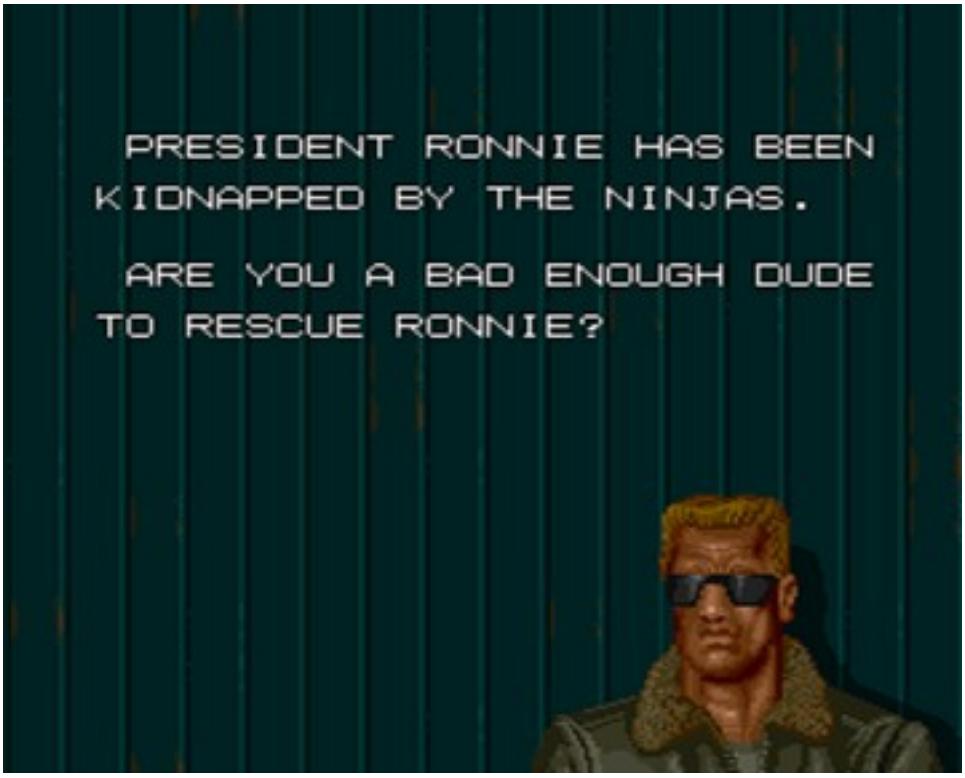
Om analys av fördunklad kod i skadlig programvara,

eller resultatet av två dygn utan sömn och fyra liter sekunda energidryck

av Benjamin Byholm

Xbo Akademi 2010

Referat


PRESIDENT RONNIE HAS BEEN
KIDNAPPED BY THE NINJAS.
ARE YOU A BAD ENOUGH DUDE
TO RESCUE RONNIE?

Innehållsförteckning

Inledning


BAD DUDES
Vs DRAGONNINJA
T.M.
NEW YORKER
DATA EAST
© 1988 DATA EAST USA, INC.

Skadeprogrammens evolution

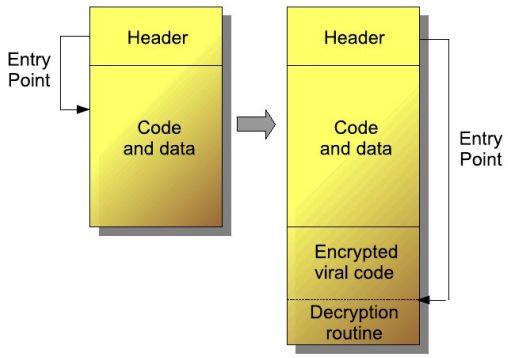
Vad kom först, hönan eller ägget? I detta fall fanns skadeprogrammen innan skadebekämpningsprogrammen. De första skadeprogrammen behövde inte dölja sig, då de inte hade några fiender. Detta ändrades i och med bekämpningsprogrammens ankomst. Terrorbalansen såg till att varje utveckling på den ena sidan kontrades från den andra sidan.

Det råder stor oenighet om när det första skadeprogrammet skrevs, delvis för att den allmänna definitionen på skadeprogram är väldigt subjektiv, men även med den striktare definitionen *virus* är det omöjligt att veta säkert. Man kan dock anta att den första datorn, som tillverkades av Charles Babbage 1822, var fri från skadeprogram.[1]

I resten av detta kapitel kommer det främst vara virus som diskuteras. Detta beror på att jag anser deras definition vara något mer objektiv än definitionen av skadeprogram samt att deras historia verkar vara något bättre kartlagd. Metoderna för att undgå upptäckt kan dock användas av samtliga skadeprogram, oberoende av huruvida de sprider sig eller ej.

Eftersom de första skadeprogrammen saknade skydd krävdes det inte särdeles mycket av de första bekämpningsprogrammen. I allmänhet gjorde de inte mer än att söka efter datasekvenser som identifierade skadeprogrammen. Under slutet av 80-talet utvecklades virus med krypterad last för att kringgå detta.[2]

Krypterad last



Med krypterad last menas att det egentliga programmet, *lasten*, är krypterat så att dess innehåll inte direkt kan observeras. För att ett krypterat program skall vara körbart krävs det dock att en liten programrutin som dekrypterar det egentliga programmet med den valda nyckeln körs först. För varje instans kan nyckeln ändras och därigenom ändras hela den krypterade lasten avsevärt.[2]

Eftersom dekrypteringsrutinen måste kunna exekveras kan den själv inte vara krypterad.[2] Visserligen kunde man lägga på ytterligare ett eller flera lager av kryptering, men det yttersta lagret måste ändå vara okrypterat. Eftersom denna rutin förblir oförändrad kan den i sin tur upptäckas utgående från sin datasignatur. [2] Detta problem ledde till utvecklandet av *oligomorfism*.

Oligomorfism

Till skillnad från virus med inget mer än krypterad last ändrar oligomorfa virus dekrypteringsrutinen mellan olika instanser. Ett enkelt sätt att göra detta på är att slumpmässigt välja dekrypteringsrutin från en uppsättning förutbestämda rutiner, ett annat sätt är att foga samman slumpmässigt valda mindre delar till en helhet. [3] Oligomorfa virus kan dock endast anta relativt få olika skepnader, varför de fortfarande kan upptäckas med en tillräcklig uppsättning statiska signaturer.[3] En avsevärt bättre vidareutveckling kom i form av *polymorfism*.

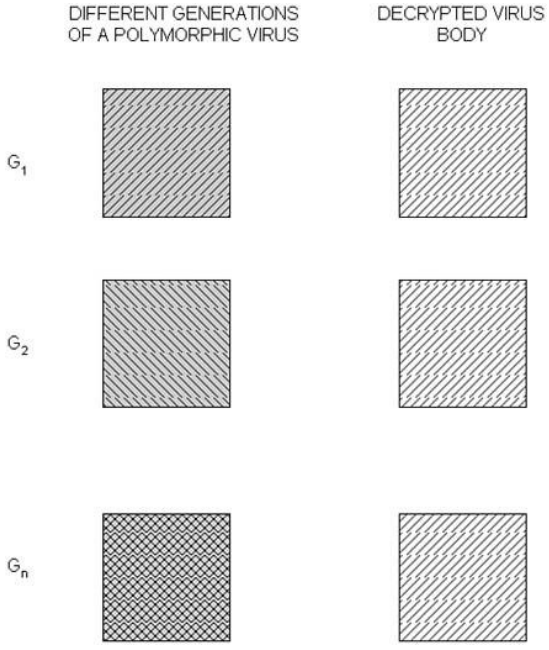
Polymorfism

För att kringgå statisk signaturigenkänning kan man ändra hela programkoden, både lasten och initieringsrutinen. Ett enkelt sätt för att uppnå polymorfism är att infoga kod som utför operationer på en variabel som aldrig används, *skräpinstruktioner*.

[Kodsnuttar som illustrerar skräpkod]

Ett tidigt polymorft virus som observerades 1989 infogade ett slumpmässigt antal onödiga operationer i sin dekrypteringsrutin för varje instans. Således fanns det ingen påtaglig datasekvens av tillräcklig längd för att vara av statistisk signifikans i identifieringssyfte.[2] Statiska signaturer räckte inte längre till för att identifiera den nya typen av virus.

Det skulle dröja fram till 1992 innan *Eugene Kaspersky* utvecklade en metod som automatiskt kunde känna igen den nya virustypen, *kodemulering*.[4] Det går ut på att under kontrollerade omständigheter låta programmet exekvera och periodvis söka efter kända virussignaturer. Eftersom polymorfa virus alltid dekrypterades till en konstant normalform kunde de således upptäckas.

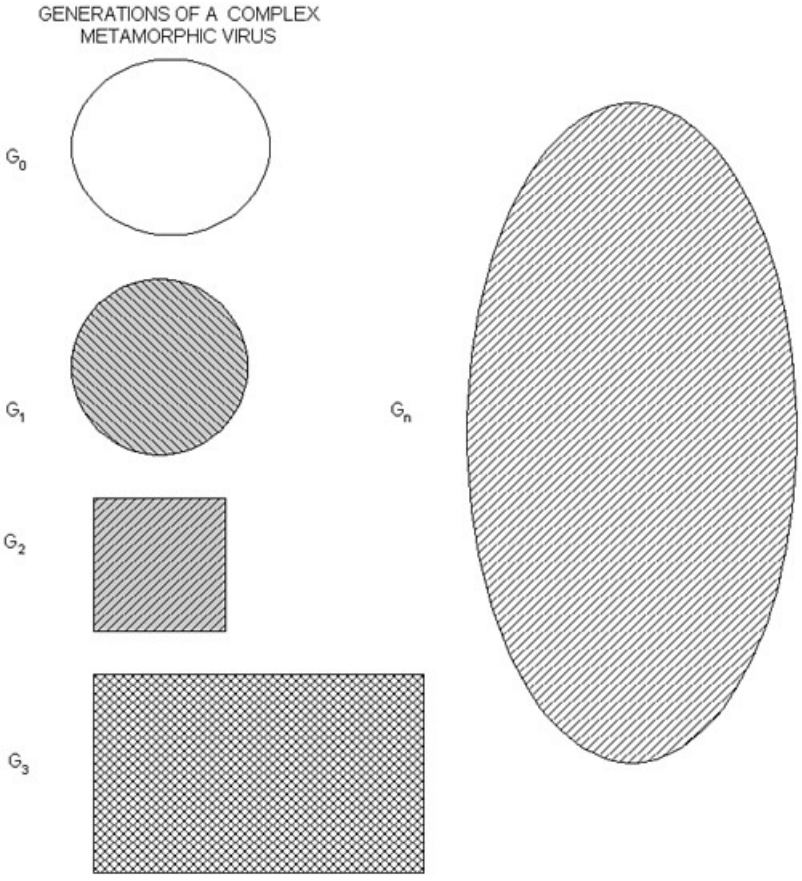


Svaret från virussidan kom dock i sedvanlig ordning. Eftersom kodemulering är

en väldigt resurskrävande operation och inte alltid fungerar fullkomligt perfekt kan metoder riktade mot emulering användas. Exempelvis kan man infoga så mycket skräpkod att emulatorn ger upp sitt försök att söka efter virussignaturer då exekveringstiden blir för lång.[4] Den mest avancerade metoden som skapades var dock *metamorfism*, en vidareutveckling av polymorfism.

Metamorfism

Kod som skriver om sig själv men behåller den underliggande funktionaliteten kallas metamorf kod. Det är en tillämpning av metakompilering, som innebär en transformation från ett program P till ett funktionellt ekvivalent program P'. Eftersom koden ändras mellan varje instans är kryptering inte längre nödvändigt, men i praktiken kombineras kryptering med metamorfism, men både lasten och dekrypteringsrutinen ändras gång på gång.[4] Bild 10000 illustrerar skepnaden av ett metamorft virus.

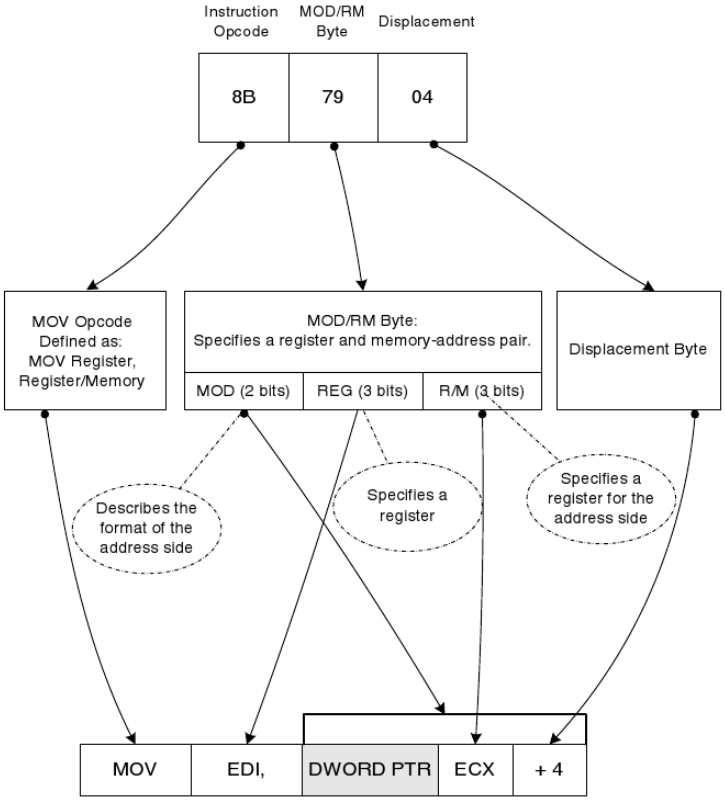


Till skillnad från polymorfa virus antar metamorfa virus aldrig antar en specifik form under något skede av deras livscykel och således kan de inte upptäckas med hjälp av automatiska dekrypteringsmetoder. Vid en tillräckligt hög grad av metamorfism skulle det krävas extrahering av programkod på en högre abstraktionsnivå, själva kärninstruktionerna, för att perfekt kunna identifiera ett virus.[3] Det är denna process jag försöker undersöka i de kommande kapitlen.

Olika sorters disassemblerare

Ett av de viktigaste verktygen för analys av skadeprogram är disassembleraren. Som namnet anger är en disassemblerare motsatsen till en assemblerare. Medan assembleraren omvandlar assembler till maskinkod, omvandlar disassembleraren maskinkod till assembler. En disassemblerare är, liksom assembleraren, maskinspecifik eftersom olika plattformer har olika maskinkod.[5]

Bild ### visar hur en disassemblerare konverterar en uppsättning maskinkodsinstruktioner till assembler. Vanligen görs detta via tabeller innehållande samtliga maskinkodsinstruktioner, deras argument och varianter.



Det finns två grundtyper av disassemblerare, *linjärt svepande* och *rekursivt traverserande*. Dessa typer har kompletterande styrkor och svagheter, varför *hybrider* ger bäst resultat i praktiken.[6]

Linjärt svepande disassemblerare

Det simplaste tillvägagångssättet när man önskar disassemblera ett program är att försöka avkoda allting i sektioner som angetts innehålla kod.[6] En möjlig utgångspunkt är att se i vilken sektion startadressen för exekvering (entry point) ligger. Den största fördelen är enkelhet, men nackdelen är att data inbakad i instruktionsströmmen kommer att antas vara kod och feltolkas.[6]

Rekursivt traverserande disassemblerare

Eftersom en linjärt svepande disassemblerare inte beaktar kontrollflödet i koden saknar den möjlighet att utesluta onåbar kod. Om man i stället beaktar kontrollflödesinstruktioner genom att märka ut deras möjliga mål och fortsätta disassembleringen på dessa adresser så kan man eliminera problemet med data tolkad som kod. Detta kallas rekursivt traverserande och beskrivs av funktionen nedan:[6]

```
proc Disassemble(Addr, instrList) {
     if (Addr has already been visited)
          return;

     do {
          instr = DecodeInstr(Addr);
          Addr.visited = true;
          add instr to instrList;
          if (instr is a branch or function call) {
               T = set of possible control flow successors of instr;

               for each (target ∈ T) {
                    Disassemble(target, instrList);
               }
          } else Addr += instr.length; /* addr of next instruction */
     } while Addr is a valid instruction address;
}
```

Om funktionen `Disassemble()` anropas med programmets startadress som argument och samtliga kontrollflödesinstruktioner hanteras korrekt så garanteras att varje instruktion som är nåbar från startadressen disassembleras korrekt.[6]

Korrekt hanterande av kontrollflödesinstruktioner visar sig vara problemet med denna metod. Indirekta hopp kan vålla problem för denna typ av disassemblerare. [6] Funktioner som anropas av andra program kommer inte heller tolkas som kod.

Hybriddisassemblerare

Kombinationen av en förbättrad linjär svepalgoritm och en rekursivt traverserande algoritm kallas hybriddisassemblerare och är det bästa alternativet trots högre beräkningskostnad, eftersom feltolkning av kod eller data kan ha katastrofala konsekvenser.[6]

Vanliga sätt att fördunkla kod

Fördunkling (obfuscation) innebär att man förändrar ett programs struktur, data, logik, och organisering så att funktionaliteten bibehålles, men att det blir svårare att utröna vad som sker.[5] Syftet är att hindra analys av koden och skadeprogram har därigenom lättare att undgå upptäckt.

I detta kapitel presenteras några vanliga sätt att fördunkla kod. Jag hoppas att de skall gå att motverka automatiskt med metoderna jag presenterar i senare kapitel, men i vissa svåra fall kan det vara omöjligt: *”Of course, just like any other method, code optimalization–based decryptor detection has its limitations and cannot be applied universally. For example, the complex polymorphic garbage of the MtE mutation engine (as discussed in Chapter 7) cannot be optimized effectively.”*[3]

Definition 1 (Fördunklande transformation)[7]

Låt $P \xrightarrow{\tau} P'$ vara en transformation av ett utgångsprogram P till ett resultatprogram P'.

$P \xrightarrow{\tau} P'$ är en *fördunklande transformation*, om P och P' har samma *observerbara uppförande*. För att $P \xrightarrow{\tau} P'$ skall vara en giltig fördunklande transformation måste följande villkor gälla:

- Om P ej avslutas eller avslutas med ett felvillkor så kan P' avslutas eller ej.
- Annars måste P' avslutas och ge samma resultat som P.

□

Med observerbart uppförande menas uppförande som det uppfattas av användaren. Detta innebär att P' får ha biverkningar (som skapande av filer, längre exekveringstid, etc.) som P saknar, så länge dessa biverkningar inte upplevs av användaren.[7]

Kontrollflödestransformationer är de enda vettiga transformationerna, utförligare text kommer stå här.

Ogenomskinliga konstruktioner

Med ogenomskinliga konstruktioner menas logiska uttryck *(predikat)* och variabler vilkas värden är kända på förhand. Syftet är att därigenom hindra analys genom att ovillkorliga hoppinstruktioner ersätts med till synes villkorliga hoppinstruktioner, vars ena möjlighet inte leder någonvart.[7]

```
if(x + 1 != x) {
     /* fortsätt exekvera programmet */
}
/* här kan man lägga godtyckligt skräp och vilseledande kod */
```

Väl valda ogenomskinliga konstruktioner kan i bästa fall omöjliggöra perfekt automatisk identifiering, då global dataflödesanalys är ett mycket svårt problem. [5] Dock kan de inte besegra en människa, och problemspecifika heuristiska lösningar kan likväl göra stor nytta.

Tabelltolkning

En ytterst kraftfull, men dyr teknik är tabelltolkning (table interpretation). Tabelltolkning kan liknas vid implementeringen av en tillståndsmaskin där programkoden delats upp i mindre delar och tillstånden, lagrade i en tabell, anger vilka delar som skall exekveras i vilken ordning. All underliggande kodstruktur kan döljas ordentligt med denna transformation.[5]

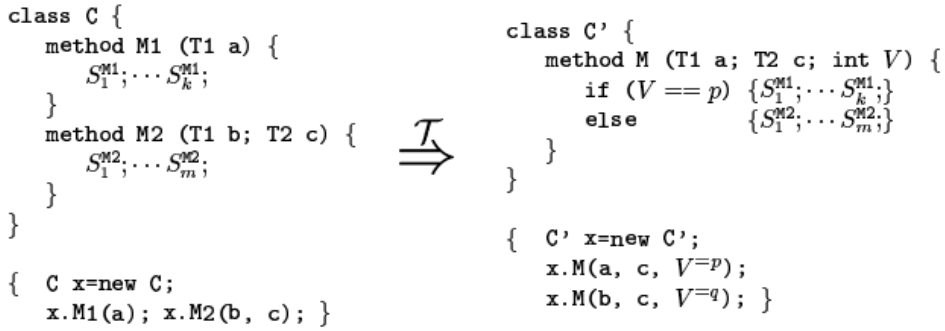
```
add r0, 2
add r0, 2
mul r0, 3
dec r0
ret
```

Jämför koden ovan med koden nedan, de har samma observerbara uppförande, men det senare stycket är betydligt svårare att tyda.

```
T[a, a, c, b, d]
r1 = 0

F:
jmp T[r1]

a:
add r0, 2
inc r1
jmp F

b:
dec r0
inc r1
jmp F

c:
mul r0, 3
inc r1
jmp F

d:
ret
```

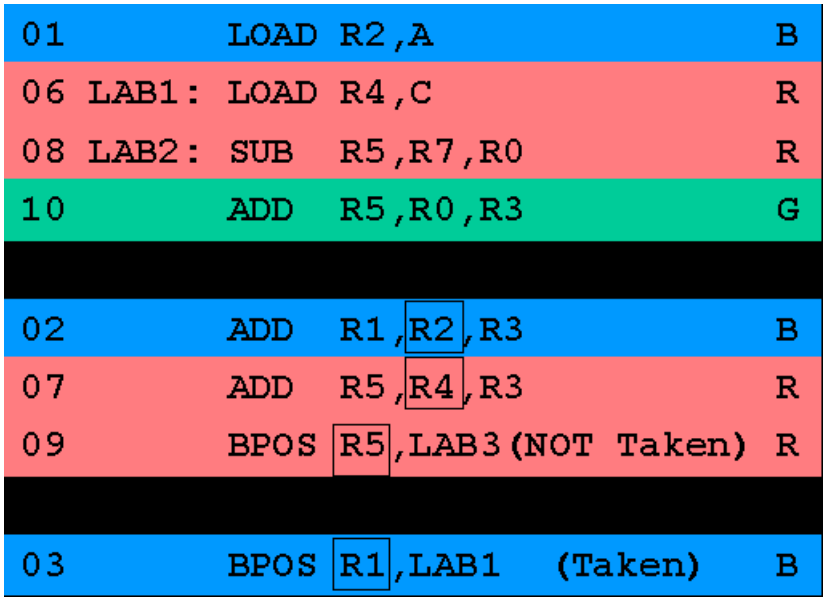
Sammanflätning

När koden för två eller flera funktioner flätas ihop så att det blir svårare att följa exekveringsvägen och att skilja olika funktioner från varandra kallas det sammanflätning. Det är en förhållandevis billig transformation som ändå kan vara tämligen effektiv.[7]



Ordningstransformationer

Det är enkelt att godtyckligt ändra ordningen på en mängd funktioner så länge alla transformationer är beroendebevarande. Resultatet är reducerad *lokalitet* i koden. [7] Det normala är ju att relaterade saker ligger i närheten av varandra, hög lokalitet, förutom att detta är intuitivt medför det också bättre prestanda när cache beaktas. Ordningstransformationer används även ofta för polymorfism.[3]



Statisk analys av fördunklad kod

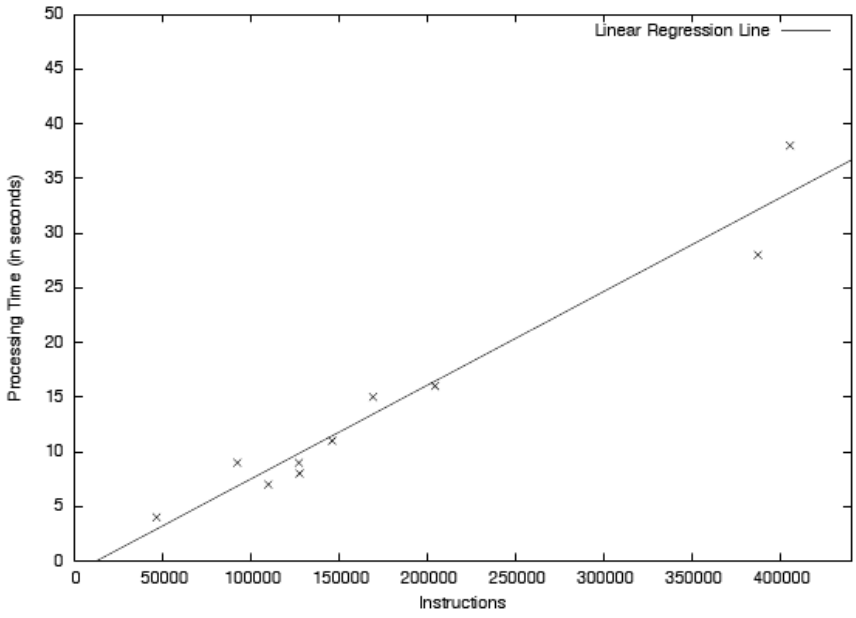
För att kunna analysera fördunklad kod måste den *klargöras* (deobfuscate). Innan klargörningstransformationer kan appliceras måste själva programkoden extraheras korrekt genom disassemblering. Detta förfarande kan även kallas *dekompilering*, dock utan målsättningen att generera faktisk källkod.

När det rör sig om fördunklad kod kan man inte göra många vanliga antaganden om gott uppförande hos ett program.[8] Så länge processorn felfritt kan exekvera koden uppför den sig således korrekt. Därför måste vissa åtgärder vidtas för att kringgå denna begränsning. Detta kapitel kommer behandla statiska metoder för att uppnå detta.

Disassemblering

När man disassemblerar ett fördunklat program börjar man med att dela in koden i funktioner för att sedan kunna analysera dem oberoende av varandra. Den främsta orsaken till detta är prestanda. Vanligen förknippas statiska analysmetoder med dålig skalbarhet. När datamängden växer leder detta till att tidskravet ökar drastiskt, eller att resultatet försämras. Funktionsindelningen medför en linjär kostnadsökning i förhållande till antalet instruktioner i programmet. Om man antar att medelstorleken för en godtycklig funktion är förhållandevis oberoende av av programmets storlek, så är även mängden arbete per funktion oberoende därav. Följaktligen måste fler funktioner analyseras när storleken på programmet växer och eftersom antalet funktioner då växer linjärt i förhållande till antalet instruktioner och mängden arbete per funktion är konstant. Således kan disassemblering fås att ha god skalbarhet.[8]

Program	Storlek (Byte)	Instruktioner	Tid (s)
openssh	263684	46343	4
compress95	1768420	92137	9
li	1820768	109652	7
ijpeg	1871776	127012	9
m88ksim	2001616	127358	8
go	2073728	145953	11
perl	2176268	169054	15
vortex	2340196	204230	16
gcc	2964740	387289	28
emacs	4765512	405535	38



För att lokalisera funktionerna används en heuristisk metod som letar efter funktionsprologer, kodsnuttar som vanligen återfinns i början på funktioner. Är även funktionsprologerna fördunklade kan det eventuellt krävas problemspecifika lösningar.[8]

Intraprocedurella kontrollflödesgrafer

En *kontrollflödesgraf* (Control Flow Graph, CFG) är en representation av samtliga vägar som kan tas under exekveringen av ett program. Ett *grundblock* (basic block) beskriver en instruktionsföljd utan hopp eller mål för hopp däri. Riktade bågar mellan grundblock representerar hopp i kontrollflödet.[8]

Definition 2 (Kontrollflödesgraf)[8]

Låt $G\langle V, E\rangle$ vara en riktad graf där noderna $u, v \in V$ representerar **grundblock** och en båge $e \in E : u \to v$ representerar ett möjligt kontrollflöde från u till v.

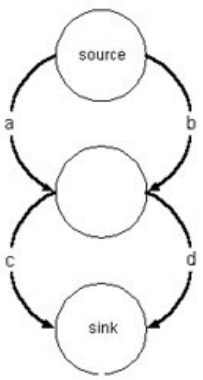


Fig 1. Simplified Control Flowgraph

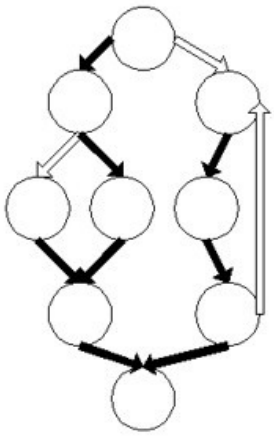


Fig 2. Flowgraph with marked edges

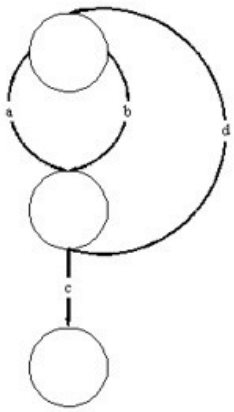


Fig 3. Simplified Flowgraph with Looping

Vid analys av fördunklad kod leder användandet av en traditionell rekursiv disassemblerare vanligen till att endast en liten del av kontrollflödesgrafen kan återskapas. En bättre metod i detta fall är att använda en tvåstegsprocess, där en hybriddisassemblerare först skapar en mer omfattande CFG, som kan vara en supermängd till den egentliga CFG:n, varpå konflikter löses och tvetydigheter avlägsnas i det andra steget.[8]

Dataflödesanalys

Dataflödesanalys är ett ytterst kraftigt verktyg som kan motverka allt från ogenomskinliga konstruktioner till tabelltolkning. För att kunna analysera dataflödet bör koden översättas till en mellanform kallad *SSA* (Single Statement Assignment).

Optimering

När koden väl konverterats till SSA-form kan vanliga optimeringsmetoder tillämpas. Eventuellt kunde man kanske mata in den rakt i back-end på en färdig kompilator.

Symbolisk exekvering

Ett alternativ till optimeringsvägen är symbolisk exekvering. Här använder man en processoremulator och tilldelar variablerna symboliska värden. I slutändan får man ett aritmetiskt uttryck som kan reduceras till en normalform enligt normala räkneregler eller användas som semantiskt fingeravtryck. På detta sätt kan man observera vad koden egentligen gör på en högre abstraktionsnivå.

Dynamisk analys av fördunklad kod

I motsats till statisk analys innebär dynamisk analys att programmet som skall analyseras faktiskt tillåts exekvera. Detta har sina för- och nackdelar. Eftersom programmet faktiskt körs finns det goda skäl för att se över säkerheten så att det inte förmår göra någon egentlig skada. Samtliga dynamiska metoder torde dock innefatta begränsningen att de endast kan utgå från information som finns tillgänglig vid ett visst exekveringstillfälle. Exempelvis kan ett virus vara gjort för att endast sprida sig en viss veckodag, råkar man köra det en annan dag kommer det skadliga beteendet aldrig observeras och det kan bli ett felaktigt frikännande. Detta kapitel kommer undersöka dynamiska metoder för klargöring av fördunklad kod.

Framtida utvecklingar

Ett möjligt kapitel som spekulerar i vad framtiden har att erbjuda.

Dekompilering

Diskussion av Hex-Rays, den bästa dekompilatorn på den öppna marknaden.

Virtualisering

I allt större utsträckning kommer virtualisering att användas. Eftersom dagens datorer är så snabba är en virtualiseringstransformation möjlig, då den trots sin extremt höga kostnad ändå kan anses ge observerbart identiska resultat.

Kvantdatorer och P ≠ NP

Jag har ingen aning om vad som skall stå här, så det är bara utfyllnad tills vidare. Troligen kommer det vara någonting helt annat.

Avslutning


HEY DUDES THANKS.
FOR RESCUING ME.
LET'S GO FOR
A BURGER....
HA! HA! HA! HA!

Litteraturförteckning

1: Richard Ford, Eugene H. Spafford, Happy Birthday, Dear Viruses, 2007
2: Thomas Chen, Jean-Marc Robert, The Evolution of Viruses and Worms, 2004
3: Peter Szor, The Art of Computer Virus Research and Defense, 2005
4: Philippe Beaucamps, Advanced Polymorphic Techniques, 2007
5: Eldad Eilam, Reversing: Secrets of Reverse Engineering, 2005
6: Benjamin Schwarz, Saumya Debray, Gregory Andrews, Disassembly of Executable Code Revisited, 2002
7: Christian Collberg, Clark Thomborson, Douglas Low, A Taxonomy of Obfuscating Transformations, 1997
8: Christopher Kruegel, William Robertson, Fredrik Valeur, Giovanni Vigna, Static Disassembly of Obfuscated Binaries, 2004