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

Wolfram Luithardt

Codeflow-Analysen und Komplexitätsmessungen an Linuxsystemen

Hes·so FRIBOURG FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

The University Of Queensland AUSTRALIA
School of
Information Technology and Electrical Engineering

Inhalt

- Warum Komplexitätsanalyse?
- Technische Aspekte
- Evolution des Linuxkernels
- Weitere Analysemethoden

Warum Codefluss- und Komplexitäts-Messungen ???

Ist die Fehlerdichte (z.B. Fehler pro 1000 Zeilen Code) abhängig von der Komplexität einer Funktion ?

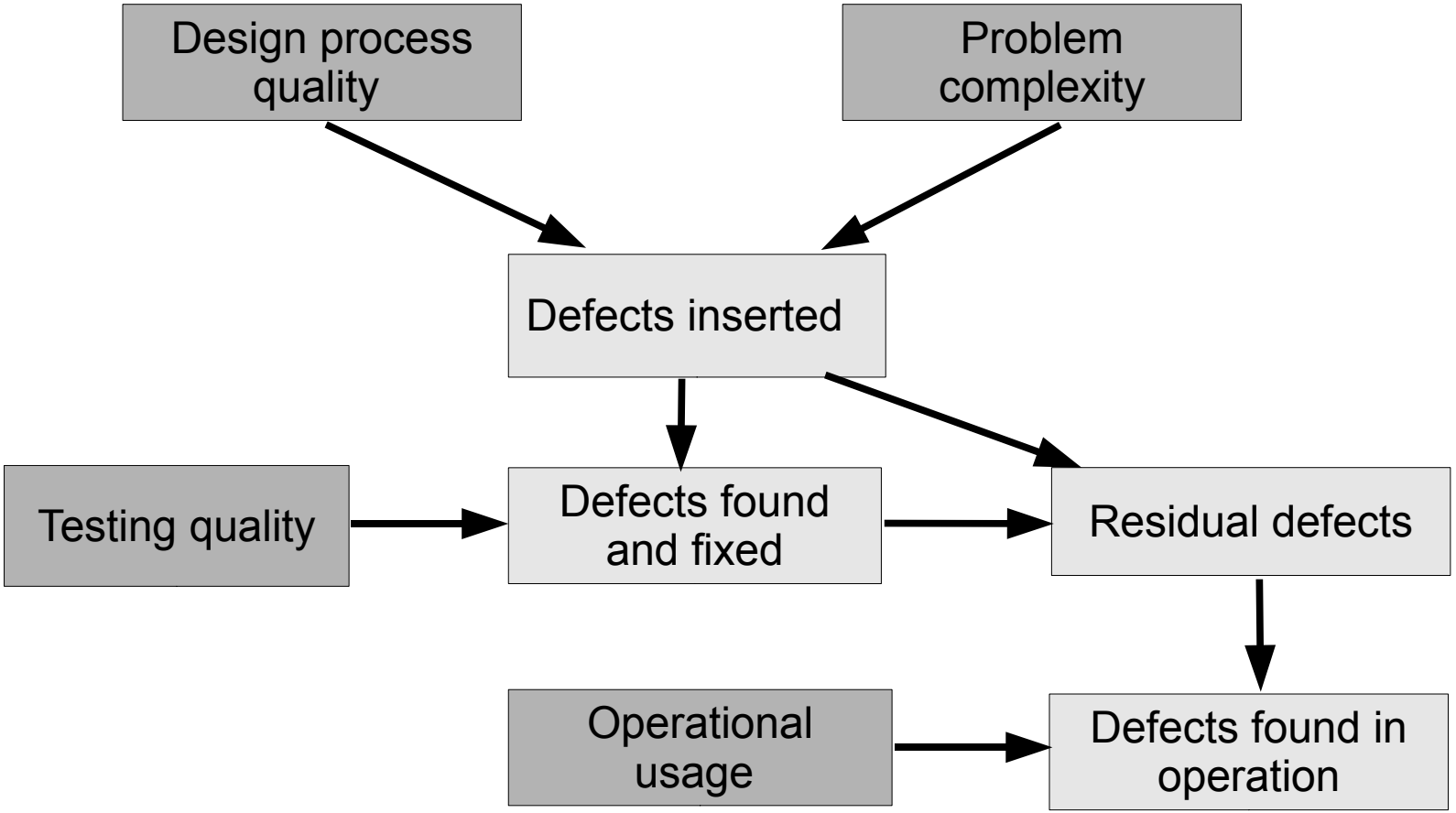
Warum? Komplexer Code ist oft besser getestet, was zu einer niedrigeren Fehlerdichte führt.

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]

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.

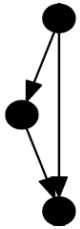
Umso erstaunlicher ist es, dass die zyklomatische Komplexität bis heute als Komplexitätsmetrik verwendet wird.

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

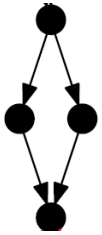
- if-else
- Schleifen-Elemente: for, do-while, while
- case und default

Graphentheoretische Deutung

Jedes Programm setzt sich aus einer Reihe von Grundelementen zusammen:

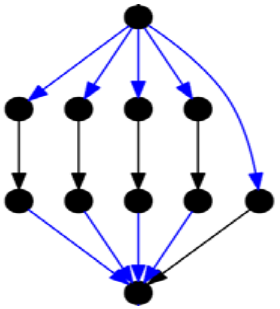


if



if-else

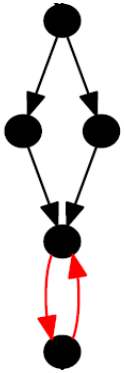
for(;;)
while()
do-while()

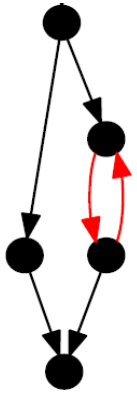


switch –
case/break
default

und den Operationen "sequenceing" und "nesting"

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





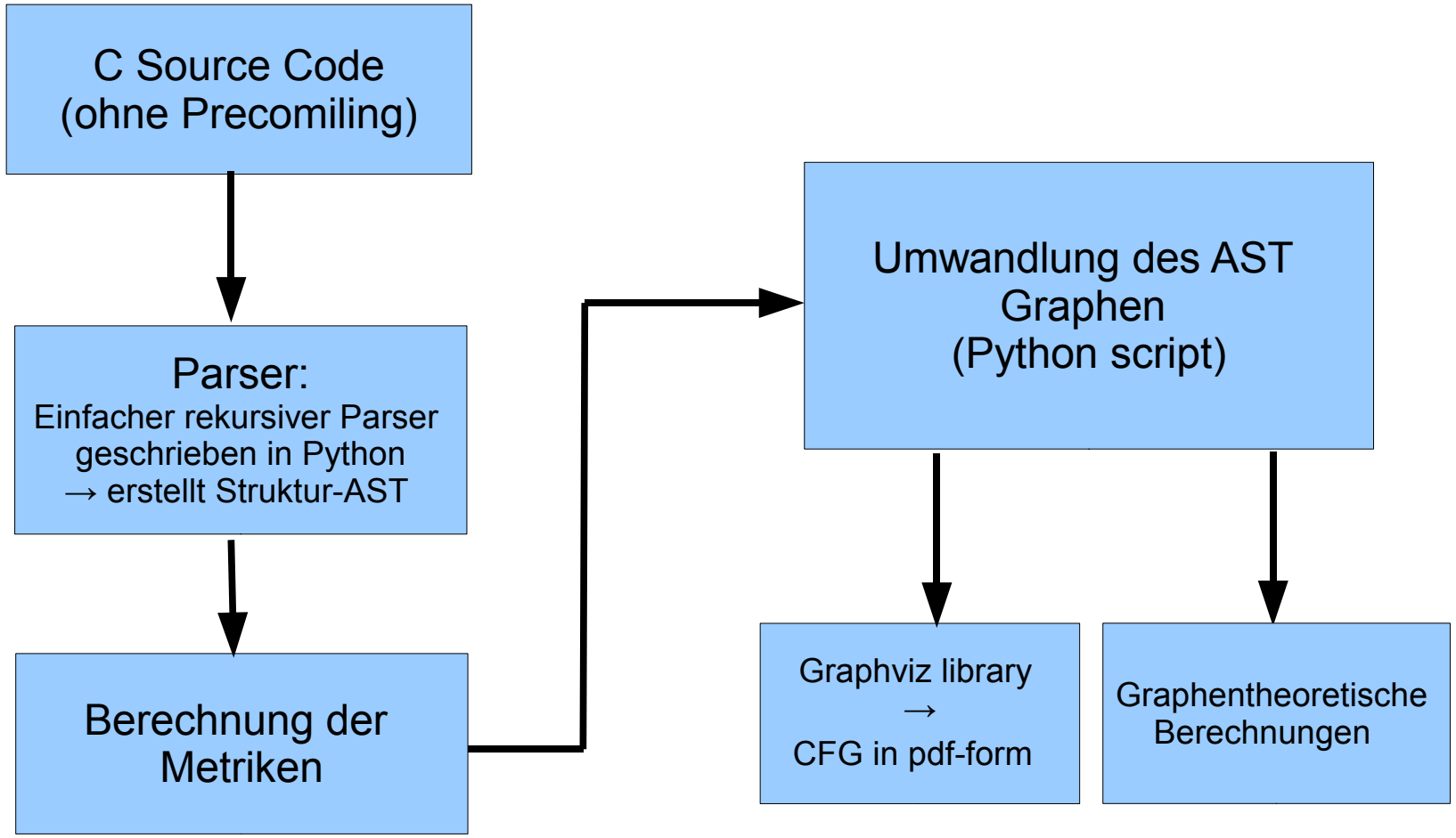
Sehr gute Erläuterung der Methode im Buch "Software Metrics" von Norman Fenton (1997)

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 ?

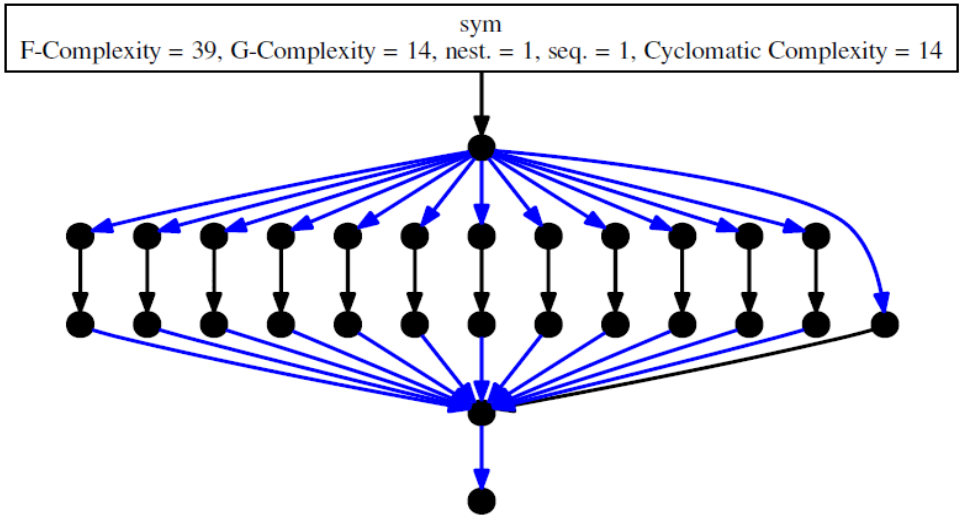


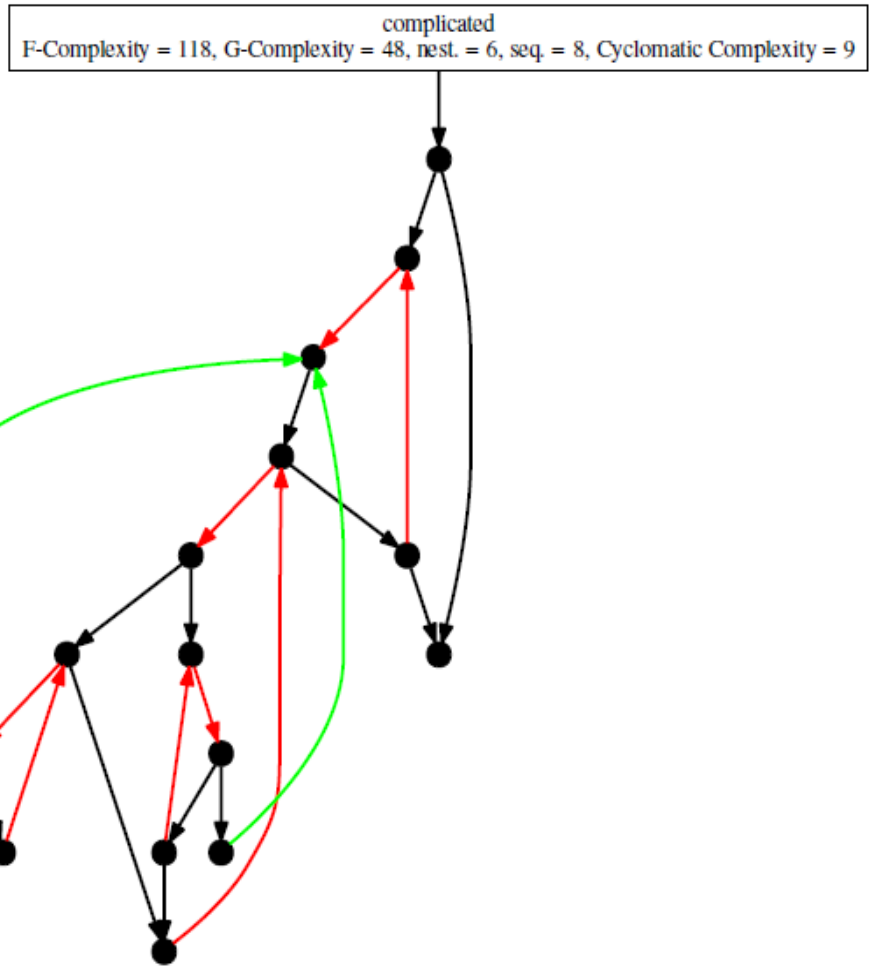
Erstes kleines Beispiel

```
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){
            strpy(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;
            }
          }
       }
     }
  }
}
```

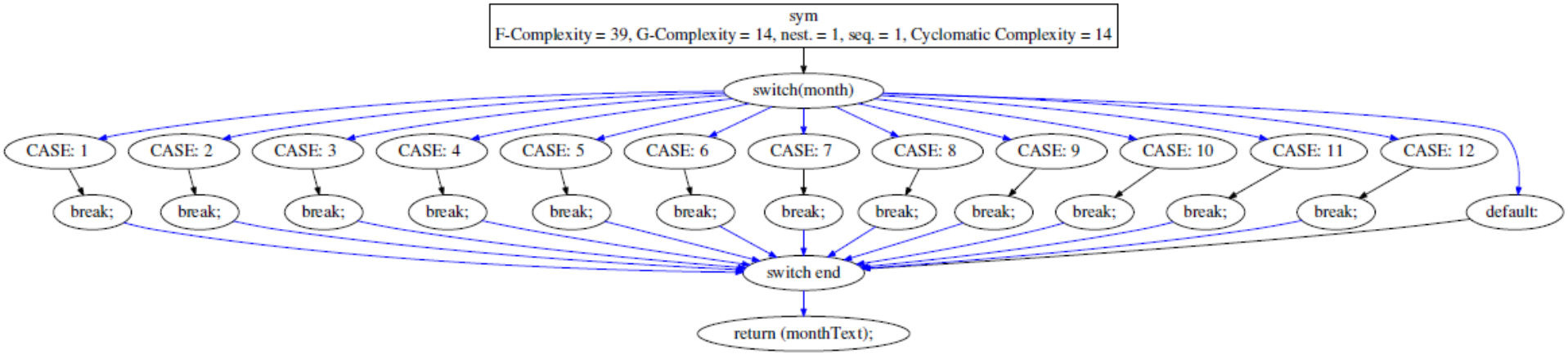
Erstes kleines Beispiel

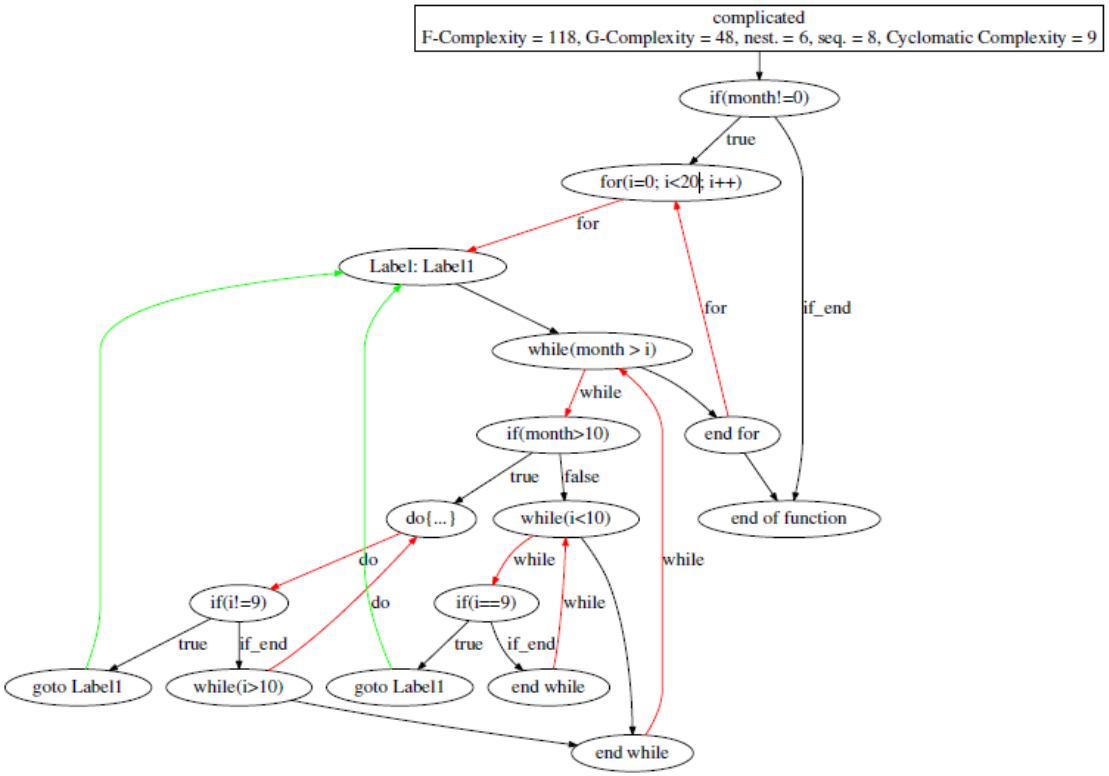




rot: loops (for, while, do-while)
grün: goto
blau: switch and break

Erstes kleines Beispiel





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

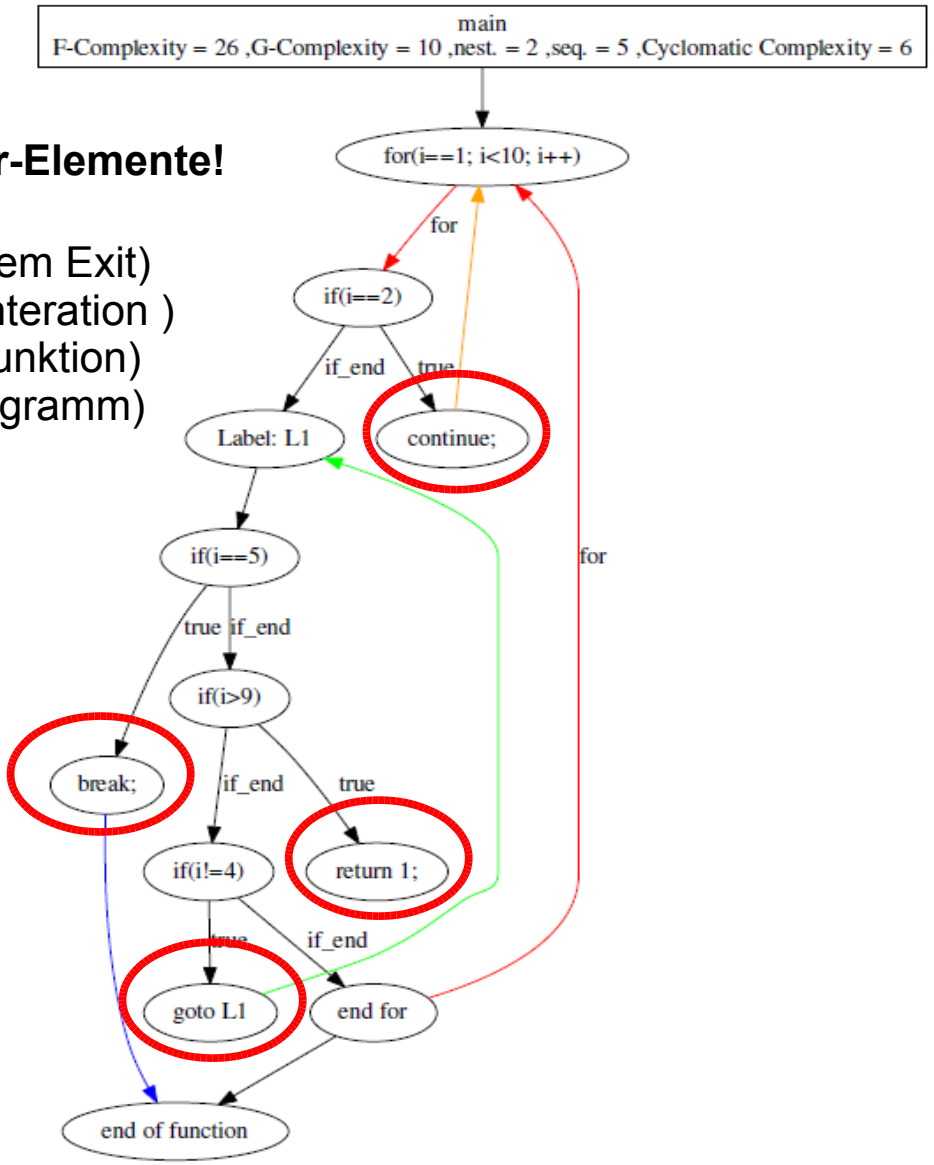
Abgeleitete Grundelemente in C

C besitzt jedoch noch weitere Struktur-Elemente!

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

Diese müssen als zusätzliche Grundelemente angesehen werden!

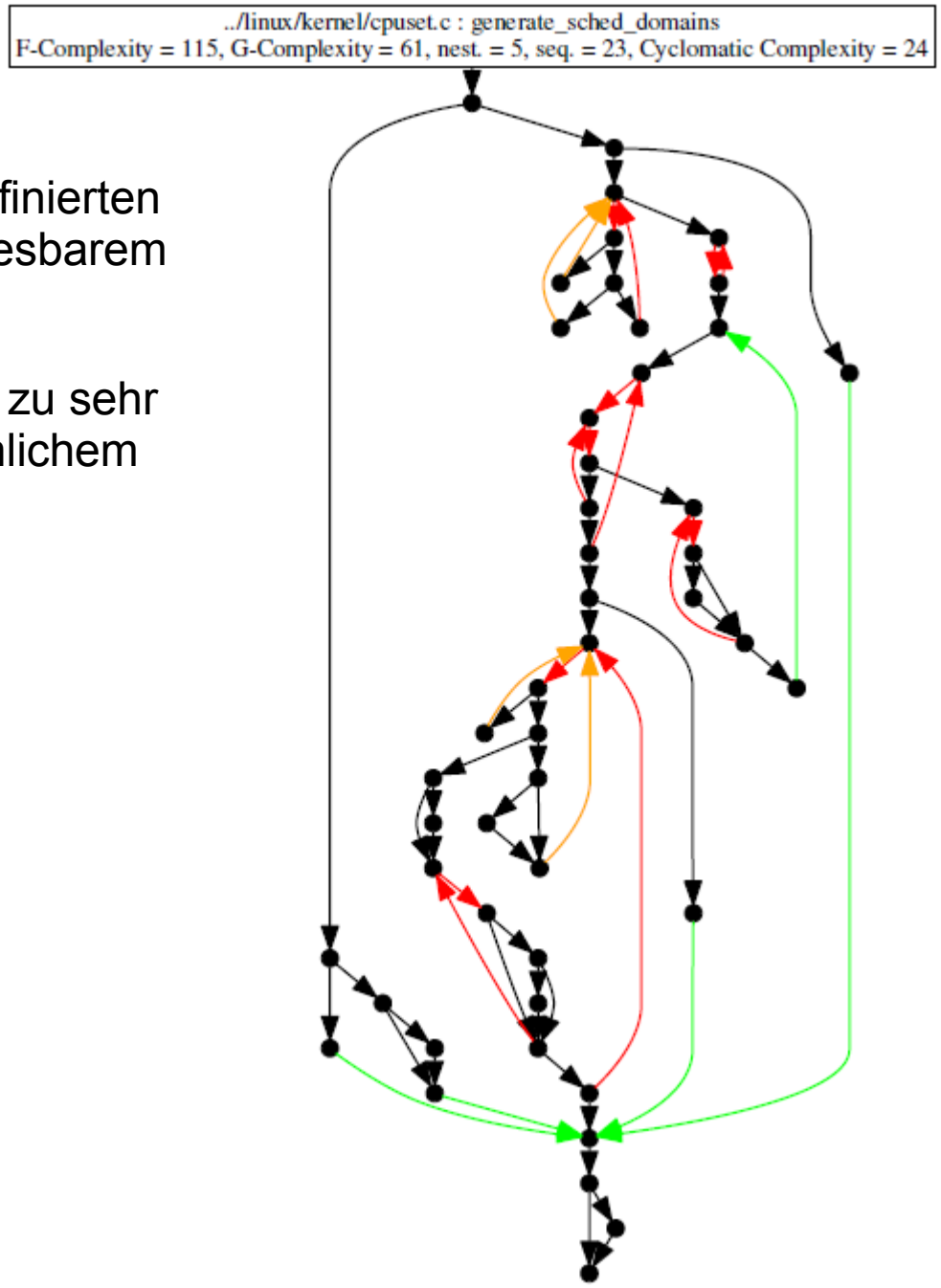
rot: Schleifen (for, while, do-while)
grün: goto
blau: switch and break
orange: continue



Sprünge

- Vorwärtssprünge zu gut definierten Labeln können zu besser lesbarem Code führen

- Rückwärtssprünge können zu sehr komplizieren Schleifen-ähnlichem Strukturen ohne klare Exit-Bedingungen führen !



Zyklomatische-, F- and G- Komplexität

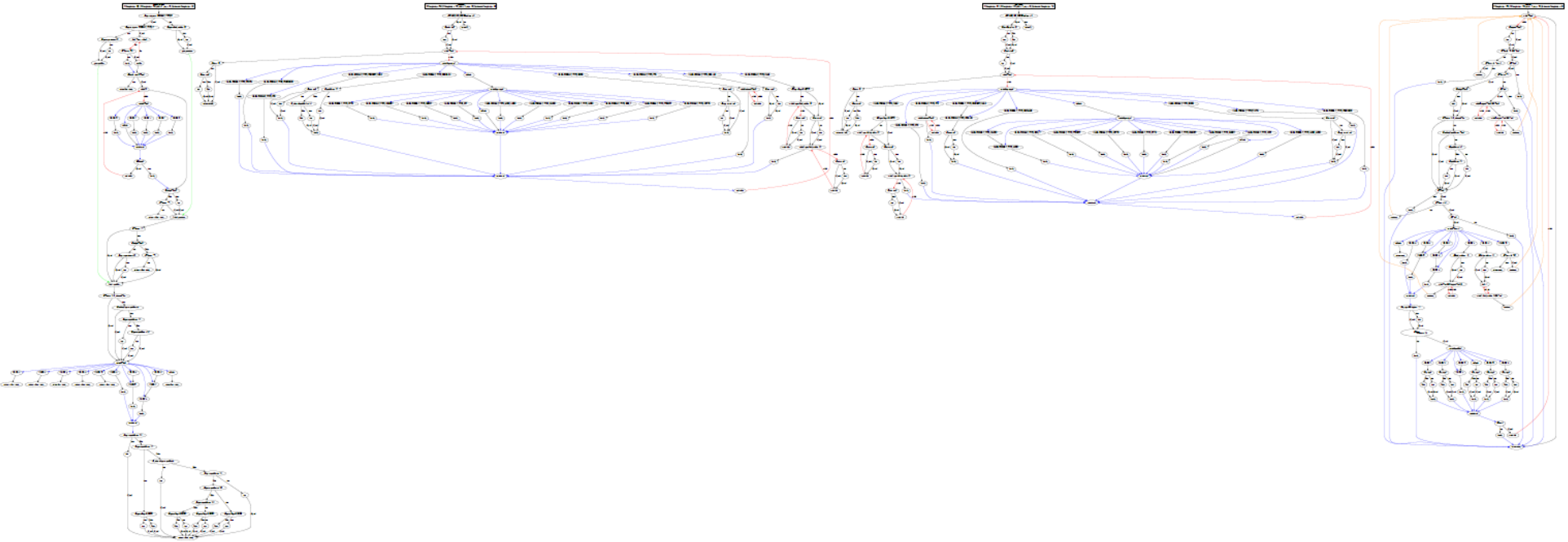
Operation	Zyklomatische Komplexität	F- und G- Komplexität
Sequenz	Summe aller Grundelemente	Summe aller Grundelemente
Nesting	Wird wie eine Sequenz behandelt	abhängig von der Nesting-tiefe: 1, 2, 3, 4, ...

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

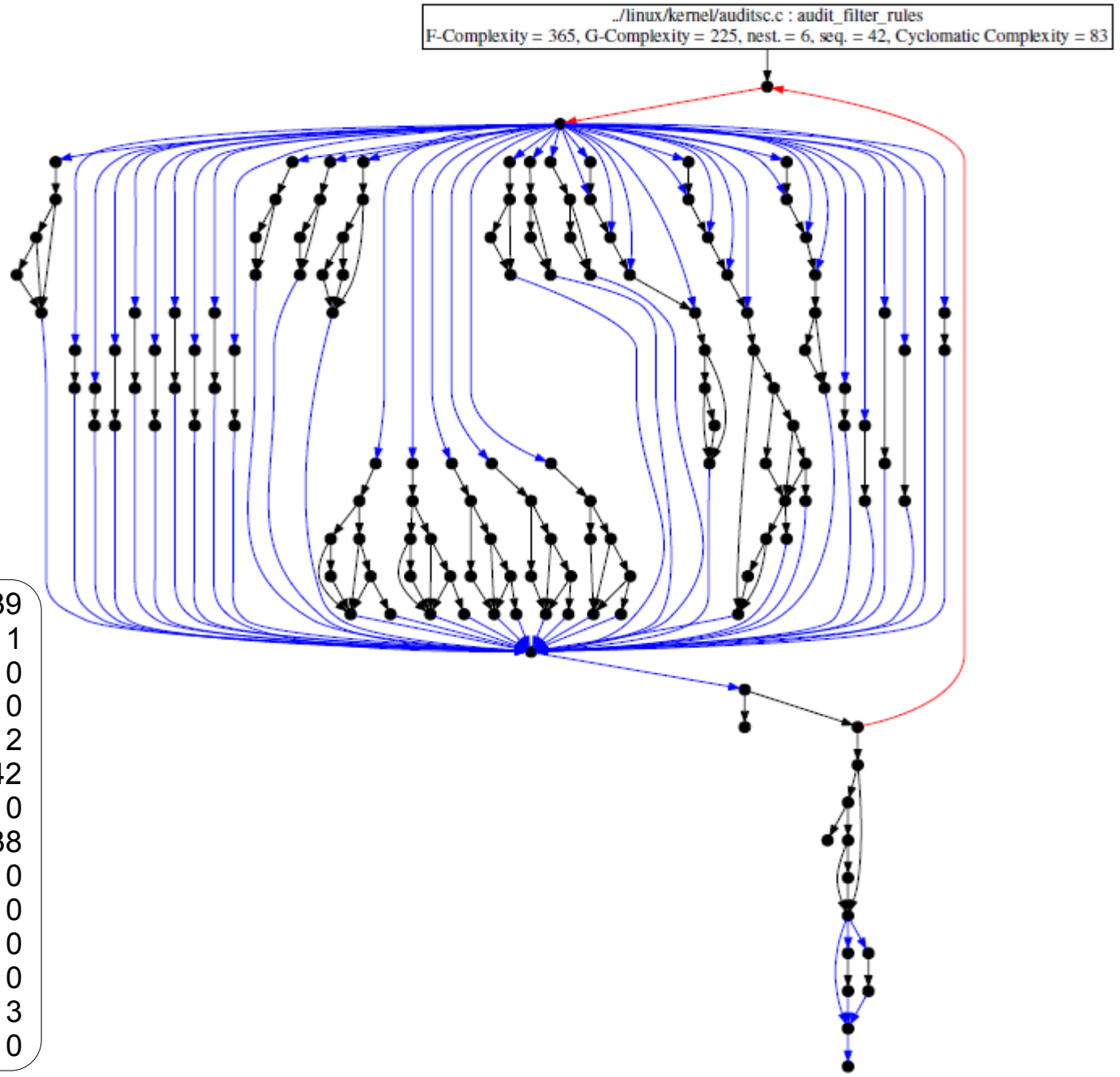
- 8679 Dateien vom Linux ″kernel” 3.7.9; Treiberpfad nicht mitberücksichtigt
- 108887 Funktionen analysiert
- mehr als 7500 Funktionen mit einer F-Komplexität > 30
- 854 Funktionen mit einer F-Komplexität > 100
- F-Komplexitätswerte bis 1250 !

- Riesenspass die Graphen anzuschauen :-)



Linux files ohne ‘drivers’ (Ver. 3.7.9)

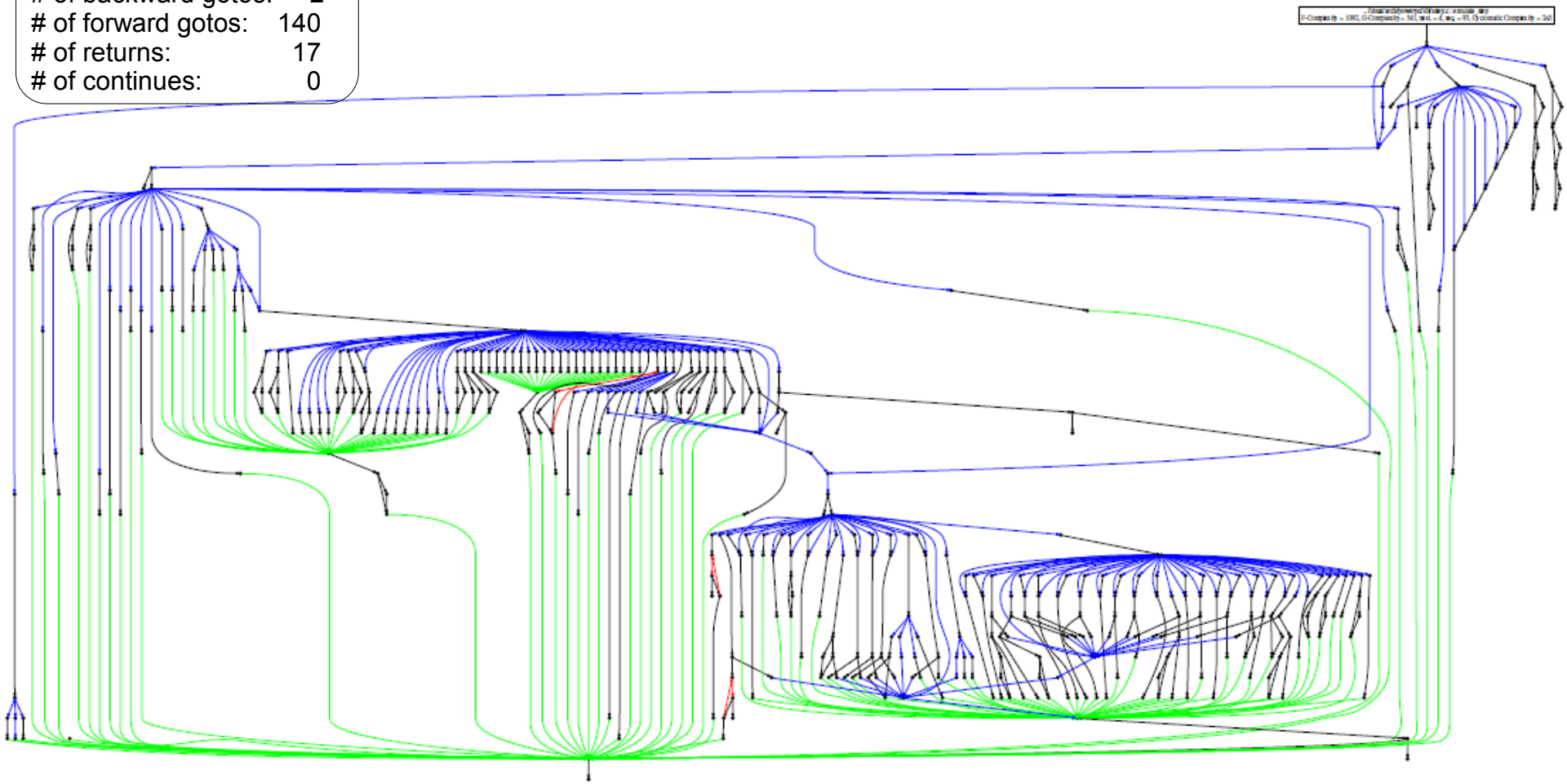
	# Dateien	# Funktionen	F-Com ≥ 30	F-Com ≥ 100	F-Max
kernel	210	6679	306	28 (0.4%)	365
lib	207	1343	104	22 (1.6%)	529
crypto	148	1190	39	1 (0.08%)	464
ipc	13	249	19	3 (1.20%)	120
arch	5103	46059	2019	236 (0.5%)	1092
block	42	897	46	8 (0.89%)	194
fs	1184	22282	2541	287 (1.2%)	1246
init	10	102	11	1 (0.98%)	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



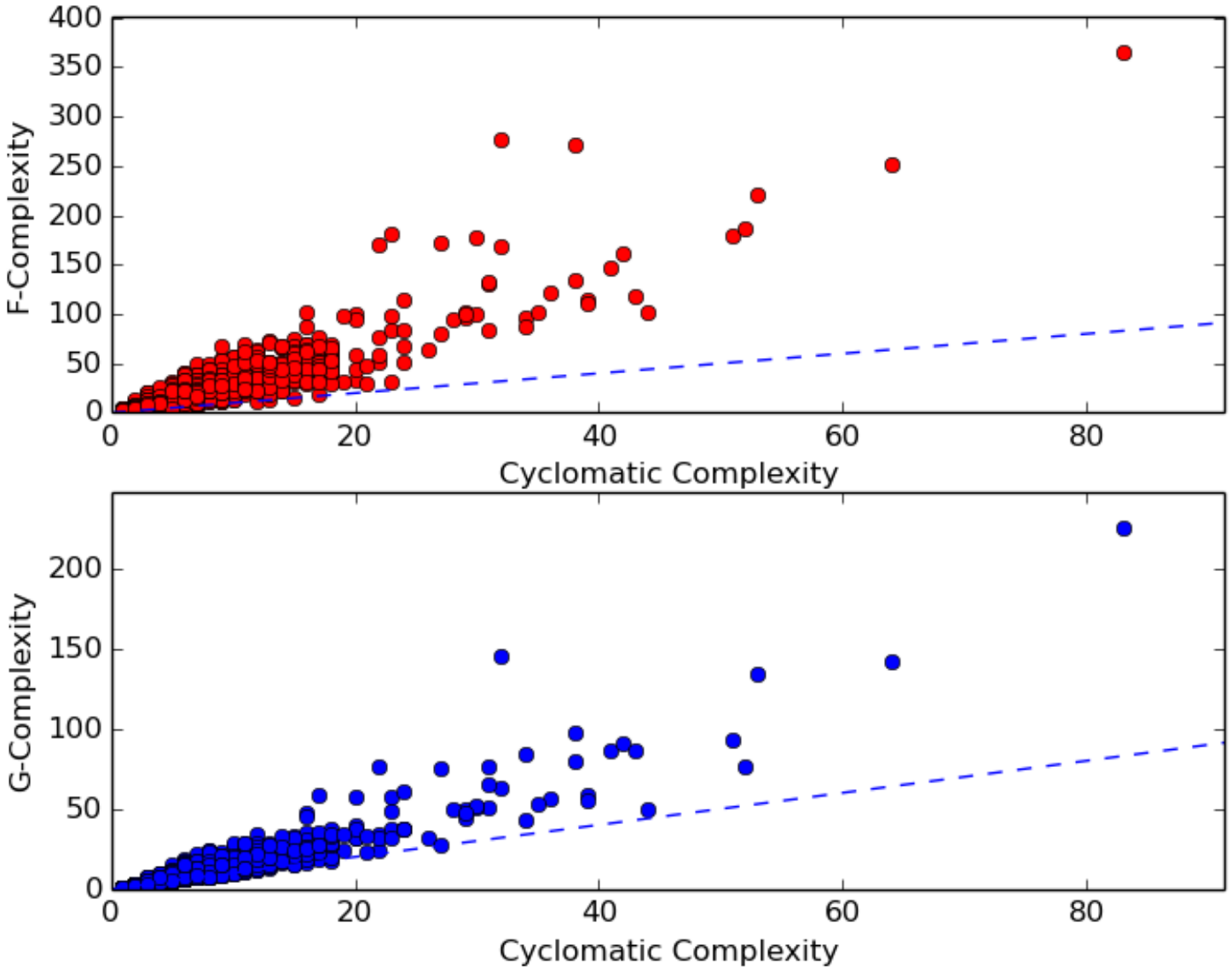
# of ifs:	39
# of fors:	1
# of Do_Whiles:	0
# of Whiles:	0
# of Switches:	2
# of Cases:	42
# of Defaults:	0
# of breaks in cases:	38
# of breaks in loops:	0
# of Labels:	0
# of backward gotos:	0
# of forward gotos:	0
# of returns:	3
# of continues:	0

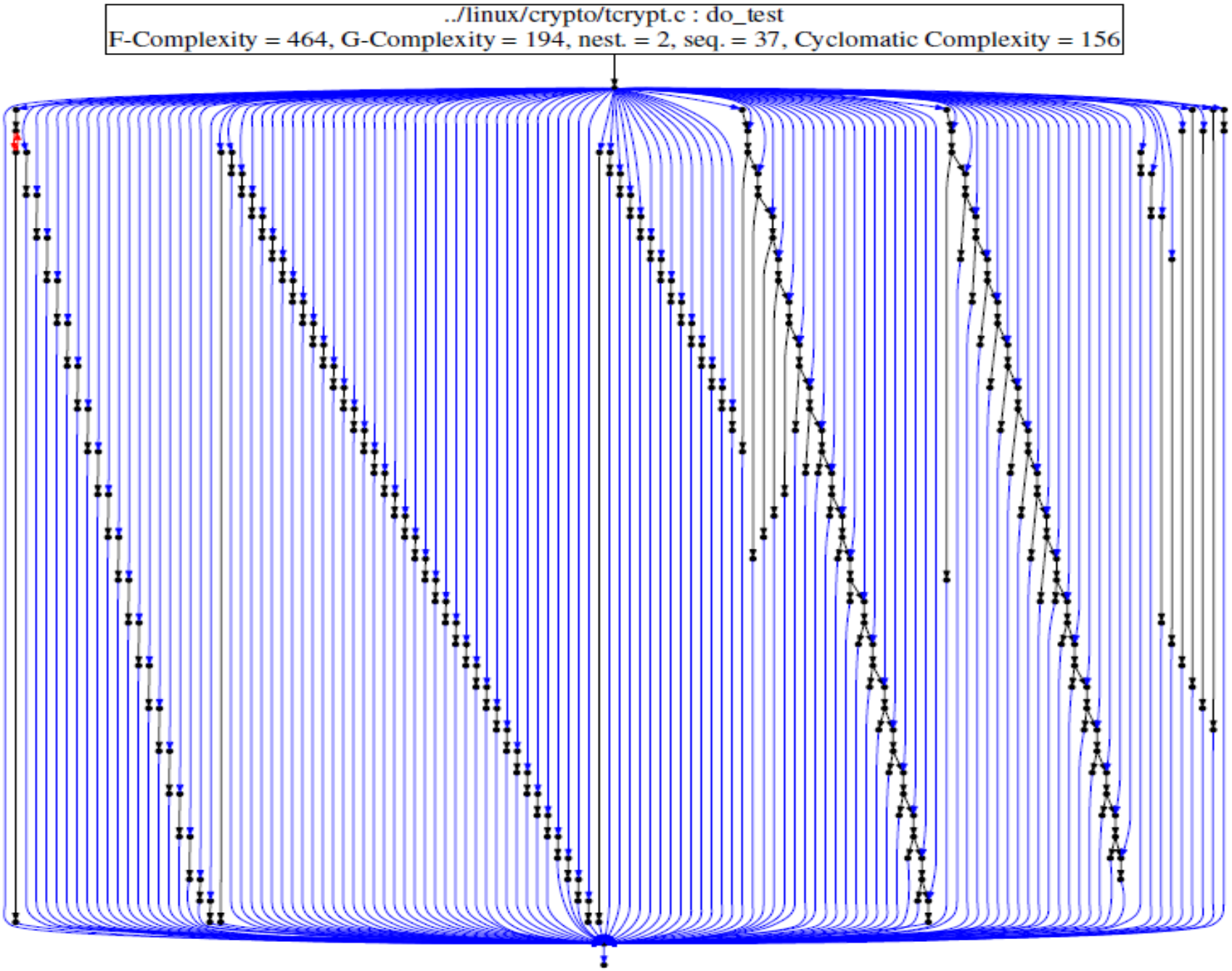
# of ifs:	76
# of fors:	1
# of Do_Whiles:	2
# of Whiles:	0
# of Switches:	14
# of Cases:	185
# of Defaults:	0
# of breaks in cases:	27
# of breaks in loops:	2
# of Labels:	4
# of backward gotos:	2
# of forward gotos:	140
# of returns:	17
# of continues:	0

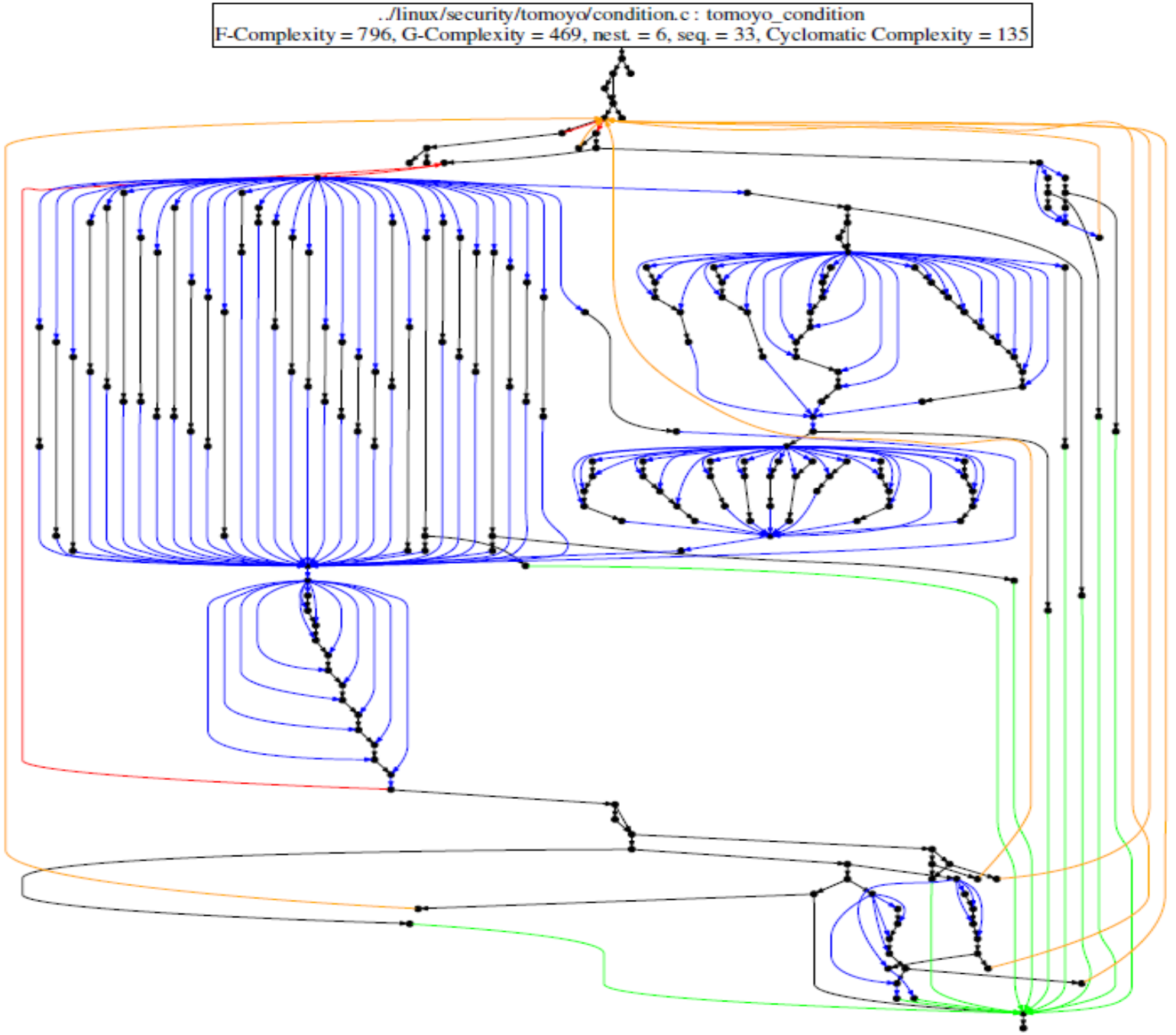
../linux/arch/powerpc/lib/sstep.c : emulate_step
F-Complexity = 1092, G-Complexity = 541, nest. = 4, seq. = 93, Cyclomatic Complexity = 265

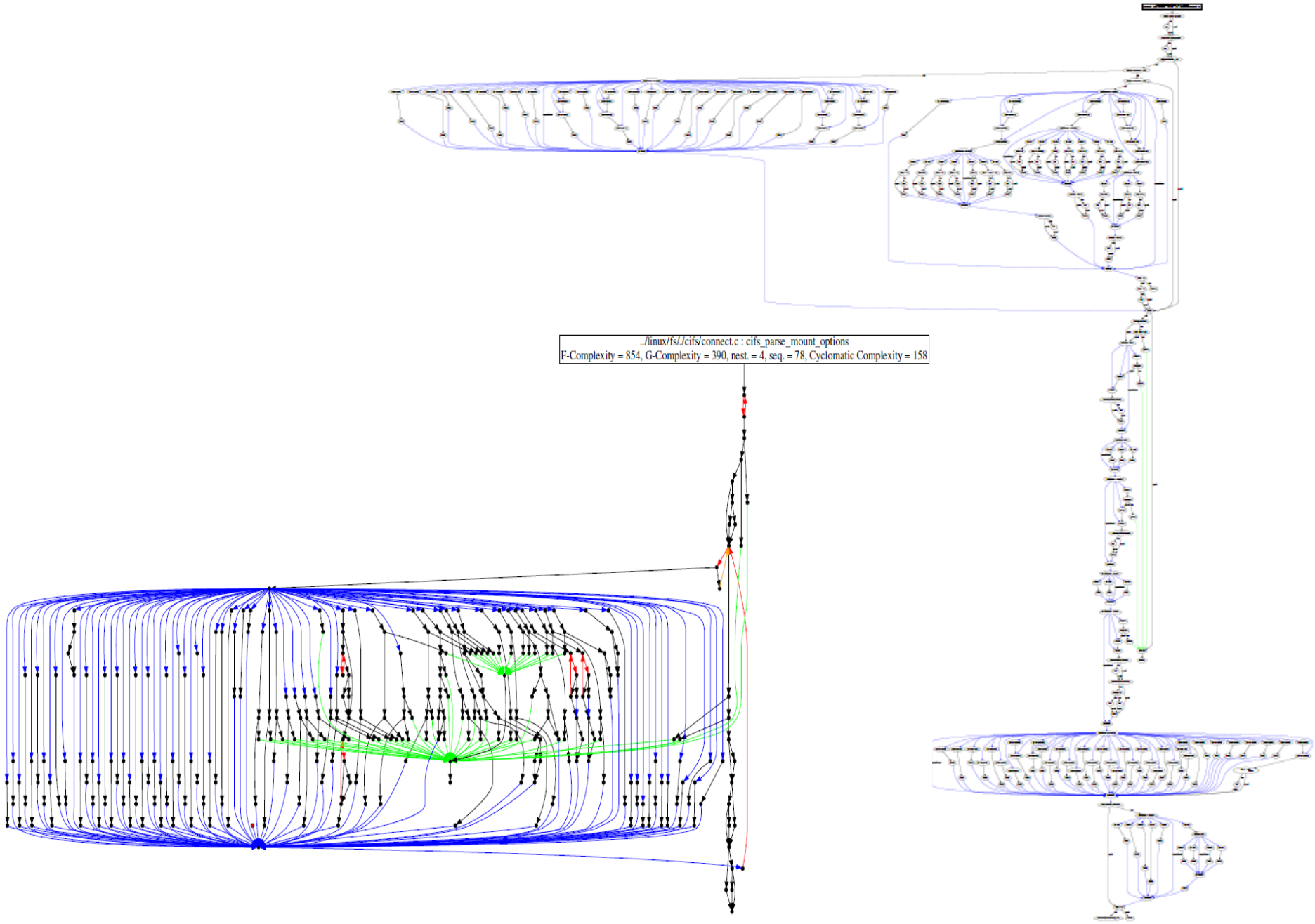


kernel files



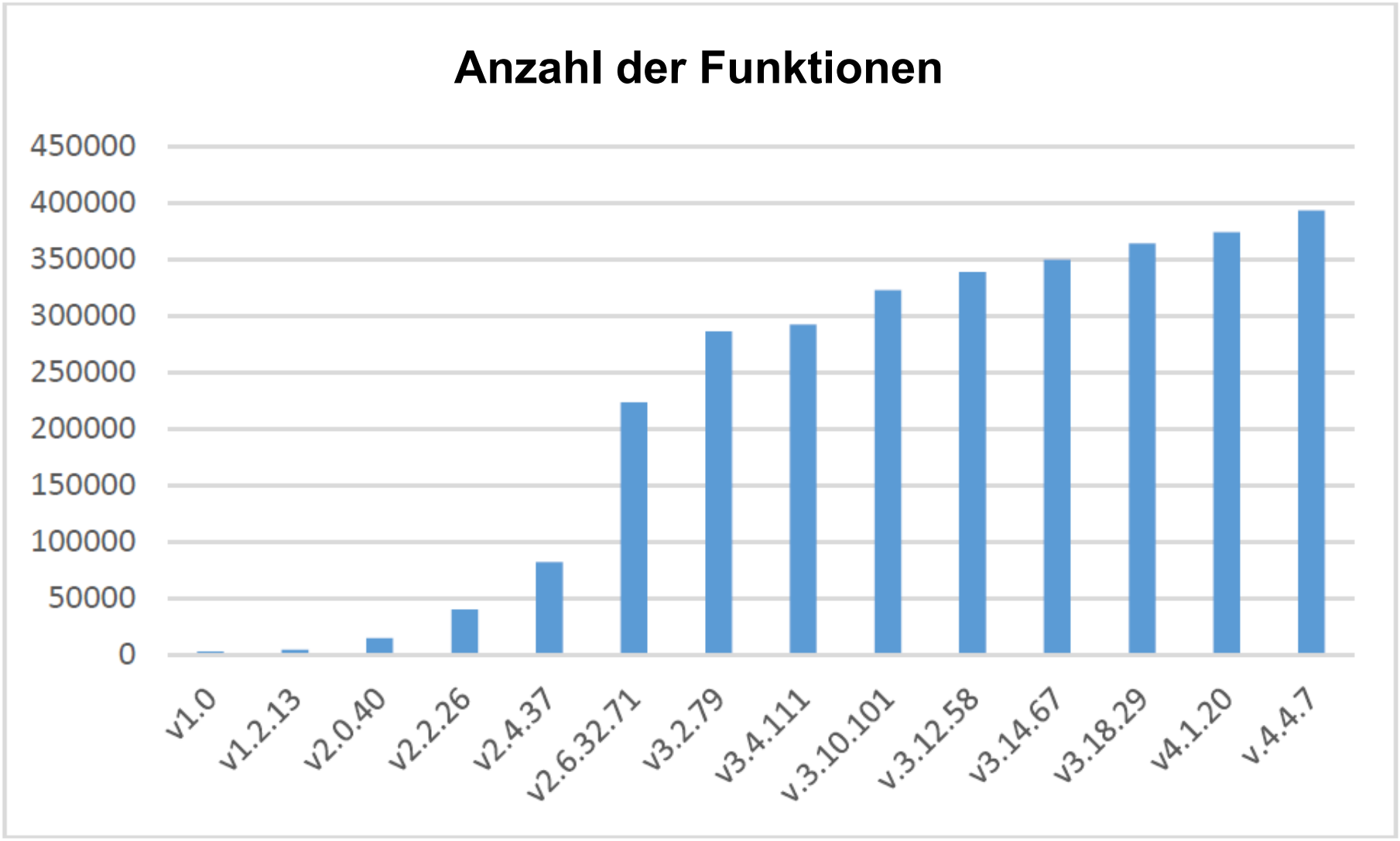

../linux/crypto/tcrypt.c : do_test
F-Complexity = 464, G-Complexity = 194, nest. = 2, seq. = 37, Cyclomatic Complexity = 156


./linux/security/tomoyo/condition.c : tomoyo_condition
F-Complexity = 796, G-Complexity = 469, nest. = 6, seq. = 33, Cyclomatic Complexity = 135

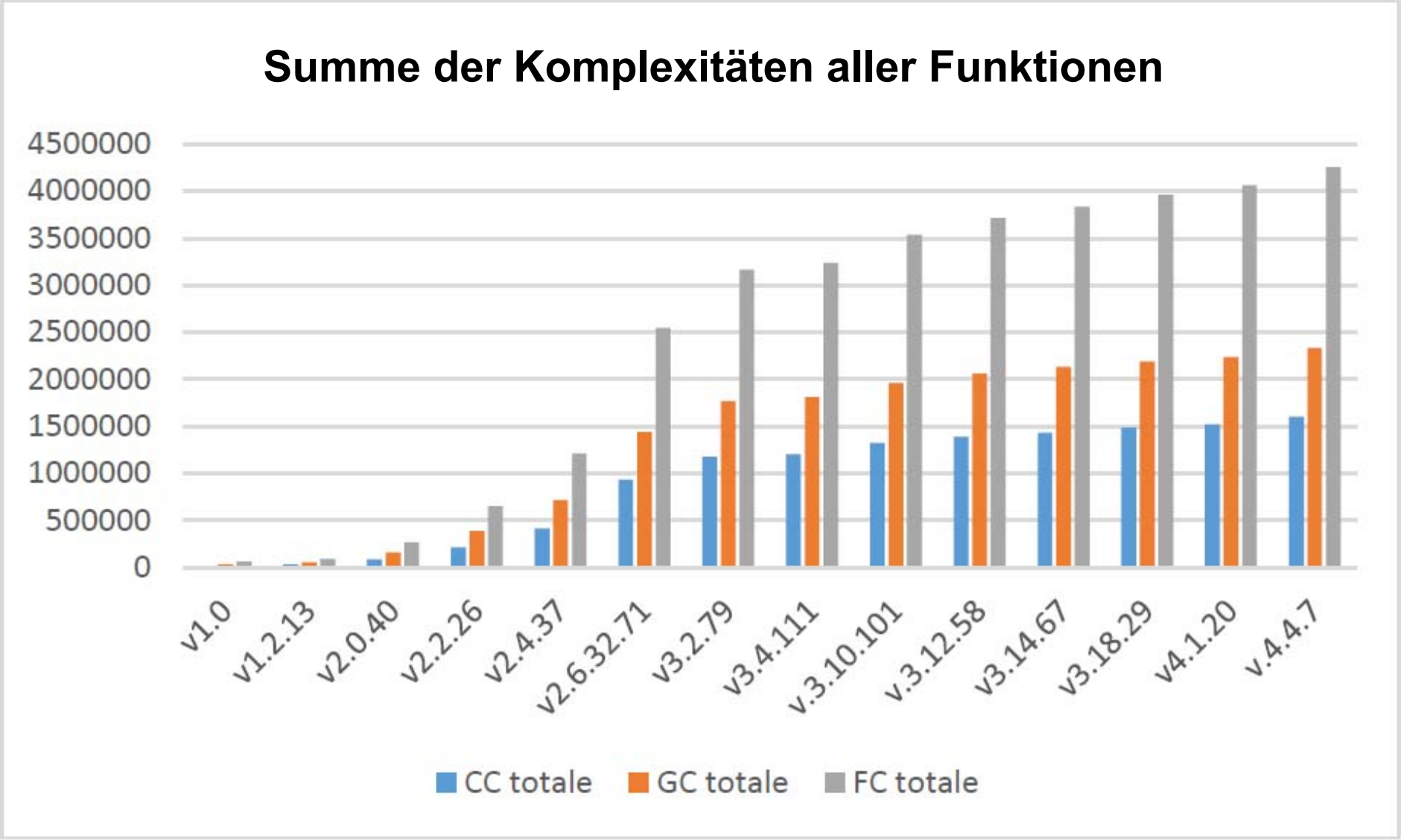

../linux/fs/./cifs/connect.c : cifs_parse_mount_options
F-Complexity = 854, G-Complexity = 390, nest. = 4, seq. = 78, Cyclomatic Complexity = 158

Evolution des Linuxkernels

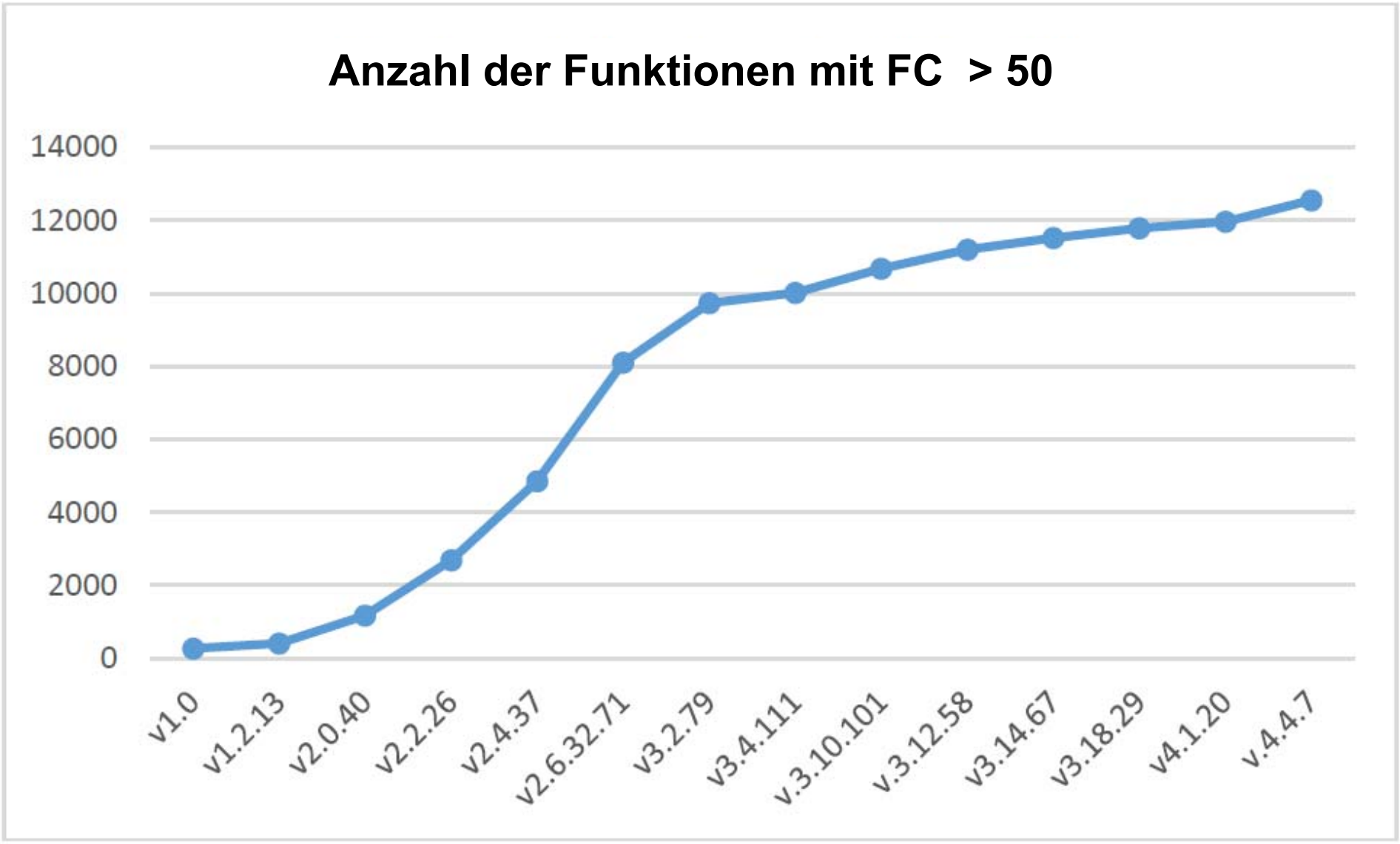
(alle Versionen von www.kernel.org)



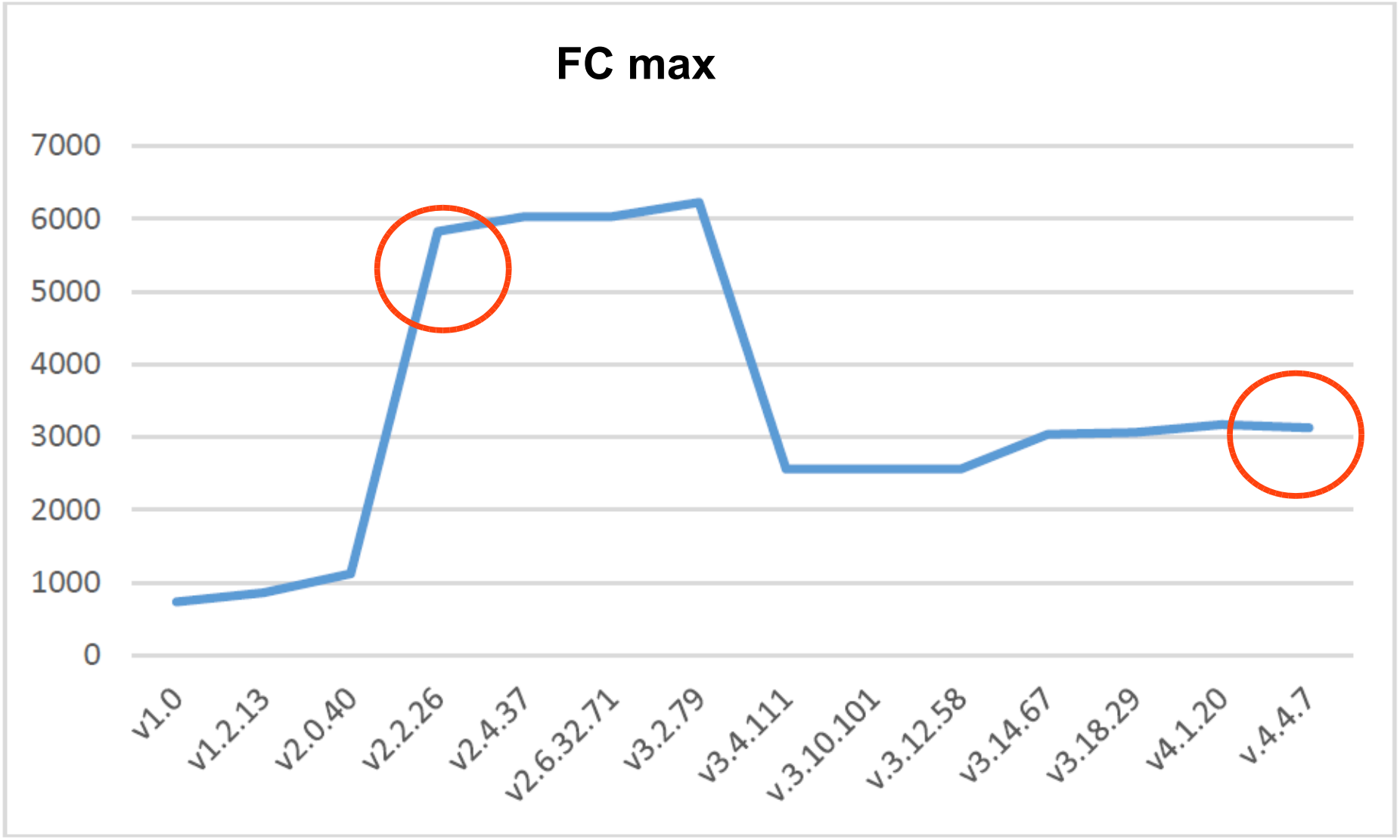
Evolution des Linuxkernels



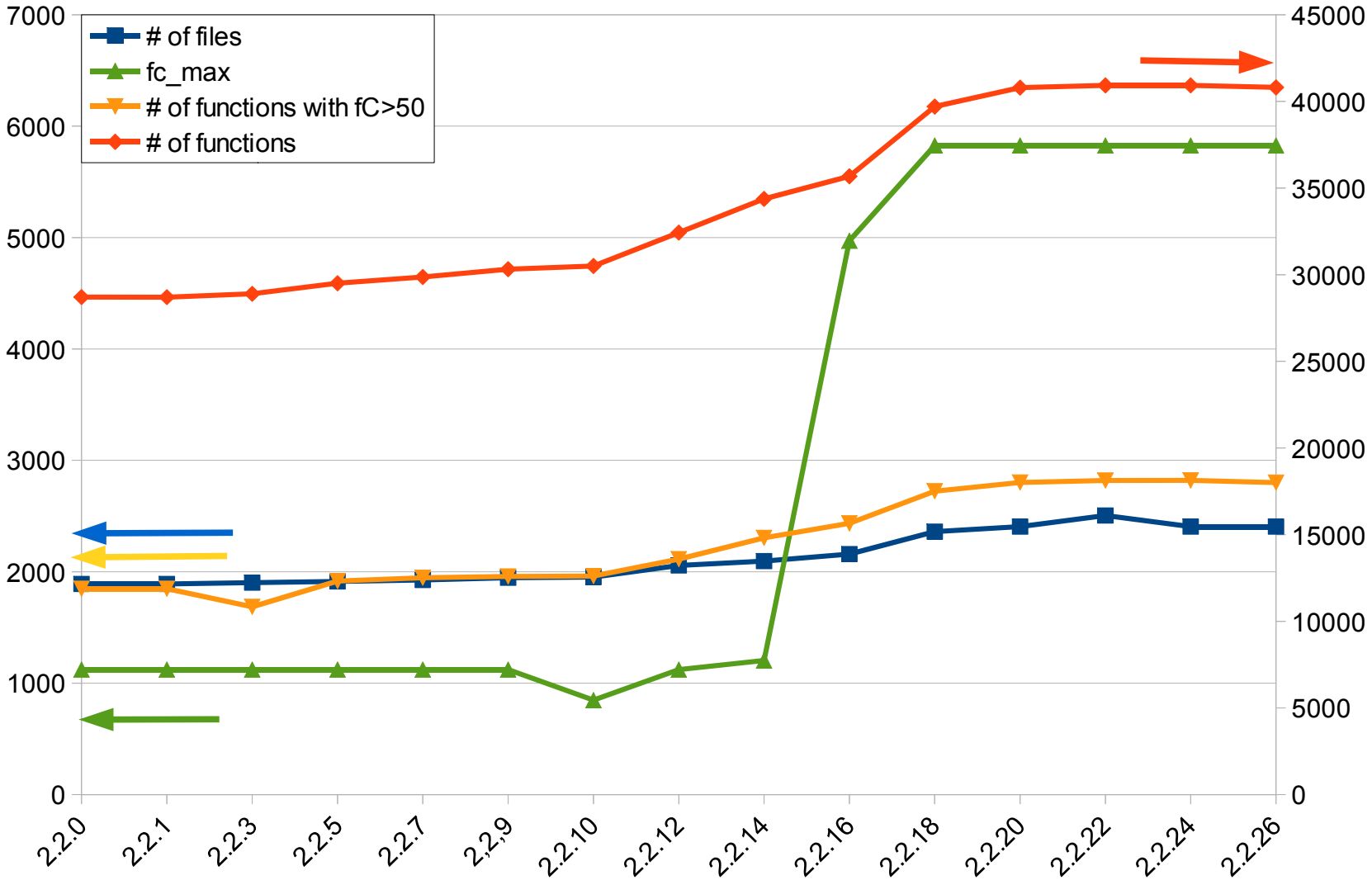
Evolution des Linuxkernels



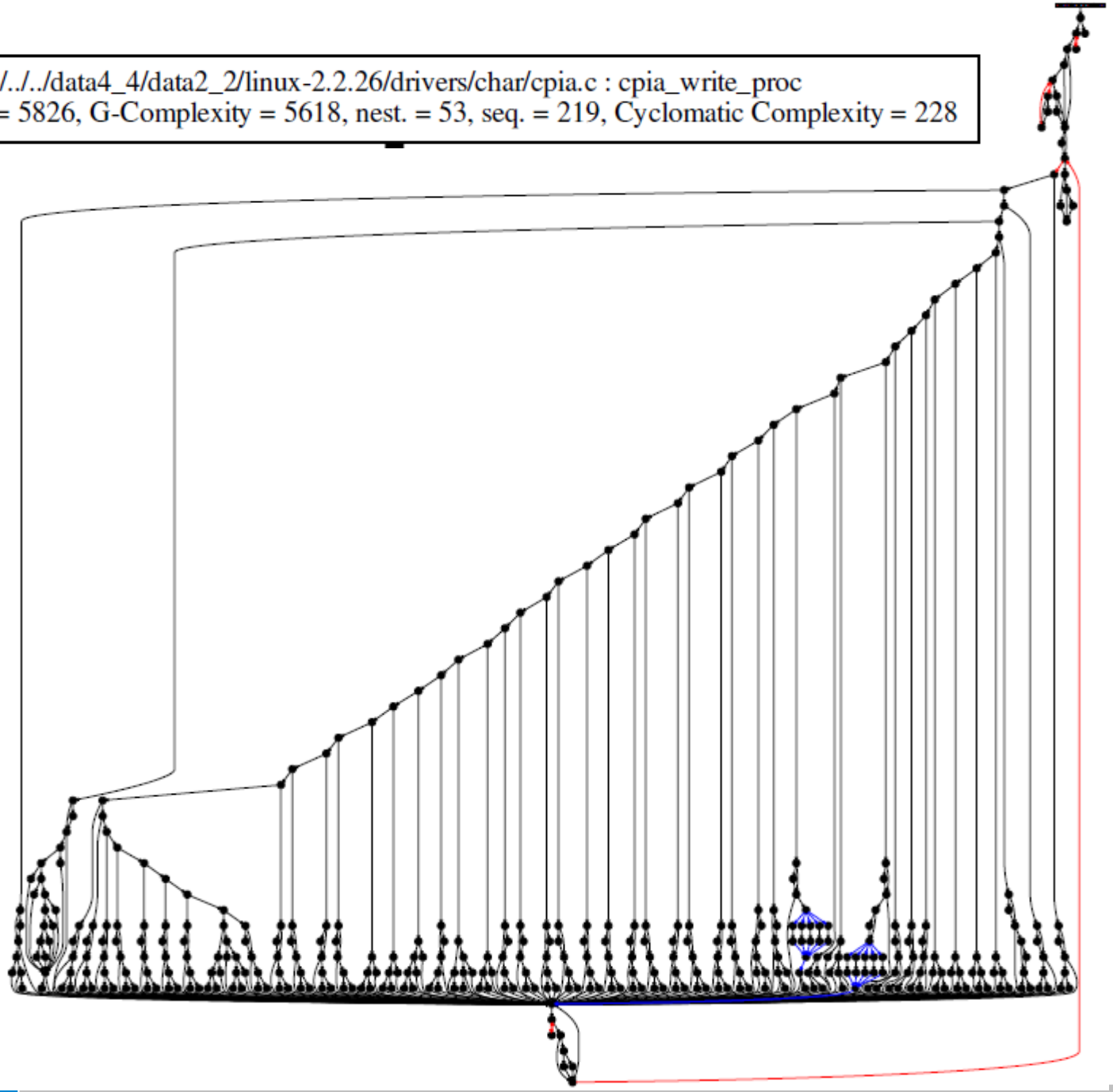
Evolution des Linuxkernels



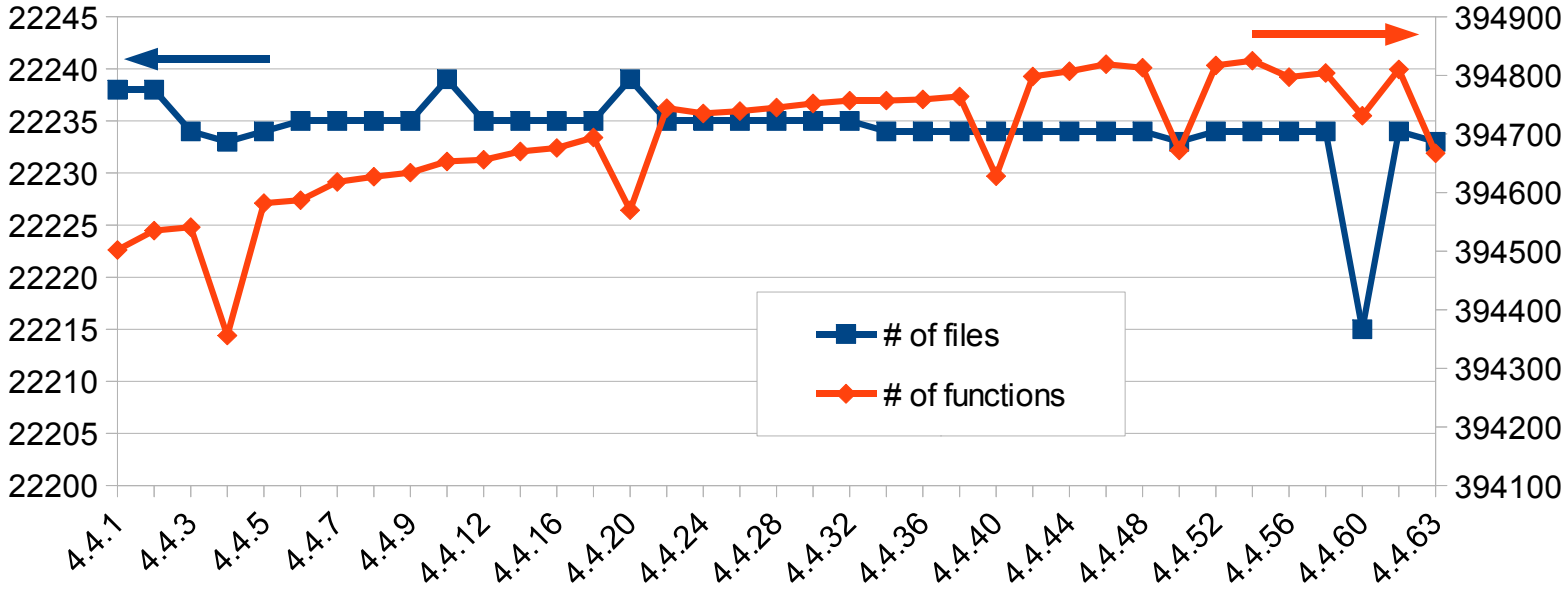
Kernel 2.2.x

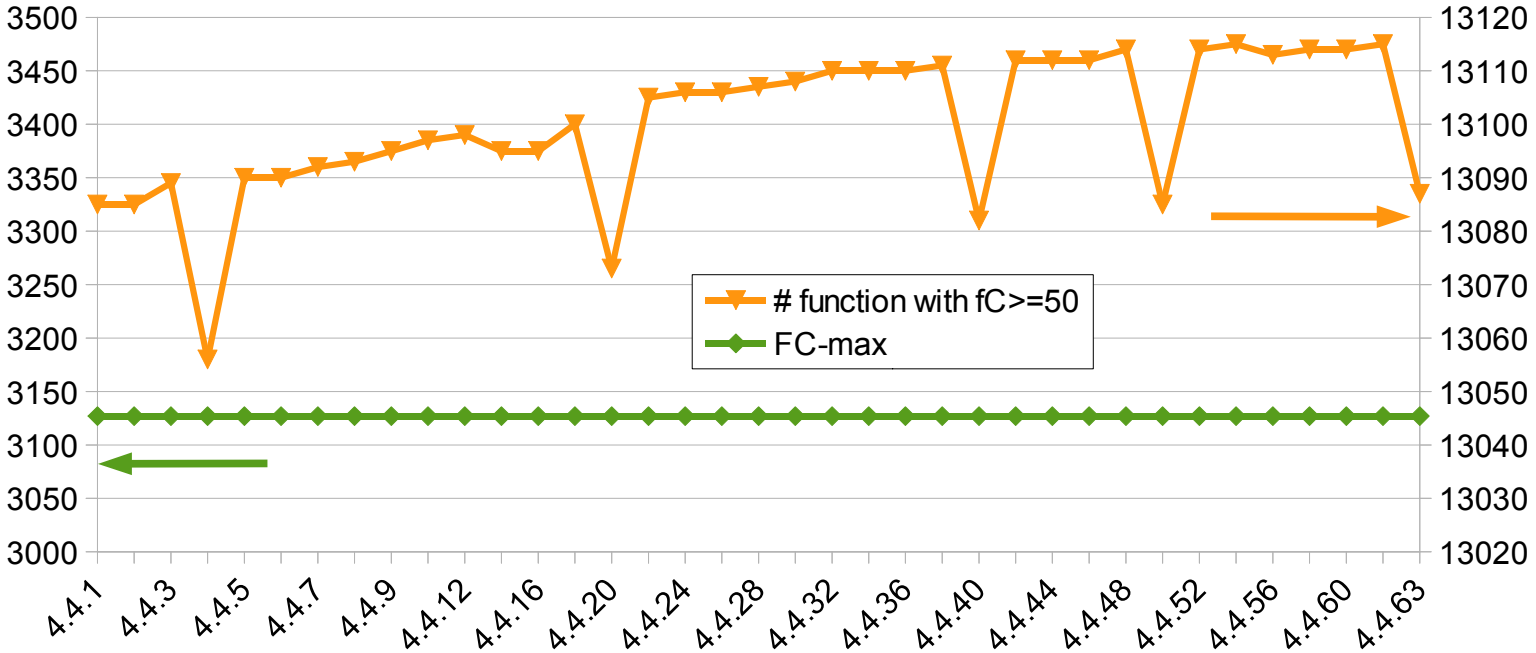

# of files
fc_max
# of functions with fC>50
# of functions
7000
6000
5000
4000
3000
2000
1000
0
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
2.2.0
2.2.1
2.2.3
2.2.5
2.2.7
2.2.9
2.2.10
2.2.12
2.2.14
2.2.16
2.2.18
2.2.20
2.2.22
2.2.24
2.2.26

../../../data4_4/data2_2/linux-2.2.26/drivers/char/cpia.c : cpia_write_proc
F-Complexity = 5826, G-Complexity = 5618, nest. = 53, seq. = 219, Cyclomatic Complexity = 228



Kernel 4_4

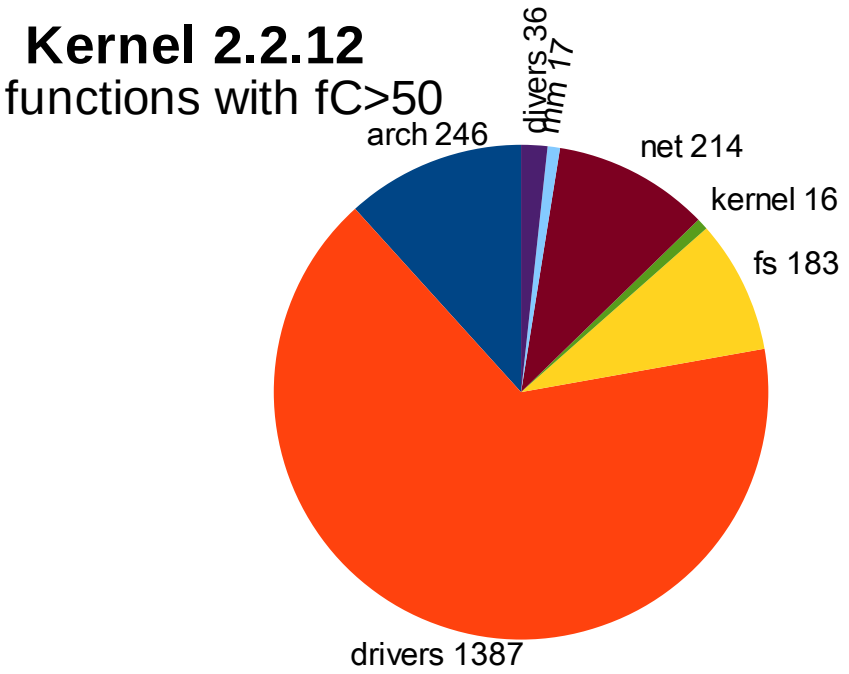

22245
22240
22235
22230
22225
22220
22215
22210
22205
22200
394900
394800
394700
394600
394500
394400
394300
394200
394100
# of files
# of functions
4.4.1
4.4.3
4.4.5
4.4.7
4.4.9
4.4.12
4.4.16
4.4.20
4.4.24
4.4.28
4.4.32
4.4.36
4.4.40
4.4.44
4.4.48
4.4.52
4.4.56
4.4.60
4.4.63

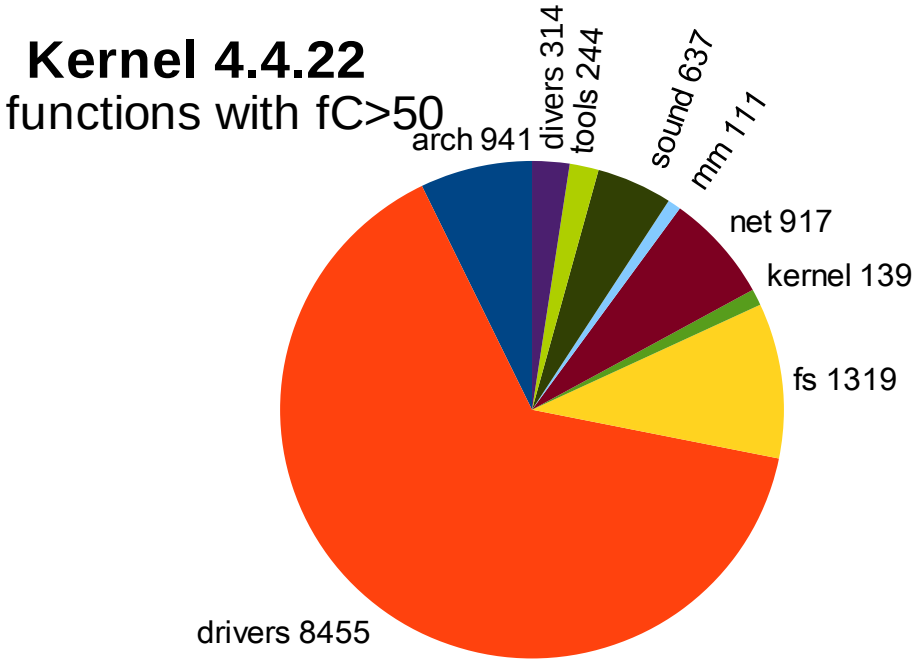

3500
3450
3400
3350
3300
3250
3200
3150
3100
3050
3000
13120
13110
13100
13090
13080
13070
13060
13050
13040
13030
13020
function with fC>=50
FC-max
4.4.1
4.4.3
4.4.5
4.4.7
4.4.9
4.4.12
4.4.16
4.4.20
4.4.24
4.4.28
4.4.32
4.4.36
4.4.40
4.4.44
4.4.48
4.4.52
4.4.56
4.4.60
4.4.63

Vergleich zweier Kernel:

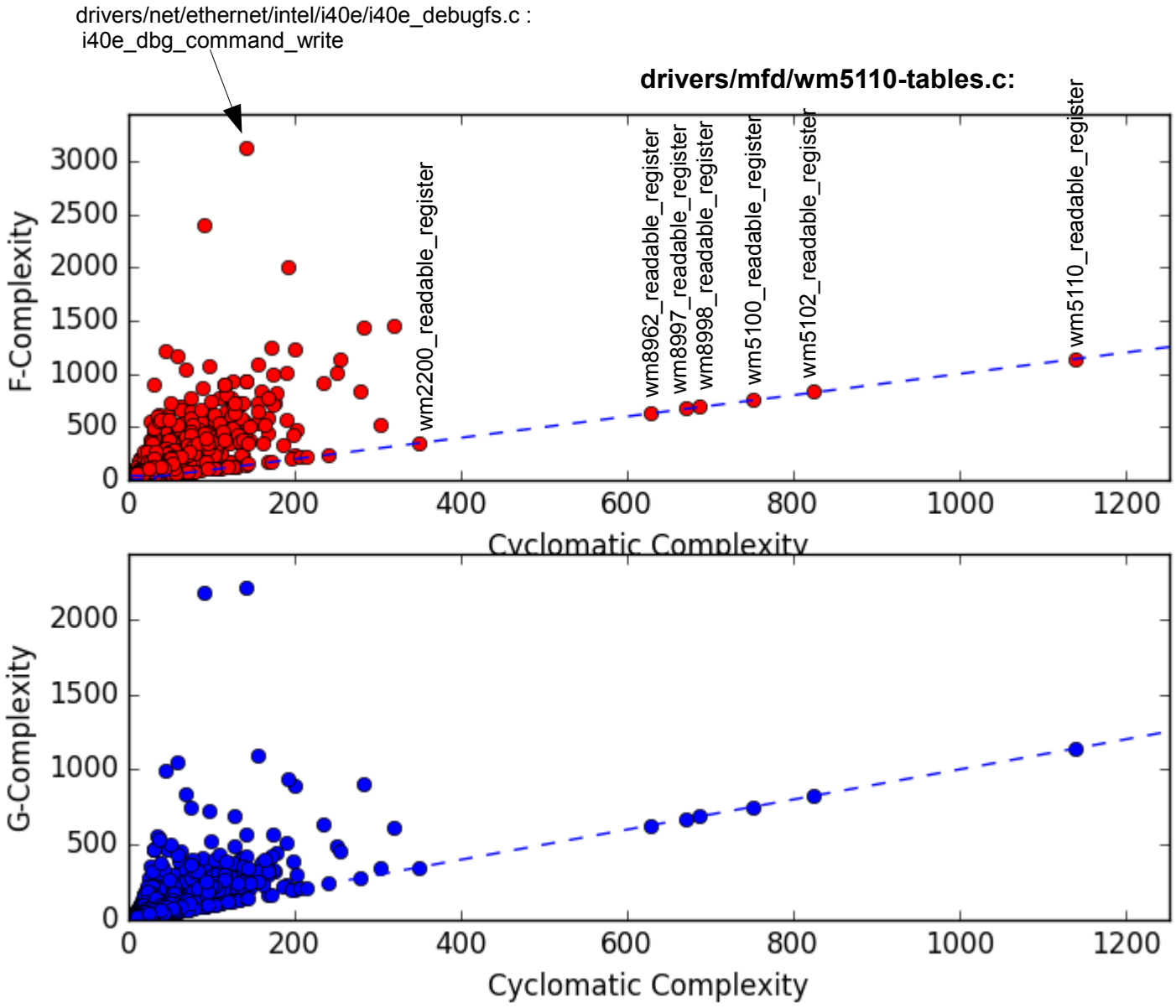
2.2.12: 2099 Funktionen mit fC>50 von insgesamt 32427 (6.5 %)

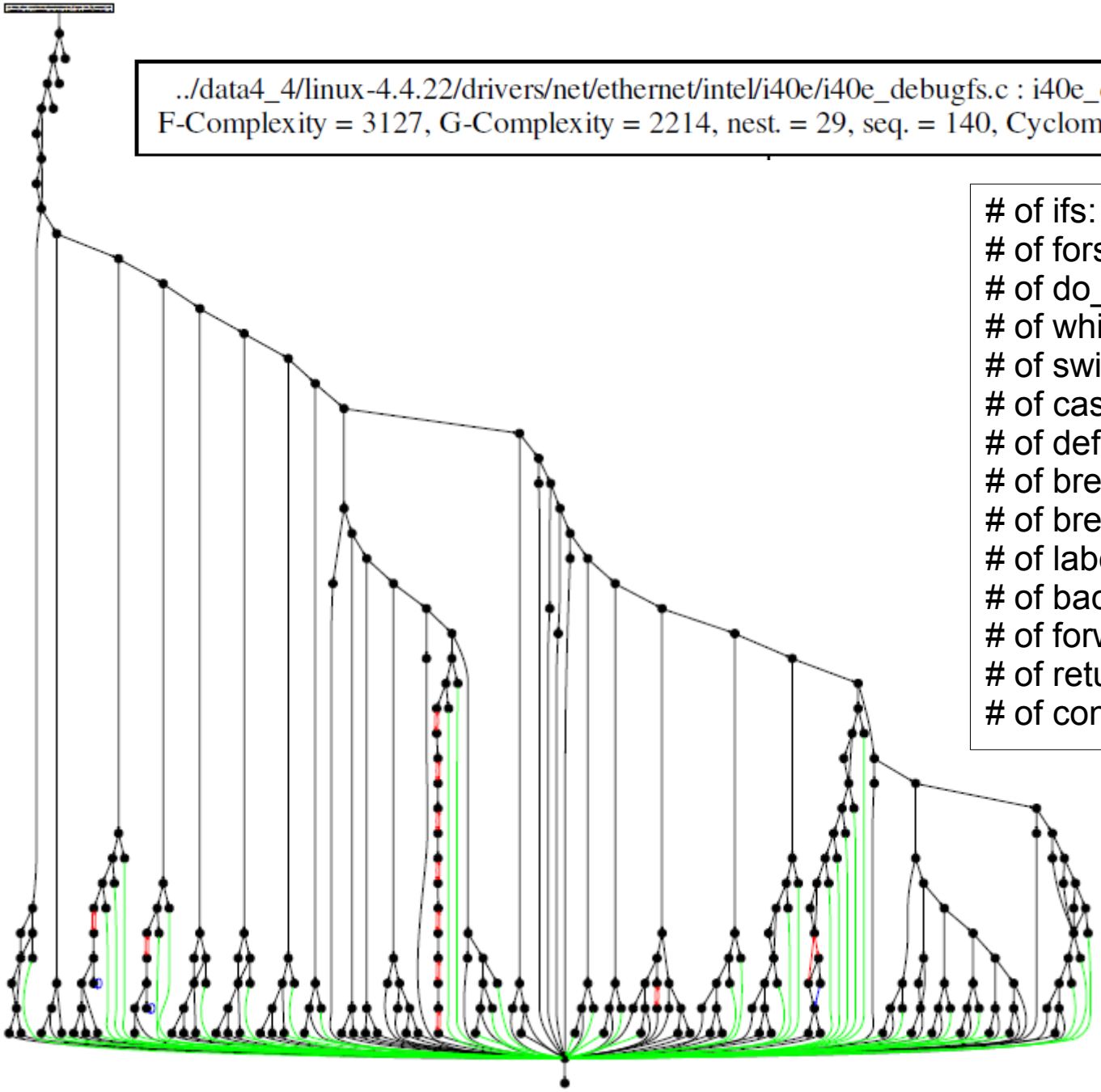
4.4.22: 13077 Funktionen mit fC>50 von insgesamt 394722 (3.3%)





Kernel 4.4.22


drivers/net/ethernet/intel/i40e/i40e_debugfs.c :
i40e_dbg_command_write
drivers/mfd/wm5110-tables.c:
wm2200_readable_register
wm8962_readable_register
wm8997_readable_register
wm8998_readable_register
wm5100_readable_register
wm5102_readable_register
wm5110_readable_register
F-Complexity
Cyclomatic Complexity
G-Complexity
Cyclomatic Complexity



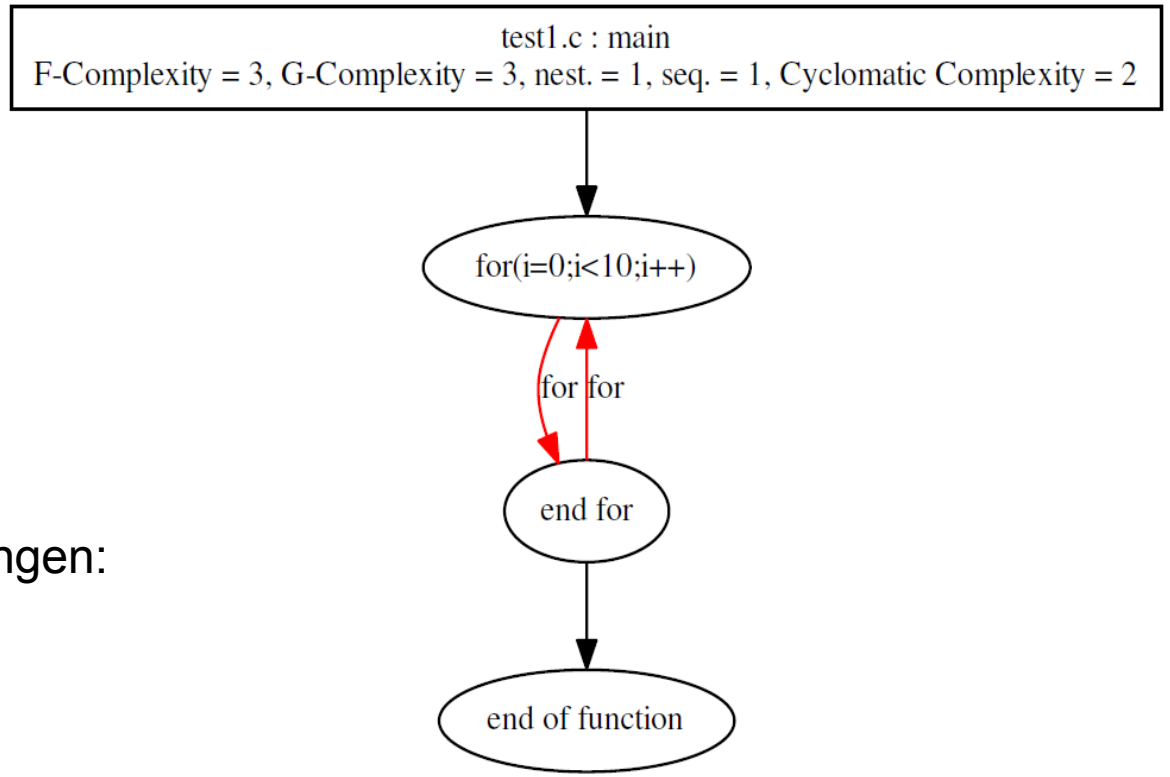
../data4_4/linux-4.4.22/drivers/net/ethernet/intel/i40e/i40e_debugfs.c : i40e_dbg_command_write
F-Complexity = 3127, G-Complexity = 2214, nest. = 29, seq. = 140, Cyclomatic Complexity = 141

# of ifs:	129
# of fors:	11
# 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:	3
# of labels:	1
# of backward gotos:	0
# of forward gotos:	50
# of returns:	4
# of continues:	0

Graphentheoretische Klassifikation

Adjazenz Matrix (n x n)

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



Algebraische Berechnungen:

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

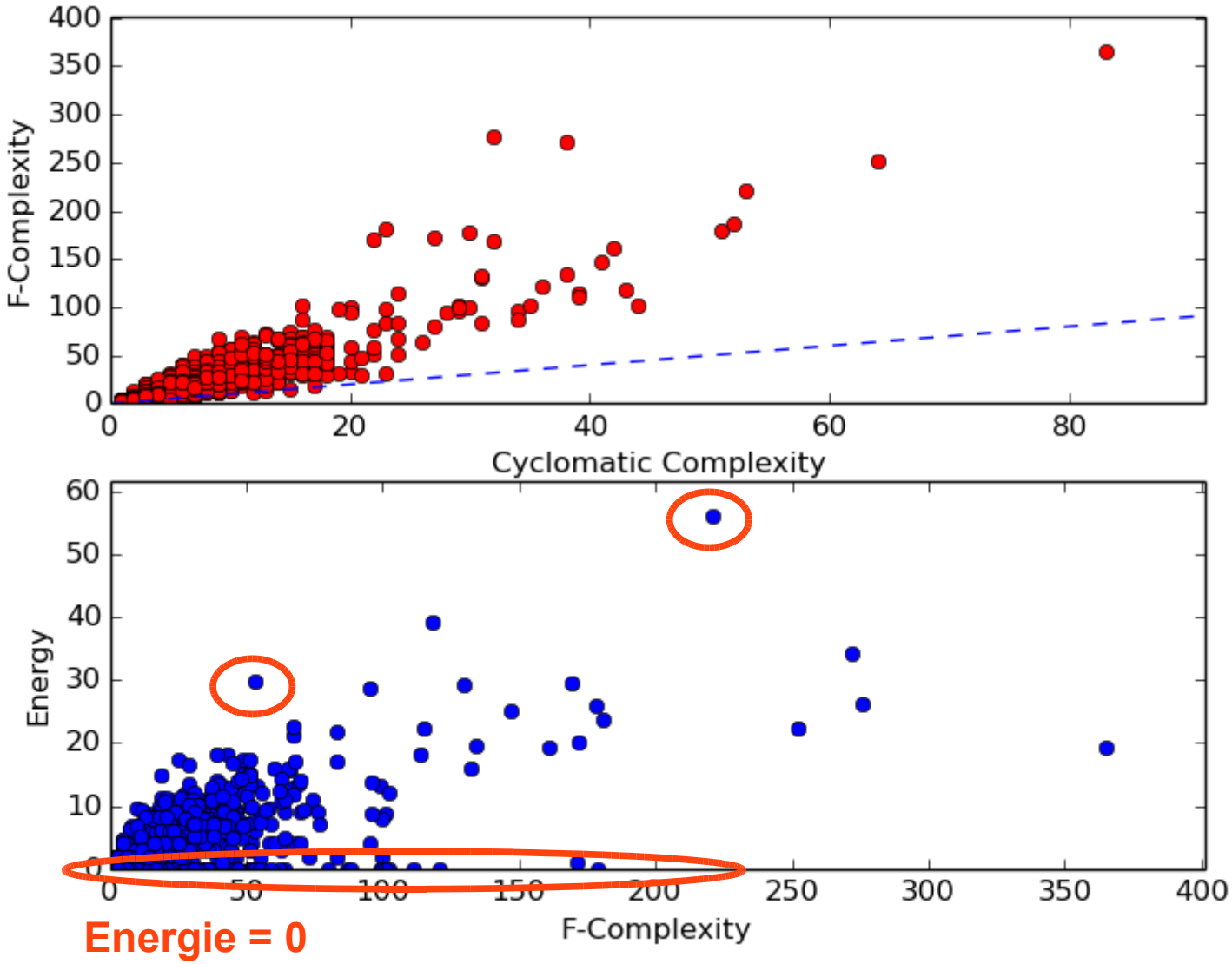
λ_i = Eigenwerte (complex)

v_i = Eigenvektoren (complex)

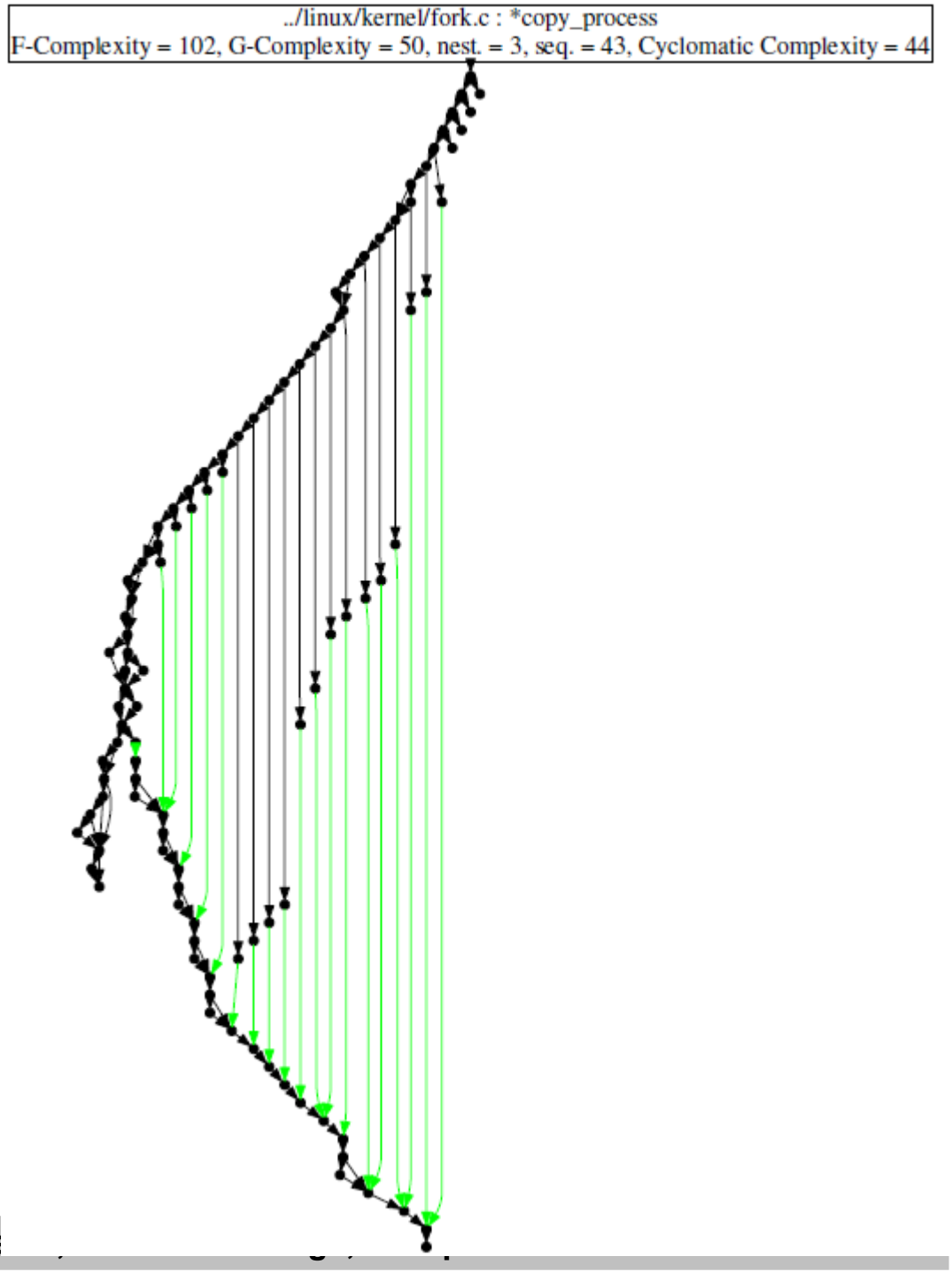
i = 0 .. n-1

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

F-Komplexität und Energie



Energie = 0: Bäume oder DAGs



Längster Pfad, hier: 66

$$\underline{\underline{A}} = \begin{pmatrix} 0\,1\,0\,0\,1/0 \\ 0\,0\,0\,0\,1/0 \\ 0\,0\,0\,1\,1/0 \\ 0\,0\,0\,0\,1/0 \\ 0\,0\,0\,0\,0\,\ldots \\ .\ .\ .\ .\ .\ldots \end{pmatrix}$$

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

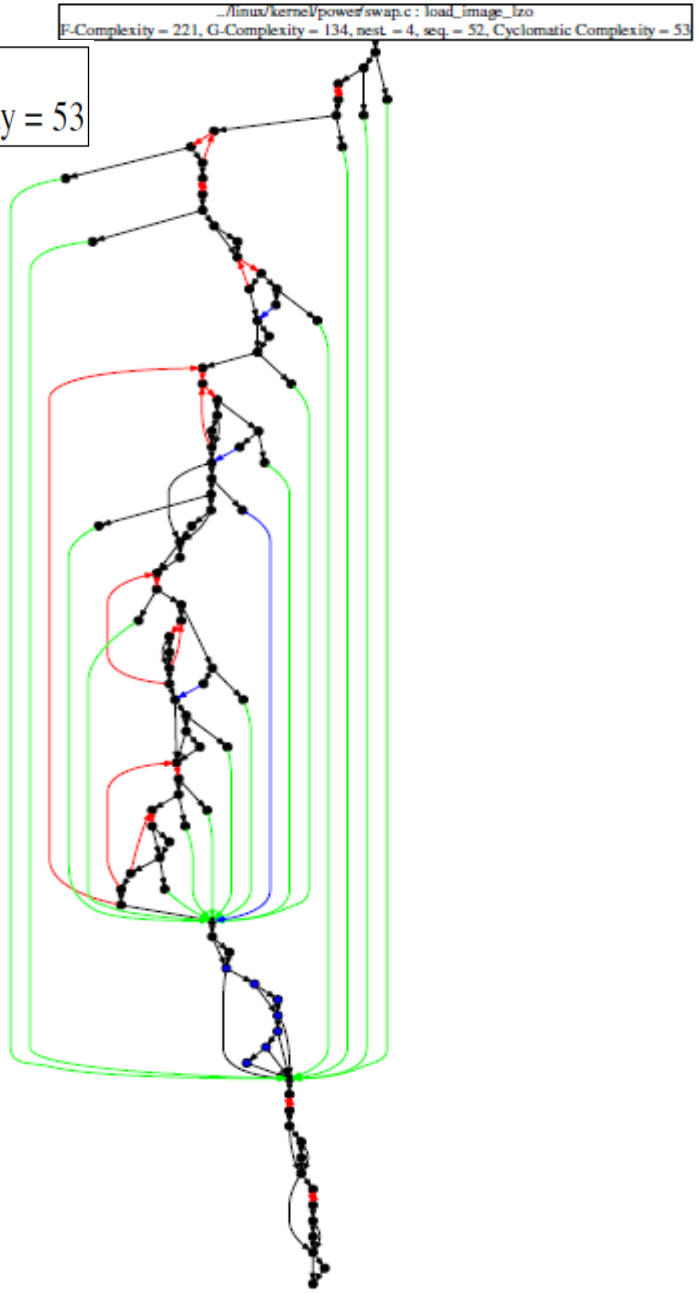
$\lambda^n = 0$

$\lambda_n = 0 \rightarrow$ Energie = 0

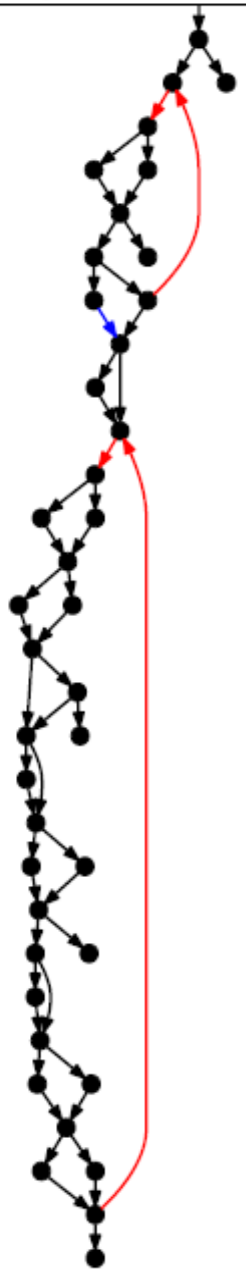
../linux/kernel/power/swap.c : load_image_lzo
F-Complexity = 221, G-Complexity = 134, nest. = 4, seq. = 52, Cyclomatic Complexity = 53

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



../linux/kernel/auditsc.c : audit_log_single_execve_arg
F-Complexity = 53, G-Complexity = 34, nest. = 3, seq. = 17, Cyclomatic Complexity = 18



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

Nächste Schritte:

- Besserer Parser
- Vollständige Analyse der verschiedenen Linuxversionen
- Interpretation der Eigenwerte und Eigenvektoren

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.

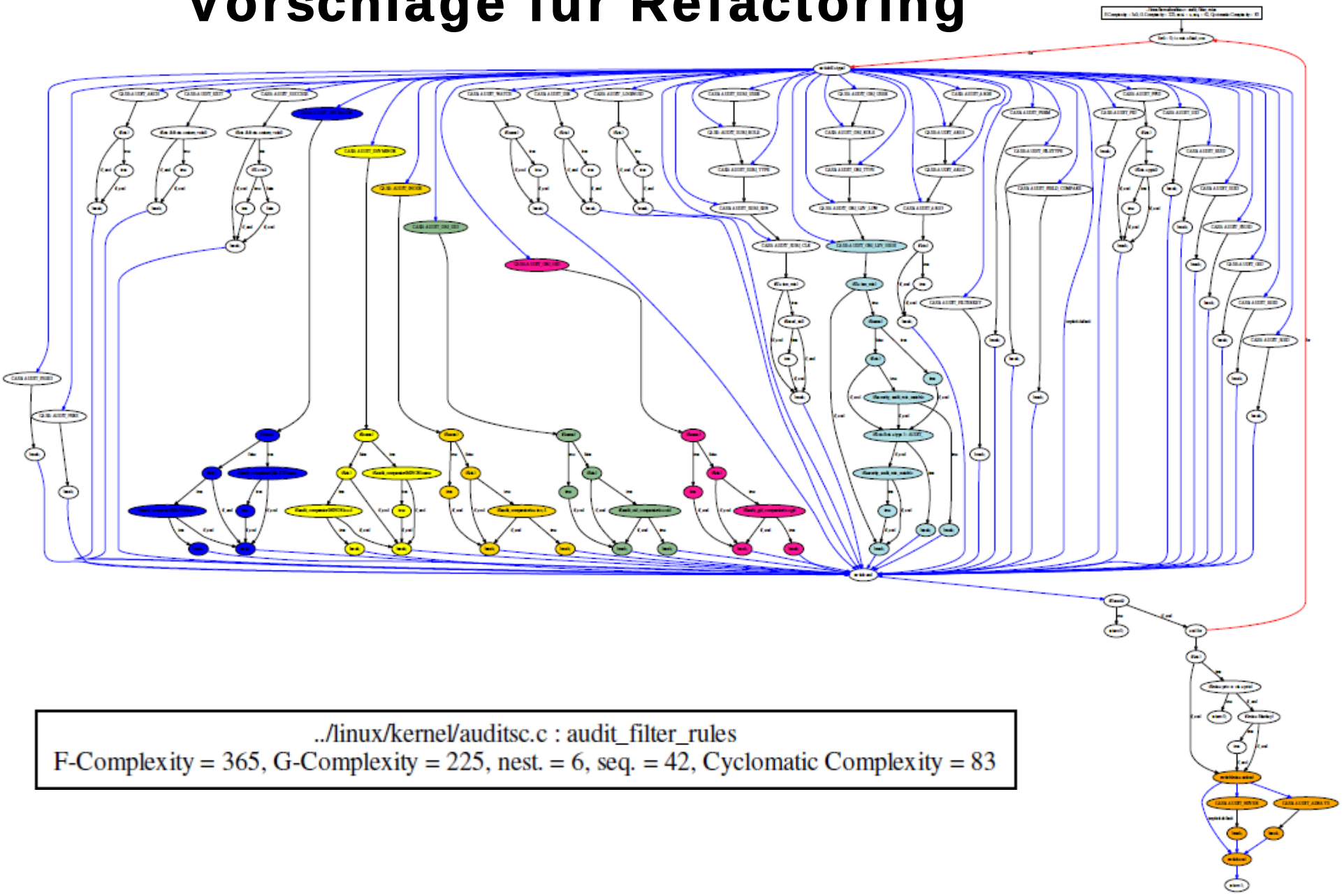
für die vielen hilfreichen Diskussionen über dieses Thema

Vielen Dank für Eure Aufmerksamkeit !

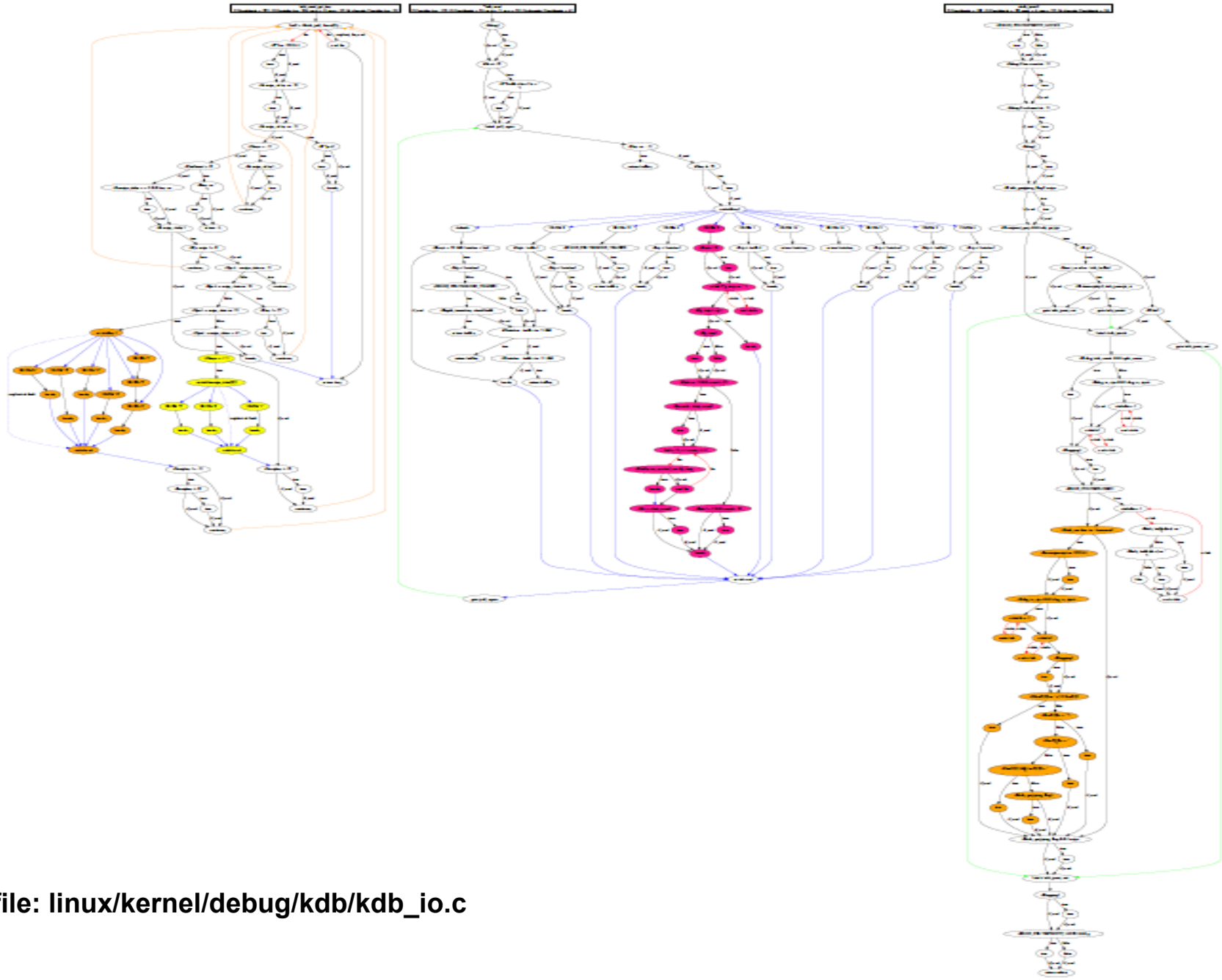
Refaktoring

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

Vorschläge für Refactoring



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



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

Linux files ohne ‘drivers’ (Ver. 3.7.9)

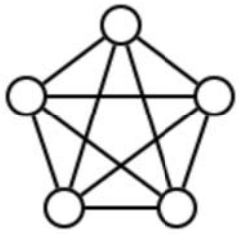
	#Dateien	#Funktionen	F-Com > 30	F-Comp > 100	F-Max	# Refs	# Refs/ #Funkionen
kernel	210	6679	306	28	365	188	2.8 %
lib	207	1343	104	22	529	92	6.8 %
crypto	148	1190	39	1	464	16	1.3 %
ipc	13	249	19	3	120	14	5.6 %
arch	5103	46059	2019	236	1092	1342	2.9 %
block	42	897	46	8	194	37	4.1 %
fs	1184	22282	2541	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

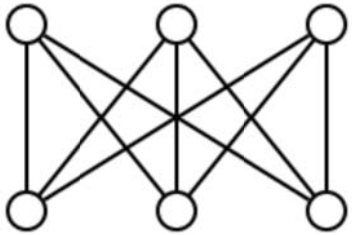
- CFG sind gerichtete Graphen
- Jedes Schleifenelement und rückwärts gerichtete Gotos kreieren Zyklen.
- Normalerweise (aber nicht immer) sind Graphen planar, d.h. die Kanten kreuzen sich nicht...

- Sind nicht-planare Graphen komplexer?

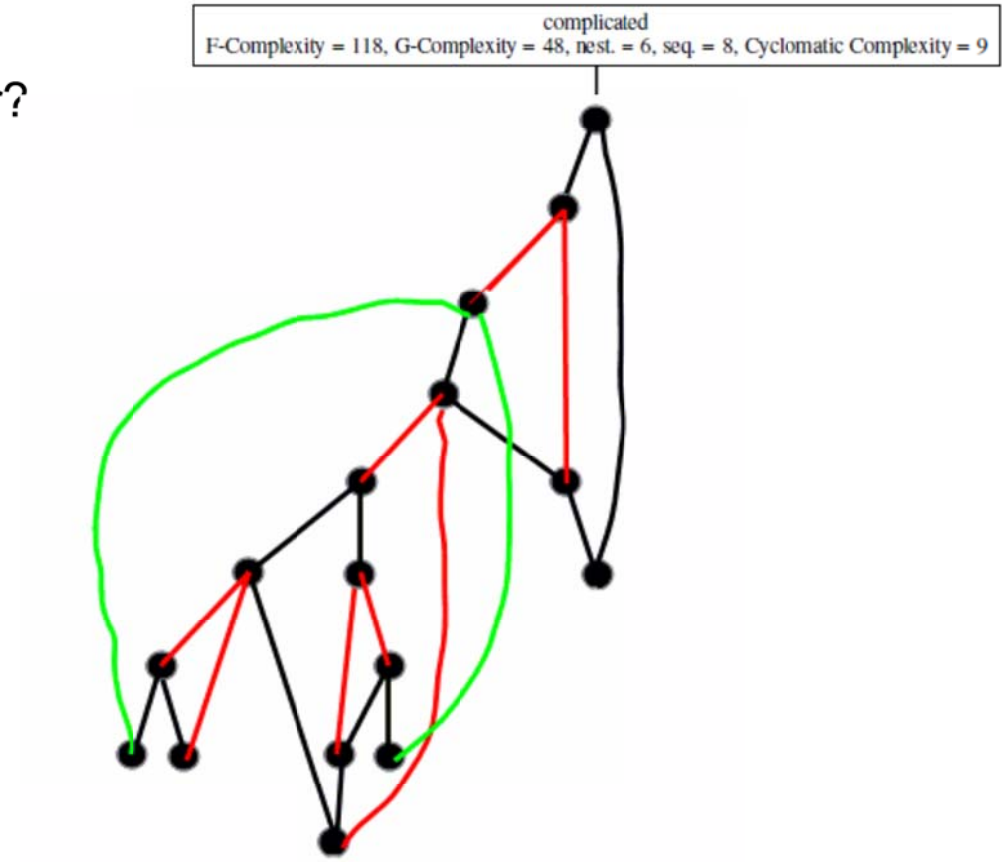
Theorem of Kuratowski



K_5

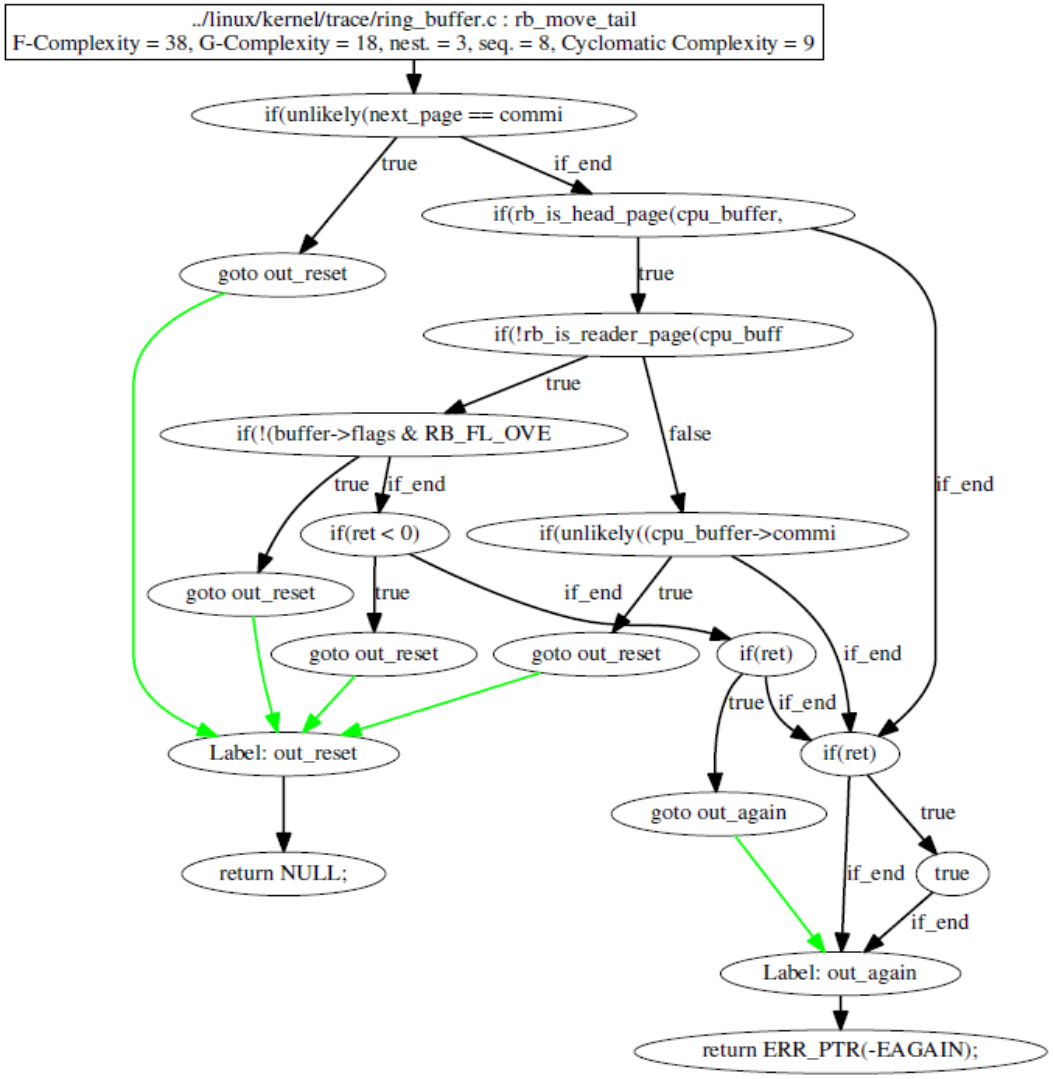


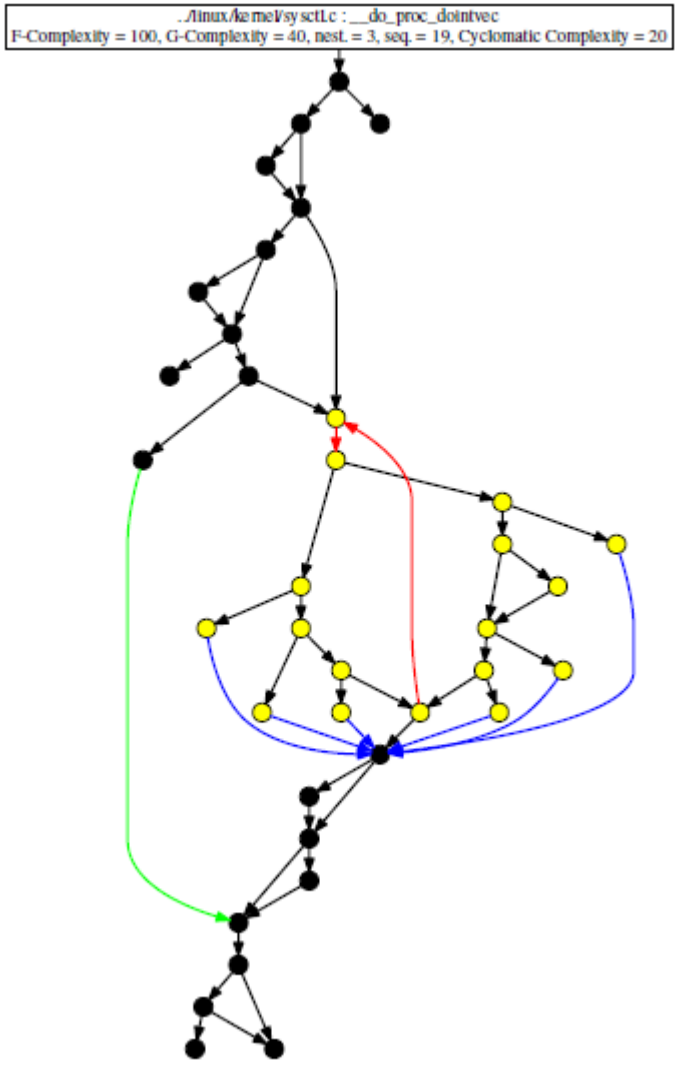
$K_{3,3}$



Nichtplanare Graphen

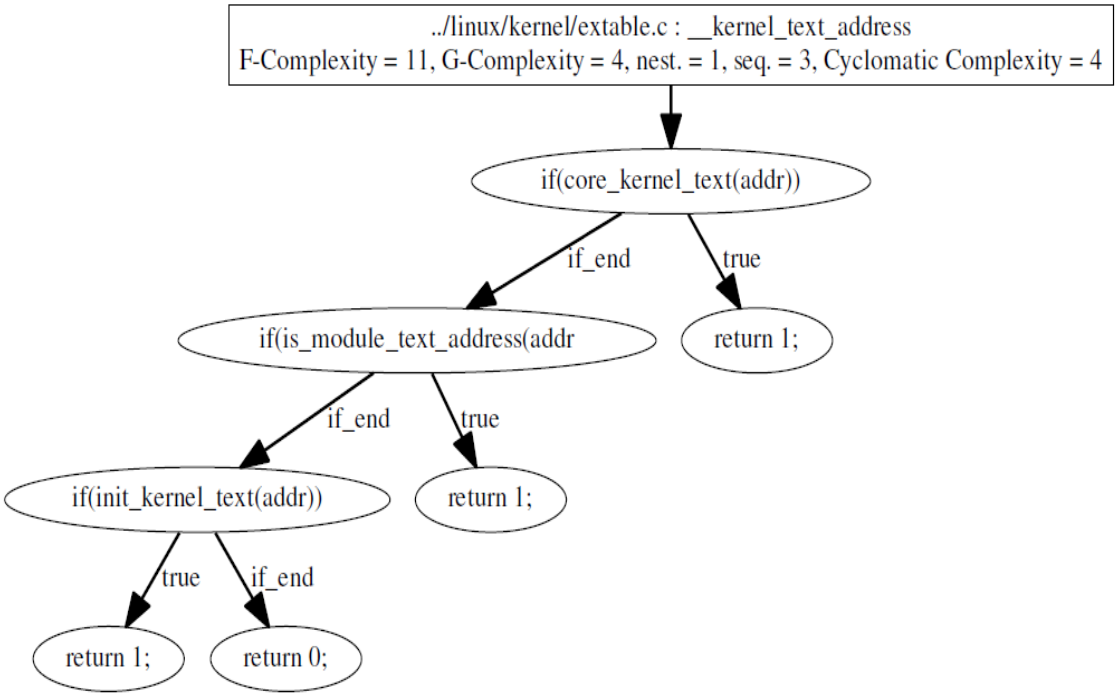
Nur 19 nichtplanare Graphs im linux/kernel/-Pfad

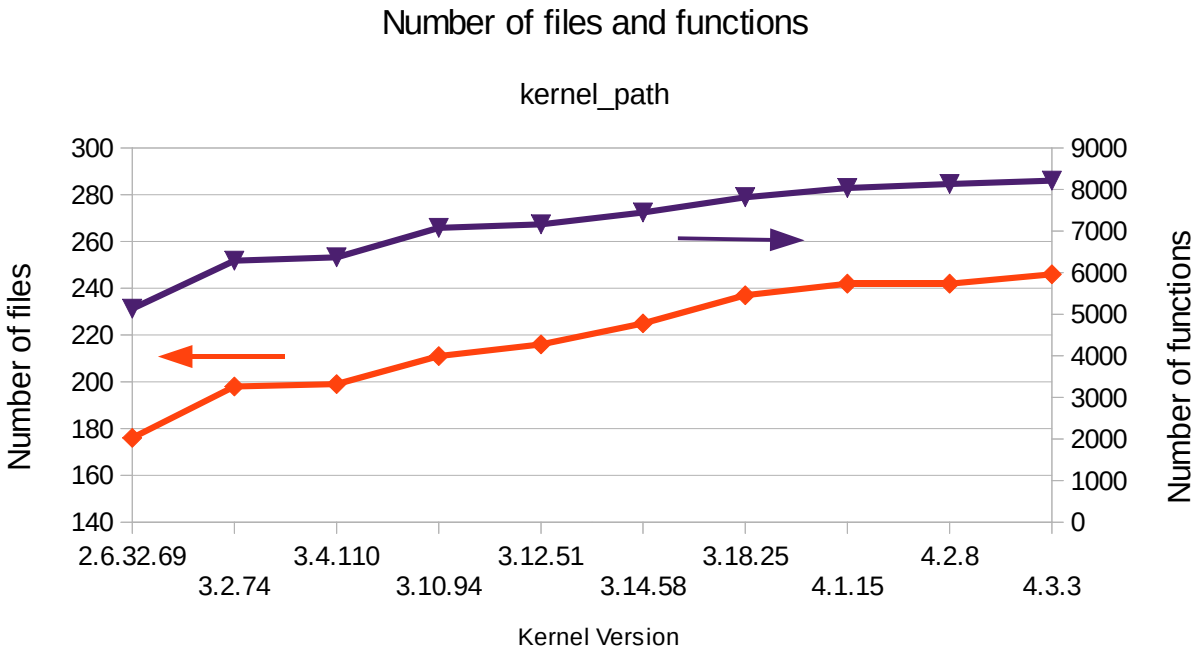


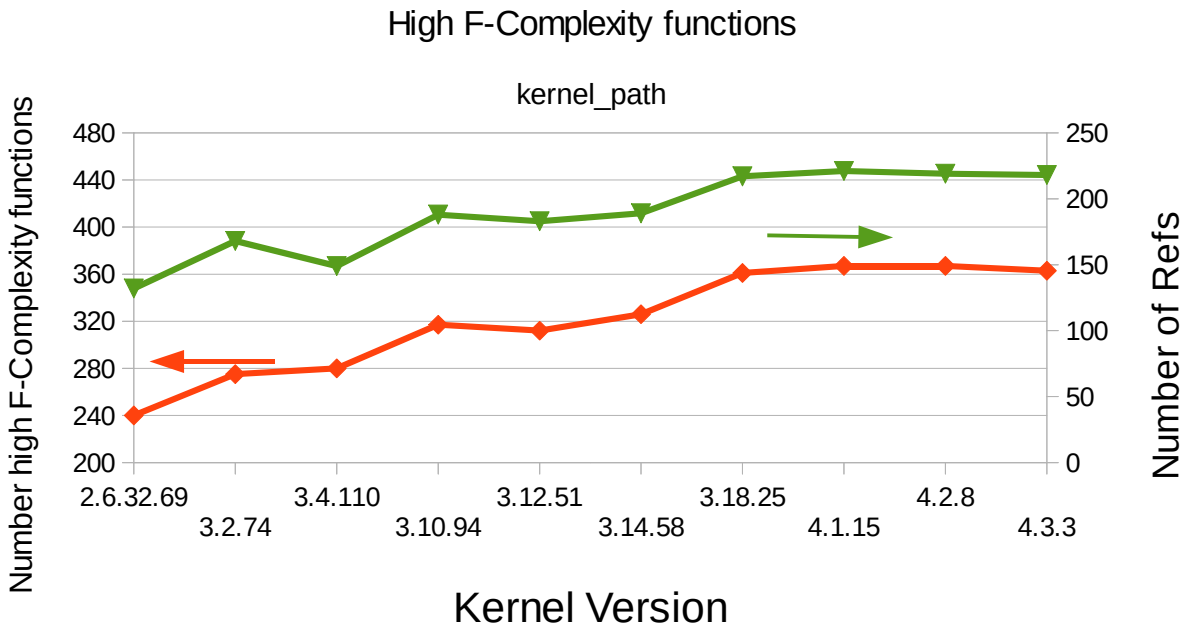


Graphentheoretische Klassifikation

Funktionen mit Baumstruktur:



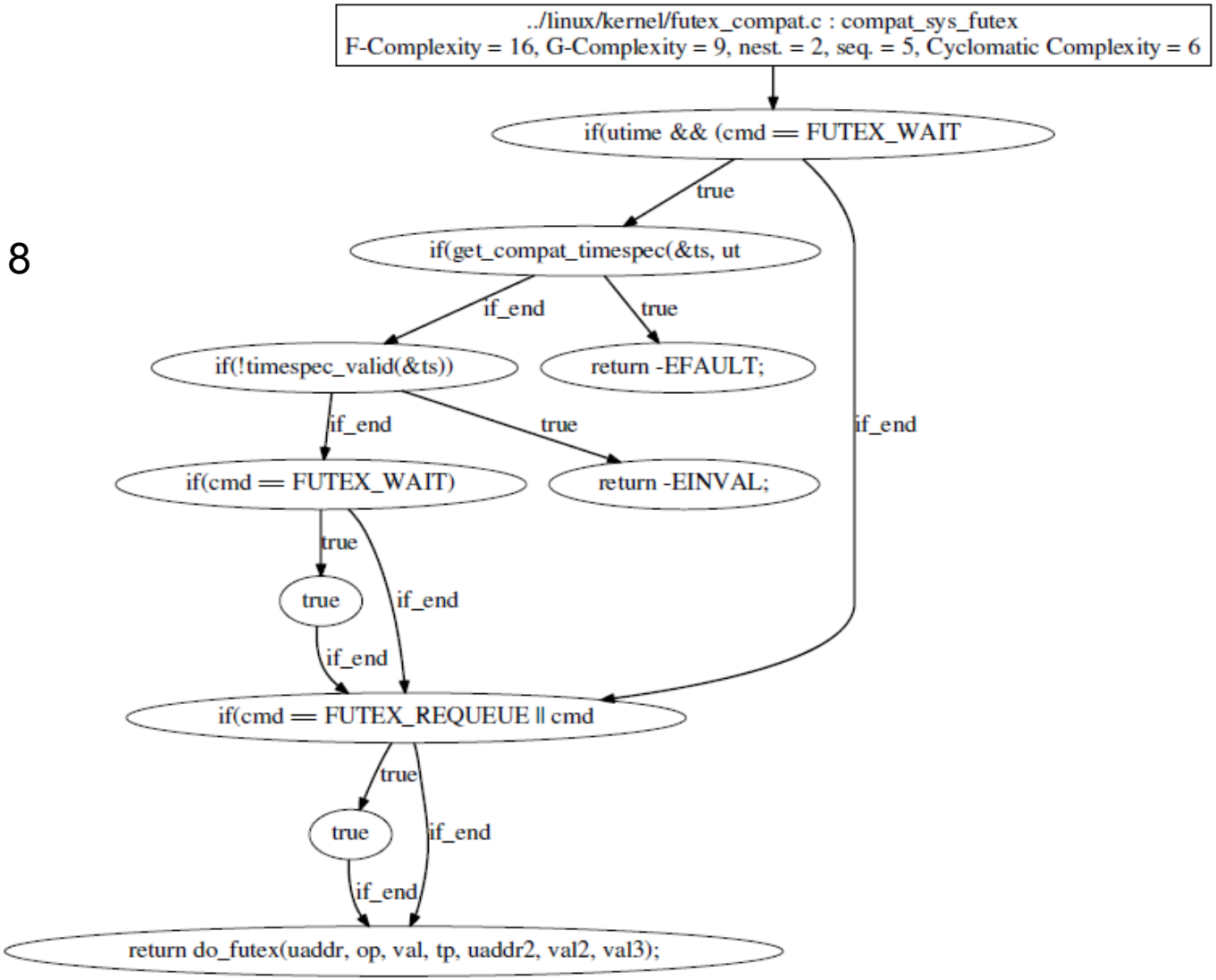

Number of files and functions
kernel_path
Number of files
Number of functions
300
280
260
240
220
200
180
160
140
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
2.6.32.69
3.2.74
3.4.110
3.10.94
3.12.51
3.14.58
3.18.25
4.1.15
4.2.8
4.3.3
Kernel Version


High F-Complexity functions
kernel_path
Number high F-Complexity functions
Number of Refs
480
440
400
360
320
280
240
200
250
200
150
100
50
0
2.6.32.69
3.2.74
3.4.110
3.10.94
3.12.51
3.14.58
3.18.25
4.1.15
4.2.8
4.3.3
Kernel Version

Graphentheoretische Klassifikation

Azyklische Graphen (Directed acyclic graphs (DAG))?

Quantitative Klassifikation:
Längster Pfad: hier 8



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**