ELEKTRONIK TIDNINGEN

Robert Krten, senioranalytiker
Sergei Sourjko, seniorutvecklare
UBM TechInsights

Avslöjar algoritmen bakom binärkoden

**Sherlock Holmes hade sitt förstoringsglas.
Robert Krten och Sergei har sin dekompilator.
De utreder misstänkta intrång på programvarupatent.
Här visar de steg för steg hur de går till väga
för att återskapa välstrukturerad C-kod
från rörig assembler.**

Redaktör
Jan Tångring
jan@etn.se
0734-17 13 09

EMBEDDED EXPERT

Avslöjar algoritmen bakom binärkoden

Steg för steg från kryptisk assembler till välstrukturerad C-kod

Av Robert Krten och Sergei Sourjko, UBM Techninsights

Robert Krten gör så kallad *reverse engineering* på mjukvara — han analyserar programkod för att se om den inkräktar på patenterade algoritmer. Han är tidigare sin egen konsult med realtidsprogramvara som expertområde och har publicerat böcker om operativsystemet QNX och artiklar i bland annat Doctor Dobb's och Byte.

Sergei Sourjko är dubbeldoktor i matematik och datorvetenskap. Tidigare arbetade han på Numerical Technologies (som köptes av Synopsys 2003). Han har publicerat flera böcker och över 90 vetenskapliga artiklar.

Ett av de vanligaste uppdragen för UBM Techinsights är att bevisa att en algoritm har utsatts för patentintrång. Det här är något som vi gör inom ramen för våra tjänster kring intellektuell egendom (IP Management Services).

Intressanta algoritmer inom inbyggda system finns i allt från sensorteknik i apparater, motorstyrning, effektstyrning, navigationsalgoritmer, användargränssnitt till filsystem i avancerade inbyggda system. Ett misstänkt patentintrång är ett av de få tillfällen man har rätt att använda reverse engineering — att återskapa konstruktionen av en produkt genom att analysera ett exemplar produkten i sig — vad det än står i licensavtalet.

Ett problem i den här typen av projekt är att maskinkoden har skapats utgående från programvara författad i högnivåspråk som C eller C++. Kompilatorns översättning till maskininstruktioner är en sofistikerad process som bland annat innefattar optimering. Det är därför mycket svårt att avgöra om det handlar om intrång genom att bara titta på maskinkoden.

Att dekompilera innebär att ta man tar maskininstruktionerna och översätter dem till en högnivårepresentation. Ett vanligt användningsområde för dekompilering är att analysera virus och annan avsiktligt skadlig programvara. Ibland används det också för att återskapa förlorad källkod eller för att skapa en kompatibel produkt.

En av de populärare dekompilatorerna säljs av Hex-Rays för i386-plattformen som en plug-in till företagets disassemblator IDAPro.

Vårt exempel i den här artikeln baseras på ett av de populäraste assemblerspråken inom konsumentelektronik med stora volymer liksom i andra inbyggda system – Arm-arkitekturen. Vi har funnit att tillgängliga dekompilatorer för Arm-baserade produkter generar kod av låg kvalitet, så vi utgick från dekompilatorn "Desquirr", som är öppen källkod, och anpassade den sedan för våra behov.

Studera det enkla C-programmet i **fil 0**, i bilden intill. Det innehåller tilldelningar, matematiska operationer, variabler, en jämförelse och ett villkorligt hopp.

Om man kompilerar koden med GNU CC för Arm32 med maximal optimering (-O3) får man koden i **fil 1**.

Man ser att *for*-loopen har implementerats som en jämförelse med slutvillkoret och ett villkorligt hopp. Variablerna *i* och *sum* har lagts i register.

Om man applicerar Desquirr på ovanstående får man koden i **fil 2**.

Resultatet är möjligen tillräckligt bra för ett enkelt exempel som detta. Men det finns några problem med Desquirr:

- det finns buggar i interpreteringen av Arm-instruktionerna
- Arm-arkitekturen är ofullständigt modellerad (särskilt med avseende på flaggor)
- det görs inga transformationer av kontrollflödet, vilket resulterar i spaghettikod med GOTO-satser

Första steget vid dekompilering är att

Fil 0

```
main (int argc, char **argv)
{
   int  i;
   int  sum;

   sum = 0;
   for (i = 0; i < 1000; i++) {
      sum += i;
   }
   printf(
      "The sum of 0..999 is %d\n", sum);
}
```

Fil 1

```
main:
   MOV    R1, #0
   MOV    R0, #0x3E4
   MOV    R3, R1
   ADD    R2, R0, #3
loc_8478:
   ADD    R1, R1, R3
   ADD    R3, R3, #1
   CMP    R3, R2
   BLE    loc_8478
   LDR    R0, =aTheSumOf0__999
   B      .printf
```

Fil 2

```
main:
   R1 = 0;
   R0 = 0x3e4;
   R3 = R1;
   R2 = R0 + 3;
loc_8478:
   R1 = R1 + R3;
   R3 = R3 + 1;
   Cond = R3 - R2;
   if (Cond <= 0) goto loc_8478;
   R0 = "The sum of 0..999 is %d\n";
   goto .printf;
```

Fil 3

```
#define sgn(__X) ((1 << 31) & (__X))
main:
   R1 = 0;
   R0 = 0x3e4;
   R3 = R1;
   R2 = R0 + 3;
loc_8478:
   R1 = R1 + R3;
   R3 = R3 + 1;
   Flag_Z = !(R3 - R2);
   Flag_N = !!sgn(R3 - R2);
   Flag_C = !(R3 < R2);
   Flag_V = sgn(R3) != sgn(R2)
            && sgn(R3-R2) == sgn(R2);
   CCEQ = R3 == R2;
   CCCS = R3 >= R2;
   CCMI = R3 < R2;
   CCVS = Flag_V;
   CCHI = R3 > R2;
   CCGE = R3 >= R2;
   CCGT = R3 > R2;
   if (CCGT) goto loc_8488;
   goto loc_8478;
loc_8488:
   R0 = * (PC + 0);
   goto .printf;
```

konvertera det hårdvaruberoende assemblerspråket till en oberoende representation. Det sker via en front-end-processor. Vi har modifierat Desquirrs standardversion och gjort ett antal justeringar av småfel i tolken och lagt till en omfattande modelleringssektion som hanterar flaggor, storlek på operander, med mera.

Det interna mellanresultatet efter det att all maskinkod har konverterats till C-kodsliknande text kan du se i **fil 3**.

Notera de temporära flaggtilldelningarna som kan strykas via dödkodseliminering (i litteraturen kallad "define/use expansion") vilket ger oss koden i **fil 4**.

Det här sker genom analys av kontrollflödet (CFA, Control Flow Analysis) med en kontrollflödesgraf (CFG) som indata.

En CFG är en representation av programmets kontrollflöde. Den består av en listning av obrutna instruktionssekvenser (som varken själva påverkar kontrollflödet eller är mål för kontrollflödesändringar) sammankopplade till en graf där pilarna representerar förändringar i kontrollflödet (via instruktioner som proceduranrop, hopp och villkorliga hopp).

Utdata från kontrollflödesanalysen är ett antal grafer som matchar de vanliga flödesstyrningsmönstren som loopar, *if*-satser, *switch*-satser, och så vidare. Utgående från dem kan vi sedan återskapa programmets högnivårepresentation.

För att bearbeta graferna från kontrollflödesanalysen använder vi så kallad intervallbaserad analys. Man inleder med en iterativ procedur som identifierar alla

Fil 4

```
main:
   R1 = 0;
   R0 = 0x3e4;
   R3 = R1;
   R2 = R0 + 3;
loc_8478:
   R1 = R1 + R3;
   R3 = R3 + 1;
   CCGT = R3 > R2;
   if (CCGT) goto loc_8488;
   goto loc_8478;
loc_8488:
   R0 = * (PC + 0);
   goto .printf;
```

grafintervall — undergrafer som innehåller ett givet hörn, kallat intervallhuvud, plus alla loopar (sekvenser med kontrollövergångar från hörn till hörn som börjar och slutar i samma hörn) som innehåller intervallhuvudet.

Därefter bygger vi en ny, härledd, graf genom att kollapsa varje grafintervall till ett nytt hörn och anse två hörn vara sammanbundna om några hörn från den ursprungliga grafen tagna från intervallen är sammanbundna.

Genom att upprepa processen rekursivt får vi en sekvens av härledda grafer. På det här sättet identifieras nästlade loopar i programmet, de motsvarar loopar som finns i sekventiellt skapade grafer i olika nivåer.

Kontrollflödesanalys omfattar ett antal algoritmer för detaljerade studier av kontrollflödesstrukturen. Till exempel är det inte tillräckligt att upptäcka var en loop finns, vi måste också identifiera villkoren, innehållet i loopen, vilken typ av loop det är (*while*, *for*, *do-while*, och så vidare), följa vilka vägar den kan ta, med mera.

Resultatet som det kan se ut finns i **fil 5**. Notera att huvudloopen har markerats med en kommentar.

Det återstående steget är att generera koden. För detta använder vi den information som samlats ihop under kontrollflödesanalysen, och översätter de kontrollstrukturer som kontrollflödesgrafen upptäckt, till en högnivårepresentation av programmet.

Den som vill veta mer om implementationerna av kontrollflödesanalysen kan studera referenserna i fotnot 1 och 2. Vi grundar oss på välkända algoritmer och utnyttjar Boostbiblioteket Graph och dess implementation av Lengauer-Tarjans algoritm för dominatorträd.

Detta ger oss till slut koden i **figur 6**, vilken som synes ligger mycket nära originalet i **fil 0**. Men den är inte identisk — varför inte?

Svaret är att en dekompilator inte är perfekt. Utöver rent tekniska begräns-

Fil 5

```
main:
   R1 = 0;
   R0 = 0x3e4;
   R3 = R1;
   R2 = R0 + 3;

   {//BEGIN loop 0
loc_8478:
      R1 = R1 + R3;
      R3 = R3 + 1;
      CCGT = R3 > R2;
      if (CCGT) goto loc_8488;

   goto loc_8478;
   }//END loop 0
loc_8488:
   R0 = * (PC + 0);
goto _printf;
```

Fil 6 — att jämföra med fil 0

```
main:
   R3 = R1 = 0;

   while (1) {
      R1 += R3;
      if (++R3 > 999) break;
   }
   return (printf (*PC));
```

ningar i dagens dekompilatorer — som deras förmåga att detektera for-loopar eller att hantera funktioner med variabelt antal argument — finns principiella problem:

- Ursprungliga variabelnamn (som *i* och *sum*) syns inte. De är bra för utvecklaren, men plockas bort av kompilatorn. I koden ovan är det uppenbarligen *R1* som motsvarar *sum* och *R3* som motsvarar *i*.
- Makron, templates och inlinefunktionsanrop tappas normalt bort innan kompilatorn börjar generera kod, som de syns inte i assemblerkoden i vilket fall som helst
- Många av de objektorienterade primitiven röner ett liknande öde. Dekompileringen försvåras exempelvis av att medlemsfunktioner accessas via flera nivåer av omdirigeringar.

■

Fotnoter

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

2. C Cifuentes, D Simon and A Fraboulet, Assembly to High-Level Language Translation.â€ Proceedings of the International Conference on Software Maintenance, Washington DC, 18-20 Nov 1998, IEEE Press, pp 228-237.