Programkód–elemzés és –módosítás

Ph.D. értekezés tézisei

Kiss Ákos
Témavezető: Dr. Gyimóthy Tibor

Matematika és Számítástudományok Doktori Iskola

Szegedi Tudományegyetem
Természettudományi és Informatikai Kar
Szoftverfejlesztés Tanszék

Szeged, 2008

Bevezetés

„Bár a szoftverfejlesztő közösség figyelme sokszor – helyesen – a rendszerfejlesztés és -evolúció más aspektusai felé irányul (például: specifikáció-, terv- és követelményelemzés), még mindig a forráskód egy rendszer működésének egyetlen pontos leírása. Emiatt a forráskód elemzése és módosítása továbbra is fontos probléma.”

A fenti mondatok a SCAM (Source Code Analysis and Manipulation – Forráskód–elemzés és –módosítás) nevű, évenként megrendezésre kerülő konferencia mottójából származnak, és ezek a mondatok motiválták a szerzőt kutatásai során. A forráskód elemzésének és módosításának területe óriási: felöleli többek között a programtranszformáció, az absztrakt interpretáció, a programszeletelés, a forráskód szintű szoftvermetrikák, a visszafordítás, a forráskód szintű tesztelés és ellenőrzés, a forráskód-optimalizálás, és a programmegértés témaköreit. Eme nagyszámú témából a szerző három területre koncentrált: a programszeletelés elméleti megalapozására, a bináris programok szeletelésére, valamint C++ nyelvű programok obfuszkálására.

A szerző beismeri, hogy a binárisok szeletelése oda nem illőnek tűnhet a forráskód–elemzés szövegkörnyezetében. A SCAM tudományos közösségén belül azonban a „forráskód” kifejezés használatos egy rendszer minden végrehajtható leírására. Így tehát ennek a definíciónak nem csak a magas szintű nyelvek felelnek meg, hanem a gépi kód is. Habár ez a megengedő definíció lefedi a szerző mindhárom kutatási témáját, a jelen tézisfüzetben összegzett értekezés címe *Programkód–elemzés és –módosítás* lett, hogy jobban illeszkedjen a szoftverfejlesztő közösség szélesebb rétegeinek szóhasználatához.

Az értekezés négy tézist tartalmaz, amelyek a következők:

1. A programprojekció elméletének egyesített keretrendszere.
2. Különböző szeletelési módszerek kapcsolatának elemzése.
3. Bináris programok függőségi gráf alapú szeletelése.
4. C++ programok vezérlési folyamának lapítása.

A továbbiakban a tézisfüzet az értekezés szerkezetét követi: három fő részből áll, melyek megfelelnek a szerző kutatási témáinak. A következő fejezetekben bemutatásra kerül a kódelemzés és -módosítás három területe, valamint kifejtésre kerülnek a fent felsorolt tézispontok. Ennek során a szerző saját hozzájárulása az eredményekhez külön hangsúlyt kap.

A programszeletelés elmélete

A programszeletelés egy eredetileg 1979-ben Mark Weiser által megfogalmazott módszer [24] annak meghatározására, hogy egy program mely részei befolyásolják egy számunkra érdekes változóhalmaz értékét. Azzal, hogy csak néhány változó kiszámítására koncentrálunk, a szeletelés lehetővé teszi a program azon részeinek az eltávolítását, amelyek nem hatnak a változókra. Így a program mérete csökkenthető. Ezt a csökkentett méretű programot nevezzük szeletnek.

A továbbiakban a szeletelés formális definícióival és tulajdonságaival fogunk foglalkozni (nem pedig a kiszámításukra használt algoritmusokkal). Ehhez felhasználjuk a programszeletelés Harman, Danicic és Binkley által kidolgozott projekciós elméletét [9, 10], amelyet ők először az amorf és a szintaxistartó (ld. Weiser) szeletelések hasonlóságainak és különbségeinek vizsgálatára használtak. Ebben az értekezésben arra használjuk a projekciós elméletet, hogy a Korel és Laski által megadott dinamikus szeletelés [14] természetét is megvizsgáljuk.

Az egyesített keretrendszer

Általános vélekedés, hogy minden statikus szelet egyben Korel és Laski féle dinamikus szelet is egyben, még ha túlságosan nagy is. Intuitíven azt várjuk, hogy a dinamikus szeletelési kritérium lazább a statikusnál, mivel a program szemantikájának megőrzését nem az összes lehetséges, hanem csak egyetlen kiválasztott bemenetre követeli meg. Emellett a dinamikus szeletelési kritérium csak egyetlen előfordulását veszi egy utasításnak a program végrehajtási útjából, ellentétben a statikus szeleteléssel, ahol a számunkra érdekes utasítás minden előfordulása fontos.

Mint azonban az 1. ábra mutatja, Korel és Laski (KL) dinamikus szeletelési

1 `x=1;`	1 `x=1;`
2 `x=2;`	
3 `if (x>1)`	3 `if (x>1)`
4 `  y=1;`	4 `  y=1;`
5 `else`	5 `else`
6 `  y=1;`	6 `  y=1;`
7 `z=y;`	7 `z=y;`
p_1: Eredeti program	q_1: Az ({y},7) kritériumhoz tartozó szelet

1. ábra. Egy statikus szelet, ami nem KL szelet.

definíciója nem összehasonlítható a statikus szeleteléssel. Az 1. ábrán látható q_1 program érvényes statikus szelete p_1-nek az ({y},7) kritériumra nézve, de nem KL dinamikus szelete a $(\langle\rangle, 7^4, \{\mathtt{y}\})$ kritériumra – mivel a KL szeletelés nem engedi meg, hogy egy program és szeletének végrehajtási útja különbözzön. Észrevehetjük, hogy a KL dinamikus szeletelés és a statikus szeletelés összehasonlíthatatlanságának az oka az, hogy a KL szeletelés egyrészt „lazább", mivel csak egyetlen bemenetre őrzi meg a program viselkedését (ezt is várjuk), de egyben szigorúbb is a végrehajtási út felé támasztott követelmény miatt.

Miután azonosítottuk a KL szeletelés és a statikus szeletelés összehasonlíthatatlanságának fő okát, megpróbáltuk a KL szeletelést beilleszteni a projekciós elmélet keretrendszerébe. Az eddig használt definíciók [9, 10] azonban nem tudták leírni a végrehajtási úttal szemben támasztott követelményt. Ezért tehát kiterjesztettük a definíciókat. A kiterjesztés lehetővé tette, hogy felismerjük, hogy a KL szeletelési kritérium még egy komponenst rejt: az iterációs számot. Ezek hatására lehetővé vált, hogy felállítsunk egy egyesített szemantikus ekvivalenciareláción alapuló egységes keretrendszert, amely képes Korel és Laski dinamikus szeletelésének a leírására is.

1. definíció (Egyesített ekvivalencia)**.** Legyed adott két program, p és q, S állapothalmaz, V változóhalmaz, P, sorszámok és természetes számok alkotta párok halmaza, valamint X függvény, mely két sorszám-halmazhoz egy sorszám-halmazt rendel. Ekkor az $\mathcal{U}$ egyesített ekvivalenciát a következőképpen definiáljuk:

$$p \quad \mathcal{U}(S,V,P,X) \quad q$$

akkor és csak akkor, ha

$$\forall\sigma\in S : Proj^*{}_{(V,P,X(\overline{p},\overline{q}))}(T_p^\sigma) = Proj^*{}_{(V,P,X(\overline{p},\overline{q}))}(T_q^\sigma)$$

ahol $\overline{p}$ és $\overline{q}$ jelöli p and q sorszámainak halmazát, és T_p^σ, T_q^σ jelöli p-nek és q-nak σ-n történő végrehajtása során keletkező állapot-trajektóriákat. A $Proj^*$ segédfüggvény definícióját az értekezés tartalmazza.

A fenti definícióban használt paraméterek szerepe a következő: S jelöli azokat a kezdeti állapotokat, amik esetén az ekvivalenciának teljesülnie kell. Ez írja le a szeletelési kritérium „bemeneti" részét. V jelöli a szeletelés szempontjából fontos változók halmazát, ez azonos mindenféle szeletelési kritérium esetén. A P paraméter jelöli ki a végrehajtási úton a számunkra érdekes pontokat (utasításokat), valamint megragadja a szeletelési kritérium „iterációs szám" részét. Végezetül X az, ami leírja a „végrehajtási út követelményt". Ezt egy függvény formájában teszi, amely meghatározza mely utasításokat kell a szelet végrehajtása során megőrizni.

Az 1. definícióba a megfelelő paramétereket illesztve egy új ekvivalencia relációt kapunk, amely leírja Korel és Laski dinamikus szeletelésének szemantikáját.

2. definíció (Korel és Laski féle dinamikus ekvivalencia)**.** Legyen σ egy állapot, V változók egy halmaza, $n^{(k)}$ pedig egy sorszám-természetes szám pár. Ekkor a Korel és Laski féle dinamikus ekvivalenciát ($\mathcal{D}_{KLi}$) a következőképpen definiáljuk:

$$\mathcal{D}_{KLi}(\sigma, V, n^{(k)}) = \mathcal{U}(\{\sigma\}, V, \{n^{(k)}\}, \cap).$$

Az értekezésben bebizonyítjuk, hogy a 2. definíció pontosan megfeleltethető Korel és Laski eredeti definíciójának.

1. tétel. *Egy p' program akkor és csak akkor Korel és Laski féle dinamikus szelete p-nek a (x, I^q, V) dinamikus szeletelési kritérium szerint, ha p' program $(\sqsubseteq, \mathcal{D}_{KLi}(\sigma, V, n^{(k)}))$ projekciója p-nek, ahol $\sigma = x$, $n = I$, és q jelöli az n utasítás T_p^σ-beli k-adik előfordulásának pozícióját.*

Az egyesített ekvivalenciával nem csak Korel és Laski dinamikus ekvivalenciáját tudjuk kifejezni, de újradefiniálhatjuk Weiser statikus ekvivalenciáját is.

3. definíció (Hagyományos statikus ekvivalencia)**.** Legyen V változók egy halmaza, n pedig egy sorszám. Ekkor

$$\mathcal{S}(V, n) = \mathcal{U}(\mathbf{\Sigma}, V, \{n\} \times \mathbf{N}, \varepsilon)$$

ahol $\mathbf{\Sigma}$ jelöli az összes lehetséges állapot halmazát, valamint $\varepsilon(x, y) = \emptyset$ minden x és y sorszám-halmazra.

Most már, hogy azonosítottuk a szeletelési kritérium ortogonális komponenseit (kezdőállapotok halmaza, végrehajtási út, iterációs szám), felismerhetjük, hogy az $\mathcal{S}(V, n)$ és $\mathcal{D}_{KLi}(\sigma, V, n^{(k)})$ szemantikus ekvivalenciarelációk nyolc lehetséges ekvivalenciareláció egy-egy végletét jelentik. Három ortogonális kritérium-elem van, tehát közöttük még létezik hat további ekvivalenciareláció, melyeket az egyesített ekvivalencia további lehetséges paraméterezéseiből kaphatunk. Ezeket az ekvivalenciarelációkat alább definiáljuk (a teljesség kedvéért feltüntetve a már bemutatott relációkat is).

4. definíció (Nyolc ekvivalencia).

$$
\begin{array}{lcl}
\mathcal{S}(V,n) & = & \mathcal{U}(\mathbf{\Sigma}, V, \{n\} \times \mathbf{N}, \varepsilon), \\
\mathcal{S}_i(V,n^{(k)}) & = & \mathcal{U}(\mathbf{\Sigma}, V, \{n^{(k)}\}, \varepsilon), \\
\mathcal{D}(\sigma,V,n) & = & \mathcal{U}(\{\sigma\}, V, \{n\} \times \mathbf{N}, \varepsilon), \\
\mathcal{D}_i(\sigma,V,n^{(k)}) & = & \mathcal{U}(\{\sigma\}, V, \{n^{(k)}\}, \varepsilon), \\
\mathcal{S}_{KL}(V,n) & = & \mathcal{U}(\mathbf{\Sigma}, V, \{n\} \times \mathbf{N}, \cap), \\
\mathcal{S}_{KLi}(V,n^{(k)}) & = & \mathcal{U}(\mathbf{\Sigma}, V, \{n^{(k)}\}, \cap), \\
\mathcal{D}_{KL}(\sigma,V,n) & = & \mathcal{U}(\{\sigma\}, V, \{n\} \times \mathbf{N}, \cap), \\
\mathcal{D}_{KLi}(\sigma,V,n^{(k)}) & = & \mathcal{U}(\{\sigma\}, V, \{n^{(k)}\}, \cap).
\end{array}
$$

A fenti nyolc ekvivalenciarelációból hat új, amik eddig még nem tárgyalt szeletelési módszerek szemantikus tulajdonságait írják le.

A fölérendeltségi reláció

Az $\mathcal{S}$, $\mathcal{S}_i$, $\mathcal{D}$, $\mathcal{D}_i$, $\mathcal{S}_{KL}$, $\mathcal{S}_{KLi}$, $\mathcal{D}_{KL}$ és $\mathcal{D}_{KLi}$ ekvivalenciarelációk valójában ekvivalenciareláció osztályok, mivel σ-val, V-vel, n-nel és k-val vannak paraméterezve. Ezen osztályok között definiálható a $\approx_B \subseteq \approx_A$ fölérendeltségi reláció, ahol $\approx$ paraméteres ekvivalenciarelációt jelöl.

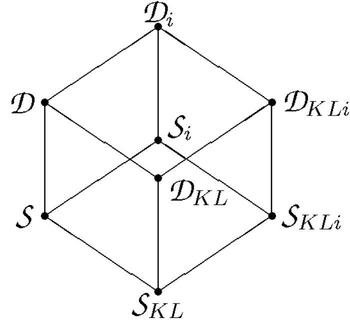
5. definíció (Fölérendeltségi reláció). Legyen $\approx_A$ és $\approx_B$ két ekvivalenciareláció, melyeket σ, V, n és k-val paraméterezünk. $\approx_A$ akkor és csak akkor fölérendeltje $\approx_B$-nek (amit a következőképpen jelölünk: $\approx_B \subseteq \approx_A$), ha

$$\forall \sigma, V, n, k : \approx_B^{(\sigma,V,n,k)} \subseteq \approx_A^{(\sigma,V,n,k)}.$$

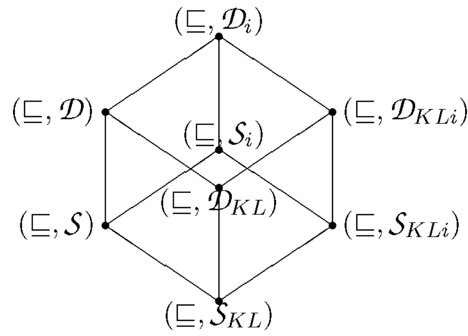
A fenti fölérendeltségi reláció a paraméteres ekvivalenciarelációk egy részben rendezése. A 2. ábra tartalmazza az $\mathcal{S}$, $\mathcal{S}_i$, $\mathcal{D}$, $\mathcal{D}_i$, $\mathcal{S}_{KL}$, $\mathcal{S}_{KLi}$, $\mathcal{D}_{KL}$ és $\mathcal{D}_{KLi}$ ekvivalenciarelációk és a fölérendeltségi reláció által alkotott hálót (pl.: $\mathcal{D}$ fölérendeltje $\mathcal{S}$-nek). Mint látható, a statikus és dinamikus szeletelések szemantikus aspektusai közötti kapcsolat nem olyan egyszerű, mint más szerzők azt korábban feltételezték [6, 8, 22]. Az értekezésben azt is bebizonyítjuk, hogy a 2. ábrán látható rajz helyes:

2. tétel. *A 2. ábrán látható háló helyes: két paraméteres ekvivalenciareláció akkor és csak akkor van összekötve az ábrán, ha azok fölérendeltségi kapcsolatban vannak.*

Eddig nyolcfajta szeletelés *szemantikus* tulajdonságai közötti kapcsolatot vizsgáltunk. Általában azonban a szeletelések közötti kapcsolatok érdekelnek minket. Hogy ezt megvizsgálhassuk, figyelembe kell vennünk a szemantikus ekvivalenciareláció mellett a szintaktikus rendezést is. Egy ilyen relációpárt



2. ábra. Az ekvivalenciarelációk közötti fölérendeltségi viszony.



3. ábra. Szeletelési módszerek fölérendeltségi viszonya.

a továbbiakban szeletelési módszernek nevezünk és rájuk is kiterjesztjük a fölérendeltségi relációt.

A szeletelési módszereken értelmezett fölérendeltség definíciója szoros kapcsolatban áll a paraméteres szemantikus ekvivalenciarelációkon éretelmezett fölérendeltségi relációval. Azaz amennyiben $\approx_A$ fölérendeltje $\approx_B$-nek, úgy a $(\lesssim, \approx_A)$–szeletelés is fölérendeltje a $(\lesssim, \approx_B)$–szeletelésnek. Ennek segítségével bebizonyítható a 3. ábra helyessége, amely megmutatja a $\sqsubseteq$ hagyományos szintaktikus rendezés és a $\mathcal{S}$, $\mathcal{S}_i$, $\mathcal{D}$, $\mathcal{D}_i$, $\mathcal{S}_{KL}$, $\mathcal{S}_{KLi}$, $\mathcal{D}_{KL}$, $\mathcal{D}_{KLi}$ paraméteres ekvivalenciarelációk párosításával kapott szeletelési módszerek kapcsolatát.

A nyolc ekvivalenciareláció és a belőlük származtatott szeletelési módszerek kapcsolatának elemzéséből született eredmények ugyan elméleti síkon is érdekesek, de gyakorlati szempontból is fontosak. Segítségükkel a szeletelést használók jobban megérthetik az egyes szeletelési definíciókat, és az adott feladathoz legmegfelelőbbet választhatják ki.

Szeletelési módszerek szintaktikus rendezése

Néha hallhatók olyan kijelentések a szeleteléssel foglalkozó kutatók között, hogy „a dinamikus szeletek kisebbek a statikus szeleteknél". Intuitíven értjük, hogy ez mit kíván jelenteni, de világos, hogy nem *minden* dinamikus szelet kisebb minden statikus szeletnél. Egy lehetséges értelmezése a fenti kijelentésnek, hogy a dinamikus szeletelés által elérhető *legkisebb* szeletek kisebbek a statikus szeletelés által elérhető *legkisebb* szeleteknél.

Hogy ezeket a meglátásokat formalizálhassuk és meghatározhassuk, hogy egy szeletelési definíció valóban kisebb szeleteket tesz-e lehetővé egy másiknál, ahhoz kiterjesztjük a szintaktikus rendezési relációt szeletelési módszerekre. A kiterjesztés minimális szeletek halmazainak összehasonlításán alapul (mivel a minimális szeletek nem feltétlenül egyediek).

Az értekezésben kimondunk egy tételt szeletek halmazai és minimális szeletek halmazai kapcsolatáról. Informálisan: ha egy adott programból A projekcióval kapott szeletek valós szeletek B projekció szerint is, akkor B minimális szeletei kisebbek, mint A minimális szeletei. (Érdekes módon a tétel nem megfordítható.)

A tézisfüzetben csak informálisan kimondott fenti tétel biztosítja az alapot a szeletelési módszerek összehasonlítására. Lehetővé teszi, hogy megmutassuk, hogy duális kapcsolat létezik a szeletelési módszerek fölérendeltségi relációja és szintaktikus rendezése között.

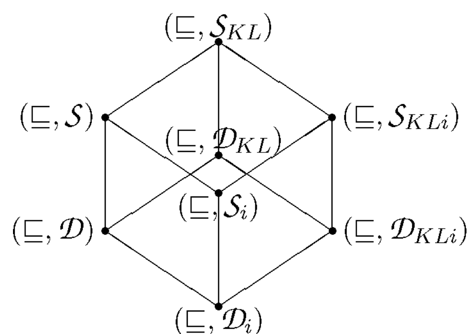
3. tétel (Szeletelési módszerek dualitása). *Bármely* $(\lesssim, \approx_A)$ *és* $(\lesssim, \approx_B)$ *szeletelési módszer esetén (ahol* $\lesssim$ *olyan szintaktikus rendezés, hogy tetszőleges programhalmazban van* $\lesssim$*-re nézve minimális elem) teljesül, hogy*

$$(\lesssim, \approx_A) \subseteq (\lesssim, \approx_B) \Rightarrow (\lesssim, \approx_B) \lesssim (\lesssim, \approx_A).$$

Ez a tétel kimondja, hogy amennyiben B szeletelési módszer fölérendeltje A-nak, akkor B minimális szeletei kisebbek lesznek mint A minimális szeletei. Azaz A várhatóan nagyobb szeleteket fog eredményezni.

Habár a 3. tétel megfordítása nem igaz általános esetben, a nyolc ekvivalenciareláció (ld. 4. definíció) és a hagyományos szintaktikus rendezés párosításából kapott szeletelési módszerekre erősebb állítást tehetünk. Ez a nyolc reláció egy, a 3. ábrán látható hálóval izomorf (bár invertált) hálót alkot. Ez látható a 4. ábrán.

A 3. tétel kimondja, hogy amennyiben két szeletelési módszer kapcsolatban van a 3. ábrán, azaz fölérendeltségi relációban állnak, akkor a szintaktikus rendezési reláció szerint fordított kapcsolatban állnak. Ezek szerint a 4. ábra helyességének „ha" iránya bizonyított. Mivel azonban a $(\lesssim, \approx_A) \nsubseteq (\lesssim, \approx_B)$ viszonyból nem következik $(\lesssim, \approx_B) \not\lesssim (\lesssim, \approx_A)$, ezért be kell lát-



4. ábra. A hagyományos szintaktikus rendezés szerint rendezett szeletelési módszerek.

ni, hogy a 4. ábrán kapcsolatban nem álló módszerek a hagyományos szintaktikus rendezés szerint sem állnak kapcsolatban. Ezt a következő tételben mondjuk ki (az értekezésben pedig be is bizonyítjuk):

4. tétel (A nyolc szeletelési módszer dualitása (csak akkor irány))**.** *Ha két szeletelési módszer nincsen kapcsolatban a 4. ábrán, akkor nincsenek hagyományos szintaktikus rendezési relációban.*

A fenti eredmények kapcsolatot teremtenek a szeletelési módszerek két alapvető relációja, a fölérendeltség és a szintaktikus rendezés között. A fölérendeltségi reláció meghatározza, hogy mely szeletelés használható egy másik helyett, míg a szintaktikus rendezés melyik módszer eredményez jobb (értsd: kisebb) szeletet.

Tézispontok és a szerző hozzájárulása az eredményekhez

1. A programprojekció elméletének egyesített keretrendszere

Weiser statikus, valamint Korel és Laski dinamikus szeletelésének összehasonlításából a szerző felismerte, hogy a dinamikus szeletelési kritérium nem csak a bemenettel bővíti a statikus kritériumot, hanem két további elemmel is rendelkezik. A dinamikus kritérium két további komponensének felfedezése lehetővé tette, hogy a szerző megalkossa a programprojekció elméletének egyesített ekvivalenciáját és egyesített keretrendszerét. A szerző így a két szeletelési módszert (a statikust és a dinamikust) egy közös keretrendszerbe helyezhette. A létrehozott keretrendszer nem csak hogy lehetővé tette Weiser valamint Korel és Laski már létező és jól ismert szeletelési módszerének

újradefiniálását, de hat új, eddig még nem ismert szeletelési módszer felismeréséhez is vezetett. (Ezen új szeletelési módszerek bemutatása a szerző és szerzőtársai közös munkája.)

2. Különböző szeletelési módszerek kapcsolatának elemzése

A szerző definiálta különféle szeletelések szemantikus aspektusain a fölérendeltségi relációt, majd az egyesített ekvivalencia segítségével megmutatta, hogy a disszertációban bemutatott nyolc szeletelés szemantikus része hálót alkot. Mindemellett a szerző azt is megmutatta, hogy a nyolc szeletelési módszer közötti fölérendeltségi viszony akkor sem változik, ha a szeletelési módszereknek nem csak a szemantikus aspektusát de a szintaktikus komponensét is vizsgáljuk.

Mivel a szeletek mérete a szeletelés minden alkalmazási területén fontos, a szerző megvizsgálta a szeletelési módszerek által lehetővé tett minimális szeleteket. A szerző úgy találta, hogy a szeletelési módszerek a minimális szeletek halmazai alapján rendezhetők, és az így kapott rendezés a fölérendeltségi reláció duálisa. A szerző megmutatta, hogy a nyolc korábban említett szeletelési módszer esetén ez a rendezés hálót alkot, ami a fölérendeltségi reláció hálójának tükörképe.

Bináris programok szeletelése

Weiser eredeti ötletének publikálása óta mások számos szeletelési algoritmust javasoltak [19, 14, 11, 6, 9, 1]. Eredetileg ezeket az algoritmusokat magas szintű, strukturált programok szeletelésére tervezték. Míg a magas szintű nyelvekben írt programok szeletelésével a szakirodalom sok cikke foglalkozik, viszonylag kevés figyelmet kapott a bináris programok szeletelése. Cifuentes és Frabuolet ugyan publikált egy módszert [7] bináris programok intraprocedurális szeletelésére, ám használható interprocedurális megoldásról nincsen tudomásunk.

A megoldások létének hiánya nehezen érthető, hiszen a binárisok szeletelésének alkalmazási területe nagyon hasonló a magas szintű nyelvekéhez. Hovatovább, a forráskód nélküli programok (pl.: assembly programok, régi rendszerek, vírusok, összeszerkesztés után módosított programok) szeletelésének speciális alkalmazásai is vannak. (Idetartozik a forráskód visszanyerés, a bináris hibajavítás és a kódtranszformáció.)

Vezérlési folyamelemzés

A kódelemzés és -módosítás több területe is igényli a vezérlési folyam gráfot (control flow graph, CFG). A programszeletelés számára is előfeltétel, hogy rendelkezésre álljon a szeletelt program CFG-je. A bináris programok vezérlési folyamának elemzése során azonban problémákba ütközünk.

Egy bináris végrehajtható fájlban a program bájtok sorozataként van tárolva. Hogy képesek legyünk a program vezérlési folyamának elemzésére, először az alacsony szintű utasítások határait kell megtalálni. Azokon az architektúrákon, ahol *az utasítások változó hosszúak*, az utasításhatárok nem mindig azonosíthatóak egyértelműen. Olyan architektúrákon, amelyek egyszerre *több utasításkészletet* támogatnak, ott az egyes helyeken használt utasításkészlet meghatározásakor jelentkezik probléma. Ha a bináris ábrázolás *összefésüli a programkódot és az adatokat*, mint az a legtöbb elterjedt architektúrán igaz, akkor azok szétválasztását is meg kell tenni.

Feltéve, hogy azonosítottuk az utasításokat, elkezdhetjük a gráf csomópontjainak létrehozását. Először az *alapblokkokat* (basic block) kell meghatározni *blokk-kezdőutasítások* alapján. A kezdőutasítások közötti utasítások alkotják a program alapblokkjait, melyeket tovább csoportosítunk *függvényekké*. Végül minden függvényhez egy speciális *kilépési pontot* hozunk létre, hogy a függvény egyetlen kilépési pontját reprezentálja.

A CFG csomópontjait *vezérlési folyam*, *hívási* és *visszatérési élek* kötik össze, amik a program végrehajtása során lehetséges vezérlésátadásokat jelképezik. A lehetséges vezérlésátadások helyes felismeréséhez szükség van a gépi

utasítások működésének elemzésére. Itt már a nagyszámú utasítás kezelése is gondot okozhat, de a legnehezebb problémát azok az utasítások jelentik, amelyek esetében *a vezérlésátadás célpontja nem határozható meg egyértelműen.* Ezeknek az utasításoknak a helyes kezeléséhez két új CFG csomópont-típust kell bevezetni: az *ismeretlen függvényt* és az *ismeretlen blokkot.* Ezek az indirekt függvényhívások és ugrások célpontjait jelölik, és össze vannak kötve az indirekt vezérlésátadások összes lehetséges célpontjával.

További problémaforrás amikor a vezérlés nem függvényhívás formájában adódik át függvények között. Az *átlapoló* és *keresztbe ugró* függvények tipikus példák erre a problémára. Ezen szerkezetek esetében a vezérlést átadó függvény kilépési csomópontja a gráfban nem elérhető. Ennek korrigálására plusz vezérlési folyam élekkel kell bővíteni a gráfot az érintett függvények kilépési csomópontjai között.

A fenti problémák tárgyalása során néhány kérdést nyitva hagytunk: Hogyan azonosíthatjuk az utasításhatárokat? Hogyan találhatjuk meg az utasításkészletek váltását? Hogyan válasszuk szét a kódot és az adatot? Hogyan határozhatjuk meg a függvények határait? Hogyan azonosítsuk az indirekt ugrások és függvényhívások lehetséges célpontjait? Szerencsére a legtöbb futtatható fájlformátum [21, 17] képes tárolni a nyers bináris adat mellett további szimbólum és relokációs információkat, melyek felhasználhatók kód és adat szétválasztásához, segíthetnek függvényhatárok és utasításkészlet váltások felismerésében, vagy lehetővé teszik nem egyértelmű vezérlésátadások célpontjainak meghatározását. Több fordítórendszer és fájlformátum használata során szerzett tapasztalatunk azt mutatja, hogy megfelelő specifikációk megléte esetén a szükséges információ viszonylag könnyen megszerezhető.

Függőségi gráf alapú szeletelés

Miután az interprocedurális vezérlési folyam gráf elkészült, vezérlési- és adatfüggőségi elemzést végzünk a CFG minden függvényén, aminek hatására előállnak a programfüggőségi gráfok (program dependence graph, PDG).

Az adatfüggőségi elemzés során foglalkozni kell a magas szintű nyelvek és az alacsony szintű utasítások alapvető különbségeivel. Magas szintű nyelvekben az utasítások argumentumai jellemzően lokális vagy globális változók, vagy formális függvényparaméterek. Ezek a fogalmak azonban általában nem jelennek meg a bináris szintjén. Az alacsony szintű műveletek regisztereket, jelzőbiteket és memóriacímeket írnak-olvasnak. Ezért tehát minden műveletet leelemzünk, hogy megállapítsuk, hogy mely regisztereket és jelzőbiteket írja vagy olvassa. Emellett elemezni kell a műveletek memória-hozzáférését is. Egy konzervatív megközelítése ennek, hogy mindössze annyit állapítunk

meg, hogy a művelet olvas-e a memóriából vagy írja-e azt. További probléma, hogy magas szintű programokkal ellentétben a binárisokban az eljárások paraméterlistája nincs expliciten megadva, hanem megfelelő interprocedurális elemzéssel kell azt felderíteni.

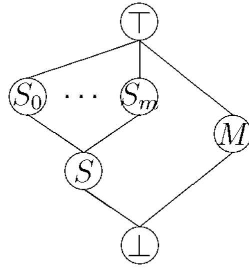
Az eddig tárgyalt módon elkészült PDG még csak intraprocedurális szeletek meghatározására használható. Ehhez a gráf vezérlési- és adatfüggőségi éleit kell bejárni [19]. Ha azonban összekötjük a PDG-ket *bemenő paraméter* és *kimenő paraméter* élekkel, valamint kiegészítjük *összegző* élekkel is, akkor létrejön a program rendszerfüggőségi gráfja (system dependence graph, SDG). Az így előálló SDG már alkalmas arra, hogy Horwitz és szerzőtársai [11] kétmenetes algoritmusát rajta végrehajtva interprocedurális szeleteket kapjunk.

A szeletelés pontosítása

Habár a fent leírt módon épített függőségi gráfok biztonságosak, sajnos túlságosan konzervatívak. Architektúra-függő információkkal azonban pontosíthatók. A legtöbb jelenlegi architektúrán ugyanis érvényben vannak különféle függvényhívási szokások, amelyek meghatározzák, hogy a függvényhívásoknak a regiszterkészlet mely részét kell módosítatlanul hagyniuk. Ha az ez miatt elmentett-visszatöltött regiszterek halmazát meg tudjuk határozni, akkor csökkenthetjük a függvények kimeneti paramétereinek számát.

Egy másik megközelítés az adatfüggőségi elemzés konzervatív memóriahasználat-kezelésének a pontosítása. Mivel a legtöbb architektúrán a rendelkezésre álló regiszterek száma véges, ezért azok az értékek és eredmények, amelyek már nem tárolhatók regiszterekbe, általában a verembe kerülnek. A korábban felvázolt memóriamodell azonban meglehetősen egyszerű, ezért az adatfüggőségi elemzés nem tudja pontosan felismerni a vermen keresztül létrejövő függéseket. Erre megoldásként egy továbbfejlesztett memóriamodellt javaslunk, amely a program minden műveletének belépési és kilépési pontjánál egy-egy hálóelemmel karakterizálja a regiszterek tartalmát. Az ehhez használt háló és annak elemei az 5. ábrán láthatók. (A $\perp$ hálóelem azt jelzi, hogy statikus elemzéssel nem eldönthető, hogy a regiszter hivatkozik-e a veremre vagy sem. Ha egy regiszterhez M van hozzárendelve, akkor az biztosan nem hivatkozik a veremre. Az S elem jelentése, hogy az adott regiszter valahova a verembe hivatkozik, de a pontos hely nem meghatározható. Végül valamely S_i azt jelzi, hogy a regiszter valamely ismert verempozícióra mutat.)

A hálóelemeknek a vezérlési folyam gráfon való propagálására használt fixpont iterációt a disszertációban írjuk le részletesebben. A propagáció eredményét felhasználva az adatfüggőségi analízis továbbfejleszthető és így elkerülhető, hogy a gráfba felesleges függőségi élek kerüljenek.



5. ábra. A regisztertartalom karakterizálására használt háló.

A szeletelés pontosítására használt harmadik megközelítés azon a megfigyelésünkön alapul, hogy a nagyszámú fel nem oldott indirekt függvényhívások sokszor okoznak túlságosan nagy szeleteket. Ezért követjük Mock és szerzőtársai ötletét [18], és *dinamikus* mutatóinformációkat használunk a statikus szeletelés során. Pontosabban fogalmazva, dinamikus információt gyűjtünk minden statikusan fel nem oldott függvényhívási helyen. Amennyiben a szeletelendő alkalmazást kontrollált környezetben, reprezentatív bemeneten hajtjuk végre, akkor lehetővé válik a statikusan fel nem oldott indirekt hívási helyek tényleges célpontjainak meghatározása. Így az ismeretlen függvény-csomóponthoz behúzott függvényhívási élek lecserélhetők a valós célpontokhoz behúzott élekkel. Kísérleteink azt mutatják, hogy a dinamikus élek felhasználása a hívási élek jelentős csökkenéséhez vezethet. Habár az így kapott hívási gráf és a belőlük számolt szeletek nem biztonságossá válhatnak, bizonyos felhasználási esetekben ez a korlátozás is elfogadható.

A statikus szeletelés kísérleti eredményei

Megvalósítottunk egy alkalmazást statikusan szerkesztett, végrehajtható ARM bináris állományokon szeletelésére és különböző tesztgyűjteményekből összegyűjtött programokon kiértékeltük. Az állományokban található kód mérete 12 és 419 kilobájt között mozgott.

Első lépésként felépítettük minden program CFG-jét a fentebb leírt módon. Mikor ezek előálltak, vezérlési- és adatfolyam elemzéseket végeztünk (mind a konzervatív, mind a statikus megoldásokkal javított módon), hogy megkapjuk minden elérhető függvény PDG-jét. Végül elkészítettük az SDG-ket. A statikus javítások az adatfüggőségi és összegző élek számában átlagosan 28% és 51%-os csökkenést eredményeztek (a legnagyobb csökkenések 44% és 58% voltak).

Miután minden tesztprogram SDG-jét előállítottuk, interprocedurális szeleteket számítottunk a függőségi gráfok alapján. Hogy egy adott kiválasztási

módszerből adódó befolyásolást elkerüljünk, minden olyan utasításra kiszámítottuk a szeletet, amely elérhető és forrásból fordított függvényben volt található (tehát nem a szerkesztési folyamat során került a programba). A konzervatív megközelítéssel kapott szeletek átlagosan 36%-71%-át tartalmazták a forrásokból származó utasításoknak, míg a javítások ezen további 1%-3%-kal csökkentettek.

Vannak olyan esetek (pl.: szerkesztés után módosított programok), amikor a függvénykönyvtárakból származó kódok is fontosak. Emiatt a könyvtárakból származó (elérhető) függvények utasításaira is számítottunk szeleteket. Az eredmények a fentiekhez hasonló tendenciát mutatnak: a konzervatív függőségi gráffal számított szeletek átlagosan az összes utasítás 52%-71%-át tartalmazták, amin a statikus javítások további 1%-4% csökkenést eredményeztek.

Vizsgálataink szerint a szeletek méretének mérsékelt csökkenésében fontos szerepe van a statikusan nem feloldott függvényhívások nagy számának.

A dinamikus pontosítás kísérleti eredményei

Hogy a kiválasztott tesztprogramokról dinamikus információt gyűjthessünk, a Texas Instruments emulátorában hajtottuk végre őket. Az összegyűjtött dinamikus információ birtokában pontosíthattuk az indirekt függvényhívásoknál a hívási gráfot. Mint az várható volt, a hívási élek száma jelentősen csökkent azon alkalmazások esetében, amelyek intenzíven használnak indirekt függvényhívást. Még azoknál az alkalmazásoknál is egyértelmű csökkenés volt kimutatható, amelyekben csak néhány indirekt függvényhívás és indirekten hívható függvény volt található.

Hogy a pontosabb hívási gráf hatását kimérhessük, szeleteket számítottunk mind a statikus, mind a dinamikusan pontosított hívási gráf felhasználásával. Mint korábban, most is el kívántuk kerülni a valamely kritérium-kiválasztási stratégiából fakadó eredménytorzulást, ezért minden olyan utasításra szeletet számítottunk, amelyek forrásból származó, és a tesztfuttatások során meghívott függvényekben találhatók. Az eredmények azt mutatják, hogy magas a korreláció a hívási élek számának csökkenése és a szeletek méretének csökkenése között. Azoknál a programoknál, amelyek nem használnak indirekt függvényhívást, – nem meglepő módon – a módszer nem hozott csökkenést. Két programnál, amelyek kevés számú indirekt hívást használtak, 6%-os méretcsökkenés volt megfigyelhető. Annál a két programnál azonban, amelyek intenzíven használtak indirekt függvényhívást, a dinamikusan pontosított hívási gráf használatával a statikus megközelítéshez képest 72% és 57%-os átlagos szeletméret-csökkenést értünk el.

Tézispontok és a szerző hozzájárulása az eredményekhez

3. Bináris programok függőségi gráf alapú szeletelése

Más kódelemzési módszerekhez hasonlóan a függőségi gráfon alapuló szeletelésnek is szüksége van vezérlési folyam gráfra. Ezért a szerző felderítette a bináris programok vezérlési folyamelemzésének problémáit és megoldásokat is javasolt. A disszertációban az elemzés leírása mellett a binárisok program- és rendszerfüggőségi gráfjainak építési módszere is megtalálható. (Ez a módszer azonban nem a szerző saját munkája.)

Mivel a bináris programok meglehetősen speciálisak, a szerző több lehetőséget is megvizsgált a szeletelés bináris-specifikus pontosítására. A pontosítások egyrészt statikusan csökkentik a függőségi gráfokban található adatfüggőségi és összegző élek számát, másrészt dinamikusan gyűjtött információ segítségével törölnek a statikus hívási gráfból éleket. Ezek a pontosítások a szerző és szerzőtársai közös munkájának eredményei, ahol a szerző saját hozzájárulása az eredményekhez a következő: a szerző dolgozta ki a veremelemzés pontosításához használt hálót, valamint a szerző részt vett a dinamikus pontosító módszer megtervezésében, és a kísérleti eredmények kiszámításához használt szeletelő prototípus megvalósításában is.

Vezérlési folyamlapítás alapú kódobfuszkálás

A szoftvergyártók számára mindig gondot okozott, hogy miként védhetik meg a programokat az illetéktelen hozzáféréstől. A cél általában a támadó dolgának a lehető legnagyobb mértékű megnehezítése. Alább a kódobfuszkálással foglalkozunk, amely első védelmi vonalként működik a programok védelmében, hiszen a célja annak megakadályozása, hogy a támadó megértse a kódot.

Habár napjainkban több nagyméretű rendszert is C++ nyelven írnak, csak kevés eszköz létezik a védelmükre. Mivel a C++ nyelvű program védelme nem elhanyagolható fontosságú, obfuszkáló technikák C++-ra való fejlesztését tűztük ki magunk elé célnak.

Fontos továbbá, hogy az algoritmusoknak nem csak a forráskód szinten, hanem a binárisra kifejtett hatására is kíváncsiak vagyunk, mivel sok támadás a bináris formában elérhető programok ellen irányul azon célból, hogy megkerüljék vagy kikapcsolják a védelmüket. Ezért azt is megvizsgáltuk, hogy egy forráskódot átalakító módszer megnehezítheti-e a belőle származó bináris kód megértését.

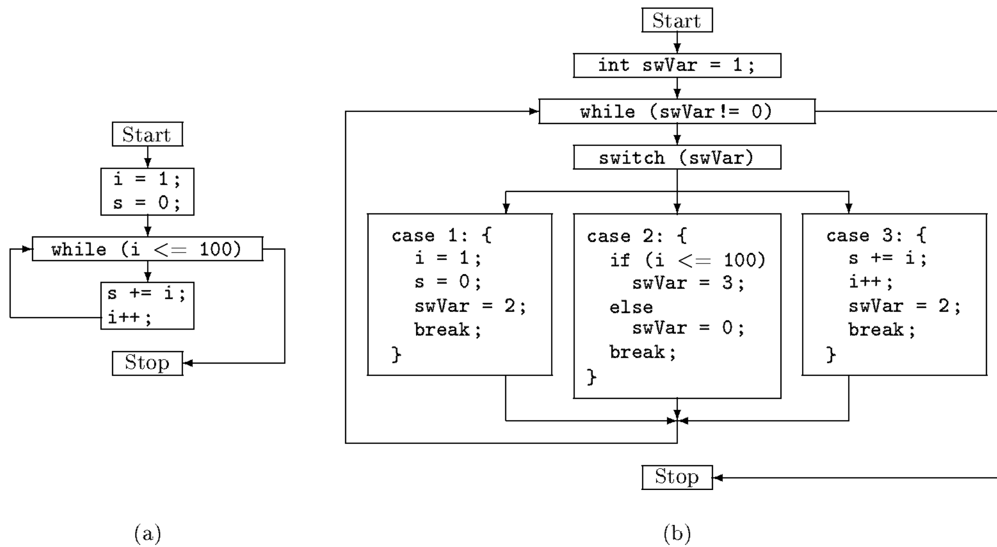
C++ programok lapítása

A továbbiakban a vezérlési folyamlapítás [23] nevű módszer C++ nyelvre adaptálását tárgyaljuk. A módszer alapötlete abban rejlik, hogy olyan módon alakítja át a forrást, hogy az átalakított kódban található ugrások célpontja statikus elemzéssel ne legyen egyszerűen meghatározható, így nehezítve a program megértését.

Egy függvény lapításának módszere alapvetően a következő: először a függvény törzsét alapblokkokra bontjuk, majd az összes így kapott blokkot, melyek eredetileg különböző módon egymásba voltak ágyazva, egymás mellé helyezzük. Ezeket a blokkokat ezután egy `switch` utasításba helyezzük úgy, hogy mindegyik blokk egy külön `case` címkét kap, majd az egész `switch` szerkezetet egy ciklusba helyezzük. Végül a vezérlés helyes folyását egy vezérlőváltozó bevezetésével biztosítjuk, amely a program állapotát jelképezi. Ezt a változót minden alapblokk végén megfelelő módon beállítjuk, és a befoglaló ciklus és `switch` utasítás predikátumában felhasználjuk. A 6. ábra példát mutat a módszer alkalmazására.

A fenti leírás alapján egy függvény lapítása meglehetősen egyszerűnek tűnik. Amikor azonban az ötletet egy valós programozási nyelvre kell alkalmazni, problémákba ütközünk.

Mint azt a 6. ábra szemléltette is, a ciklusok alapblokkokra bontása nem egyszerűen a ciklus fejének és törzsének szétválasztását jelenti. Az eredeti nyelvi szerkezet, azaz a `while`, `do` vagy `for` megtartása a lapított kódban



6. ábra. A vezérlési folyam lapításának hatása a vezérlési folyam gráfra.
(a: eredeti, b: lapított).

helytelen eredményre vezetne, mivel egy magányos ciklusfej a törzs nélkül nyilvánvalóan nem tudja reprodukálni az eredeti viselkedést.

Egy másik összetett utasítás, amelynek kezelése nem triviális, a `switch` szerkezet. Esetében a probléma oka a `switch` utasítás laza specifikációja, amely csak annyit ír elő, hogy a `switch` által vezérelt utasítás egy nyelvtanilag helyes (összetett) utasítás legyen, amiben a `case` címkék bármely részutasítás előtt elhelyezkedhetnek. (Egy érdekes példa, ami kihasználja ezt a laza specifikációt, a „Duff's device" [20].)

Nem hagyhatjuk ki a nem struktúrált vezérlésátadási utasítások említését sem. Ha módosítatlanul hagynánk őket a lapított kódban, a `break` és `continue` utasítások gondot okoznának, hiszen ahelyett, hogy azt a ciklust, vagy `switch` utasítást terminálnák vagy indítanák újra, amire az eredeti kód írója szánta őket, a lapítás után a vezérlő szerkezetre lennének hatással.

A C nyelvhez képest a C++ egy új vezérlési szerkezetet is bevezetett: a kivételkezelő `try-catch` szerkezetet. Ha a vezérlési folyam lapításának alapötletét közvetlenül alkalmaznánk egy `try` törzsére, azaz meghatároznánk az alapblokkokat és a vezérlő `switch` szerkezetbe helyeznénk őket, akkor azzal megsértenénk a kivételkezelés logikáját. Ekkor azokat az utasításokat, amelyeket kimozgatnánk a `try` törzséből, többé nem védené a kivételkezelő mechanizmus és a dobott kivételeket az eredeti kezelők nem tudnák elkapni.

Az értekezésben példák is találhatók a fent felsorolt problémák megoldásának segítésére, de egy algoritmus formális leírását is megadtuk, amelyet C++

függvények lapítására dolgoztunk ki.

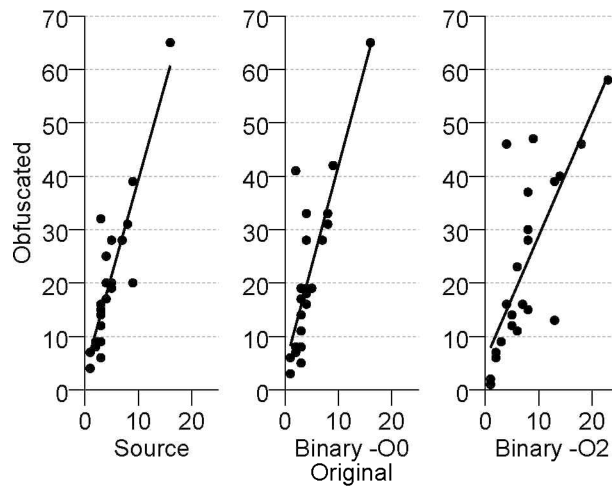
Kísérleti eredmények

Annak meghatározására, hogy mennyire hatékony a vezérlési folyam lapítása akár a forráskód akár az obfuszkált kódból fordított bináris kód védelmére, összegyűjtöttünk egy 23 függvényből álló teszthalmazt. Ezeket a függvényeket egy prototípus eszközzel obfuszkáltuk, amelyben megvalósítottuk az értekezésben leírt módszert. Az obfuszkálás előtt és után minden függvény forráskódján kimértük a McCabe féle ciklomatikus komplexitásmetrikát [16], hogy megállapítsuk a bonyolultságukban és megérthetőségükben bekövetkezett változást. Ezután mind az eredeti, mind az obfuszkált kódot lefordítottuk ARM architektúrára (be- és kikapcsolt fordítóprogram optimalizációkal egyaránt). A keletkezett binárisokat ugyanazzal az eszközzel elemeztük, amelyet a bináris szeleteléshez használtunk. Az eszköz segítségével kiszámoltuk az eredeti és az obfuszkált kód McCabe metrikáját és így a bináris programokra is kimértük a bonyolultságukban és megérthetőségükben elért változást.

A McCabe metrika változásait vizsgálva jelentős, 4,63-szoros növekedést mértünk átlagosan a forráskódon. A mérések továbbá igazolják a feltevésünket, hogy a forráskód obfuszkálásával a bináris programok bonyolultsága is növekszik. A bináris programokon mért McCabe metrikanövekedés hasonló a forráskódon mérthez: átlagosan 5,19-szoros optimalizálatlan és 3,26-szoros optimalizált kódon. Az optimalizált binárisokon elért, némileg kisebb növekmény a fordítóprogram által alkalmazott hatékony optimalizálási módszereknek köszönhető. A több mint háromszoros növekmény azonban így is jelentősnek tekinthető, és azt mutatja, hogy a fordítóprogram optimalizálások nem érvénytelenítik a forráskód obfuszkálásának a hatását.

Az eredmények elemzése azt mutatja, hogy az algoritmus hatása a bonyolultságra az eredeti bonyolultság lineáris függvénye. A 7. ábrán látható az obfuszkált kód bonyolultsága az eredeti kód bonyolultságának függvényében (mind forráskód, mind a bináris verziók esetére), valamint az adatokra lineáris regresszióval illesztett egyenesek.

A vezérlési folyamlapításnak a bonyolultságra tett hatásának kimérése mellett megvizsgáltuk a módszer erőforrásigényét is. Ennek érdekében megvizsgáltuk, hogy miként változott a tesztfüggvények mérete. Az eredmények azt mutatják, hogy az obfuszkált források mérete átlagosan kevéssel több mint kétszerese az eredetinek, míg a nem optimalizált és az optimalizált binárisokon mért méretnövekedés átlagosan mindössze 1,55-szoros és 1,57-szoros. Emellett megszámoltuk a végrehajtott utasításokat is. A mérések szerint a végrehajtott utasítások száma átlagosan 2,03-szoros és 2,39-szoros növekedést mutat nem optimalizált és optimalizált programok esetében. (Meg



7. ábra. Az eredeti és a lapított kód bonyolultsága közötti kapcsolat.

kell jegyeznünk, hogy valós környezetben a lapítást valószínűleg nem a teljes programra alkalmaznák, csak annak bizonyos kritikus függvényeire vagy moduljaira. Így valós alkalmazás esetén a teljes program erőforrásigényét érintő statikus és dinamikus hatások kisebbek lehetnek.)

Tézispontok és a szerző hozzájárulása az eredményekhez

4. C++ programok vezérlési folyamának lapítása

Annak érdekében, hogy lehetővé váljon C++ programok vezérlési folyamának a lapítása, a szerző azonosította azokat a nyelvi elemeket, amelyek kezelése nem triviális, majd megoldásokat adott a kezelésükre. A szerző továbbá egy algoritmust is kidolgozott, amely C++ nyelvű függvények lapítására képes és megadta annak a formális leírását. A szerző – szerzőtársával együttműködve – részt vett egy prototípus obfuszkáló-eszköz létrehozásában, és kísérleteket hajtott vele végre a vezérlési folyamlapításnak a kód megérthetőségére gyakorolt hatásának kiértékelése céljából. A kísérletek a forráskódot átalakító módszerek bináris kódobfuszkálásra való felhasználhatóságát is megvizsgálták.

Összegzés

A szerző a programkód elemzésének és -módosításának területéről három témakört mutatott be: a programszeletelés elméleti megalapozását, a programszeletelés alkalmazását bináris programokra, valamint a C++ nyelven írt programok obfuszkálását. A témákban végzett kutatások alább kerülnek összegzésre.

Mivel a szerző érdeklődik a szeletelés formális definíciói iránt, összevetette Weiser statikus szeletelését Korel és Laski dinamikus szeletelésével. Ennek eredménye a programprojekció elméletének a szerző által létrehozott egyesített keretrendszere. A létrehozott keretrendszer nem csak a már jól ismert szeletelési módszerek újradefiniálását tette lehetővé, de hat új, eddig nem ismertetett szeletelési módszer felismeréséhez is vezetett. A szerző definíciót adott továbbá a szeletelés módszerek fölérendeltségi relációjára és megmutatta, hogy nyolc szeletelési módszer hálót alkot. A szerző azt is felismerte, hogy a szeletelési módszerek a minimális szeletek halmazai alapján rendezhetőek, és az így kapott rendezés a fölérendeltségi reláció duálisa. Ezek az eredmények elméleti síkon is érdekesek, de gyakorlati szempontból is fontosak, mivel segítségükkel a szeletelést használók mindig az adott feladathoz legmegfelelőbb szeletelési definíciót választhatják ki.

Bár a szakirodalomban sok cikk jelent meg magas nyelven írt programok szeleteléséről, viszonylag kevés figyelmet kapott a bináris programok szeletelése. Az megoldások hiánya nehezen érthető, hiszen a forráskód nélküli programok szeletelésének speciális alkalmazási területei is vannak. Így tehát a szerző felderítette a bináris programok vezérlési elemzésének problémáit – mivel a függőségi gráf alapú szeleteléshez szükség van vezérlési folyam gráfra –, és megoldásokat is javasolt rájuk. Mivel a bináris programok elég speciálisak, a szerző ezért a szeletelés bináris-specifikus pontosításának több lehetőségét megvizsgálta (ide értve statikus és dinamikus megközelítéseket is). A kísérletek során kifejlesztésre és felhasználásra került egy szeletelő eszköz prototípusa.

Mivel a C++ programok védelme nem elhanyagolható fontosságú, a szerző olyan obfuszkáló technikák C++-ra való fejlesztését tűzte ki maga elé célnak, amelyek a programvédelem első védelmi vonalaként működhetnek. A szerző bemutatta a vezérlési folyamlapítás nevű technika C++ nyelvre történő adaptálását. A szerző azonosította azokat a nyelvi elemeket, melyek kezelése nem triviális, és megoldásokat adott a kezelésükre. A szerző továbbá egy algoritmust is kidolgozott C++ nyelvű függvények lapítására. A módszer implementálásra került egy obfuszkáló eszköz prototípusában, valamint a forráskódra és a bináris kódra gyakorolt hatása kiértékelésre került.

	[3]	[2]	[5]	[4]	[13]	[12]	[15]
1.	•		•				
2.		•	•	•			
3.					•	•	
4.							•

1. táblázat. Kapcsolat az értekezés tézispontjai és a felhasznált publikációk között.

Végül az 1. táblázat foglalja össze, hogy mely publikációk az értekezés mely tézispontjait fedik le.

Köszönetnyilvánítás

Először is köszönetet kívánok mondani Gyimóthy Tibornak, a témavezetőmnek, hogy fiatal kutatóként irányított, és mindig érdekes, megoldásra váró problémák elé állított. Köszönet illeti Dave Binkley-t, Sebastian Danicicot és Mark Harmant is, akiket mentoraimnak és barátaimnak is tartok egyben. Köszönet továbbá kollégáimnak és társszerzőimnek: Beszédes Árpádnak, David Curley-nek, Dobóczky Adrienne-nek, Ferenc Rudolfnak, Frittman Patríciának, Herczeg Zoltánnak, Jász Juditnak, Bogdan Korelnek, László Tímeának, Lehotai Gábornak, Lahcen Ouarbyának, Siket Istvánnak és Siket Péternek. Legyen a hozzájárulásuk kicsi vagy nagy, mind segítettek az értekezés elkészítésében. Végezetül ki szeretném fejezni a hálámat az állandó támogatásért édesanyámnak, édesapámnak és feleségemnek, Kingának.

Hivatkozások

[1] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. IEEE Computer Society, March 2001.

[2] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. Minimal slicing and the relationships between forms of slicing. In *Proceedings of the 5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005)*, pages 45–54, Budapest, Hungary, September 30 – October 1, 2005. IEEE Computer Society. Best paper award.

[3] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Lahcen Ouarbya. Formalizing executable dynamic and forward slicing. In *Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 43–52, Chicago, Illinois, USA, September 15–16, 2004. IEEE Computer Society.

[4] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. A formalisation of the relationship between forms of program slicing. *Science of Computer Programming*, 62(3):228–252, October 2006.

[5] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. Theoretical foundations of dynamic program slicing. *Theoretical Computer Science*, 360(1–3):23–41, August 2006.

[6] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[7] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural static slicing of binary executables. In *Proc. International Conference on Software Maintenance*, pages 188–195, October 1997.

[8] Chris Fox, Sebastian Danicic, Mark Harman, and Robert Mark Hierons. ConSIT: a fully automated conditioned program slicer. *Software—Practice and Experience*, 34:15–46, 2004. Published online 26th November 2003.

[9] Mark Harman, David Wendell Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, October 2003.

[10] Mark Harman and Sebastian Danicic. Amorphous program slicing. In 5th *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[11] Susan Horwitz, Thomas Reps, and David Wendell Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[12] Ákos Kiss, Judit Jász, and Tibor Gyimóthy. Using dynamic information in the interprocedural static slicing of binary executables. *Software Quality Journal*, 13(3):227–245, September 2005.

[13] Ákos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy. Interprocedural static slicing of binary executables. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 118–127, Amsterdam, The Netherlands, September 26–27, 2003. IEEE Computer Society.

[14] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[15] Tí mea László and Ákos Kiss. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientiarum de Rolando Eötvös Nominatae – Sectio Computatorica*, XXX, 2008. Accepted for publication.

[16] Thomas J. McCabe and Arthur H. Watson. Software complexity. *Crosstalk, Journal of Defense Software Engineering*, 7(12):5–9, December 1994.

[17] Microsoft Corporation. Microsoft Portable Executable and Common Object File Format specification version 6.0, February 1999. http://www.microsoft.com/hwdev/hardware/PECOFF.asp.

[18] Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. In *Proc. 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 71–80, November 2002.

[19] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184, 1984.

[20] Bjarne Stroustrup. *The C++ Programming Language*, chapter Expressions and Statements, page 141. Addison-Wesley, 3rd edition, 1997.

[21] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) version 1.2, May 1995.
http://www.x86.org/ftp/manuals/tools/elf.pdf.

[22] Guda A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada, June 1991. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[23] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, May 2000.

[24] Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, MI, 1979.