aláírást alkalmazó módszerekkel szemben a sémánk legfontosabb előnye az, hogy egyaránt megengedi szabadon terjeszthető és másolásvédett szoftverek futtatását is.

2. Elméleti háttér

A szoftver másolásvédelemi megoldásoknak két fő kategóriája létezik: az önálló rendszerek, valamint azok, amelyek külső eszközöket használnak működésük során [10].

Az önálló rendszerek védelmét önmagába a szoftverbe építik bele, így annak biztonsága csak a felhasznált programozási technikáktól függ. Ilyen technikák lehetnek különböző integritásvédelmek, program obfuszkálás, ellenőrző összegek, titkosítás, a kódvisszafejtés a megfigyelt futtatás (debug) megakadályozása, illetve egyéb, a cracker-ek feladatát megnehezítő megoldások [8]. Ezen módszerek alapja szükségképpen az, hogy a program *önmagát* ellenőrizze. Mivel az ellenőrzés a program részét képezi, ezért annak visszafejtésével az ellenőrzést végző programrészletek lokalizálhatóak és a kód módosításával a védelem kiiktatható

[10]. Ezek a technikák tehát elméleti szempontból és a tapasztalatok alapján gyakorlati szempontból sem tekinthetőek elegendően biztonságosnak [2].

A védelmi mechanizmusok másik kategóriáját a külső segítséget igénybe vevő megoldások alkotják. Ezek a másolásvédett programok általában valamilyen megbízható processzort, operációs rendszert, vagy más biztonságos hardver, esetleg szoftver megoldást használnak. A külső támogatás lehet on-line vagy off-line [10]. Az on-line támogatás esetében néhány ellenőrző függvény olyan távoli számítógépen fut le, amelyekhez a támadónak nincs hozzáférése. Ezzel ellentétben az off-line együttműködés során a védelem valamilyen biztonságos hardvert vagy szoftvert igényel. A biztonságos hardver általában egy smart card használatát jelenti, míg a megbízható szoftver komponenst általában egy sérthetetlen operációs rendszer biztosítja.

A jogosult felhasználók hozzákapcsolását a program egy példányához általában Nyílt Kulcsú Infrastruktúrán (Public Key Infrastructure, PKI) [9] alapuló digitálisan aláírt licenszek használatával valósítják meg. Másolásvédett szoftverek esetében ilyenkor a programhoz csatolni kell egy úgynevezett licensz fájlt, ami információkat tartalmaz a felhasználóról, a gyártóról, a forgalmazóról és magáról a termékről (például az adott szoftver hash kódja). Mivel a licensz sértetlenséget digitális aláírás védi, az operációs rendszer már biztonsággal ellenőrizheti a jogosultságot, és a megfelelő licensz nélkül az alkalmazás futtatását megtagadhatja.

Egy programvédelmi megoldásnak támogatnia kell a különböző üzleti modelleket, a védett alkalmazás felhasználásának különböző eseteit, valamint a kényelmes szoftverfejlesztést is, de – ami a legnagyobb technikai kihívást jelenti – az ezzel felvértezett operációs rendszernek futtatnia kell tudni mind a másolásvédett, mind pedig a szabad terjesztésű programokat. Elengedhetetlen feltétel tehát, hogy az operációs rendszer minden olyan esetben felismerje, hogy a program eredetileg védett volt, amikor a szoftver egy részlete megváltozott, vagy ha a licensz fájlt eltávolították mellőle.

A hagyományos hang, kép, illetve video fájlok védelmi megoldásaival ellentétben, amelyek a vízjelet a tartalom eredetének a megállapítására használják, a szoftver esetében annak a jelzésére is használhatjuk a vízjelet, hogy a program másolásvédett-e vagy sem. A média fájlokhoz képest sokkal könnyebben megoldható az, hogy a program kódját úgy módosítsuk, hogy közben a felhasználó által tapasztalt működés ne változzon érezhető módon.

A *szoftver vízjelek* két típusát különböztethetjük meg: a statikust és a dinamikust. A statikus vízjelek esetében az információt a végrehajtható állomány hordozza. Az ilyen vízjeleket tipikusan az inicializált *data, code és text* szakaszokban szokták elhelyezni [1,7,8,11,13,14]. Ezzel ellentétben a dinamikus vízjelek nem a végrehajtható program kódjából, hanem a program végrehajtási állapotából nyerhetők ki. Ez azt jelenti, hogy a programot le kell futtatni, hogy a programállapot valamely tulajdonsága jelezze a vízjel jelenlétét [5,6,12].

A vízjel egyszerű eltávolíthatóságának a megakadályozására *szoftver obfuszkálást* alkalmazhatunk. Ez a módszer különböző kód-transzformációs eljárások gyűjtőneve, amelyeket azzal a közös céllal hajtunk végre, hogy a program visszafejtését és megértését nehezebbé tegyük. Az obfuszkálás mind az automatikus eszközökkel történő, mind az emberi megértés által végzett visszafejtési támadást számottevően nehezíti [4,15].

3. Követelmények és minőségi célok

A továbbiakban meghatározzuk a másolásvédelmi megoldásunkhoz szükséges követelményeket és a minőségi célok eléréséhez szükséges további feltételeket.

Megbízható operációs rendszer – sértetlenség és bizalmasság

Egy megbízható operációs rendszer esetében feltételezhetjük a futó folyamatok *sértetlenségét*, esetünkben azonban nem feltételezzük az operációs rendszer, vagy a rajta keresztül áramló információk bizalmasságát. Ez azzal a feltétellel függ össze, miszerint a vízjel ellenőrző módszert nem tarthatjuk titokban, tehát a támadó azt megismerheti, ennek ellenére azonban a vízjel eltávolításának a védett programból nehéz feladatnak kell lennie.

A fentieken túl az általunk használt megbízható operációs rendszernek képesnek kell lennie arra, hogy az alkalmazáson lévő digitális aláírást ellenőrizze, és támogatnia kell a PKI különböző egyéb elemeit is, mint például a tanúsítvány lánc kezelését vagy a tanúsítvány (kulcspár) visszavonását.

Azonos megfigyelhető működés

Esetünkben a *megfigyelő* a program felhasználója, akinek a szemszögéből nézve a transzformált programnak funkcionálisan ugyanúgy kell működnie, mint az eredetinek. Az alkalmazásnak például ugyanazokat az ablakokat kell tartalmaznia, ugyanazokat a fájlokat és kapcsolatokat kell fenntartania a külvilággal, és mindezeknek ugyanolyan módon kell működniük. A sebességnek, a memória használatnak, a program végrehajtása közbeni belső állapotoknak, valamint a program kódjának azonban kis, vagy akár nagyobb mértékű változása is megengedett.

Nehezebb visszafejthetőség

Az obfuszkálási módszerek alkalmazásával a programnak egyrészt az *automatikus* visszafordítását kell tudnunk meggátolni, illetve megnehezíteni, másrészt pedig a transzformációk hatására a kód *emberi megértését* is nehezíteni kell [8]. Célunk tehát az, hogy a visszafejtést, és így a védelem megszüntetését annyira időigényessé tegyük, hogy már ne érje meg a hasonló törések végrehajtásához szükséges időráfordítást.

A *nehezítés*, illetve a nehéz esetünkben azt jelenti, hogy egy védett program visszafordítása olyan komplex feladat megoldását kívánja meg, amelynek nehézsége visszavezethető a kriptográfiában elfogadottan nehéznek tartott problémára, tipikusan legyen azzal egyenértékű, mintha egy kriptográfiai algoritmust kellene feltörni.

Vízjel nehezebb eltüntethetősége

Ez a minőségi cél szorosan összefügg az előzővel, hiszen az erős obfuszkálási megoldások használata nem csak a visszafejtést nehezíti, de a vízjel könnyű eltávolíthatóságát is megakadályozza.

Mivel esetünkben a megbízható operációs rendszernél csak a sértetlenséget feltételezzük, de a megbízhatóságot nem, ezért feltételeznünk kell azt is, hogy a vízjel felismerő algoritmust sem lehet titokban tartani. Ebből adódóan a vízjel eltávolításának még abban az esetben is kellően nehéznek kell lennie, ha a vízjel detektáló algoritmus nyilvánosan ismert. A vízjelet ezért az eredeti kódba olyan mélyen kell elhelyezni, hogy azt ne lehessen eltávolítani a *teljes* kód megértése nélkül. A teljes megértés követelménye azt jelenti, hogy mindennek mindennel össze kell függnie ahhoz, hogy a támadó ne legyen képes dekomponálni a megértés feladatát.

A transzformációk költségeinek skálázhatósága

Mivel mindegyik transzformációnak van valamekkora költsége, amely általában az alkalmazás végrehajtási idejére és memóriahasználatára vonatkozik, ezért fontos, hogy a védelem ebből a szempontból is hatékony legyen. A különböző követelmények, és a kód különböző területein felmerülő transzformációs költségek miatt a megvalósított transzformációknak tehát *skálázhatóaknak* kell lenniük, és a rendszernek elsősorban a transzformált program futási sebessége és a memória használat tekintetében kell támogatnia a megfelelő munkapont beállítását.

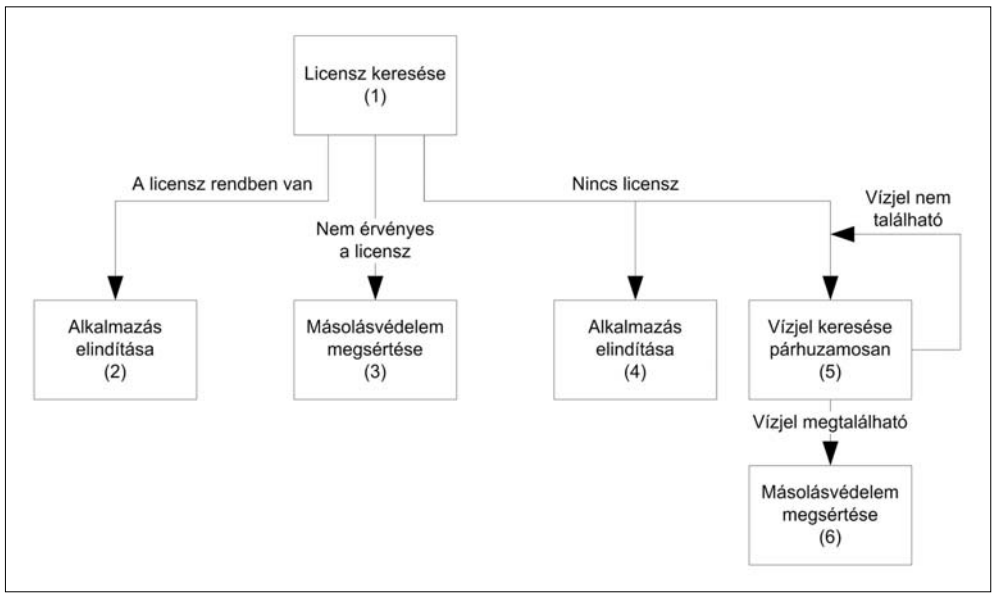
4. A javasolt másolásvédelmi megoldás

A másolásvédelmi megoldásunk tehát a fentebb ismertetett követelmények figyelembevételével, az említett építőkövek felhasználásával valósul meg. A módszer lényege, hogy a rendszer egy digitálisan aláírt licensz segítségével ellenőrzi a védett program sértetlenségét, és amennyiben a digitális aláírása vagy a licensz nem érvényes, megakadályozza a futtatást. Abban az esetben, ha nincs licensz csatolva egy alkalmazáshoz, akkor a rendszer a program futása alatt folyamatosan keresi a dinamikus vízjel jelenlétének a nyomait, amely jelzi az operációs rendszer számára, hogy másolásvédett szoftverről van szó, tehát a licensz fájl megléte kötelező lenne. A vízjel eltávolítása pedig megfelelő obfuszkálási technikák alkalmazása miatt nehéz, ezért a támadó nem tudja olyan módon változtatni, feltörni a védett alkalmazást, hogy az a licensz nélkül is működőképes legyen.

A másolásvédelmi megoldás operációs rendszeren belüli ellenőrző része az *1. ábrán* látható algoritmussal írható le.

1. Digitálisan aláírt licensz keresése a programhoz.
2. Ha van ilyen licensz, és mind a licensz (például az abban található hash), mind pedig a digitális aláírás megfelelő, akkor az alkalmazás minden további vízjel-ellenőrzés nélkül futtatható.
3. Ha programhoz kapcsolódó licensz vagy annak aláírása nem megfelelő, a végrehajtás azonnal leáll, valamint a rendszer megteheti az egyéb szükséges lépéseket (például naplózás vagy jelentés készítése), mivel a másolásvédelem, vagy az alkalmazás integritása sérült.
4. Amennyiben a programhoz nincs licensz mellékelve, akkor az lehet egy szabad felhasználású, vagy egy védett, de manipulált program is, ezért a program elindul.
5. Ezzel párhuzamosan az operációs rendszernek el kell kezdenie a vízjel folyamatos keresését.
6. Ha a vízjel megtalálható a programban, akkor a végrehajtást azonnal fel kell függesztenie, és ismételten a szükséges lépéseket kell megtennie, hiszen a vízjel jelenléte azt jelzi, hogy a program másolásvédett, így érvényes licensszel kellene rendelkeznie, és enélkül illegális másolatnak számít.

1. ábra Licensz ellenőrzés algoritmusa



A védelmi sémánkban a vízjelet a program másolásvédettségének jelzésére használjuk, ezért a vízjelnek a következő tulajdonságokkal kell rendelkeznie:

– csak egy „bitnyi” információ tárolására kell alkalmasnak lennie, amelynek a jelentése az, hogy a program másolásvédett,
– nem használhat titkos módszert a vízjel detektálására, mivel a támadónak lehetősége van arra, hogy megismerje a vízjelet felismerő algoritmust,
– ne lehessen felismerni az összes vízjelet a programban statikus elemzéssel, még abban az esetben se, ha a támadó ismeri a felismerő algoritmust.

Ezen követelmények mindegyikének a kielégítéséhez a legjobban a dinamikus vízjel felel meg, mivel esetében a vízjel bináris reprezentációja a program végrehajtása során jelenik csak meg. A dinamikus vízjel detektálásához azonban a programot egy ideig futtatni kell, amiből az következik, hogy a program állapotát folyamatosan kell figyelni a végrehajtás alatt. Emiatt a vízjel keresés ugyan lelassíthatja az alkalmazások végrehajtását, de csak abban az esetben, ha nincs licensz fájl mellékelve a programhoz. A fejlesztőknek tehát különösen érdekük licenszet biztosítani a termékeikhez még akkor is, ha az szabad felhasználású, mivel csak ily módon tudják az eszköz teljesítményét teljesen kihasználni.

A program és az operációs rendszer közötti kapcsolatot a vízjel felismerése közben a *2. ábra* illusztrálja.

Az általunk használt dinamikus vízjel csak egy bit információt tárol egy speciális szám formájában, ami egy véletlen számból és ennek a véletlen számnak a transzformált értékéből áll elő úgy, hogy a két értéket egyszerűen egymás mögé írjuk. Az *f* függvénnyel végrehajtott transzformáció lehet digitális aláírás, hash érték, CRC vagy egyéb, például egy egyszerű konstanssal való XOR művelet is; szerepe abban van, hogy egy nem védett programban a két egymást követő érték előfordulási valószínűsége elegendően kicsi legyen, így a védett programban való előfordulása egyértelműen jelezze a vízjel meglétét. A vízjelet tehát a következő módon definiálhatjuk:

$$WM = (RND; f(RND))$$

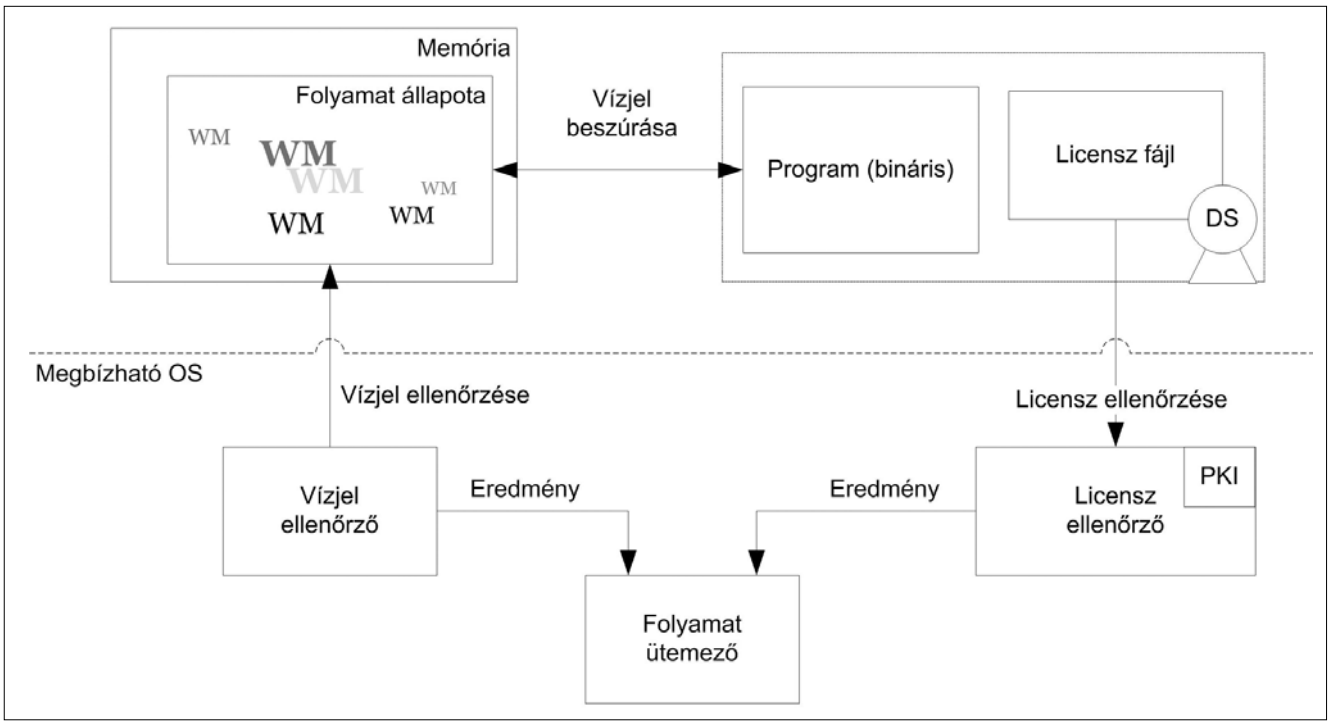
A véletlen számtól függő különböző *WM* vízjel értékpárokat tehát olyan sokszor kell elrejteni az alkalmazásban – azaz a védett programnak a futása során annyiszor kell előállítania ilyen érték párost – amennyiszer csak lehetséges. Ezen cél elérése érdekében többféle, elsősorban az adat obfuszkálással összekapcsolható technikát alkalmazhatunk; egy ciklusváltozót például olyan módon transzformálhatunk, hogy annak egy bizonyos értékére, amelyet egyszer biztosan felvesz, egy a fenti módon definiált *WM* szám álljon elő.

Vegyük észre, hogy sem az operációs rendszer, sem a támadó nem tudja a vízjel összes lehetséges értékét, csak felismerni tudja azt, amikor a megfelelő bemenetek hatására előáll a program állapotában. Így ha a támadó az összes vízjelet el akarja tüntetni, akkor végre kell hajtania a program összes elágazását az összes lehetséges bementettel, hogy megbizonyosodjon róla, hogy az eltávolítás maradéktalanul sikeres.

5. A rendszer felépítése

A következő, *3. ábrán* látható a másolásvédelmi megoldásunk keretrendszere, annak főbb elemei (moduljai) és azok kapcsolata. Az obfuszkálási és vízjel elhelyező

2. ábra A program és az operációs rendszer közötti kapcsolat a vízjel felismerése közben



transzformációk a szokványos C/C++ fordítási folyamatba integrálhatóak. A rendszer bemenetét a védendő alkalmazás C/C++ forrás fájljai képezik. A forráskódban elhelyezett direktívák szabályozhatják az obfuszkálás és a vízjel elhelyezés folyamatát azáltal, hogy behatárolják az egyes kódrészletek transzformálása során megengedett költségeket (futásidő, memória használat). Ezen direktívákat az előfeldolgozás során összegyűjtjük, majd a fordító az eredeti forrásból előállítja az assemblyre lefordított kódot tartalmazó kimeneti fájlt (LST) és az egyéb debug információkat tartalmazó fájlokat.

Az összegyűjtött információk egy fordító-független absztrakt reprezentáció *(Virtual Machine Code, VMC)* előállításának az alapjául szolgálnak, amin az obfuszkáló és vízjelező transzformációk végrehajtása történik. Az absztrakt reprezentáció létrehozásához azonban nem csak a fentebb említett forrásfájlok használhatók, hanem esetleg valamely dissasembler eszközzel nyert LST fájl, a Map fájl vagy akár a különböző Profile információk is.

A direktívák összegyűjtése és az előkészítés után a rendszer elemzi az LST fájlban és a többi forrásként használt fájlban található kódot, illetve egyéb információt a kóddal kapcsolatban. Ezen elemzés részét képezi a vezérlési folyam és az adatfüggőségi gráfok megállapítása is, melyek létrehozásával teljessé válik a program belső, platform független absztrakt reprezentációja. Ez a belső reprezentáció, amelyet *Code Model*-nek nevezünk, tartalmazza az összes olyan információt, amelyre szükség van a különböző transzformációk megtervezéséhez és végrehajtásához.

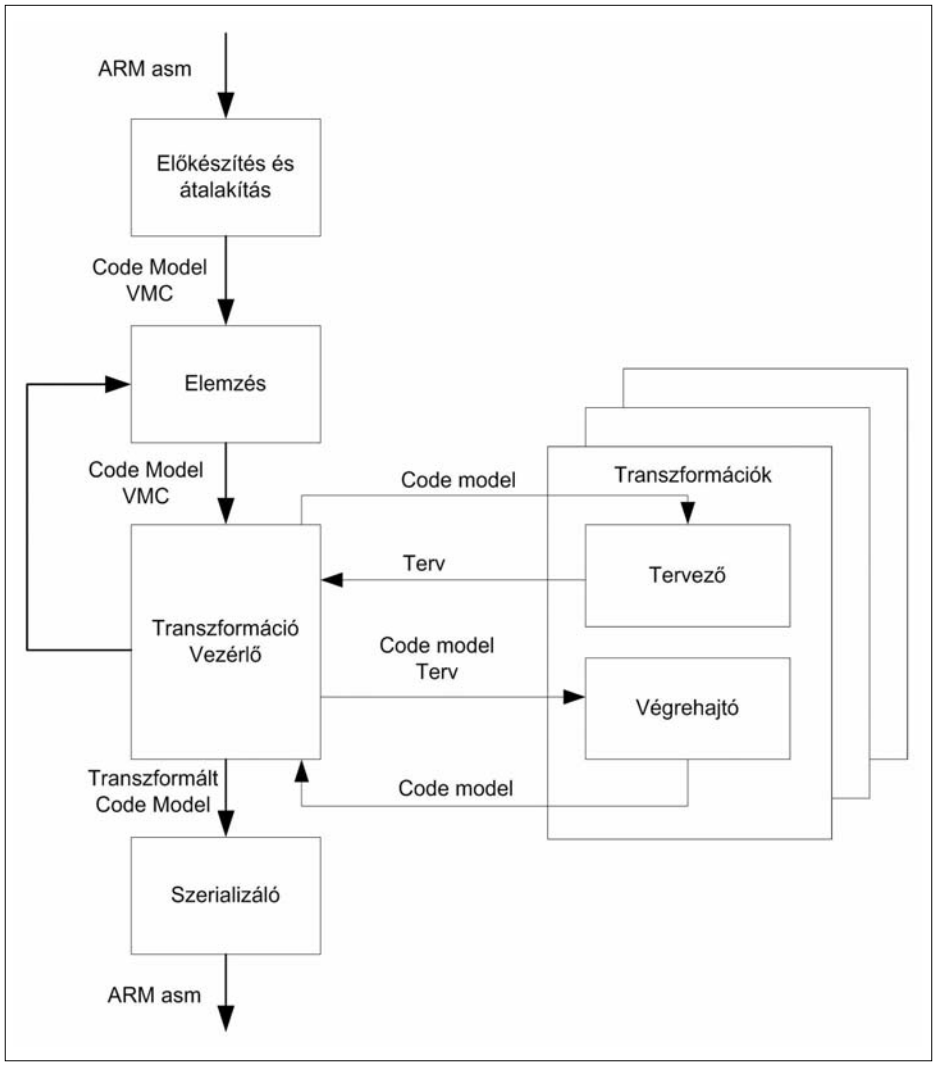
A transzformációk végrehajtása több lépésben történik. A *Transzformáció Vezérlő* minden lépés előtt egy részletes tervet készít az elvégzendő műveletekről és azok sorrendjéről. Minden transzformációs lépést követően a belső absztrakt reprezentációnak konzisztensnek kell maradnia, ami azt jelenti, hogy minden iterációs lépésnél a programnak funkcionálisan azonosnak kell lennie az eredeti transzformálatlan kóddal.

A hatékonyság növelés és a funkcionális azonosság biztosítása érdekében minden transzformációs lépés két fázisban zajlik. Az első fázisban tervek készülnek a lehetséges végrehajtási módokról és a végrehajtás esetleges paramétereiről, továbbá az egyes lépések végrehajtásának hatásairól.

Az egyes transzformációk *Tervezője* által felajánlott paraméterek alapján elkészül a terv, amely tartalmazza a kiválasztásra került transzformációk sorrendjét és paraméterezését, majd a második fázisban a transzformációk *Végrehajtója* végrehajtja azokat. Az egyes transzformációkhoz tartozó *Tervező* feladata az első fázisban a paraméterek meghatározása, a *Végrehajtó* pedig elvégzi a tényleges transzformációt és biztosítja a funkcionális azonosságot. Így a helyes működés bizonyításához elég csak a *Végrehajtó* eljárások helyességét bizonyítani.

A transzformációs lépések addig követik egymást, amíg a meghatározott minőségi célok meg nem valósulnak, azaz elegendő vízjel nem kerül elhelyezésre és a kód bonyolultsága el nem éri a kívánt szintet. A transzformációs lépések sorozata után az absztrakt reprezentáció szerializálásával ismét előáll a közvetlenül futtathatóvá fordítható assembly kód. A teljes folyamat eredménye tehát egy lefordított tárgy kód, amely egyrészt obfuszkált, másrészt tartalmazza a vízjelet is.

3. ábra A rendszer moduljai és köztük lévő kapcsolatok



6. Eredmények

Az általunk tervezett másolásvédelmi rendszer értékeléséhez implementáltuk a keretrendszert, illetve megvalósítottunk számos transzformációt. A transzformációk közül egy egyszerű példával, a rendszerhívások elrejtését megvalósító obfuszkálási technika gyakorlatba ültetésével illusztráljuk az általunk javasolt séma és a megvalósított keretrendszer lehetőségeit.

A legtöbb program intenzíven használja a különböző szabványos programozói könyvtárakat, illetve az API-kat az operációs rendszer szolgáltatásainak eléréséhez. Ezek a függvényhívások jól dokumentáltak és a legtöbb programozó által jól ismertek, így nagy segítséget jelentenek a program megértésében és visszafejtésében. Ennek elkerülésére alkalmazható a rendszerhívások elrejtését megvalósító obfuszkálási technika, amely eltünteti a hasonló nyomokat a programból.

Különböző eljárások léteznek az ilyen ismert függvények hívásainak elrejtésére. Az alapvető ötlet ezek mögött az, hogy az eredeti rendszerhívás lecserélésre kerül egy belső úgynevezett fedő *(wrapper)* függvény meghívásával, amely tovább hívja az eredeti függvényt. Az obfuszkáló tetszőleges számú ilyen interfész függvényt hozhat létre, de akár elhelyezheti valamennyi API hívást egyetlen ilyen interfész függvényben is. Ilyenkor általában egy változó értéke dönti el, hogy az interfész függvénynek ténylegesen melyik API függvényt kell meghívnia.

Ezen obfuszkálási transzformáció esetében a *Tervező* meglehetősen egyszerű, mivel csak az ismert függvények hívási helyeit kell megkeresnie, majd ezekből kiválasztania az elrejtendőket (akár mindet). Így az elkészült terv ezen hívások listáját fogja majd tartalmazni.

A hívások elrejtésének algoritmusa a terv ismeretében a következő:

1. Egy új függvény létrehozása, ami interfész függvényként fog szolgálni az API hívásokhoz.
2. Az API hívásokhoz azonosítók rendelése, és az interfész függvény feltöltése a megfelelő blokkokkal és utasításokkal a lehetséges API hívásoknak megfelelően.
3. Az API függvények hívási helyeinek módosítása úgy, hogy azok az új interfész függvényre mutassanak.
   A hívandó API függvények azonosítójának beállítása az interfész függvény meghívása előtt a megfelelő változóban.

7. Összefoglalás

A fentiekben egy olyan megoldást mutattunk be, amelyik ötvözi a kriptográfiát, a szoftver vízjelezést és az obfuszkálást annak érdekében, hogy egy megalapozott és megbízható technikai megoldást eredményezzen a szoftver másolásvédelem területén, elsősorban a mobiltelefon alkalmazási területét célozva meg. A bemutatott módszerre alapozva terveztük meg egy olyan másolásvédelmi eszköz architektúráját, amely integrálható bármely fejlesztői környezetbe annak érdekében, hogy megfelelő másolásvédelmi szolgáltatásokat biztosítson.

A rendszer felépítése robusztus és nyílt abban az értelemben, hogy az az alrendszer, ami mind a vízjelezéssel, mind pedig az obfuszkálással járó átalakításokért felelős teljes mértékben független a processzortól, az operációs rendszertől és a fejlesztői környezettől, hiszen a forráskód egy absztrakt reprezentációján működik. Ily módon, lecserélve az előfeldolgozó, fordító és szerializáló modulokat, számos platformra és fejlesztői környezetbe is integrálhatjuk az általunk kifejlesztett rendszert.

A rendszer megbízható működése érdekében bármely kód-transzformáció esetében az elvégzett műveletek helyességének a formális bizonyítása elengedhetetlen. Ennek megfelelően minden transzformációt két lépésben valósítunk meg: az egyes transzformációk megtervezése, azaz a transzformációk céljainknak legjobban megfelelő sorozatának a létrehozása után a különálló és sokkal egyszerűbb transzformációs lépéseket úgy kell végrehajtanunk, hogy az általuk végrehajtott műveletek helyessége már formálisan is bizonyítható legyen.

A keretrendszer elkészülte után a kutatás-fejlesztési projektünk következő lépéseként további transzfor-

A példa egy API hívást szemléltet a rendszerhívások elrejtését megvalósító transzformáció végrehajtása után:

```
    ldr lr, LI11            @ Visszatérési cím elmentése
    mov ip, #1              @ Ip beállítása a hívandó függvény azonosítójára
    ldr r5, LI12            @ Globális változó címének betöltése
    str ip, [ r5, #0]       @ Azonosító elmentése a globális változóba
    b HideCalls_2           @ Interfész függvény meghívása
LI11:
    .align 0
    .word .L12              @ Következô blokk címe
LI12:
    .align 0
    .word .LD110            @ Globális változó címe
.L12:
```

mációk megvalósítása következik. Célunk egyrészt különböző kontroll és adat obfuszkáló eljárások kifejlesztése és hatékonyságának a tesztelése, másrészt pedig a transzformációk révén a dinamikus vízjelek elrejtése a kódban, majd ezek detektálhatóságával kapcsolatos mérések elvégzése.

Köszönetnyilvánítás

A kutatási projektet a Gazdasági Versenyképesség Operatív Programja (GVOP 3.1.1/AKF) támogatta.

Irodalom

[1] G. Arboit:
A Method for Watermarking Java Programs via Opaque Predicates,
In The Fifth International Conference on Electronic Commerce Research (ICECR-5), 2002.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan and K. Yang:
On the (im)possibility of obfuscating programs,
In: Proc. CRYPTO'01.
Lecture notes in computer science, Vol .2139.
Springer, pp.1–18, 2001.

[3] First Annual BSA and IDC
Global Software Privacy Study, 2004.
Business Software Alliance and IDC Global Software

[4] C. Collberg, C. Thomborson, and D. Low:
A Taxonomy of Obfuscating Transformations,
Technical Report 148, Dept. of Computer Science,
The Univ. of Auckland, 1997.

[5] C. Collberg, and C. Thomborson:
On the Limits of Software Watermarking,
Technical Report 164, Dept. of Computer Science,
The Univ. of Auckland, 1998.

[6] C. Collberg, C. Thomborson, and G. M. Townsend:
Dynamic Graph-Based Software Watermarking,
Technical Report TR04-08, 2004.

[7] R. Davidson, and N. Myhrvold:
Method and system for generating and auditing a signature for a computer program,
US Patent 5,559,884, Microsoft Corporation, 1996.

[8] G. Hachez:
A Comparative Study of Software Protection Tools Suited for E-Commerce with Contributions to Software Watermarking and Smart Cards,
Ph.D. thesis,
Universite Catholique de Louvain, 2003.

[9] International Telegraph and Telephone Consultative Committee (CCITT):
The Directory – Authentication Framework,
Recommendation X.509, 1988.

[10] A. Mana, J. Lopez, J. J. Ortega, E. Pimentel and J. M. Troya:
A framework for secure execution of software,
International Journal of Information Security,
Vol. 2, Issue 4, pp.99–112, Springer,
November 2004.

[11] A. Monden., H. Iida, and K. Matsumoto:
A Practical Method for Watermarking Java Programs,
The 24th Computer Software and Applications Conference (compsac2000),
Taipei, Taiwan, October 2000.

[12] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang:
Expreience with Software Watermarking
In Proc. of the 16th Annual Computer Security Applications Conference, ACSAC'00,
pp.308–316, 2000.

[13] J. P. Stern, G. Hachez, F. Koeune, and J.-J. Quisquater:
Robust Object Watermarking: Application to Code,
In A. Pfitzmann, editor, Information Hiding '99,
Vol. 1768 of Lectures Notes in Computer Science,
pp.368–378, Dresden, Germany, 2000.

[14] R. Venkatesan, V. Vazirani, and S. Sinha:
A Graph Theoretic Approach to Software Watermarking,
In Proceedings of the 4th International Workshop on Information Hiding table of contents,
pp.157–168, 2001.

[15] G. Wroblewski:
General Method of Program Code Obfuscation,
Ph.D. thesis, Wroclaw University of Technology,
Institute of Engineering Cybernetics, 2002.