Decompilare fără goto cu DREAM

Adrian Manea

510, SLA

Problema și soluția propusă

Articolul: **No More Gotos: Decompilation Using Pattern-Independent Control Flow Structuring and Semantics-Preserving Transformations**, K. Yakdan et al., NDSS 2015

Decompilatoarele, bazate pe analiză structurală pe baza grafului de control (CFG), generează instrucțiuni `goto` atunci cînd nu găsesc o continuare așteptată în graf.

DREAM elimină `goto` prin transformări care păstrează semantica și alte prelucrări logice pe graf.

Concurența: Hex-Rays și Phoenix.

Benchmark: GNU `coreutils`.

Procedura de ansamblu

Eliminarea goto se va putea face ținînd cont că:

- Structurile de control din program au un singur punct de intrare și un singur punct succesor $\Rightarrow$ se pot simplifica;
- Tipul și posibilele ramificații ale structurilor de control se pot analiza folosind *logică* pe CFG.

Etape, în mare:

1. Se alcătuiește CFG;
2. Se structurează CFG folosind și transformări invariante semantic;
3. Se obține AST (arborele de sintaxă abstractă);
4. Se fac optimizări post-structurare (redenumiri de variabile, evidențierea funcțiilor pe șiruri de caractere, simplificarea ramificațiilor).

Etapele DREAM

1. *Dezasamblează* binarul folosind IDA Pro $\Rightarrow$ CFG;
2. *Analiza fluxului de date*, inclusiv propagarea constantelor și eliminarea codului mort (inaccesibil);
3. *Inferă tipurile variabilelor*, folosind TIE;
4. **Structurează CFG**:
   - **Nu folosește pattern matching** ("pattern independent");
   - Folosește DFS (depth-first search);
   - Tratează separat zonele cu cicluri de cele fără cicluri;
   - Aplică transformări care păstrează semantica;
   - Aplică optimizări finale.

Transferul controlului (*Reaching Condition*)

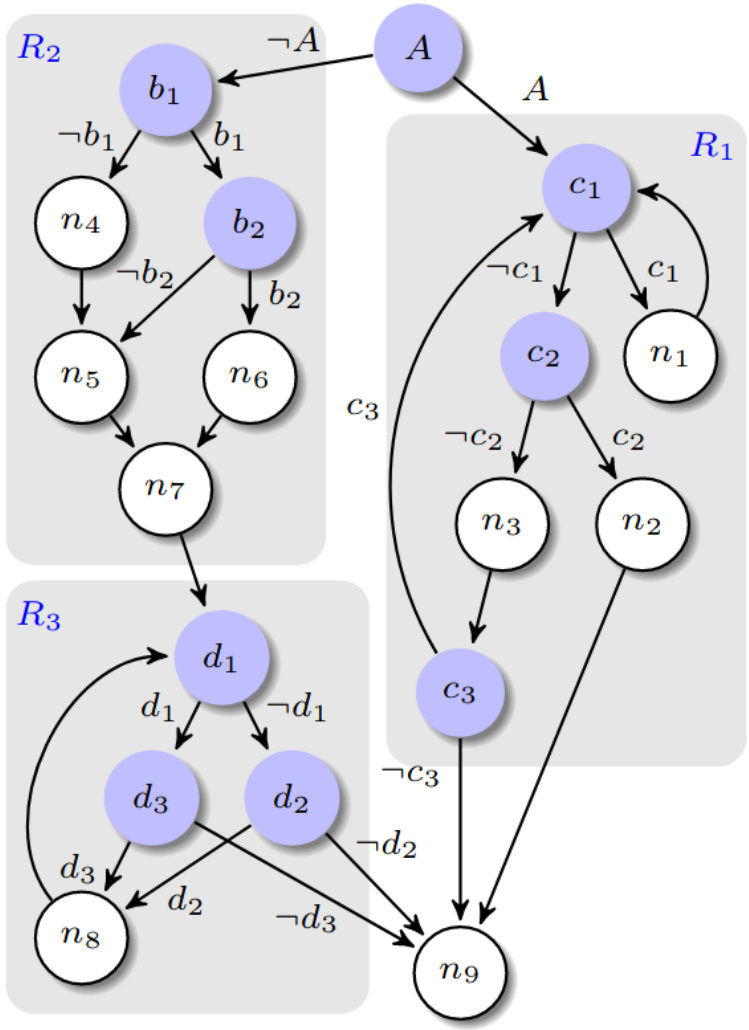
Fie $G(N, E, n_{in})$ graful de control.
Cu DFS între n_{src} și n_{exit} fixate obținem un subgraf aciclic $S_G(n_s, n_e)$ ("slice") și drumurile *simple* între n_s și n_e (similar: Cifuentes).

Algorithm 1: *Graph Slice*

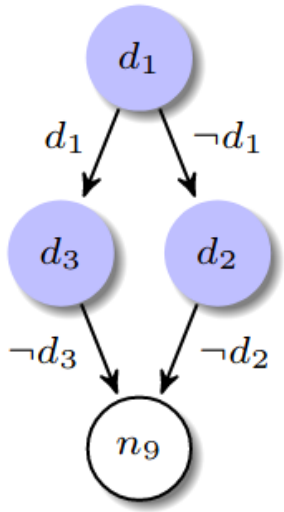
Input : Graph $G = (N, E, h)$; source node n_s; sink node n_e
Output: $S_G(n_s, n_e)$

```
1  S_G ← ∅;
2  dfsStack ← {n_s};
3  while E has unexplored edges do
4  |   e := DFSNextEdge(G);
5  |   n_t := target(e);
6  |   if n_t is unvisited then
7  |   |   dfsStack.push(n_t);
8  |   |   if n_t = n_e then
9  |   |   |   AddPath(S_G, dfsStack)
10 |   |   end
11 |   else if n_t ∈ S_G ∧ n_t ∉ dfsStack
   |   then
12 |   |   AddPath(S_G, dfsStack)
13 |   end
14 |   RemoveVisitedNodes()
15 end
```

Exemplu



(a) CFG – exemplu de lucru



(b) $S_G(d_1, n_9)$

$$d_1 \rightarrow n_9 \Leftrightarrow (d_1 \wedge \neg d_3) \vee (\neg d_1 \wedge \neg d_2)$$

Structurarea zonelor aciclice

Ideea: Putem ordona un graf orientat aciclic inversînd ordinea parcurgerii DF (ordine topologică);

Calculăm condiția de accesibilitate de la intrare la fiecare nod;

Obținem un AST al nodurilor în ordine topologică, *ne-optim*;

Rafinăm acest AST folosind logică (condiții complementare, `switch` pentru clustere de control) și eventuale `if-then-else` în cascadă.

Exemplu: În regiunea R_2, avem *condițiile complementare*

`if (b1 AND b2) then n6` și `if (`$\tilde{\texttt{b1}}$ `OR` $\tilde{\texttt{b2}}$`) then n4 or n5`.

Putem structura sub forma `if (b1 AND b2) then n6 else n5`.

Structurarea zonelor cu cicluri

1. Găsim nodurile care conduc la bucle;
2. Refacem zonele ciclice $\Rightarrow$ o singură intrare, un singur succesor;
3. Obținem AST pentru buclă;
4. Determinăm tipul buclei și condiția de intrare prin analiza AST;

Algorithm 2: *Loop Successor Refinement*

Input : Initial sets of loop nodes N_{loop} and successor nodes N_{succ}; loop header n_h
Output: Refined N_{loop} and N_{succ}

```
1  N_new ← N_succ;
2  while |N_succ| > 1 ∧ N_new ≠ ∅ do
3  |   N_new ← ∅;
4  |   forall the n ∈ N_succ do
5  |   |   if preds(n) ⊆ N_loop then
6  |   |   |   N_loop ← N_loop ∪ {n};
7  |   |   |   N_succ ← N_succ \ {n};
8  |   |   |   N_new ← N_new ∪
   |   |   |   {u : u ∈ [succs(n) \ N_loop] ∧ dom(n_h,u)};
9  |   |   end
10 |   end
11 |   N_succ ← N_succ ∪ N_new
12 end
```

Identificarea logică a buclelor

$$\frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Seq\left[n_i\right]^{i\in 1..k}\right] \quad n_1 = \mathcal{B}_r^c}{n_\ell \rightsquigarrow Loop\left[\tau_{\textbf{while}}, \neg c, Seq\left[n_i\right]^{i\in 2..k}\right]}\textsc{While} \qquad \frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Seq\left[n_i\right]^{i\in 1..k}\right] \quad n_k = \mathcal{B}_r^c}{n_\ell \rightsquigarrow Loop\left[\tau_{\textbf{dowhile}}, \neg c, Seq\left[n_i\right]^{i\in 1..k-1}\right]}\textsc{DoWhile}$$

$$\frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Seq\left[n_i\right]^{i\in 1..k}\right] \quad \forall i \in 1..k-1 : \mathcal{B}_r \notin \sum[n_i] \quad n_k = Cond\left[c, n_t, -\right]}{n_\ell \rightsquigarrow Loop\left[\tau_{\textbf{endless}}, -, Seq\left[Loop\left[\tau_{\textbf{dowhile}}, \neg c, Seq\left[n_i\right]^{i\in 1..k-1}\right], n_t\right]\right]}\textsc{NestedDoWhile}$$

$$\frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Seq\left[n_i\right]^{i\in 1..k}\right] \quad n_k = \acute{n_k} \Downarrow \mathcal{B}_r}{n_\ell \rightsquigarrow Seq\left[n_1, \ldots, n_{k-1}, \acute{n_k}\right]}\textsc{LoopToSeq}$$

$$\frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Cond\left[c, n_t, n_f\right]\right] \quad \mathcal{B}_r \notin \sum[n_t] \quad \mathcal{B}_r \in \sum[n_f]}{n_\ell \rightsquigarrow Loop\left[\tau_{\textbf{endless}}, -, Seq\left[Loop\left[\tau_{\textbf{while}}, c, n_t\right], n_f\right]\right]}\textsc{CondToSeq}$$

$$\frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Cond\left[c, n_t, n_f\right]\right] \quad \mathcal{B}_r \in \sum[n_t] \quad \mathcal{B}_r \notin \sum[n_f]}{n_\ell \rightsquigarrow Loop\left[\tau_{\textbf{endless}}, -, Seq\left[Loop\left[\tau_{\textbf{while}}, \neg c, n_f\right], n_t\right]\right]}\textsc{CondToSeqNeg}$$

\+ Transformări care *păstrează semantica* buclei prin înregistrarea punctelor de intrare.

Optimizări finale (pentru lizibilitate)

- Simplificarea structurilor de control:
  - `if (c) then (x = v) else (x = w)` $\rightsquigarrow$ `x = c ? v : w;`
  - Se transformă `while` în `for` oricît de des este posibil;
- Identificarea funcțiilor pe șiruri de caractere:
  - `strcpy`, `strlen`, `strcmp` sînt înlocuite de definițiile lor de compilator, așa că DREAM le evită sau le evidențiază în mod special;
- Redenumirea variabilelor, e.g. cele corespunzătoare API-urilor folosite.

Teste și comparații

Considered Functions F	$\|F\|$	Number of goto Statements			Lines of Code			Compact Functions		
		DREAM	Phoenix	Hex-Rays	DREAM	Phoenix	Hex-Rays	DREAM	Phoenix	Hex-Rays
`coreutils` functions with duplicates										
$T_1 : F_p^r \cap F_h^r$	8,676	0	40	47	93k	243k	120k	81.3%	0.3%	32.1%
$T_2 : F_d \cap F_p \cap F_h$	10,983	0	4,505	3,166	196k	422k	264k	81%	0.2%	30.4%
`coreutils` functions without duplicates										
$T_3 : F_p^r \cap F_h^r$	785	0	31	28	15k	30k	18k	74.9%	1.1%	36.2%
$T_4 : F_d \cap F_p \cap F_h$	1,821	0	4,231	2,949	107k	164k	135k	75.2%	0.7%	31.3%
Malware Samples										
ZeusP2P	1,021	0	N/A	1,571	42k	N/A	53k	82.9%	N/A	14.5%
SpyEye	442	0	N/A	446	24k	N/A	28k	69.9%	N/A	25.7%
Cridex	167	0	N/A	144	7k	N/A	9k	84.8%	N/A	12.3%

TABLE III: Structuredness and compactness results. For the `coreutiles` benchmark, we denote by F_x the set of functions decompiled by compiler x. F_x^r is the set of recompilable functions decompiled by compiler x. d represents DREAM, p represents Phoenix, and h represents Hex-Rays.