



Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Wolfram Luithardt



Softwariemetriken und Codeflussanalysen am Linux Kernel



School of
Information Technology and Electrical Engineering



Idee

2014: Sehr interessantes Paper von A Jbara et al. (School of Computer Science and Engineering, Jerusalem):

“High-MCC Functions in the Linux Kernel”
erschieden in Empir. Software Eng.

- Analyse der Evolution von Funktionen mit hoher Zyklomatischer Komplexität

→ Zyklomatische Komplexität ist kein gutes Mass für Komplexität

Warum also keine andere Komplexitätsmetriken verwenden ?

Zyklomatische Komplexität

1976 von Mc Cabe vorgeschlagen: Sie ist definiert als Anzahl linear unabhängiger Pfade auf dem Kontrollflussgraphen eines Moduls. Damit ist die Zahl eine obere Schranke für die minimale Anzahl der Testfälle, die nötig sind, um eine vollständige Zweigüberdeckung des Kontrollflussgraphen zu erreichen. [Wikipedia]

McCabe hielt einen Wert von 10 als noch vertretbar. Andere Organisationen liessen jedoch etwas höhere Werte zu (z.B. 15 oder 25).

Bereits McCabe erkannte, dass seine Zyklomatische Komplexität kein gutes Mass für die eigentliche Komplexität (im objektiven Sinne) ist.

Umso erstaunlicher ist es, dass die zyklomatische Komplexität bis heute als Komplexitätsmetrik verwendet wird.

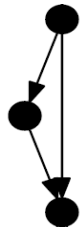
In strukturierten Programmiersprachen berechnet sich die zyklomatische Komplexität aus 1 plus der Summe aller verzweigenden Elementen.

In C sind dies:

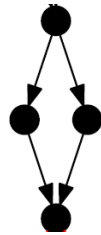
- if-else
- Schleifen-Elemente: for, do-while, while
- case und default

Graphentheoretische Deutung

Jedes Programm setzt sich aus einer Reihe von Grundelementen zusammen:



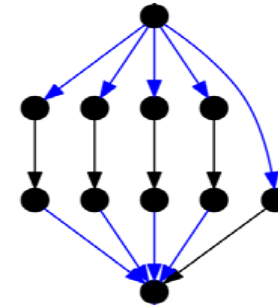
if



if-else



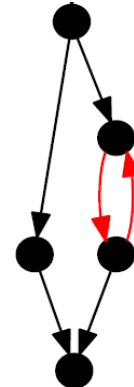
for(;;)
while()
do-while()



switch –
case/break
default

und den Operationen "sequenceing" and "nesting"

Die graphische
Darstellung einer
Funktion und/oder
Modul heisst:
Codeflussgraph
(CFG)



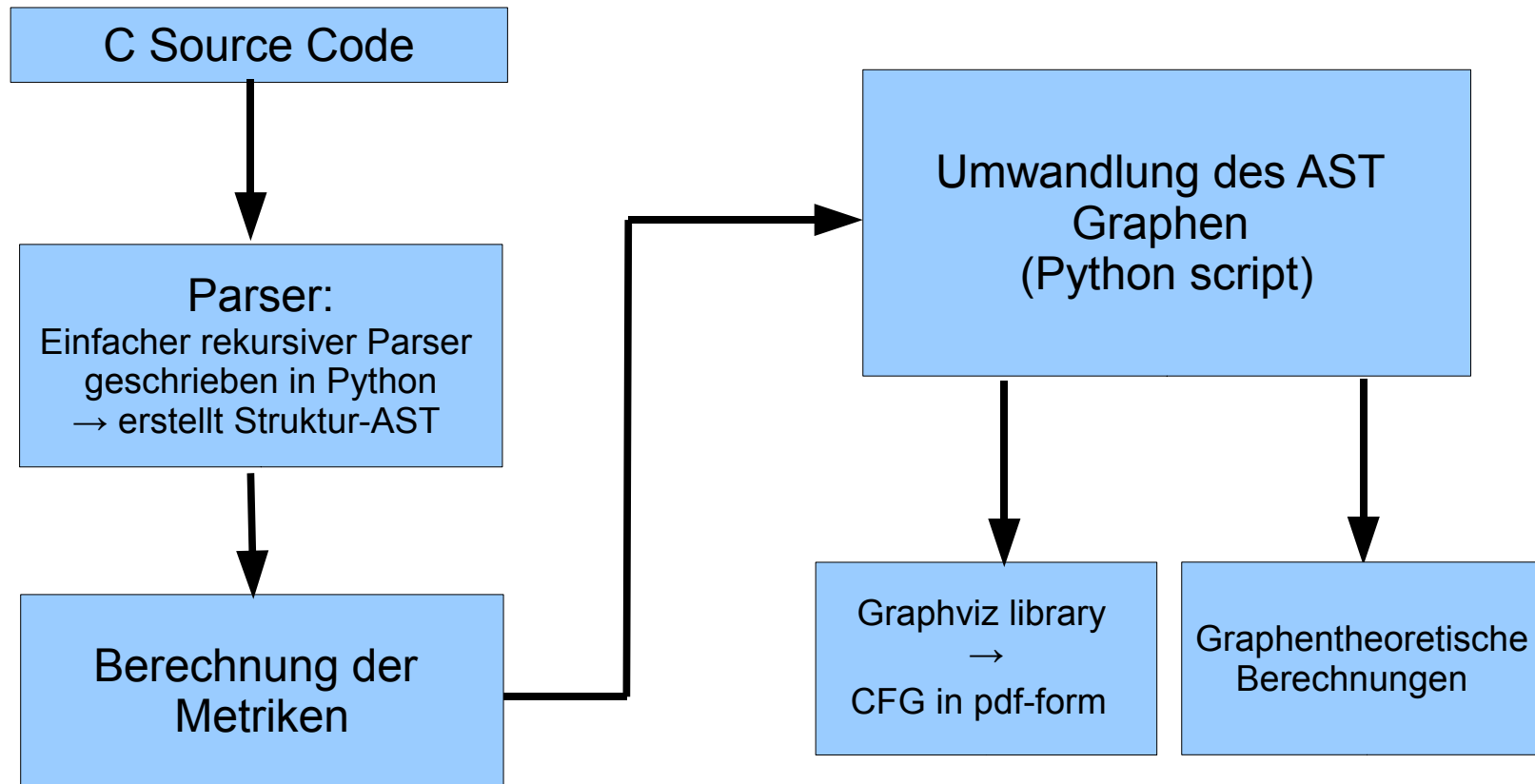
Sehr gute
Erläuterung der
Methode im Buch
"Software Metrics"
von Norman Fenton
(1997)

Zyklomatische Komplexität

Grundelement	Zyklomatische Komplexität
if or if-else	1
for	1
while	1
do-while	1
case + default	1

Operation	Zyklomatische Komplexität
Sequenz	Summe der Grundelemente
Nesting	Wird wie Sequenz behandelt

Wie wird gemessen ?



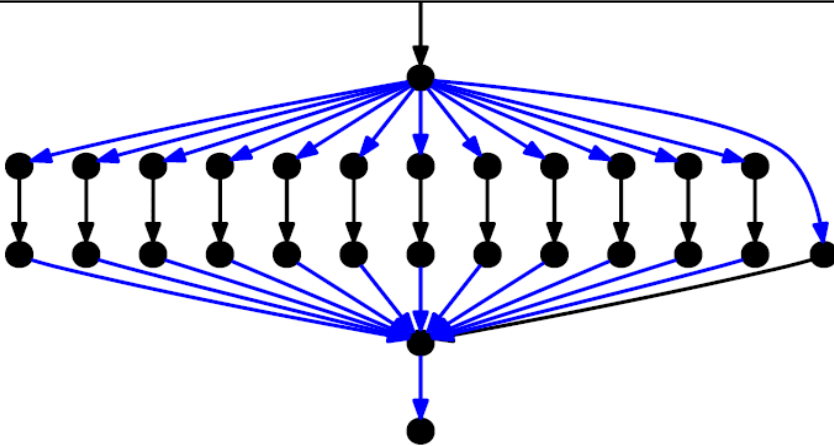
Erstes kleines Beispiel

```
unsigned char * sym(int month){
    unsigned char monthText[4];
    switch (month){
        case 1 : strcpy(monthText, "Jan");
                  break;
        case 2 : strcpy(monthText, "Feb");
                  break;
        case 3 : strcpy(monthText, "Mar");
                  break;
        case 4 : strcpy(monthText, "Apr");
                  break;
        case 5 : strcpy(monthText, "May");
                  break;
        case 6 : strcpy(monthText, "Jun");
                  break;
        case 7 : strcpy(monthText, "Jul");
                  break;
        case 8 : strcpy(monthText, "Aug");
                  break;
        case 9 : strcpy(monthText, "Sep");
                  break;
        case 10: strcpy(monthText, "Oct");
                  break;
        case 11: strcpy(monthText, "Nov");
                  break;
        case 12: strcpy(monthText, "Dec");
                  break;
        default: strcpy(monthText, "NN");
    }
    return(monthText);
}
```

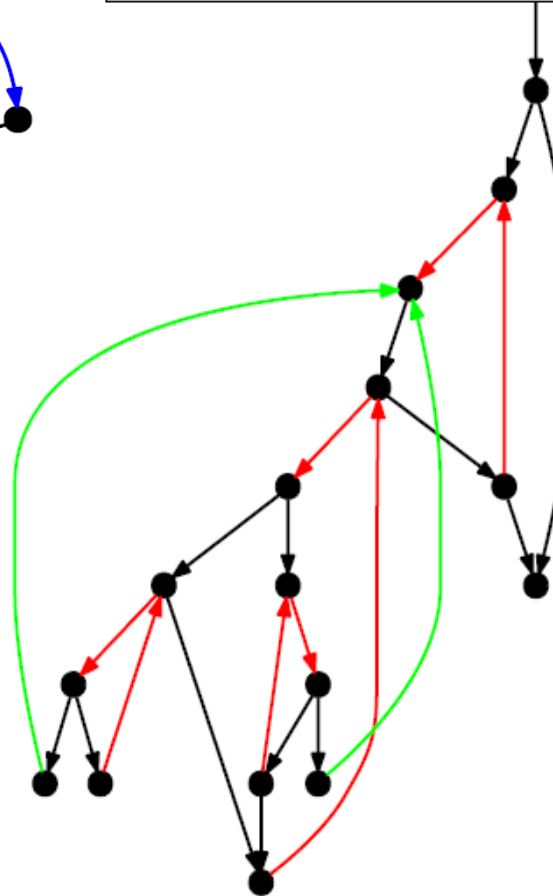
```
unsigned char * complicated(int month){
    int i;
    if (month!=0){
        for (i=0; i<20; i++){
            Label1:
                while(month > i){
                    strcpy(monthText, "hello");
                    if(month>10){
                        do {
                            printf("Hello World");
                            i=i+1;
                            if (i!=9) goto Label1;
                        } while (i>10);
                    } else {
                        while(i<10){
                            printf("Hello World");
                            i=i+2;
                            if (i==9) goto Label1;
                        }
                    }
                }
            }
        }
    }
}
```

Erstes kleines Beispiel

sym
F-Complexity = 39, G-Complexity = 14, nest. = 1, seq. = 1, Cyclomatic Complexity = 14

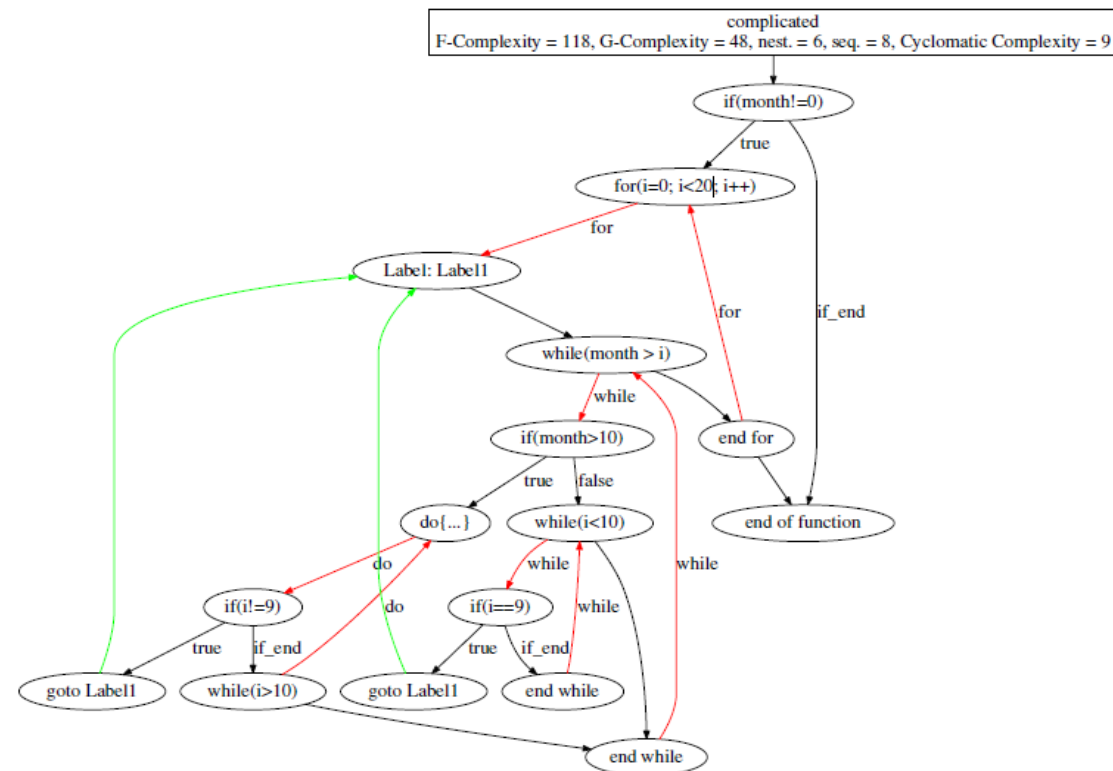
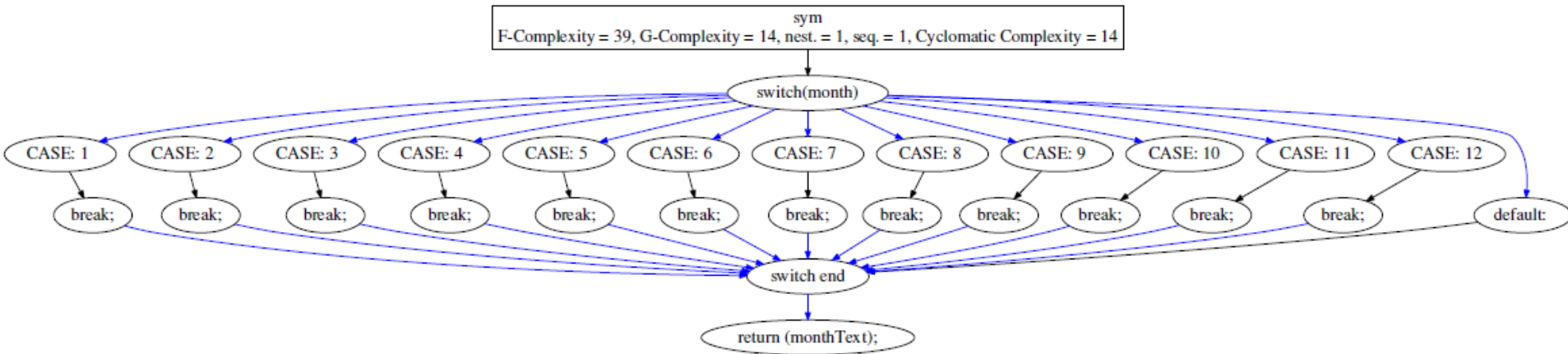


complicated
F-Complexity = 118, G-Complexity = 48, nest. = 6, seq. = 8, Cyclomatic Complexity = 9



rot: loops (for, while, do-while)
grün: goto
blau: switch and break

Erstes kleines Beispiel



rot: loops (for, while, do-while)
grün: goto
blau: switch and break

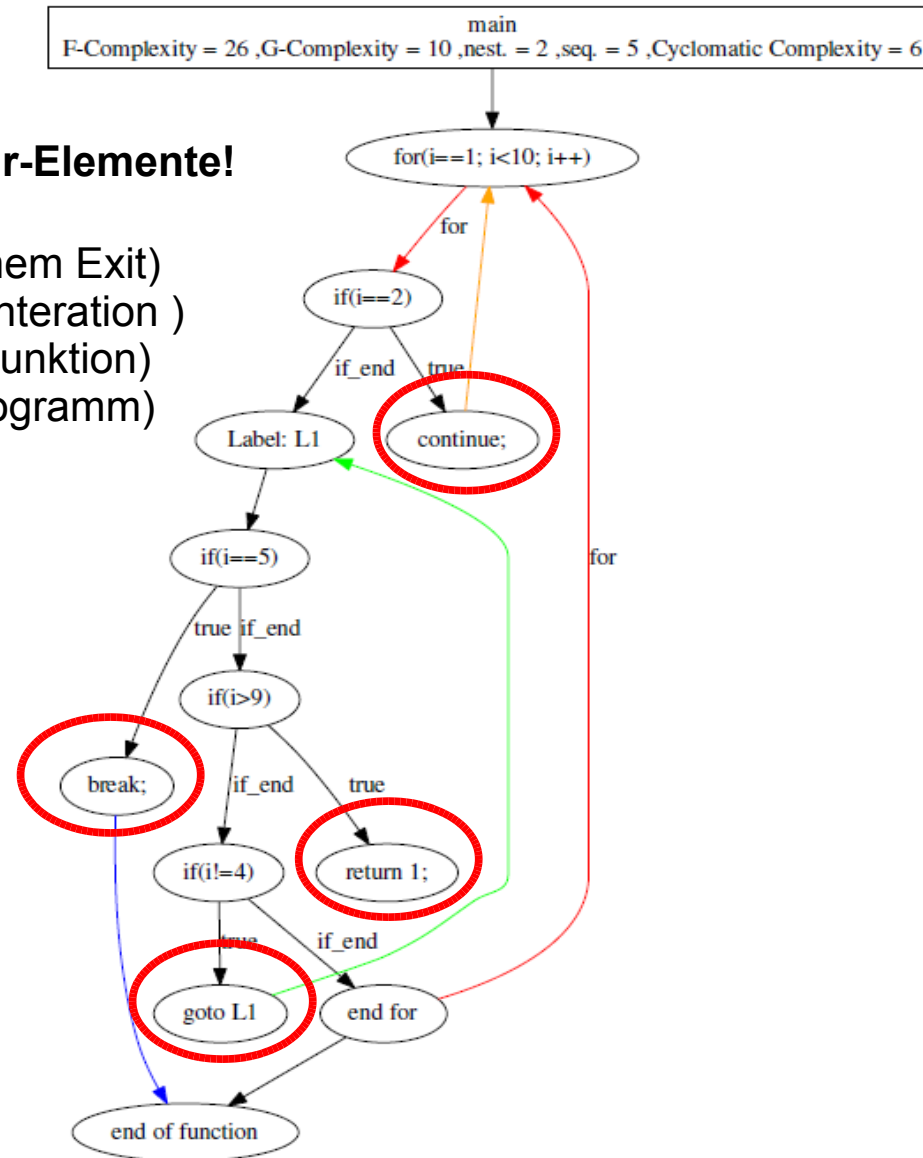
Abgeleitete Grundelemente in C

C besitzt jedoch noch weitere Struktur-Elemente!

- break (Schleife mit mehr als einem Exit)
- continue (Schleife mit vorzeitiger Iteration)
- return (vorzeitiges Ende einer Funktion)
- goto (beliebiger Sprung im Programm)

Diese müssen als zusätzliche Grundelemente angesehen werden!

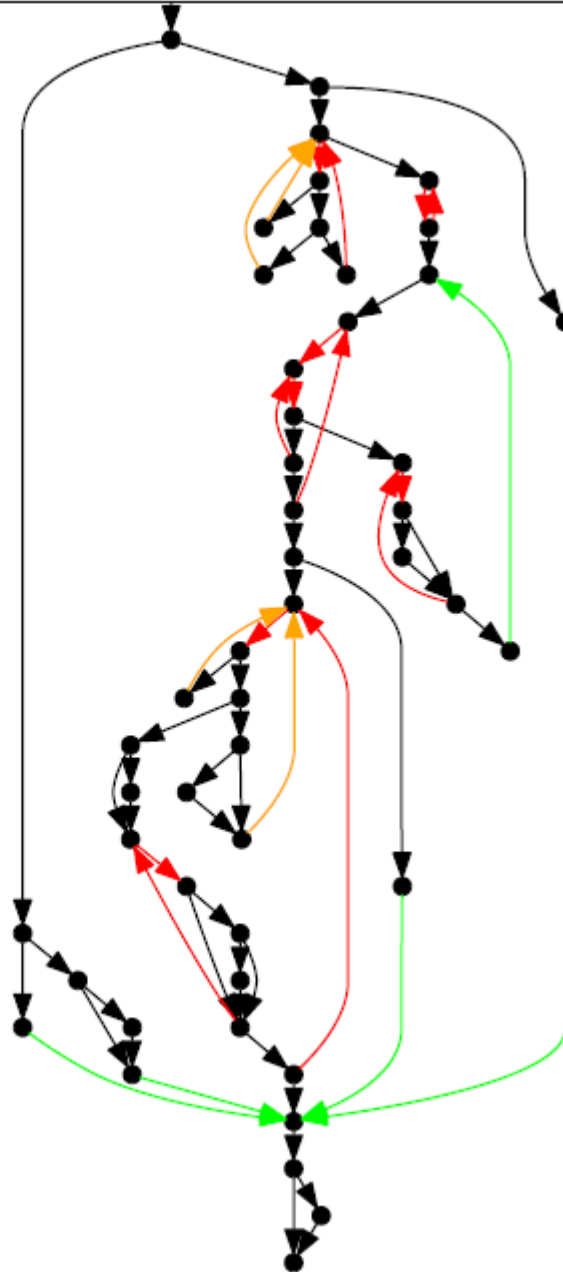
rot: Schleifen (for, while, do-while)
grün: goto
blau: switch and break
orange: continue



Sprünge

../linux/kernel/cpuset.c : generate_sched_domains
F-Complexity = 115, G-Complexity = 61, nest. = 5, seq. = 23, Cyclomatic Complexity = 24

- Vorwärtssprünge zu gut definierten Labeln können zu besser lesbarem Code führen
- Rückwärtssprünge können zu sehr komplizierten Schleifen-ähnlichem Strukturen ohne klare Exit-Bedingungen führen !



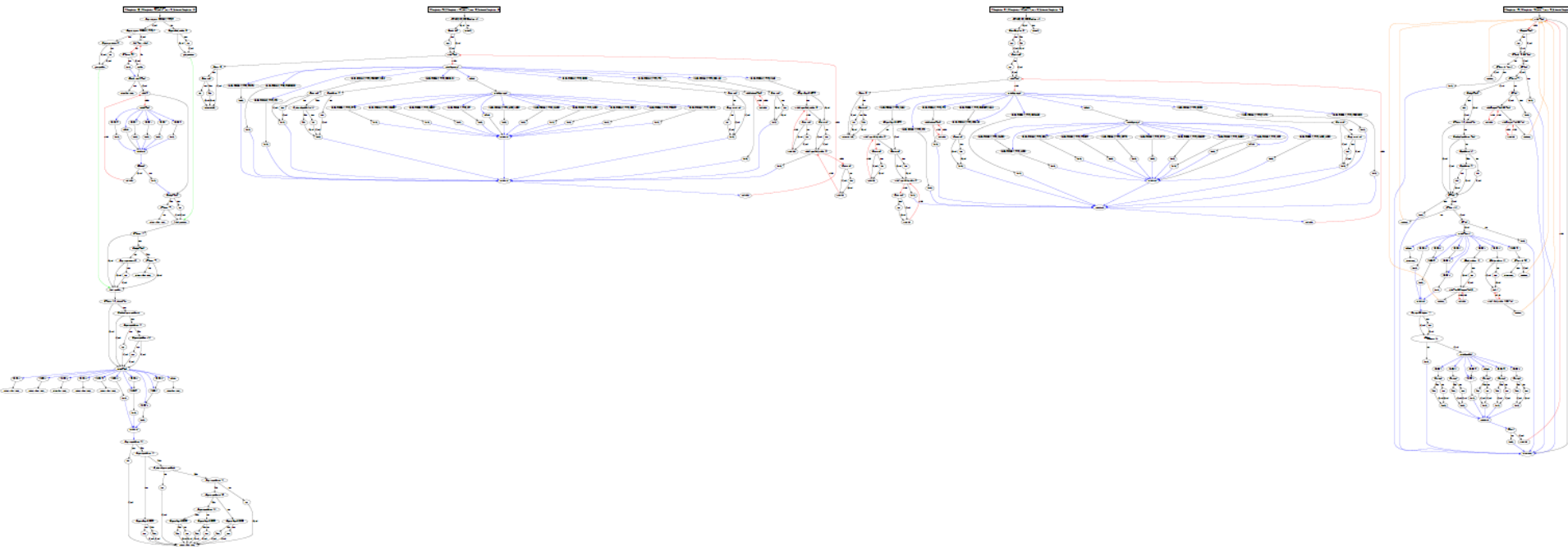
Zyklomatische-, F- and G- Komplexität

Operation	Zyklomatische Komplexität	F- und G- Komplexität
Sequenz	Summe aller Grundelemente	Summe aller Grundelemente
Nesting	Wird wie eine Sequenz behandelt	abhängig von der Nesting-tiefe: 1, 2, 3, 4, ...

Grundelement	Zyklomatische Komplexität	F-Komplexität	G-Komplexität
if or if-else	1	1	1
for	1	2	1
while	1	2	1
do-while	1	2	1
case + default	1	1	1
goto	0	1 für Vorwärtssprünge 5 für Rückwärtssprünge	0
break	0	1 für breaks nach cases 2 für breaks aus Schleifen	0
continue	0	2	0
return	0	1	0

Erste Ergebnisse

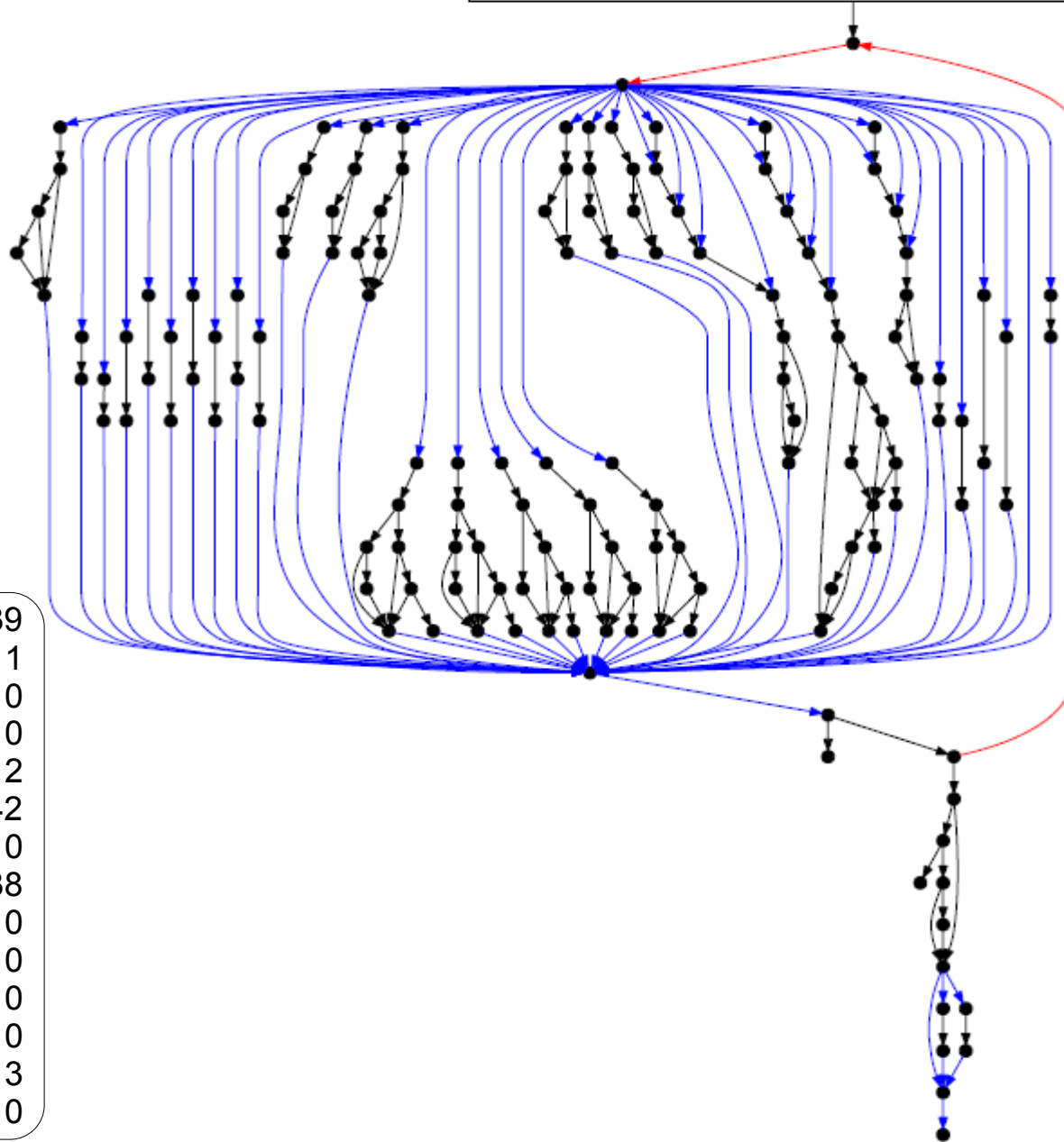
- 8679 Dateien vom Linux "kernel" 3.7.9; Treiberpfad nicht mitberücksichtigt
 - 108887 Funktionen analysiert
 - mehr als 7500 Funktionen mit einer F-Komplexität > 30
 - 854 Funktionen mit einer F-Komplexität > 100
 - F-Komplexitätswerte bis 1250 !
- Riesenspass die Graphen anzuschauen :-)



alle Linux files (Ver. 3.7.9)

	# Dateien	# Funktionen	F-Com ≥ 30	F-Com ≥ 100	F-Max
kernel	210	6679	306	28 (0.4%)	365
lib	207	1343	104	22 (1.6%)	529
crypto	148	1190	39	1 (0.08%)	464
ipc	13	249	19	3 (1.20%)	120
arch	5103	46059	2019	236 (0.5%)	1092
block	42	897	46	8 (0.89%)	194
fs	1184	22282	2541	287 (1.2%)	1246
init	10	102	11	1 (0.98%)	105
mm	76	2800	218	18 (0.64%)	283
net	1399	20397	1823	185 (0.91%)	1043
security	95	2059	177	27 (1.31%)	796
sound	192	4830	381	38 (0.78%)	343

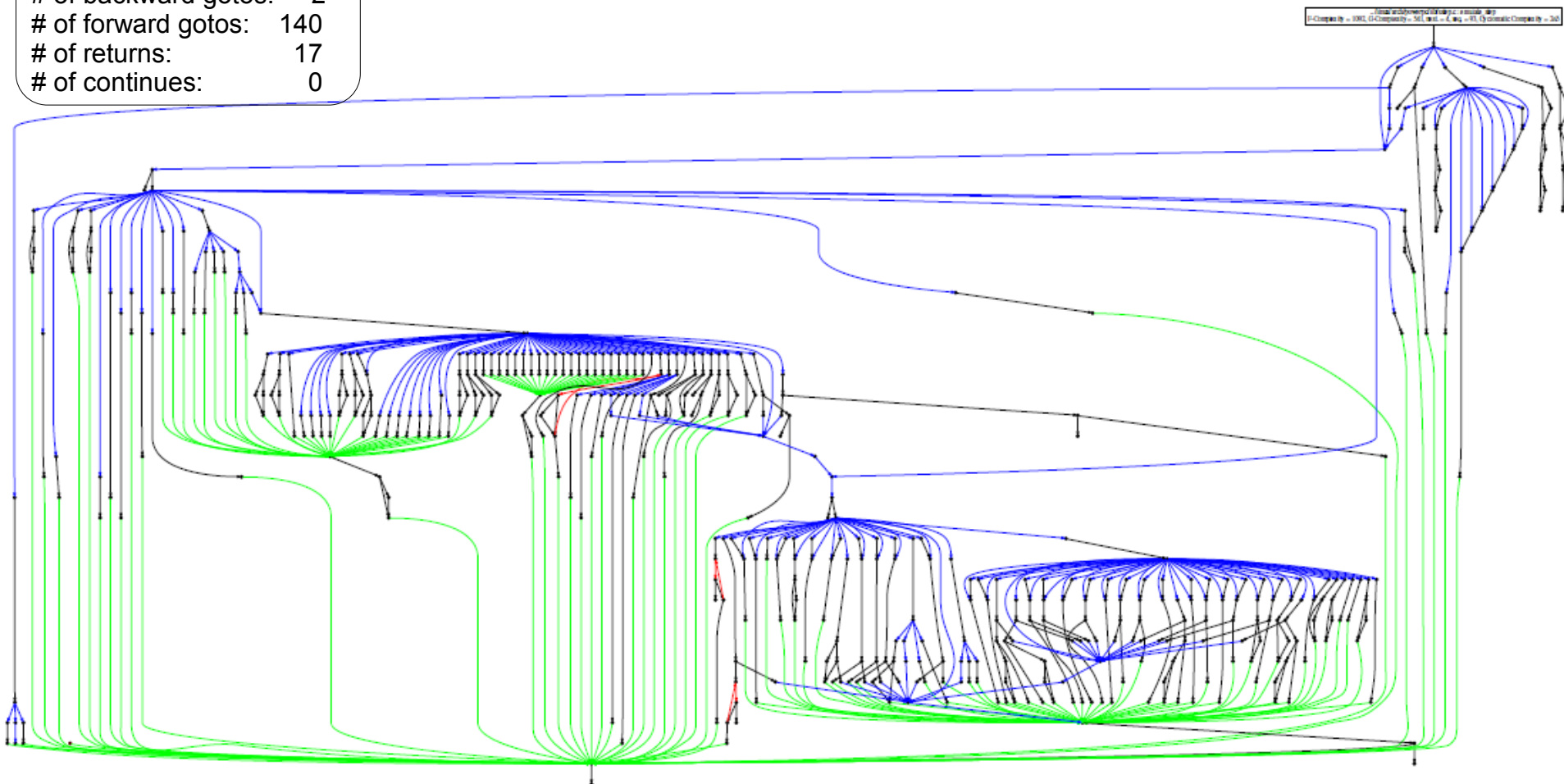
../linux/kernel/auditsc.c : audit_filter_rules
 F-Complexity = 365, G-Complexity = 225, nest. = 6, seq. = 42, Cyclomatic Complexity = 83



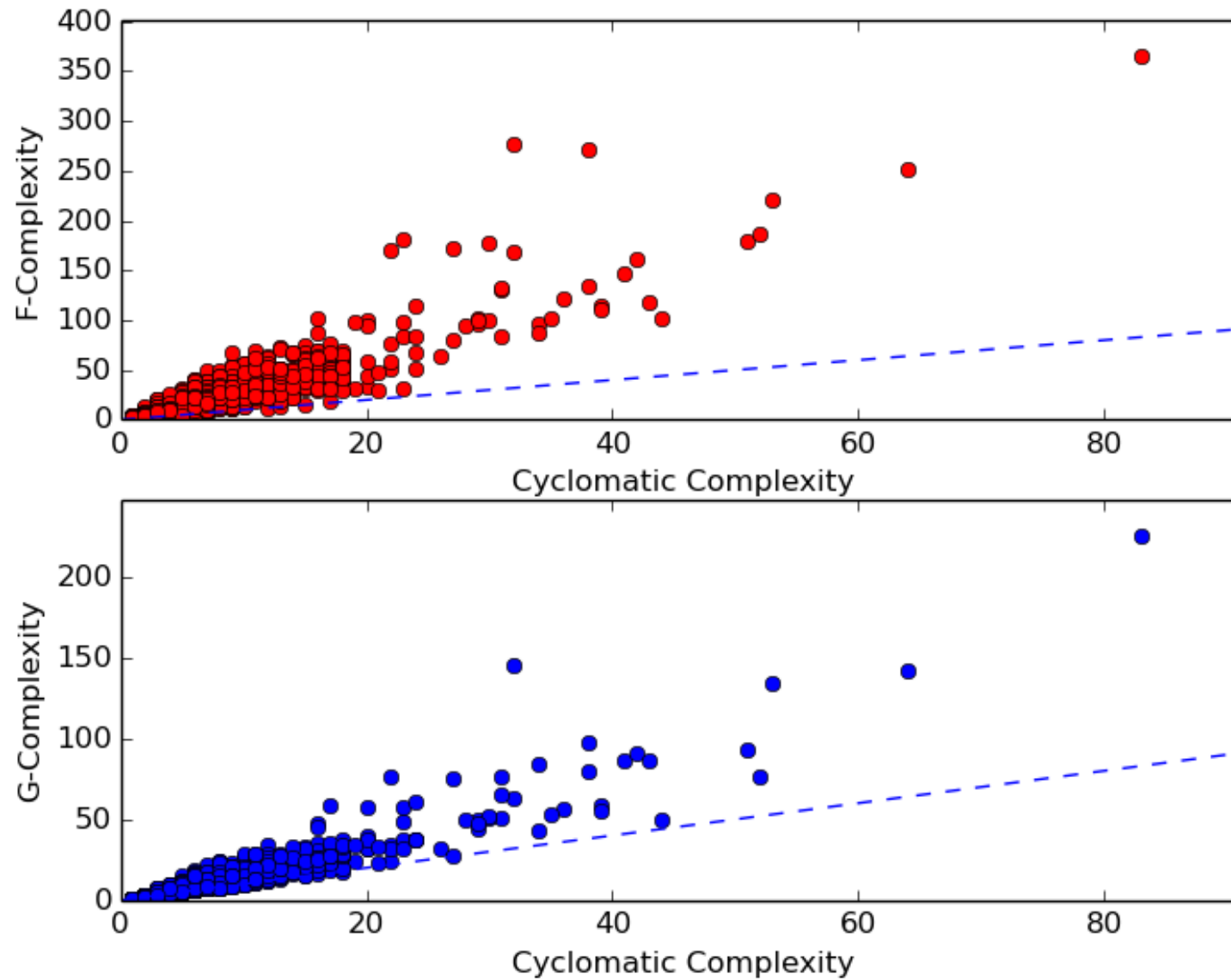
# of ifs:	39
# of fors:	1
# of Do_Whiles:	0
# of Whiles:	0
# of Switches:	2
# of Cases:	42
# of Defaults:	0
# of breaks in cases:	38
# of breaks in loops:	0
# of Labels:	0
# of backward gotos:	0
# of forward gotos:	0
# of returns:	3
# of continues:	0

# of ifs:	76
# of fors:	1
# of Do_Whiles:	2
# of Whiles:	0
# of Switches:	14
# of Cases:	185
# of Defaults:	0
# of breaks in cases:	27
# of breaks in loops:	2
# of Labels:	4
# of backward gotos:	2
# of forward gotos:	140
# of returns:	17
# of continues:	0

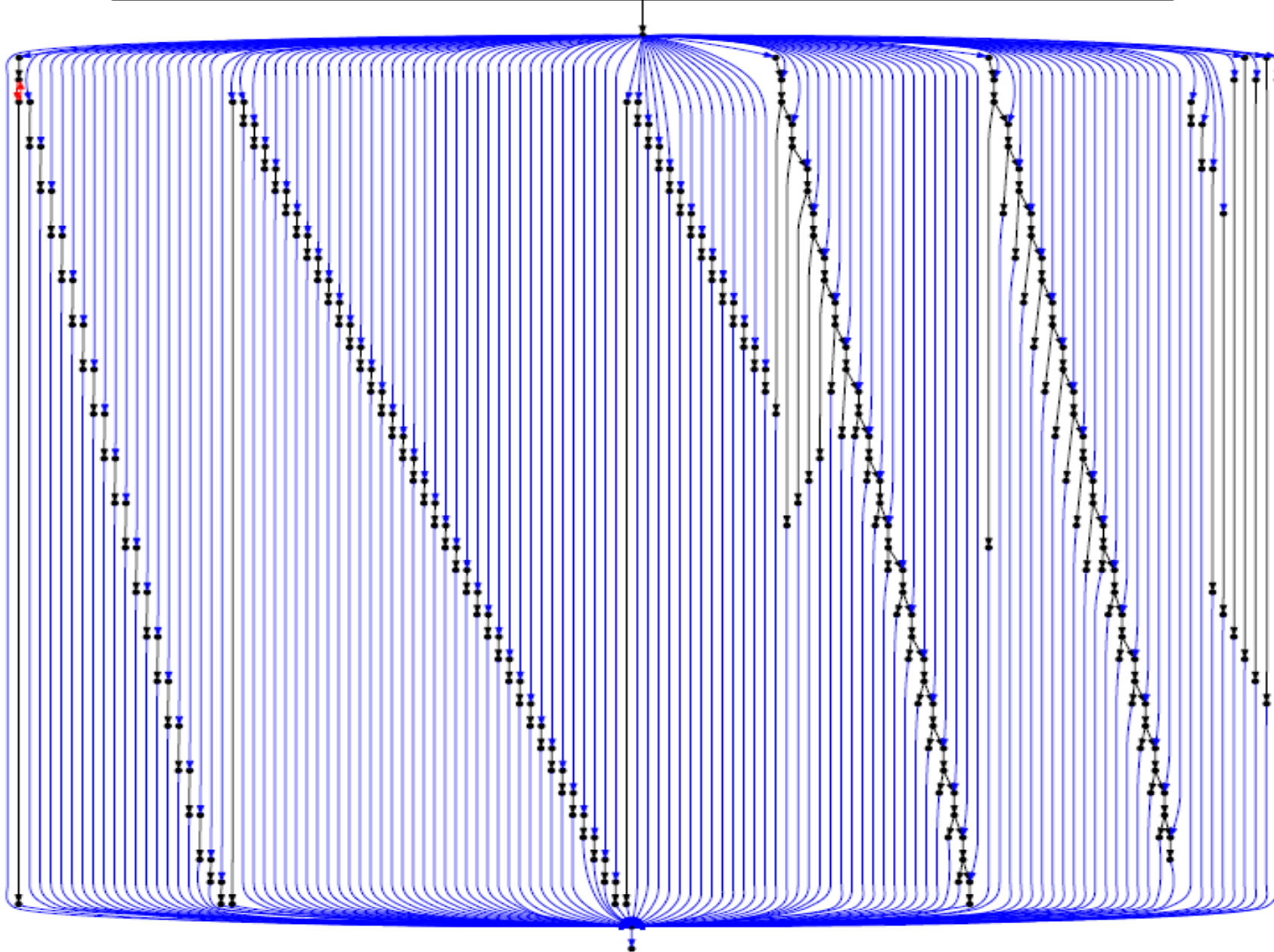
./linux/arch/powerpc/lib/sstep.c : emulate_step
 F-Complexity = 1092, G-Complexity = 541, nest. = 4, seq. = 93, Cyclomatic Complexity = 265

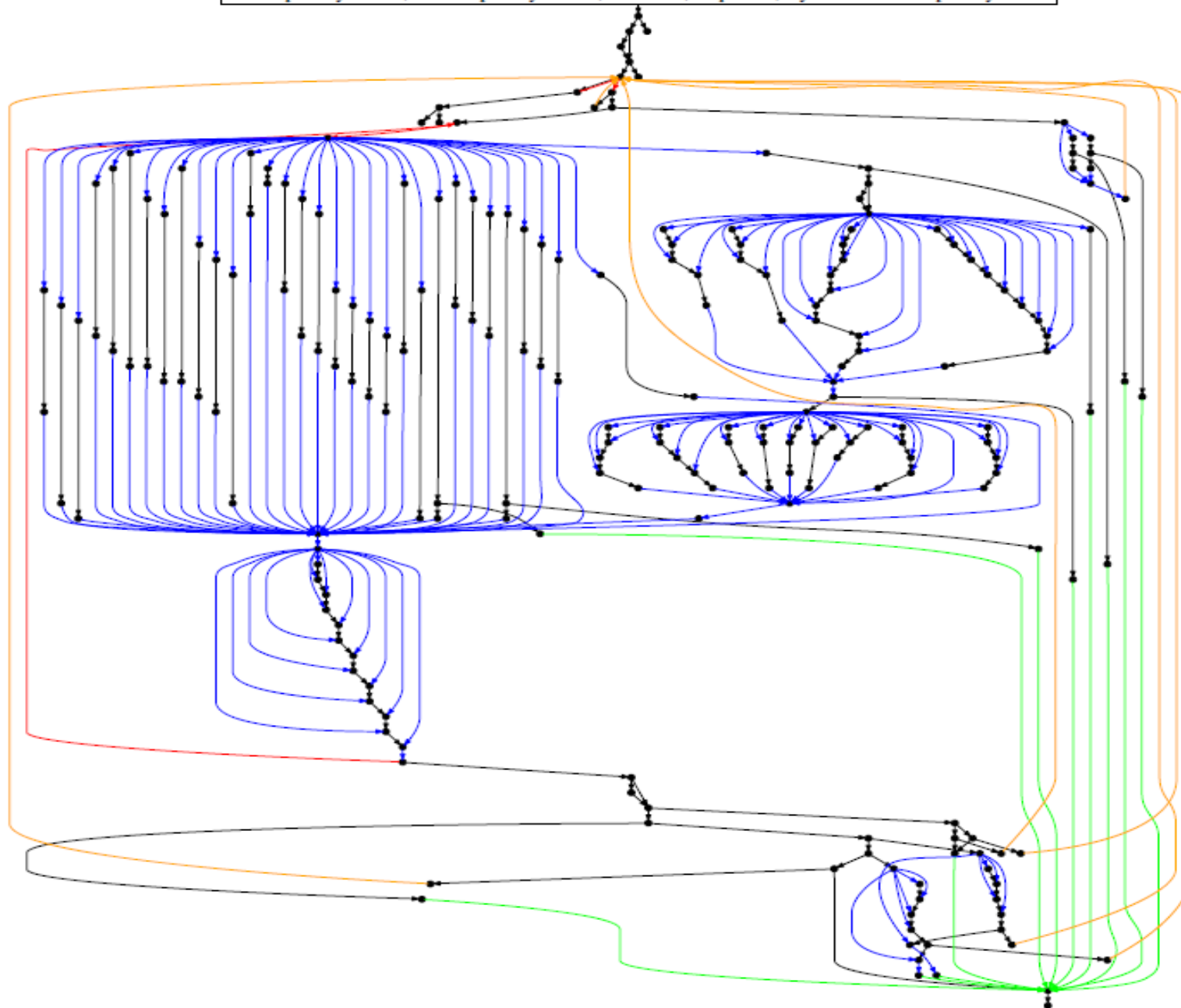


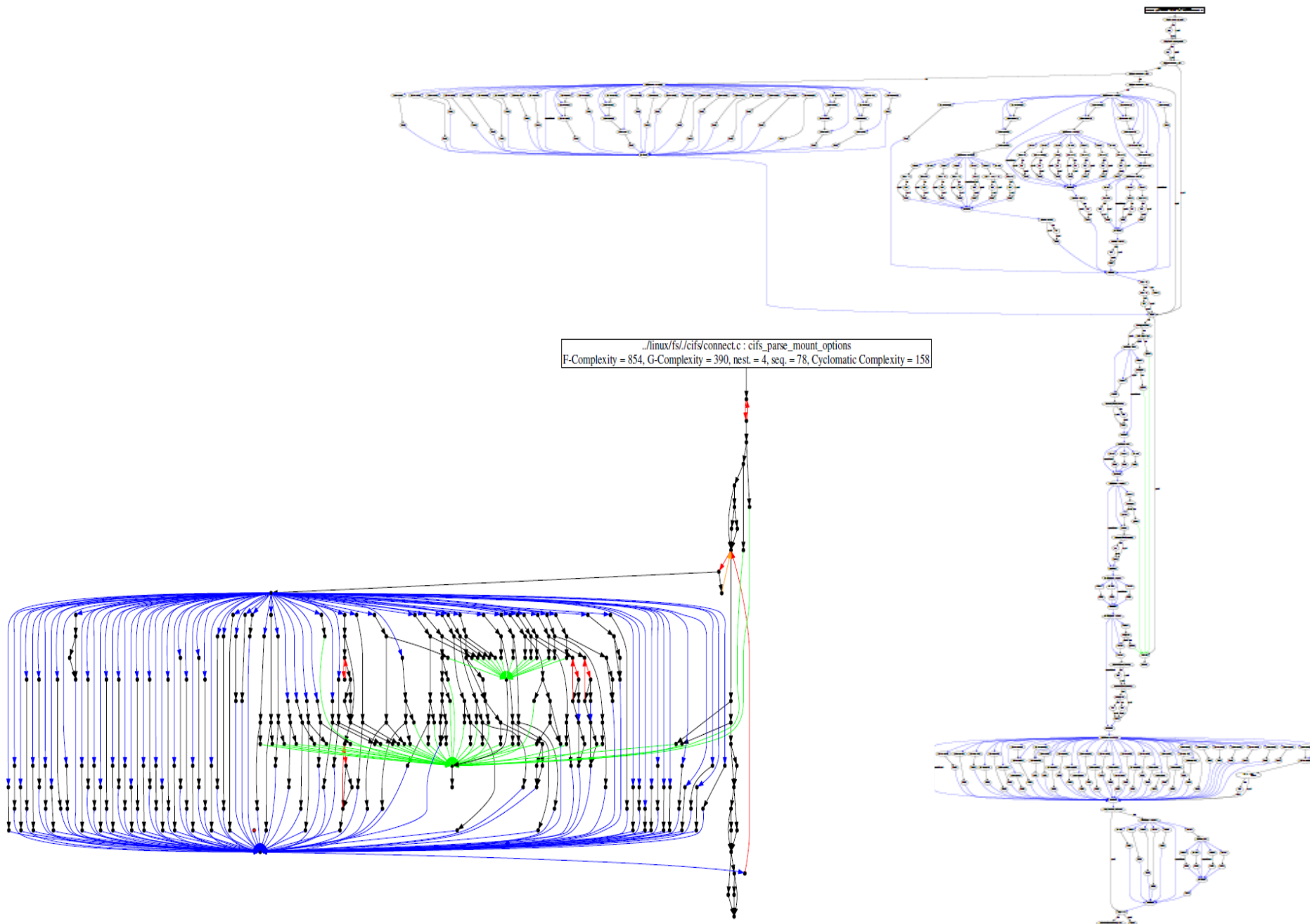
kernel files



../linux/crypto/tcrypt.c : do_test
F-Complexity = 464, G-Complexity = 194, nest. = 2, seq. = 37, Cyclomatic Complexity = 156

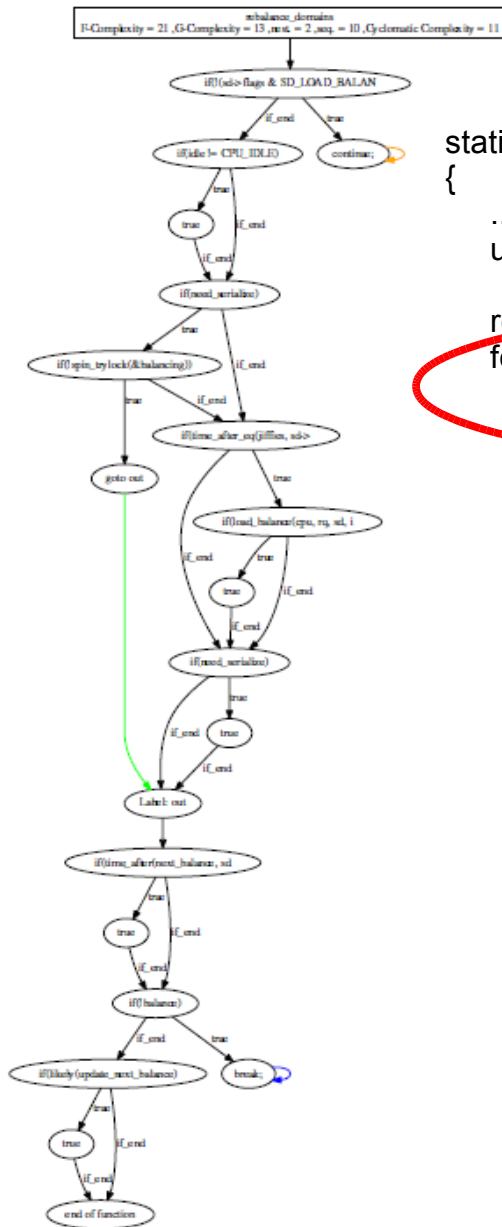






Einfluss von Macros

file: linux/sched/fair.c



```
static void rebalance_domains(int cpu, enum cpu_idle_type idle)
{
```

```
    ...
    update_shares(cpu);
```

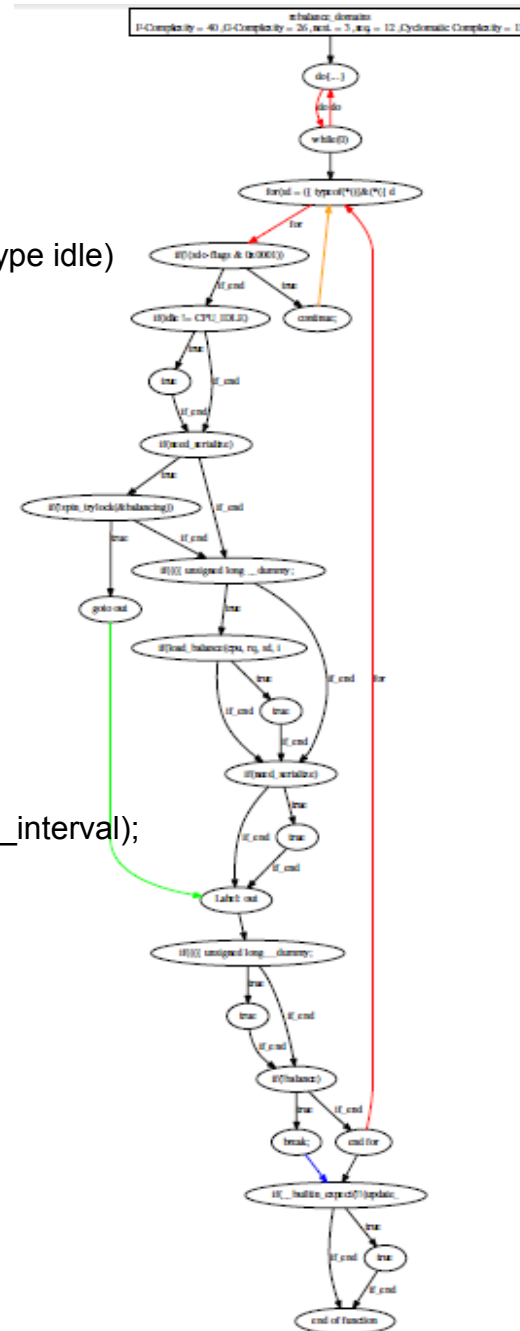
```
    rcu_read_lock();
    for_each_domain(cpu, sd) {
        if (!(sd->flags & SD_LOAD_BALANCE))
            continue;
```

```
        interval = sd->balance_interval;
        if (idle != CPU_IDLE)
            interval *= sd->busy_factor;
```

```
        /* scale ms to jiffies */
        interval = msecs_to_jiffies(interval);
        interval = clamp(interval, 1UL, max_load_balance_interval);
        need_serialize = sd->flags & SD_SERIALIZE;
```

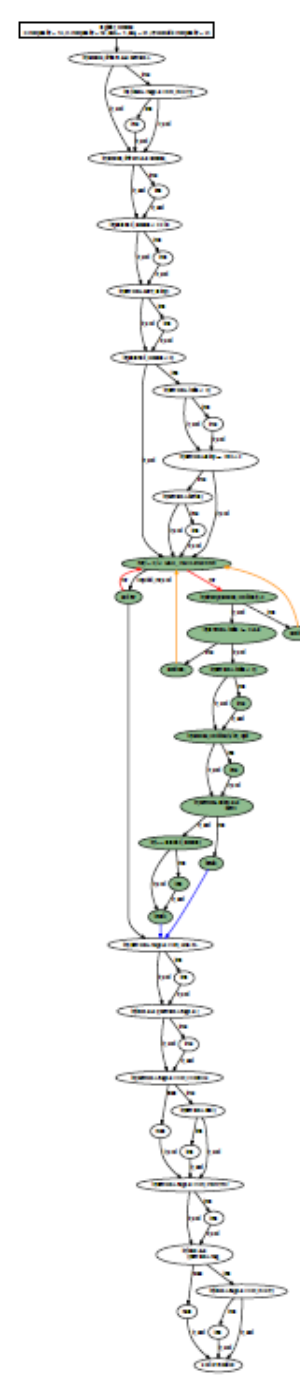
```
        if (need_serialize) {
            if (!spin_trylock(&balancing))
                goto out;
```

```
        }
    ...
}
```

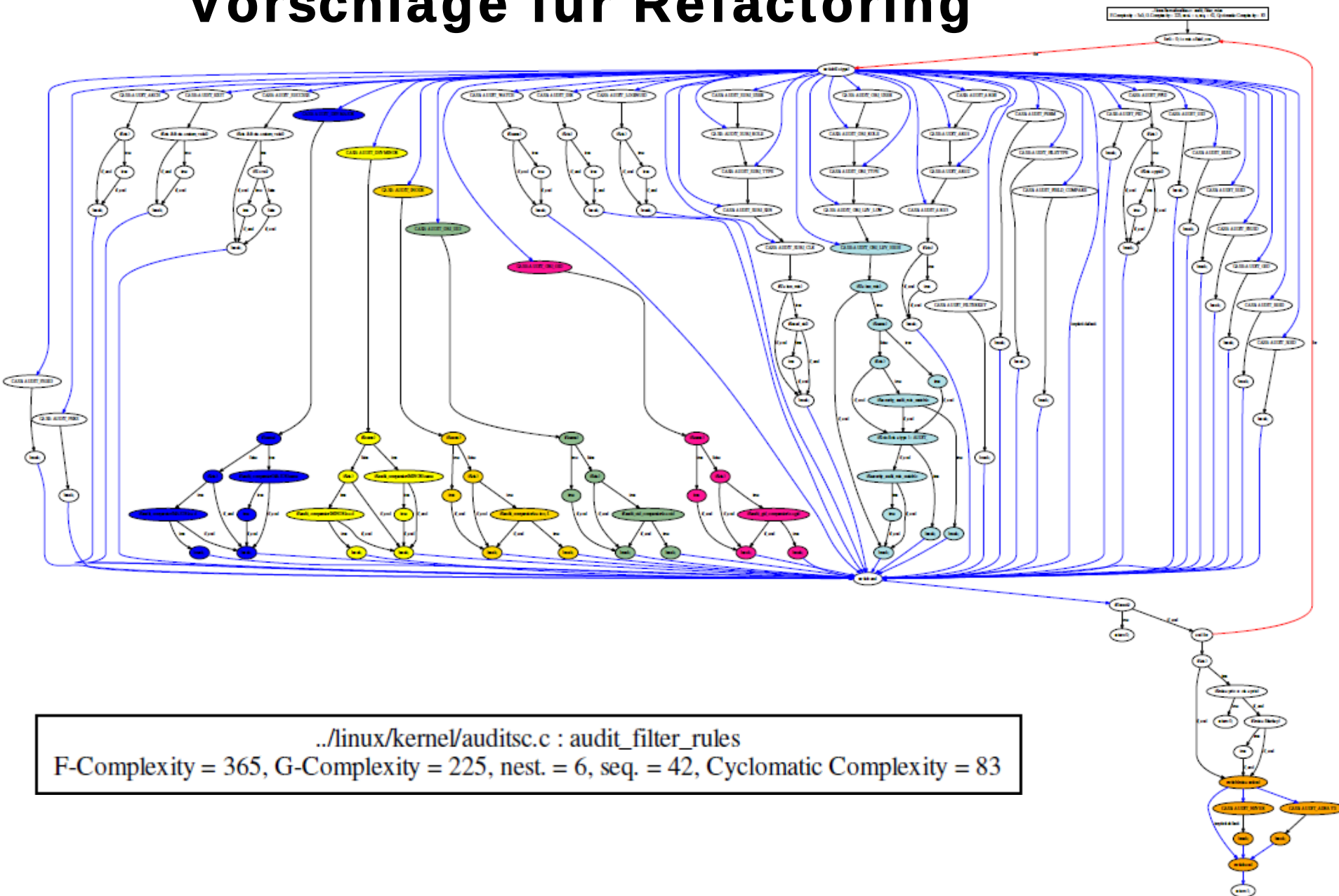


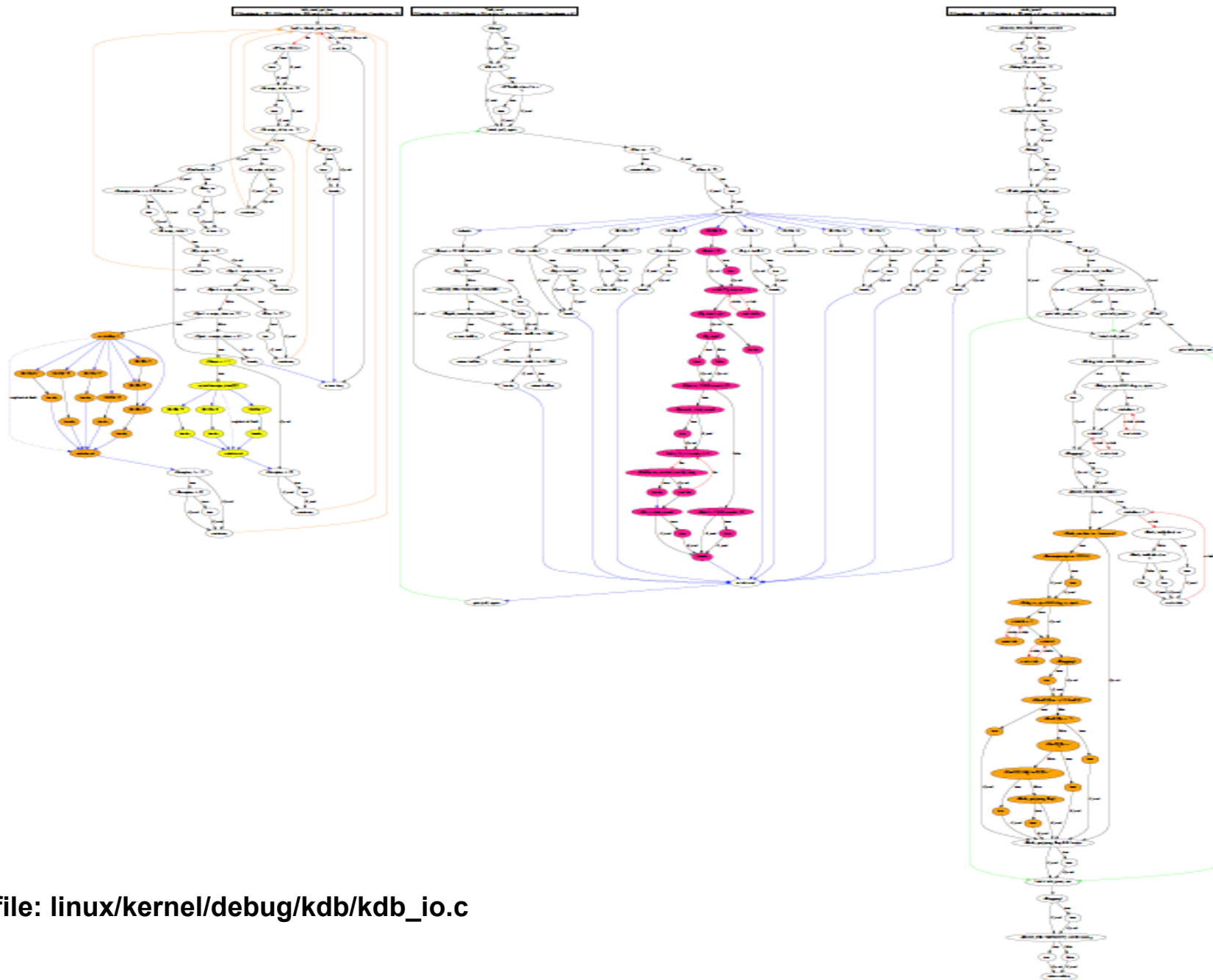
Refaktoring

CFG erlauben es sehr leicht, tief genestete Strukturen zu finden, welche nur einen Ein- und einen Ausgang besitzen.



Vorschläge für Refactoring

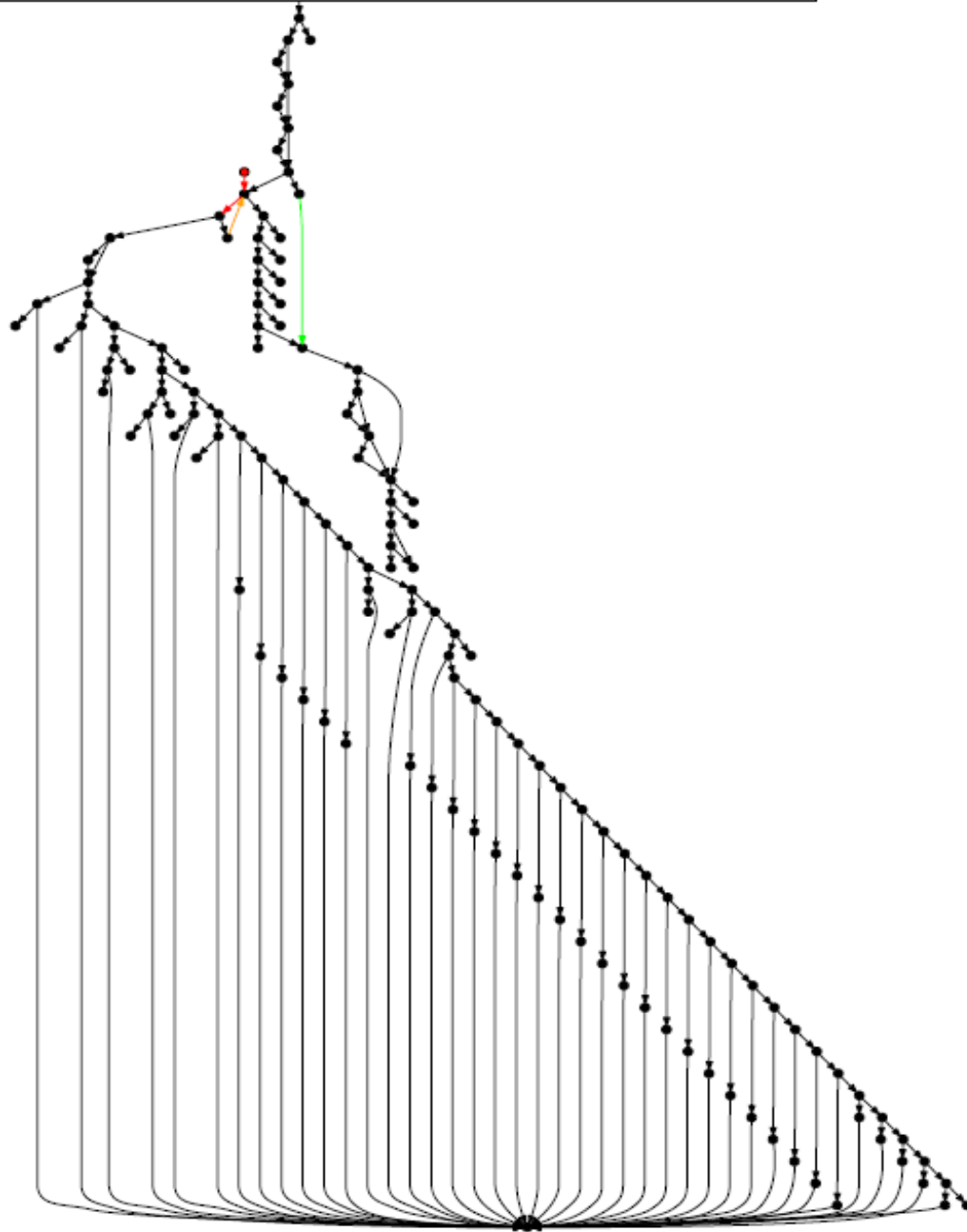




file: linux/kernel/debug/kdb/kdb_io.c

alle Linux files (Ver. 3.7.9)

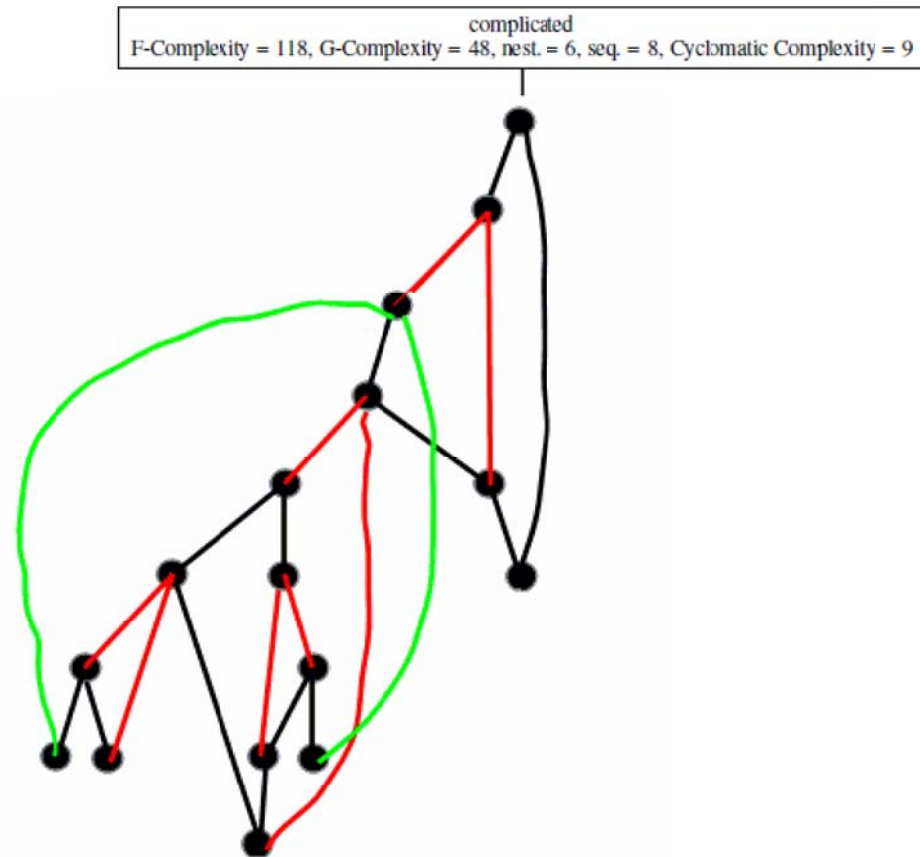
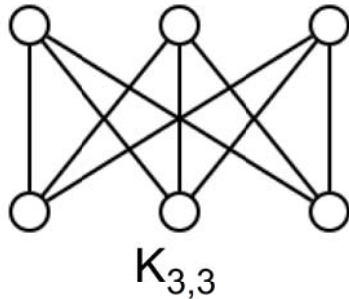
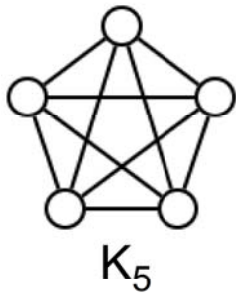
	#Dateien	#Funktionen	F-Com > 30	F-Comp > 100	F-Max	# Refs	# Refs/ #Funktionen
kernel	210	6679	306	28	365	188	2.8 %
lib	207	1343	104	22	529	92	6.8 %
crypto	148	1190	39	1	464	16	1.3 %
ipc	13	249	19	3	120	14	5.6 %
arch	5103	46059	2019	236	1092	1342	2.9 %
block	42	897	46	8	194	37	4.1 %
fs	1184	22282	2541	287	1246	1121	5.0 %
init	10	102	11	1	105	8	7.8 %
mm	76	2800	218	18	283	72	2.5 %
net	1399	20397	1823	185	1043	1218	6.0 %
security	95	2059	177	27	796	108	5.2 %
sound	192	4830	381	38	343	373	7.7 %



Graphentheoretische Deutung

- CFG sind gerichtete Graphen
- Jedes Schleifenelement und rückwärts gerichtete Kanten kreieren Zyklen.
- Normalerweise (aber nicht immer) sind Graphen planar, d.h. die Kanten kreuzen sich nicht...
- Sind nicht-planare Graphen komplexer?

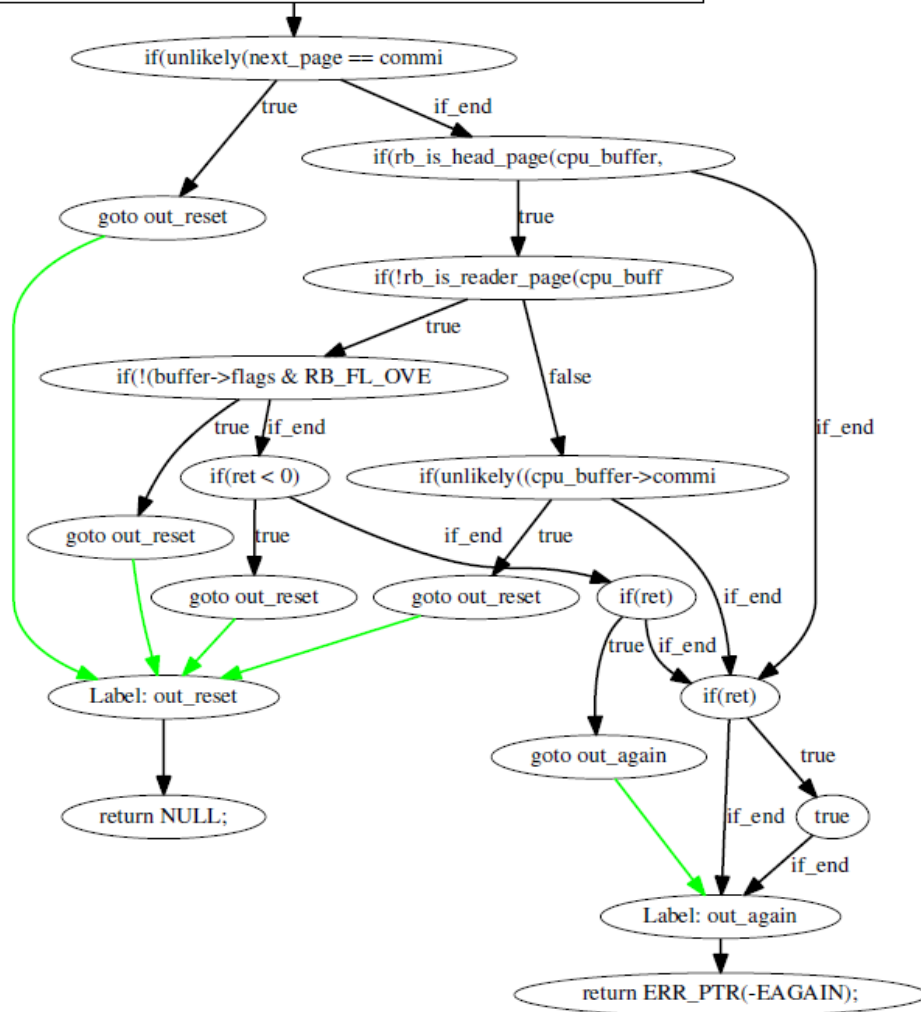
Theorem of Kuratowski



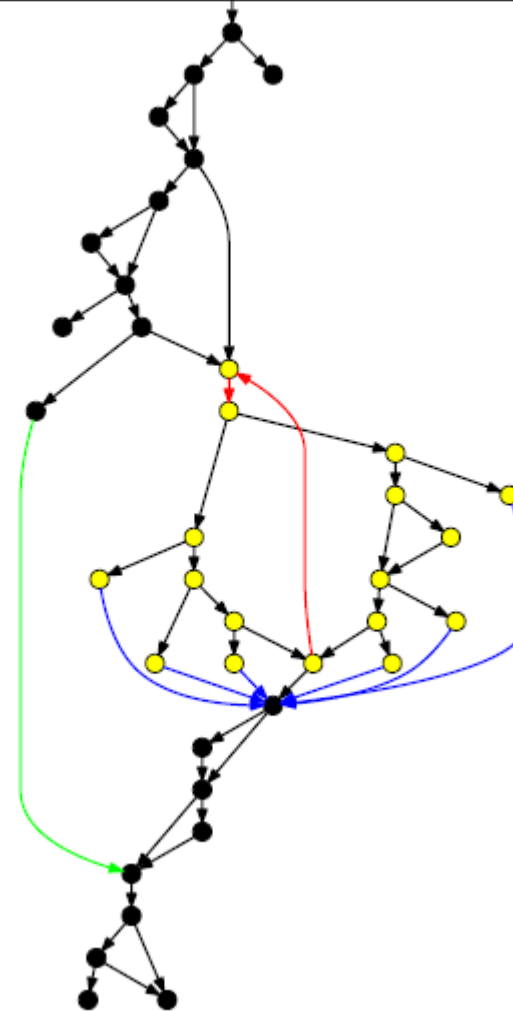
Nichtplanare Graphen

Nur 19 nichtplanare Graphs im linux/kernel/-Pfad

../linux/kernel/trace/ring_buffer.c : rb_move_tail
F-Complexity = 38, G-Complexity = 18, nest. = 3, seq. = 8, Cyclomatic Complexity = 9

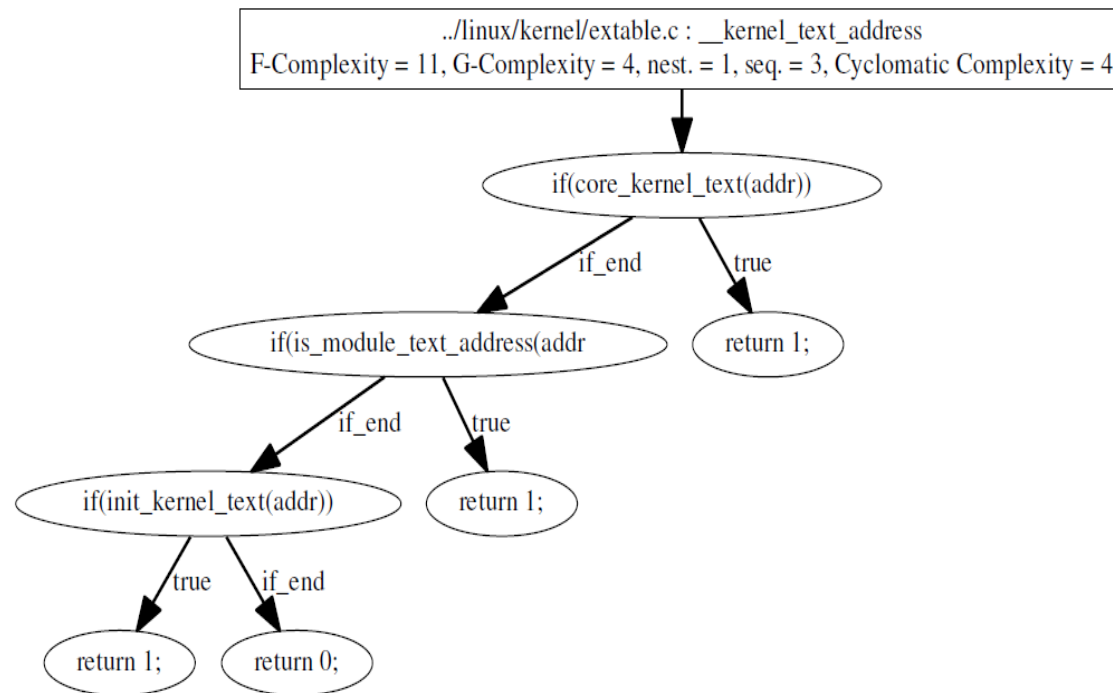


../linux/kernel/sysctl.c : __do_proc_dointvec
F-Complexity = 100, G-Complexity = 40, nest. = 3, seq. = 19, Cyclomatic Complexity = 20



Graphentheoretische Klassifikation

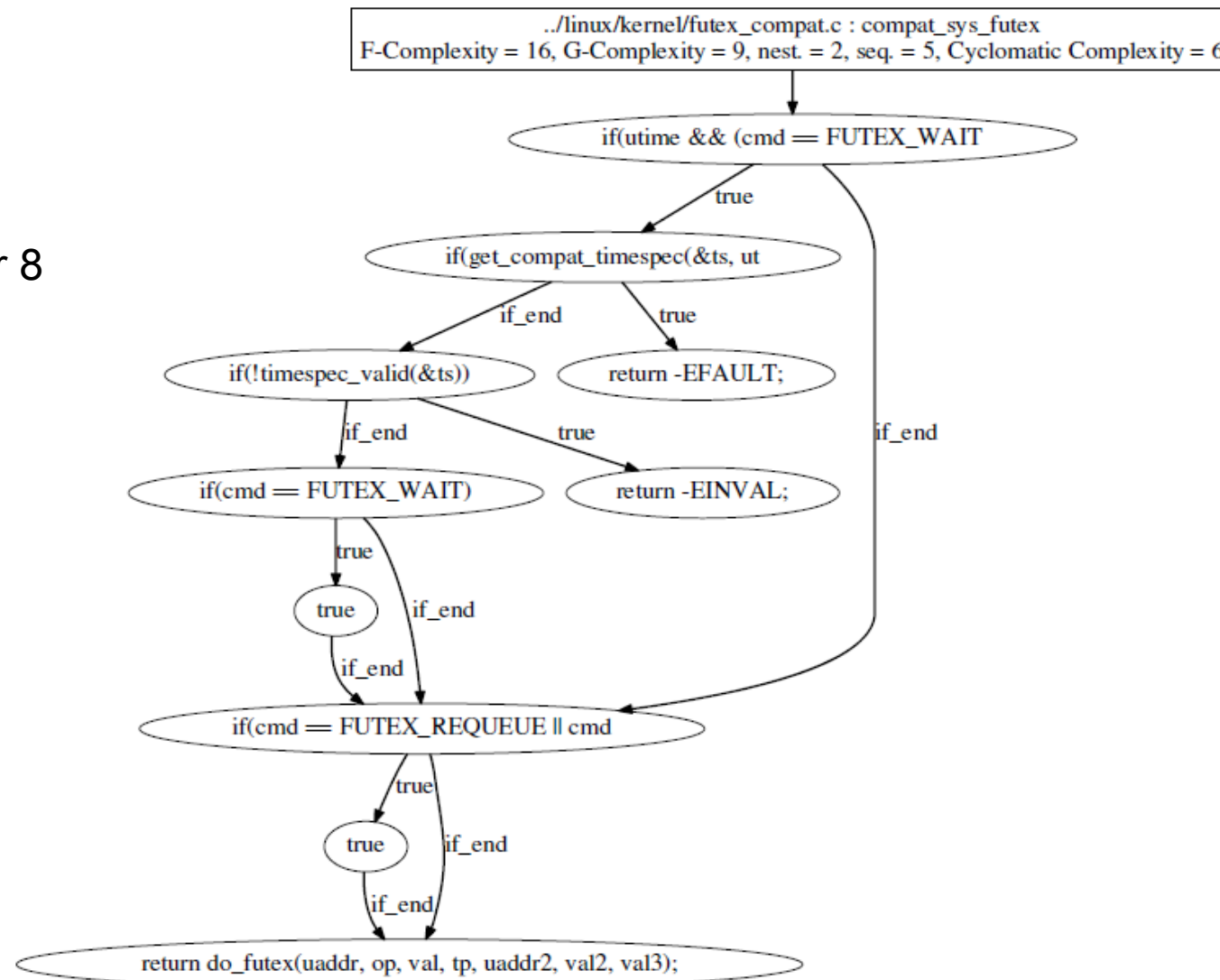
Funktionen mit Baumstruktur:



Graphentheoretische Klassifikation

Azyklische Graphen (Directed acyclic graphs (DAG))?

Quantitative
Klassifikation:
Längster Pfad: hier 8



Graphentheoretische Klassifikation

Adjazenz Matrix (n x n)

$$\underline{\underline{\mathbf{A}}} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Algebraische Berechnungen:

$$\underline{\underline{\mathbf{A}}} \cdot \underline{\underline{\mathbf{v}}} = \underline{\underline{\lambda}} \cdot \underline{\underline{\mathbf{v}}}$$

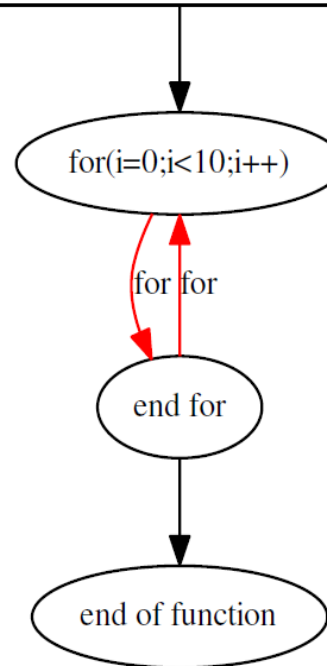
λ_i = Eigenwerte (complex)

$\mathbf{v}_i$ = Eigenvektoren (complex)

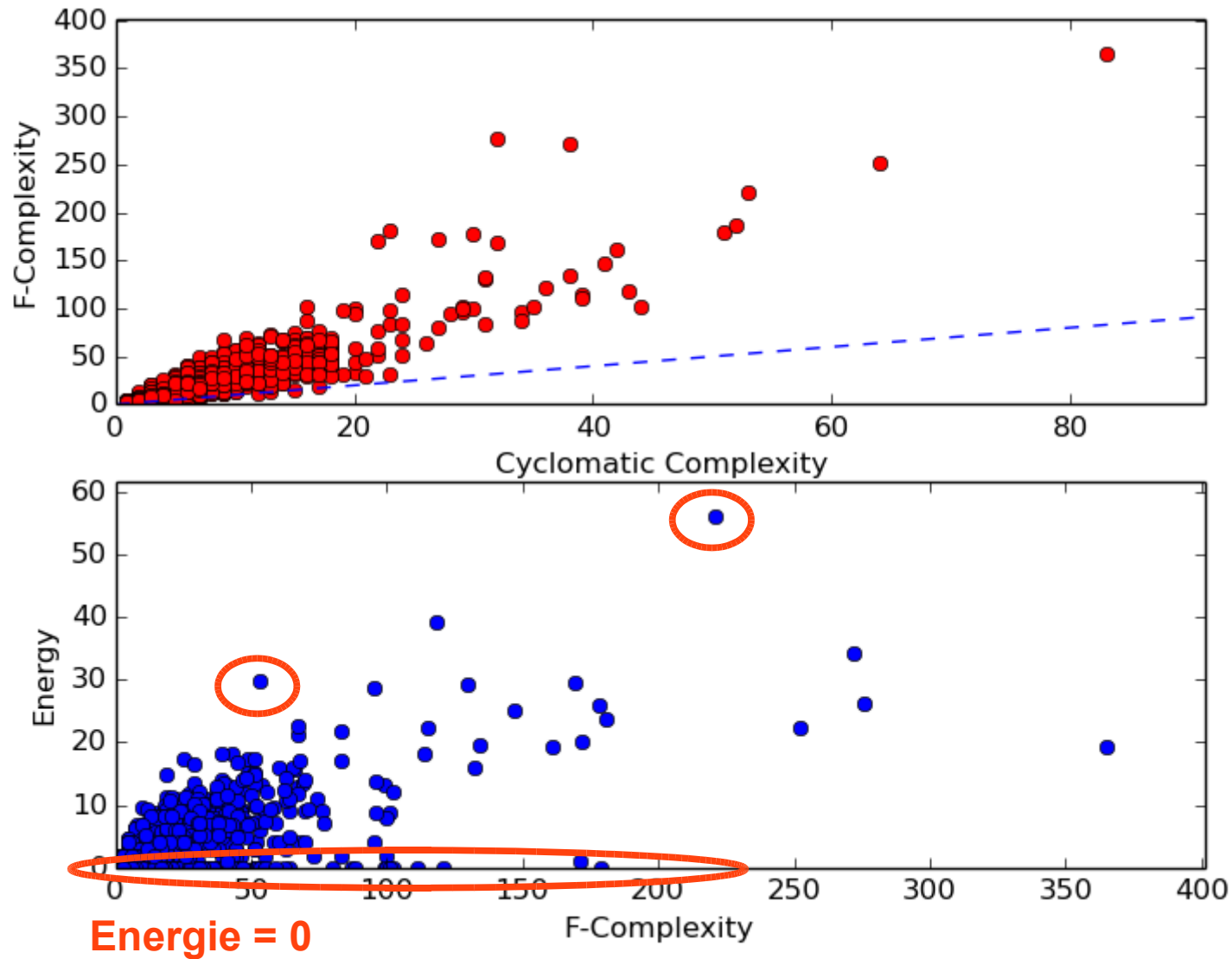
$i = 0 \dots n-1$

Graphenenergie: $E = \sum |\lambda_i|$

test1.c : main
F-Complexity = 3, G-Complexity = 3, nest. = 1, seq. = 1, Cyclomatic Complexity = 2



F-Komplexität und Energie



Energie = 0: Bäume oder DAGs

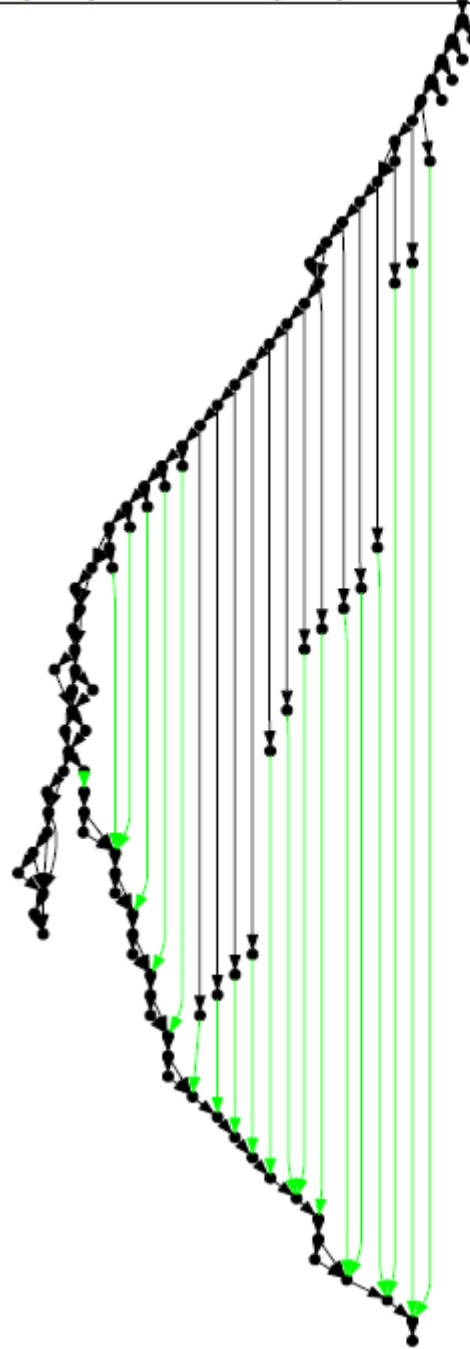
Längster Pfad, hier: 66

$$\underline{\underline{A}} = \begin{pmatrix} 0 & 1 & 0 & 0 & 1/0 \\ 0 & 0 & 0 & 0 & 1/0 \\ 0 & 0 & 0 & 1 & 1/0 \\ 0 & 0 & 0 & 0 & 1/0 \\ 0 & 0 & 0 & 0 & 0 \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \dots \end{pmatrix}$$

$$|(\underline{\underline{A}}) \cdot (\underline{\underline{E}} - \lambda)| = 0$$

$$\lambda^n = 0$$

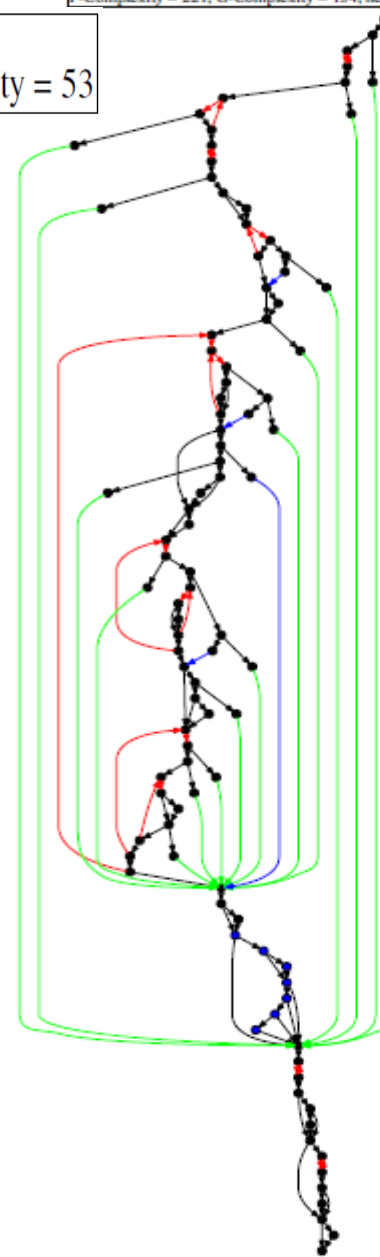
$$\lambda_n = 0 \rightarrow \text{Energie} = 0$$



../linux/kernel/power/swap.c : load_image_lzo
F-Complexity = 221, G-Complexity = 134, nest. = 4, seq. = 52, Cyclomatic Complexity = 53

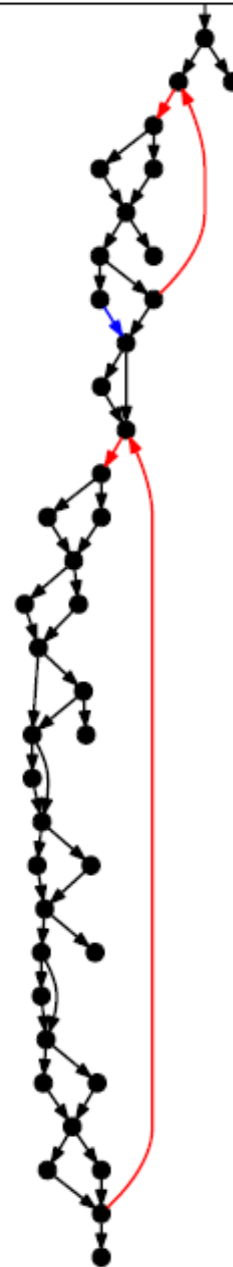
F-Comp: 221
Graphenenergie: 55.40 (höchster Wert)

# of ifs:	40
# of fors:	12
# of Do_Whiles:	0
# of Whiles:	0
# of Switches:	0
# of Cases:	0
# of Defaults:	0
# of breaks in cases:	0
# of breaks in loops:	4
# of Labels:	2
# of backward gotos:	0
# of forward gotos:	15
# of returns:	1
# of continues:	0



F-Comp: 53
Graphenenergie: 29.77

# of ifs:	15
# of fors:	1
# of Do_Whiles:	1
# of Whiles:	0
# of Switches:	0
# of Cases:	0
# of Defaults:	0
# of breaks in cases:	0
# of breaks in loops:	1
# of Labels:	0
# of backward gotos:	0
# of forward gotos:	0
# of returns:	5
# of continues:	0



Graph theoretical Classification

Version 3.7.9: kernel-path: 6679 Files

	# Funktionen	längster Pfad	planar	Max F- Komplexität	max Energie
Baum-Funktionen	3348	10	0	52	0
Azyklische Funktionen (inkl. Bäume)	5577	66	5	186	0
zyklische Funktionen	1102	∞	14	365	55.4

Was bringen Codefluss-komplexitäts-Messungen ???

Intuitiv denkt man, dass die Fehlerdichte (z.B. errors/kLOC) mehr oder weniger linear abhängig von der Komplexität einer Funktion ist.

Empirische Studien zeigen aber, dass dies nicht der Fall ist.

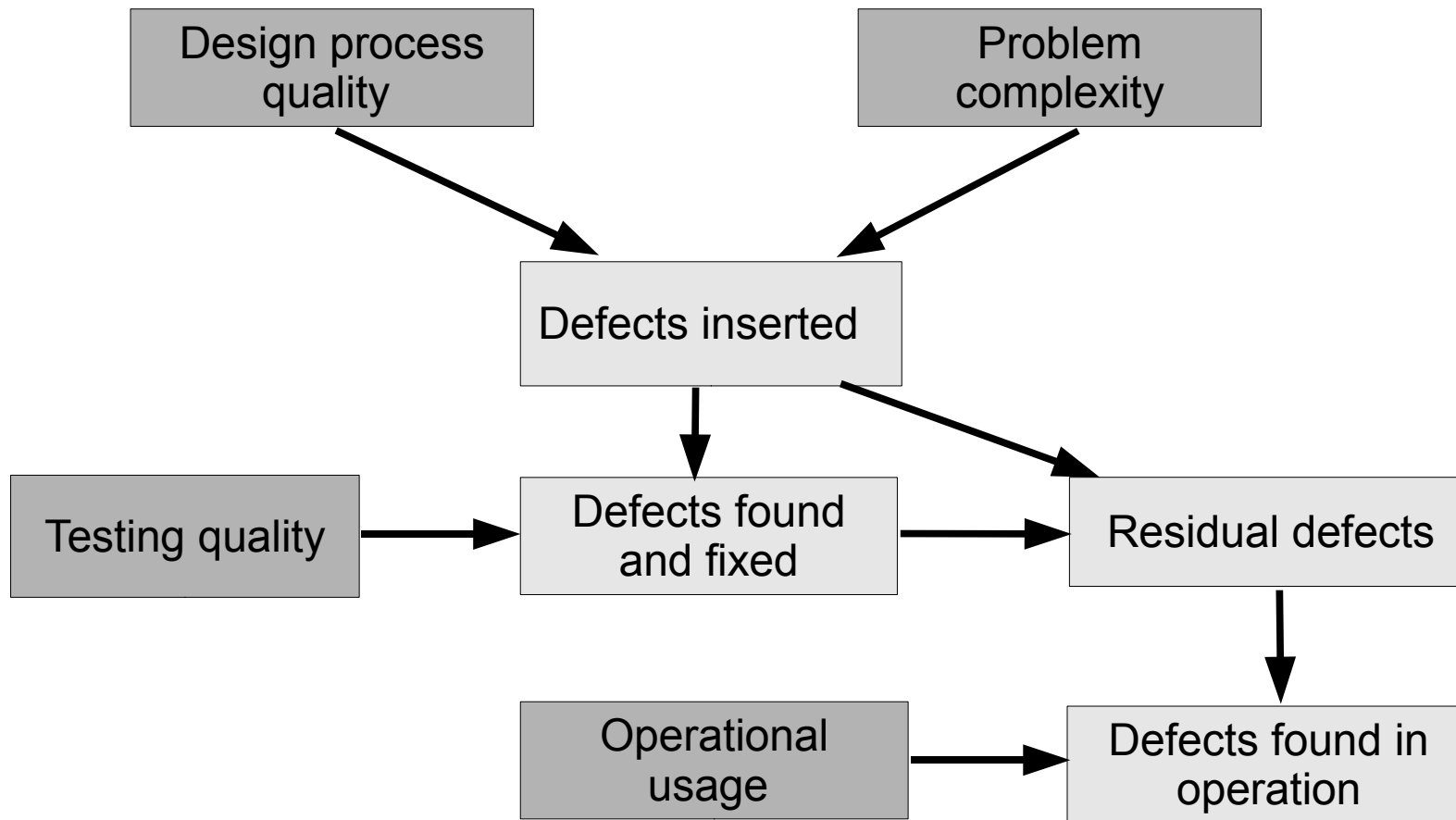
Komplexer Code ist oft besser getestet, was zu einer niedriger Fehlerdichte führt.

Die Fehlerdichte ist abhängig von :

- Qualität der Entwickler
- Zeitlicher Aufwand einer Entwicklung
- Testqualität
- Testaufwand
- Verwendete Werkzeuge und Methodologien
- und vieles mehr....
- Komplexität !!!

Diese Parameter sind oft rein subjektiv!

Bayes'sche Belief Netzwerke für Softwarequalität



Simple BBN for Software defects

[Fenton et al., Risk Assessment and Decision Analysis with Bayesian Networks, 2012]

Nächste Schritte:

- Interpretation der Eigenwerte and Eigenvektoren
- Integration von Datenkomplexität (z.B. LOC zwischen den Elementen)
- Kanten zwischen den Funktionen
- Besserer Parser

Herzlichen Dank an

- Kourosh Parsa (University of Queensland, Brisbane)
- Cristina Cifuentes und Nathan Hawes: Oracle Research Labs
- Rolf Riedi (HEIA-FR)
- Pamela Torche, Yannick Hervieux und Leonardo Antúnez.

für die vielen hilfreichen Diskussionen über dieses Thema

Vielen Dank für Eure Aufmerksamkeit !

Nächstes Jahr Ende Januar/Februar: Fri-Sw-Days, Fribourg, Schweiz
<https://fri-software-days.ch>