Konfigurierbare Systemsoftware (KSS)

VL 5 – Variability Management in the Large:
The VAMOS Approach

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 14 – 2014-05-15

http://www4.informatik.uni-erlangen.de/Lehre/SS14/V_KSS

About this Lecture

Problem Space
Solution Space

Specif. Problem
Specif. Solution

Domain Expert

Features and Dependencies

Architecture / Developer

Configuration

System User

Variant

Real-world software uses them all!
Typical Configurable Operating Systems...

- 320 features
- 1,250 features

more variants than atoms in the universe!
Typical Configurable Operating Systems...

1,250 features

Challenges: → VAMOS∗

- How to maintain this?
- How to test this?
- Why so many features anyway?

∗ Variability Management in Operating Systems

12,000 features

The Linux Configuration and Generation Process

1. Configuration with an \texttt{CONFIG} frontend
2. Compilation of a subset of files
3. Selection of a subset of CPP Blocks
4. Linking of the kernel and loadable kernel modules

Agenda

5.1 Motivation
5.2 Variability in Linux
5.3 Configuration Consistency
5.4 Configuration Coverage
5.5 Automatic Tailoring
5.6 Summary
5.7 References

Domain and Hierarchy of Variability

l₀: Feature Modeling 12,000 features
l₁: Coarse-grained: \texttt{KBUILD} 31,000 source files
l₂: Fine-grained: CPP 89,000 \texttt{#ifdef} blocks
l₃: Language-level: GCC → if(CONFIG_SMP) ...
l₄: Link time: LD → branches in linker scripts
l₅: Run time: INSMOD, MODPROBE, ...
Challenges with Implemented Variability

Central declaration of configurability: \texttt{KCONFIG}

Distributed implementation of configurability: \texttt{MAKE}, \texttt{CPP}, \texttt{GCC}, \texttt{LD}

Problem Analysis: Configuration Consistency

Problem Analysis: Symbolic Inconsistency

Agenda

5.1 Motivation
5.2 Variability in Linux
5.3 Configuration Consistency
   - Problem Analysis
   - Solution Approach
   - Results
5.4 Configuration Coverage
5.5 Automatic Tailoring
5.6 Summary
5.7 References

---

Result:
Fix for a critical bug
Problem Analysis: Logic Inconsistency

- Feature DISCONTIGMEM implies feature NUMA
- Inner blocks are not actually configuration-dependent
  - Block2 is **always** selected $\implies$ **undead**
  - Block3 is **never** selected $\implies$ **dead**

  Linux contains **superfluous** #ifdef Blocks!

---

Solution Approach: Consistency Validation

Problem and solution space are analyzed for configuration points:

$$ C = (\text{FLATMEM} \rightarrow \text{MEMORY MODEL}) \land (\text{DISCONTIGMEM} \rightarrow \text{MEMORY MODEL}) \land (\text{SPARSEMEM} \rightarrow \text{MEMORY MODEL}) \land (\text{NUMA} \rightarrow \text{MEMORY MODEL}) \land (\text{DISCONTIGMEM} \rightarrow \text{NUMA}) $$

$$ I = (\text{Block1} \leftrightarrow \text{DISCONTIGMEM}) \land (\text{Block2} \leftrightarrow \text{Block1} \land \text{NUMA}) \land (\text{Block3} \leftrightarrow \text{Block1} \land \neg\text{Block2}) $$

---

Implementation: The UNDERTAKER

**Job:** Find (and eventually bury) **dead** #ifdef-code!

- We have found 1776 configurability defects in Linux v2.6.35
- Submitted 123 patches for 364 defects
- 20 are confirmed **new bugs** (affecting binary code)
- Cleaned up 5129 lines of cruft code
Implementation: The **UNDERTAKER**

**Job:** Find (and eventually bury) dead `#ifdef-code`!

How good is this, really?

---

**Common Beliefs About Variability in Linux**

1. Most variability is expressed by boolean (or tristate) switches.
2. arch-x86 is the largest and `allyesconfig` selects most features.
3. Variability is mostly implemented with the `CPP`.
4. The Linux *kernel* is highly configurable.

⇒ Almost all features in Linux are **option-like**

---

**Linux v3.1: Feature Distribution by Type**

- **Most variability is expressed by boolean (or tristate) switches**

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>KCONFIG features</td>
<td>11,691</td>
<td>100%</td>
</tr>
<tr>
<td>Option-like</td>
<td>10,907</td>
<td>93.3%</td>
</tr>
<tr>
<td>Value-like</td>
<td>784</td>
<td>6.7%</td>
</tr>
<tr>
<td>Boolean</td>
<td>6,024</td>
<td>55.2%</td>
</tr>
<tr>
<td>Tristate</td>
<td>4,883</td>
<td>44.8%</td>
</tr>
<tr>
<td>String</td>
<td>87</td>
<td>11.1%</td>
</tr>
<tr>
<td>Integer/Hex</td>
<td>697</td>
<td>88.9%</td>
</tr>
</tbody>
</table>

⇒ How good is this, really?
Linux v3.1: Coverage of arch-x86 / allyesconfig

- arch-x86 is the largest and allyesconfig selects most features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Config Features</th>
<th>Build Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch-x86</td>
<td>7,776 [66.5%]</td>
<td>70.5%</td>
</tr>
<tr>
<td>non-arch-x86</td>
<td>3,915 [33.5%]</td>
<td>29.5%</td>
</tr>
<tr>
<td>allyesconfig</td>
<td>5,482 [46.9%]</td>
<td>not considered</td>
</tr>
<tr>
<td>non-allyesconfig</td>
<td>2,294 [19.6%]</td>
<td></td>
</tr>
</tbody>
</table>

⇒ arch-x86/allyesconfig is not nearly a full configuration

Linux v3.1: Distribution by Granularity

- Variability is mostly implemented with the CPP

<table>
<thead>
<tr>
<th>Feature</th>
<th>Config Features</th>
<th>Build Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch-x86</td>
<td>11,691 [100%]</td>
<td></td>
</tr>
<tr>
<td>non-arch-x86</td>
<td>7,749 [66.3%]</td>
<td>75.5%</td>
</tr>
<tr>
<td>allyesconfig</td>
<td>3,915 [33.5%]</td>
<td>24.5%</td>
</tr>
<tr>
<td>non-allyesconfig</td>
<td>2,294 [19.6%]</td>
<td></td>
</tr>
</tbody>
</table>

⇒ KBUILD implements more than two thirds of all variation points

Linux v3.2: Distribution by HW/SW

- The Linux kernel is highly configurable

<table>
<thead>
<tr>
<th>Feature</th>
<th>Config Features</th>
<th>Build Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software related</td>
<td>1,487 [12.4%]</td>
<td>30.1%</td>
</tr>
<tr>
<td>Hardware related</td>
<td>10,551 [87.6%]</td>
<td>63.9%</td>
</tr>
<tr>
<td>net</td>
<td>530 [4.4%]</td>
<td>37.4%</td>
</tr>
<tr>
<td>misc</td>
<td>447 [3.7%]</td>
<td>3.7%</td>
</tr>
<tr>
<td>drivers</td>
<td>5,330 [44.3%]</td>
<td>5.1%</td>
</tr>
<tr>
<td>sound</td>
<td>536 [4.5%]</td>
<td></td>
</tr>
<tr>
<td>arch</td>
<td>4,685 [38.9%]</td>
<td></td>
</tr>
</tbody>
</table>

⇒ Software features account for only twelve percent of all variation points

Linux Feature Growth over Time

- All features

<table>
<thead>
<tr>
<th>Year</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
<td>2,000</td>
</tr>
<tr>
<td>2008</td>
<td>4,000</td>
</tr>
<tr>
<td>2009</td>
<td>6,000</td>
</tr>
<tr>
<td>2010</td>
<td>8,000</td>
</tr>
<tr>
<td>2011</td>
<td>10,000</td>
</tr>
<tr>
<td>2012</td>
<td>12,000</td>
</tr>
</tbody>
</table>

⇒ SW features (everything else)
Linux Feature Growth over Time

<table>
<thead>
<tr>
<th>Year</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
<td>1,000</td>
</tr>
<tr>
<td>2008</td>
<td>2,000</td>
</tr>
<tr>
<td>2009</td>
<td>3,000</td>
</tr>
<tr>
<td>2010</td>
<td>4,000</td>
</tr>
<tr>
<td>2011</td>
<td>5,000</td>
</tr>
<tr>
<td>2012</td>
<td>6,000</td>
</tr>
</tbody>
</table>

Results: Where Have all the Features Gone?

1. Most variability is expressed by boolean (or tristate) switches
   - more than 93 percent of all features are option-like
   - it is acceptable for tools to ignore value-type features

2. arch-x86 is the largest and allyesconfig selects most features
   - more than 53 percent are not covered by this configuration
   - other parts of Linux are probably less tested and error-prone!

3. Variability is mostly implemented with the CPP
   - more than 66 percent of all features are handled
   - by the build system, only 17 percent are handled by CPP only
   - variability extraction from KBUILD is necessary

4. The Linux kernel is highly configurable
   - only 12 percent of all features configure software only
   - variability is mostly induced by advances in hardware
   - complexity will increase further

Challenges: Variability Extraction from the Build System

- Variability extraction \(\rightarrow\) which file is selected by which feature?
- Usual approach for variability extraction [6, 10] (KCONFIG, CPP, ...):
  - source \(\xrightarrow{\text{parse & transform}}\) propositional formula
- Parsing does not work well for MAKE-languages
  - declarative and Turing-complete languages
  - special features, like shell, foreach, eval, addprefix, ...
- Linux’s KBUILD is built on top of (GNU) MAKE
  - nevertheless, researchers have tried parsing to extract variability
    - KBUILDMINER by Berger, She, Czarnecki, et al. [1]
    - Nadi parser by Nadi and Holt [5]
  - resulting tools are too brittle at best
    - work for a (few) Linux version(s) only
    - each usage of a special feature requires manual tailoring

Linux Build Process Revisited
Variability Extraction from KBUILD with GOLEM [2]

Basic idea: Systematic probing and inferring of implications

**Dancing Makefiles**
- Identification of KCONFIG references
- Recursion into subdirectory while considering constraints

Robust with respect to architecture and version
⇒ no adaptations on or for KBUILD!

<table>
<thead>
<tr>
<th>Kernelversion</th>
<th>found inferences</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2.6.25</td>
<td>6,274 (93.7%)</td>
</tr>
<tr>
<td>v2.6.28.6</td>
<td>7,032 (93.6%)</td>
</tr>
<tr>
<td>v2.6.33.3</td>
<td>9,079 (94.9%)</td>
</tr>
<tr>
<td>v2.6.37.6</td>
<td>10,145 (95.1%)</td>
</tr>
<tr>
<td>v3.2</td>
<td>11,050 (94.5%)</td>
</tr>
</tbody>
</table>

**Case Study: Configuration Consistency**

Configuration defects in Linux v3.2:

**Without KBUILD constraints**
- Code defects: 1835
- Referential defects: 415
- Logical defects: 83
- Sum: \( \Sigma \ 2333 \)

**With KBUILD constraints**
- Code defects: 1835
- Referential defects: 439
- Logical defects: 299
- Sum: \( \Sigma \ 2573 \) Result: +10%

**Implementation Space Coverage**

Issue: Decompositional Implementation of Variability

```
#ifdef CONFIG_NUMA
  Block1
#else
  Block2
#endif
```

Developer has to derive at least two configurations to ensure that every line of code **even compiles!**

Make sure that the submitted code...

"8. has been carefully reviewed with respect to relevant KCONFIG combinations. This is very hard to get right with testing – brain-power pays off here."

Linux kernel patch submission checklist (Documentation/SubmitChecklist)
**The VAMPYR Driver for Static Checkers**

**Goal:** Maximize configuration coverage of existing tools
- Every configuration-conditional part should be covered at least once
- Statement coverage

⇒ Create a set of configurations and scan each individually

---

**Results with GCC as Static Checker**

**USENIX ‘14 [7]**

<table>
<thead>
<tr>
<th>Software Project</th>
<th>allyesconfig CC%</th>
<th>uniform CC%</th>
<th>Overhead: increase of GCC invocations</th>
<th>GCC warnings</th>
<th>GCC errors</th>
<th>Σ Issues</th>
<th># Header blocks per reported issue (hit)</th>
<th>Result: increase of GCC messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux/i386</td>
<td>78.6%</td>
<td>88.4%</td>
<td>21.5%</td>
<td>203 (176)</td>
<td>1 (0)</td>
<td>215</td>
<td>351</td>
<td>202 (+15%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>26.7%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>82.7%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux/arm</td>
<td>59.9%</td>
<td>84.4%</td>
<td>22.7%</td>
<td>417 (294)</td>
<td>92 (35)</td>
<td>508</td>
<td>46</td>
<td>190 (+64%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>51.2%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>80.1%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10.5%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>37 (32)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux/ppc</td>
<td>54.5%</td>
<td>90.0%</td>
<td>22.0%</td>
<td>220 (187)</td>
<td>29 (1)</td>
<td>249</td>
<td>85</td>
<td>91 (+68%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>42.1%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>80.2%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10.5%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>37 (32)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux/mips</td>
<td>79.3%</td>
<td>90.5%</td>
<td>21.6%</td>
<td>174 (121)</td>
<td>17 (1)</td>
<td>191</td>
<td>72</td>
<td>69 (+57%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>79.3%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>21.2%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12 (12)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>46 (36)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L4/FIASCO</td>
<td>90.1%</td>
<td>99.9%</td>
<td>see test</td>
<td>20 (5)</td>
<td>1 (0)</td>
<td>21</td>
<td>see test</td>
<td>16 (+30%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>see test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Busybox</td>
<td>74.2%</td>
<td>97.3%</td>
<td>60.3%</td>
<td>44 (35)</td>
<td>0 (0)</td>
<td>44</td>
<td>72</td>
<td>9 (+26%)</td>
</tr>
</tbody>
</table>

Example: arch-arm
- Increased CC compared to allyesconfig from 60% to 84%
- 199 (+64%) additional issues reported by GCC
- 91 reported issues have to be considered as serious bugs
- 7 patches submitted – all got immediately accepted

Just by letting the compiler see all the code!
Idea: Automated Tailoring of Linux

- Distribution kernels today come with a **maximum** configuration.
- As side-effect, this maximizes the **attack** surface!
- Each use-case needs its specific, **ideal** configuration.

→ Automatically derive an **ideal** configuration for a given use-case.

Approach

- **Specific Scenario**
- **Tailored Configuration**

- **CONFIG_X86=y**
- **CONFIG_SCSI=n**

Tailored Configuration

- **FTRACE** observe debug symbols

Identify in Source Code

automatically derive

**Holistic Variability Model**

employ SAT checker

<table>
<thead>
<tr>
<th>Evaluation</th>
<th>Ubuntu 12.04 with Linux 3.2 kernel; two use cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Web server setup with Apache, MySQL, PHP (LAMP)</td>
<td></td>
</tr>
<tr>
<td>- Workstation setup with NFS (Desktop)</td>
<td></td>
</tr>
<tr>
<td>- Trace time: 15 min, running defined workload</td>
<td></td>
</tr>
<tr>
<td>- LAMP: Google Skipfish ~ 5377 unique kernel functions</td>
<td></td>
</tr>
<tr>
<td>- Desktop: iozone, bonnie++ ~ 6933 unique kernel functions</td>
<td></td>
</tr>
<tr>
<td>- Black and whitelist for manual tailoring</td>
<td></td>
</tr>
<tr>
<td>- <strong>Blacklist:</strong> CONFIG_FTRACE</td>
<td></td>
</tr>
<tr>
<td>- <strong>Whitelist:</strong> CONFIG_UNIX, CONFIG_PACKET, CONFIG_DEVMPFS, CONFIG_DEVMPFS_MOUNT, CONFIG_ATA.PIIX, CONFIG_SATA.AHCI, CONFIG_ATA.GENERIC, CONFIG_DRM.I915.MS, CONFIG_BLK.DEV.INITRD</td>
<td></td>
</tr>
</tbody>
</table>

| Options set to 'y' | 1,537 | 452 (29%) |
| Options set to 'm' | 3,142 | 43 (1%) |
| Compiled source files | 8,670 | 1,121 (13%) |

<table>
<thead>
<tr>
<th>Kernel size in Bytes</th>
<th>Baseline</th>
<th>LAMP</th>
<th>Workstation/NFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>9,933,860</td>
<td>4,228,235 (44%)</td>
<td>4,792,508 (48%)</td>
<td></td>
</tr>
<tr>
<td>62,987,539</td>
<td>2,139,642 (3%)</td>
<td>2,648,034 (4%)</td>
<td></td>
</tr>
<tr>
<td>1,353,281</td>
<td>492 (32%)</td>
<td>63 (2%)</td>
<td></td>
</tr>
<tr>
<td>1,121,423</td>
<td>63 (2%)</td>
<td>1,423 (18%)</td>
<td></td>
</tr>
</tbody>
</table>

Options set to 'm' 1,537 452 (29%) 492 (32%)

Compiled source files 8,670 1,121 (13%) 1,423 (18%)
Evaluation: Reduction for LAMP

- 90% less executable code
- 10% less functions with known vulnerabilities (with published CVE issues)

Results: Automatic Tailoring

- TCB is significantly smaller
- Easy to use: process is fully automated
- If necessary, the tailoring can be guided with whitelists and blacklists
- Going further: Dynamic ASR
  - Even if present: Who is allowed to call what → CFG analysis
  - At runtime: Block illegal invocations.

**No observable performance impact**
Summary

- Real-world system software offers thousands of features
- eCos: 1,250 features mostly induced by hardware!
- Linux: 12,000 features mostly induced by hardware!
- central declaration (ecosConfig, CONFIG)
- distributed, multi-paradigm implementation (MAKE, CPP, GCC, ...)

- This imposes great challenges for management and maintenance
  - how to ensure configurability consistency?
  - how to ensure configuration coverage?
  - how to keep pace with the constant feature increase?

- A strong call for adequate tool support VAMOS
  - already found thousands and fixed hundreds of defects and bugs
  - more to come!

Referenzen (Cont’d)


