On our way to apply model-checking to the kernel

Linux Driver Verification Workshop – ISoLA 2012

Alexandre Lissy

Mandriva, Paris

Laboratoire d'Informatique de Tours (EA2101) – Équipe Ordonnancement et Conduite (ERL CNRS 6305)

October 15th, 2012

Outline

1. Bibliography results

Applying model-checking to kernel

Few references

Not a lot of references can be found in literature

- SLAM/SDV at Microsoft
- Coccinelle for Linux

Model-Checking: limited by state explosion $\Rightarrow$ Limiting number of states

Model-Checking of dynamic structures

Recent work

Introduction of the Abstract Regular Tree Model Checking technique [?] and application to linked lists [?]

Prototype GCC plugin, seems very promising

2. Explo(d|r)ing the kernel

Clustering the Kernel ?

Clusters ?

Finding independent parts inside the kernel

- First, study kernel topology
- Tool: graph of symbols dependencies
- Extract as many informations as possible from this graph

Definitions

Graph

A set of directed edges and vertices

Vertices

Object file in the kernel build process

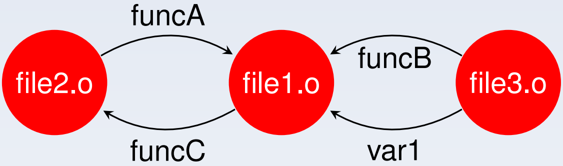
Directed edge

Symbol usage between two object files. Direction is used to known which one is exporting and importing

Example

Small example

Three source files, three corresponding object files

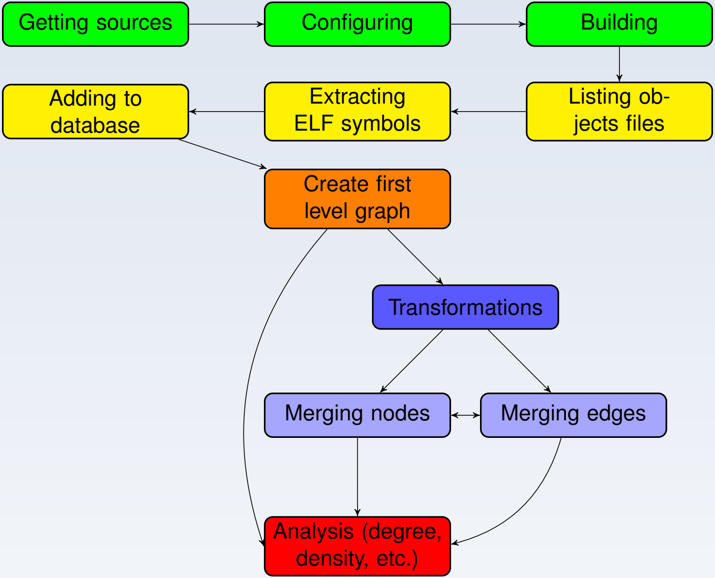


Exports, usages

- `file1.c` uses `funcA()`, `funcB()` and `var1`
- `file2.c` uses `funcC()`

3. Building a graph of the kernel
   - How to build it
   - Graph size
   - Symbols
   - Density
   - Average Path Length
   - Degrees
   - HeatMaps

Graph creation process



Generating object files

Using kernel build system

Two build configurations

- `defconfig` (2000 nodes, 50000 edges)
- `allyesconfig` (10000 nodes, 320000 edges)
- Limited to *current build hardware*

- Allows easy comparison, *light* versus *complete* system; *base system* versus *full system with drivers*.
- Using object files avoid complex, risky C source code parsing.

Working with object files

How to discover *useful* object files ?

Finding

Naive parsing of kernel Makefile's

- Find variables assignments which contains `.o`
- Extract each object file referenced
- Check that it exists really on the filesystem
- If it is the case, then add it to the list of object files to analyze

Works well, finding object files that have a legitimate existence in the kernel build system.

Symbols extraction

How do we extract them ? And which one to extract ?

Extracting

Using `ELF` extraction tools:

- First implementation, with `readelf`
- Second implementation, through `libelf` (`libelfg0`)

Extracting which ones ? All of them!

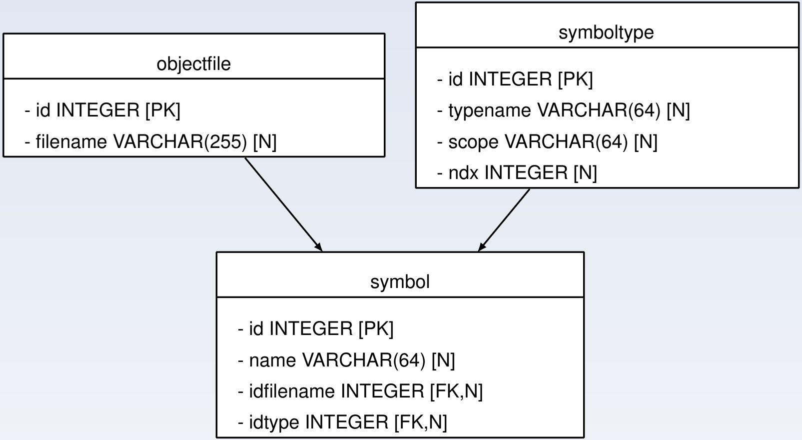
Using a database to store the result.

Storing informations

Three main objects, representing symbols, symbol types, object files and how they relates.

Slowest part of the process . . .

Database schema



First level graph

Definition

The first level graph is a naive graph built directly from the database. It serves as a basis for some analysis and more important it will be the source for transformations.

Building

How to build it

- Nodes: using all object files from the database. Label: full path of the object file
- Edges: using all symbols from database

Transformations

Goal

Producing new graph, using the “naive” as a source, that allows and/or ease analysis

Examples

- Merging nodes
- Merging edges

Studying graph through its properties

The goal is to be able to characterize the graph associated to a kernel.

- Size: number of nodes, number of edges
- Degrees: in and out
- Symbol occurrences
- Density
- Average path length
- Heatmaps

Transformations performed

Merging edges

- Relations between two nodes
- Computing “attraction”:

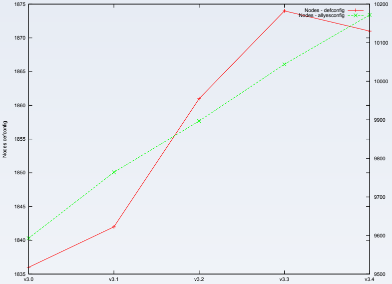
$$A_{n,m} = Card(Edges(N, M))$$

Merging nodes

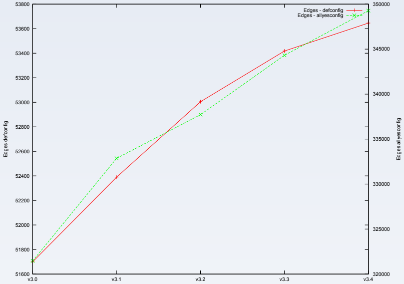
- Looking at “root” directories: `mm/`, `kernel/`, `drivers/`, etc.
- Or deeper: inside `drivers/`

Graph size: nodes and edges

	Nodes		Edges	
Version	`defconfig`	`allyesconfig`	`defconfig`	`allyesconfig`
v3.0	1836	9593	51700	321463
v3.1	1842	9764	52390	332865
v3.2	1861	9897	53005	337717
v3.3	1874	10044	53418	344314
v3.4	1871	10172	53646	349271



(a) Nodes



(b) Edges

Graph size variations: nodes and edges

	Nodes		Edges	
Version	`defconfig`	`allyesconfig`	`defconfig`	`allyesconfig`
v3.0	-	-	-	-
v3.1	$+0.33\%$	$+1.78\%$	$+1.33\%$	$+3.55\%$
v3.2	$+1.03\%$	$+1.36\%$	$+1.17\%$	$+1.46\%$
v3.3	$+0.70\%$	$+1.49\%$	$+0.78\%$	$+1.95\%$
v3.4	-0.16%	$+1.27\%$	$+0.43\%$	$+1.44\%$

Code base size variations

	SLOCCount		Evolution	
Version	`defconfig`	`allyesconfig`	`defconfig`	`allyesconfig`
v3.0	9614824	9612505	-	-
v3.1	9704743	9702470	+0.94%	+0.94%
v3.2	9862036	9860466	+1.62%	+1.63%
v3.3	9977312	9976172	+1.17%	+1.17%
v3.4	10120350	10119606	+1.43%	+1.44%

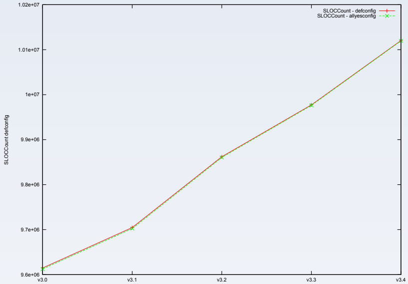
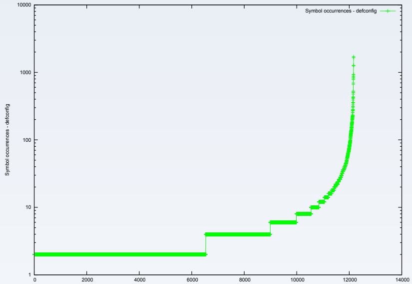
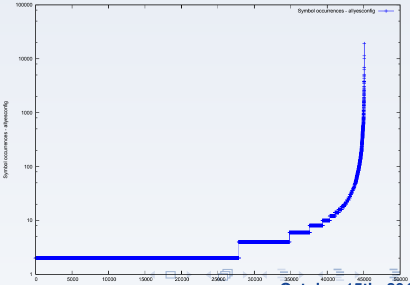


Figure: Code base size evolution

Symbol occurrences

defconfig		allyesconfig	
_raw_spin_lock	782	mutex_lock_nested	4614
_cond_resched	806	mutex_unlock	4898
__kmalloc	846	__kmalloc	5156
current_task	864	__stack_chk_fail	6258
mutex_lock	912	kmem_cache_alloc_trace	6922
mutex_unlock	936	kmalloc_caches	6950
kmem_cache_alloc_trace	1254	kfree	10152
kmalloc_caches	1270	printk	11336
printk	1658	__gcov_init	19014
kfree	1706	__gcov_merge_add	19014





Graph density

	Density	
Version	`defconfig`	`allyesconfig`
v3.0	0.015346	0.003494
v3.1	0.015449	0.003492
v3.2	0.015313	0.003448
v3.3	0.015219	0.003413
v3.4	0.015333	0.003376

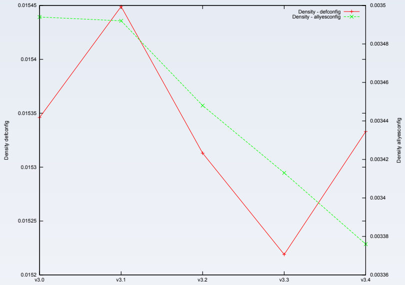


Figure: Density among versions

Graph density per subdirectories, kernel 3.0

Subdir	`defconfig`	`allyesconfig`
arch	0.039320	0.035418
block	0.268398	0.281667
crypto	0.241935	0.073537
drivers	0.021583	0.002376
fs	0.063002	0.018673
init	0.291667	0.291667
ipc	0.712121	0.719697
kernel	0.122087	0.126854
lib	0.019572	0.016000
mm	0.309949	0.299454
net	0.060322	0.015070
security	0.288762	0.103541
sound	0.173263	0.024607

Graph density per subdirectories, kernel 3.0

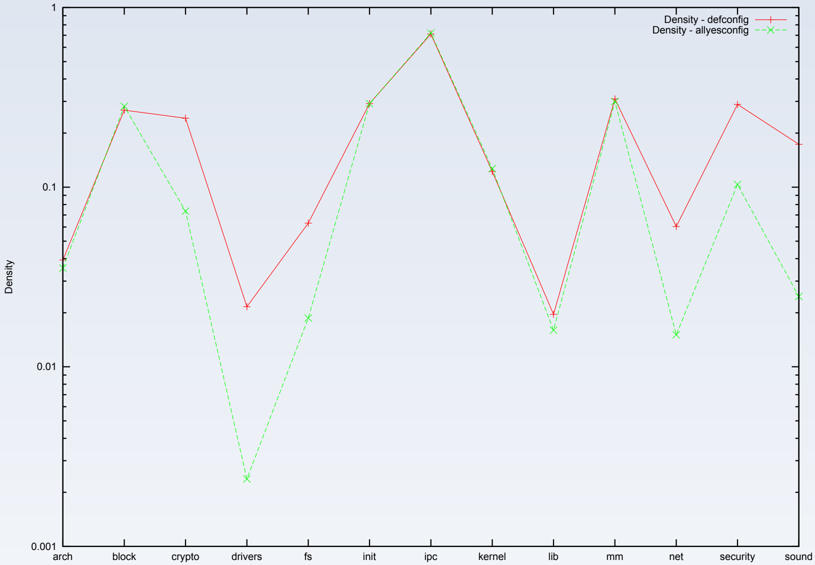


Figure: Density over subdirectories

Average Path Length

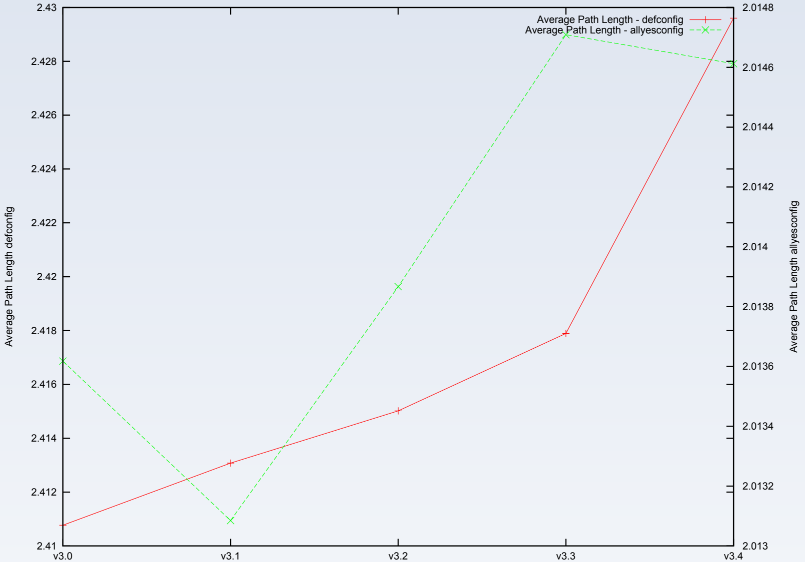
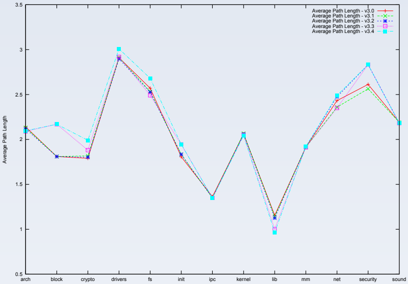
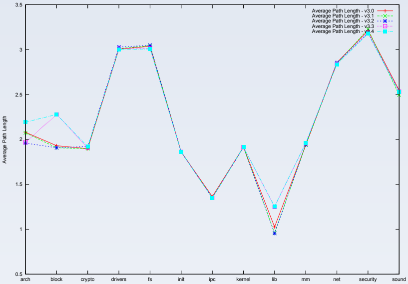


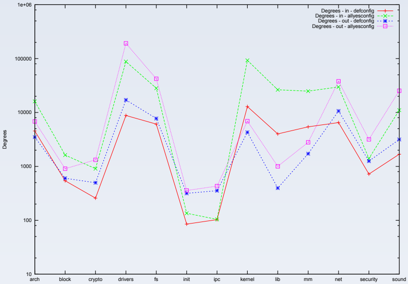
Figure: Average Path Length



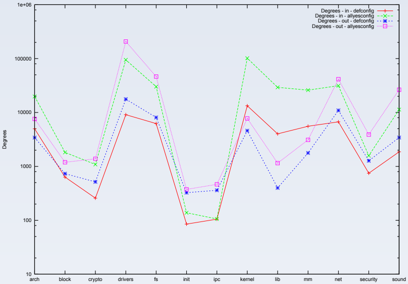
(a) `defconfig`



(b) `allyesconfig`



(c) Kernel v3.0



(d) Kernel v3.4

Degrees comparison

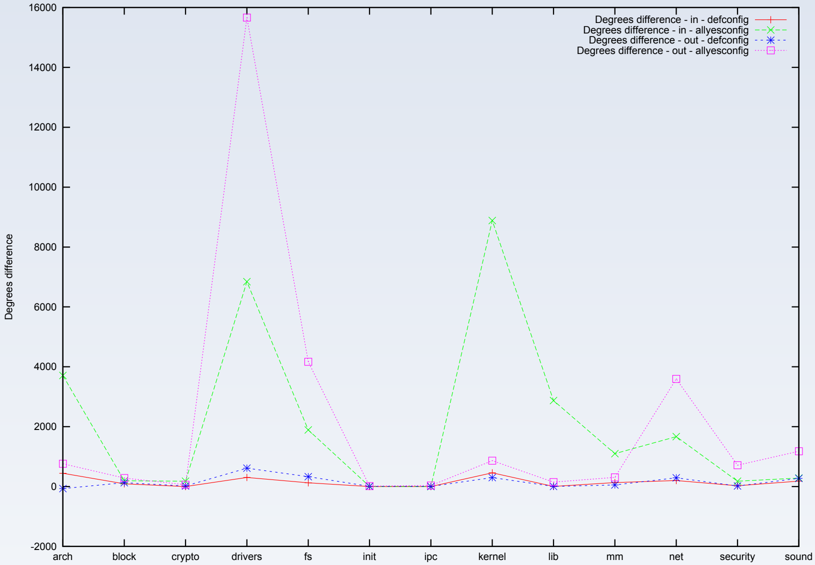


Figure: Kernel v3.4 vs v3.0

HeatMap – Why ?

Visualisation issues

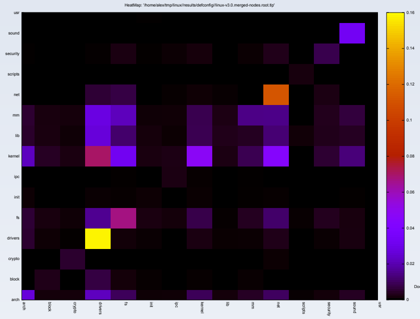
- Easy way to see symbol usage
- Compact, efficient

Proposition

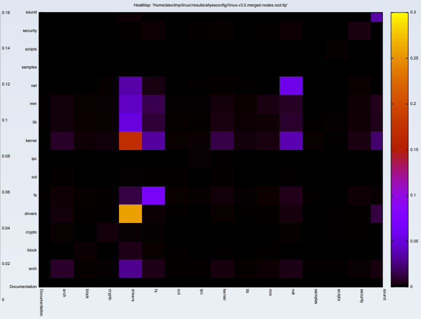
“HeatMap”, showing intensity of dependencies; roughly equivalent to an adjacency matrix.

- Axis: subdirectories
- Values: normalized number of edges

HeatMap - Kernel 3.0

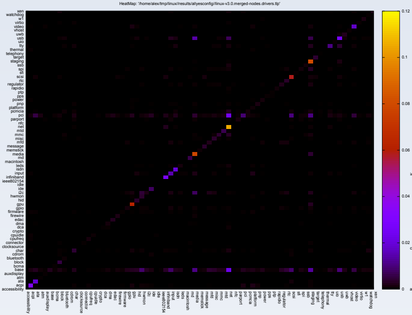


(a) `defconfig`

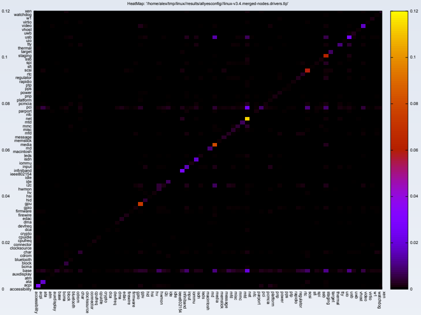


(b) `allyesconfig`

- Scale for `defconfig`: 0 to 0.16
- Scale for `allyesconfig`: 0 to 0.3



(c) Kernel v3.0



(d) Kernel v3.4

Same scale for both

4 Conclusion

- On the kernel graph
- On model-checking the kernel

First results

- Caracterizing a kernel via a graph
- Number of nodes, edges
- Relations between subcomponents of the kernel
- Foundation of a process, tools

Limitations

- Flat graph
- Edges qualifications
- Quite slow process
- Not enough kernel

Evolutions

- Re-use tree informations
- Using more symbols informations
- Running over more kernel
- Running on other code base

What has been done for Linux ?

- “Porting” SDV work as LDV
  - Hard to find up-to-date information about it
  - Publications were quite enthusiasts
- Coccinelle
  - First targetting evolutions
  - Pattern matching tool

What about ARTMC ?

- Technique for checking complex data structures
- Running inside GCC
- Quite new but very promising

Probably the best fit for using on kernel code, could allow to verify things not being verified right now

Questions ?

Thanks ! Any question ?