Real Bugs
Real Projects
Real Impact

Andrzej Wąsowski

joint work with (lexicographically)
Iago Abal, Claus Brabrand
Jonathan Hechtbauer
Gijs van der Hoorn
Alexandru F. Iosif Lazar
Jean Melo, Marcio Ribeiro
Stefan Stanciulescu
Andre Santos, Chris Timperley
The ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS) has since 1998 covered all aspects of modeling, from languages and methods, to tools and applications. Attendees of MODELS come from diverse backgrounds, including researchers, academics, engineers and industrial professionals. MODELS 2018 is a forum for participants to exchange cutting-edge research results and innovative practical experiences around modeling and model-driven software and systems. This year’s edition will provide an opportunity for the modeling community to further advance the foundations of modeling, and come up with innovative applications of modeling in emerging areas of cyber-physical systems, embedded systems, socio-technical systems, cloud computing, big data, security, open source, and sustainability.

The Premier Conference on Model-Driven Engineering
Ariane V (1996)

A floating point cast bug, Throws overflow exception
A decade of development, $7B development budget, $0.5B lost rocket & cargo, but ...
Ariane V (2013)

- 98 launches since 1996
- 3 crashes since 1996
- Only the first linked to a software bug (Is HW really more reliable?)
- Most recent launch: **Apr 5th, 2018**
  Have you heard about it?
- They never show you this slide ...
1.27 fatality per 100 million miles

including human failures

0.76 fatality per 100 million miles

0.03 fatalities per 100 million miles

including human errors

If we are Doing so well, Why are we still SO OBSESSED with correctness?

© Andrzej Wasowski, IT University of Copenhagen
Lesson 1

Don’t drive your research by problems that are abstract (remote) for you
AGENDA

- (variability) Bugs in the Linux kernel
- Bugs in the Robot Operating System (ROS)
Bugs are beautiful and fascinating

- **You are not a V&V researcher** if you don’t touch real bugs and systems in your work.
- **Real data gives a rich research context**, enables a lot of fascinating work.
- Real **data does not exclude work in clean "lab conditions"** when appropriate. It supports it.
What is Linux Kernel?

Incredibly versatile operating system

GNU/Linux runs supercomputers and internet servers

68-98% webservers run on Linux

Android phones, tablets, smartTVs etc.

Cloud infrastructure

The most popular OS kernel on the planet!

Sources: Gartner and https://en.wikipedia.org/wiki/Usage_share_of_operating_systems
https://techcrunch.com/2016/11/16/microsoft-joins-the-linux-foundation/
Linux Kernel is very large

The source code has **700 million characters**, **21 million lines of code**
(quick measurements on the Raspberry Pi version of Linux)

Boeing 747 has **6 million mechanical parts**, half of them simple fasteners

Are humans able to understand the entire kernel?
Linux Kernel Moves Fast

- **4000 programmers** from **440 companies** contributed to the kernel (approximate numbers from 2015 only)
- **10,800** lines of code added, **5,300** removed, **1,875** modified Every. Single. Day. (on average)
- Over **8 changes per second**

- **Is any human able to comprehend this evolution speed?**

- Incidentally, this makes it **impossible to verify** with current state of the art
- Nobody has **access to all hardware** on which others work
- Each potentially breaks things for others
Let's indulge! Look! A bug!

Dereferencing uninitialized pointer causes Kernel crash

```c
void pts_sb_from_inode(struct inode * inode)
{
    #ifdef CONFIG_DEVPTS_MULTIPLE_INSTANCES
    if (inode->i_sb->s_magic == ... ) ... 
    #endif
}

void pty_close(struct tty_struct * tty)
{
    #ifdef CONFIG_UNIX98_PTY
    pts_sb_from_inode (tty->driver_data);
    #endif
}
...

    tty = kzalloc(sizeof (*tty), GFP_KERNEL);
    pty_close (tty)
}
```

Domain knowledge
- Data flow
- Inter-procedural data-flow
- Pointers
- Nested structs
- [real bug] cross compilation unit and subsystem
- [real bug] function pointers (pty_close)


Bug 7acf6cd, see http://vbdb.itu.dk/#bug/linux/7acf6cd
Let’s look at another bug

Control-flow

```c
extern int preempt_count;

void tcp_twsk_destructor() {
    #ifdef CONFIG_TCP_MD5SIG     // ENABLED
        preempt_count--;
    #endif
}

void inet_twdr_hangman(long data) {
    void (*fn)();        // function pointer
    fn = (void (*)(())) data;  // cast to funptr
    fn();                // dynamic invocation
}

void __run_timers() {
    long data = (long) &tcp_twsk_destructor;
    int pc = preempt_count;    // save
    inet_twdr_hangman(data);
    if (pc != preempt_count) BUG();  // check
}
```

- Unsafe casts help **generic programming** of data structures
- Type **casts**, pointers to **ints**; Do not loose **shapes** and **aliasing** info
- Function pointers used heavily (OO)
- Inter-procedural data-flow not possible without **control-flow**
- [elsewhere] **conditional struct components** (with incompatible casts)

Bug 657e964e74, http://vbdb.itu.dk/#bug/linux/7acf6cd
Lesson 2: Hunting bugs in software
Is for tough warriors
Not for those of faint heart

It’s not (only) about \( \lambda \)-calculus
or the tiny term grammar in Fig. 2, your paper
Warning!
You may get dirty
A closer look at a bug

index: kernel/git/stable/linux-stable.git

Linux kernel stable tree

<table>
<thead>
<tr>
<th>summary</th>
<th>refs</th>
<th>log</th>
<th>tree</th>
<th>commit</th>
<th>diff</th>
<th>stats</th>
</tr>
</thead>
</table>

author: Peter Hurley <peter@hurleysoftware.com> 2013-01-30 17:43:49 (GMT)
committer: Greg Kroah-Hartman <gregkh@linuxfoundation.org> 2013-02-04 23:40:28 (GMT)
commit: 7acf6cd80b201f77371a5374a786144153629be8 (patch)
tree: 5cd6e6e60b6873d3b749f6795f12df96
parent: 10539ae48c76f1ce9b970b719de602b77e5b8b67 (diff)

pty: Fix BUG() is when ptmx_open() errors out

If ptmx_open() fails to get a slave inode or fails the ptty_open(),
the tty is released as part of the error cleanup. As evidenced by the
first BUG stacktrace below, ptty_close() assumes that the linked pty has
a valid, initialized inode* stored in driver_data.

Also, as evidenced by the second BUG stacktrace below, pty_unix98_shutdown()
assumes that the master pty's driver_data has been initialized.

1) Fix the invalid assumption in pty_close().
2) Initialize driver_data immediately so proper devpts fs cleanup occurs.

Fixes this BUG:
[ 815.86844] BUG: unable to handle kernel NULL pointer dereference at 0000000000000028
[ 815.86916] IP: [fffffff81207bccc] devpts_ptty_kill+0x1c/0xa0
[ 815.86916] PGO 7c775867 PUD 7dbeb67 PDM 0
[ 815.869443] Modules linked in: kvm_intel kvm snd_hda_intel snd_hda_codec snd_hwdep snd_pcm snd_seq_midi

http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=7acf6cd80b201f77371a5374a786144153629be8
Dereferencing uninitialized pointer causes Kernel crash

During the initialization of a UNIX98 pseudo-terminal by ptmx_open, a tty_struct structure is allocated. But before its pointer field link->driver_data is properly initialized, ptmx_open will try to allocate an inode structure for the PTY slave. If this allocation fails, some cleanup code must be executed to free the already allocated resources. Namely, pty_close will be called to release the previously opened tty, and this eventually dereferences tty->link->driver_data, which is assumed to have been already initialized.

But fixed by commit 7acf6cd80b2
Parent commit tree here

See http://vbdb.itu.dk/, and add your own bugs
## Subject Systems

As of December 2015

<table>
<thead>
<tr>
<th>System</th>
<th>Domain</th>
<th>LOC</th>
<th>#Features</th>
<th>#Commits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marlin</td>
<td>3D-printer firmware</td>
<td>0.04M</td>
<td>821</td>
<td>2,783</td>
</tr>
<tr>
<td>BusyBox</td>
<td>UNIX utilities</td>
<td>0.20M</td>
<td>551</td>
<td>13,878</td>
</tr>
<tr>
<td>Apache</td>
<td>Web Server</td>
<td>0.20M</td>
<td>681</td>
<td>27,677</td>
</tr>
<tr>
<td>Linux kernel</td>
<td>Operating system</td>
<td>12.70M</td>
<td>14,295</td>
<td>448,314</td>
</tr>
</tbody>
</table>


42 variability bugs in the Linux kernel: A qualitative analysis. ASE 2014 + TOSEM’18

© Andrzej Wąsowski, IT University of Copenhagen
What do we see? Diversity!

**Linux**

<table>
<thead>
<tr>
<th>Errors</th>
<th>CWE ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory errors</td>
<td>15</td>
</tr>
<tr>
<td>null pointer dereference</td>
<td>476</td>
</tr>
<tr>
<td>buffer overflow</td>
<td>120</td>
</tr>
<tr>
<td>read out of bounds</td>
<td>125</td>
</tr>
<tr>
<td>insufficient memory</td>
<td>-</td>
</tr>
<tr>
<td>memory leak</td>
<td>401</td>
</tr>
<tr>
<td>use after free</td>
<td>416</td>
</tr>
<tr>
<td>write on read only</td>
<td>-</td>
</tr>
<tr>
<td>compiler warnings</td>
<td>8</td>
</tr>
<tr>
<td>uninitialized variable</td>
<td>457</td>
</tr>
<tr>
<td>unused function (dead code)</td>
<td>598</td>
</tr>
<tr>
<td>unused variable</td>
<td>563</td>
</tr>
<tr>
<td>void pointer dereference</td>
<td>-</td>
</tr>
<tr>
<td>type errors</td>
<td>7</td>
</tr>
<tr>
<td>undefined symbol</td>
<td>-</td>
</tr>
<tr>
<td>undeclared identifier</td>
<td>-</td>
</tr>
<tr>
<td>wrong number of args to function</td>
<td>-</td>
</tr>
<tr>
<td>assertion violations</td>
<td>7</td>
</tr>
<tr>
<td>fatal assertion violation</td>
<td>617</td>
</tr>
<tr>
<td>non-fatal assertion violation</td>
<td>617</td>
</tr>
<tr>
<td>API violations</td>
<td>2</td>
</tr>
<tr>
<td>Linux API contract violation</td>
<td>-</td>
</tr>
<tr>
<td>double lock</td>
<td>764</td>
</tr>
<tr>
<td>arithmetic errors</td>
<td>1</td>
</tr>
<tr>
<td>numeric truncation</td>
<td>197</td>
</tr>
</tbody>
</table>

**BusyBox**

<table>
<thead>
<tr>
<th>Errors</th>
<th>CWE ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory errors</td>
<td>4</td>
</tr>
<tr>
<td>null pointer dereference</td>
<td>476</td>
</tr>
<tr>
<td>memory leak</td>
<td>401</td>
</tr>
<tr>
<td>use after free</td>
<td>416</td>
</tr>
<tr>
<td>compiler warnings</td>
<td>6</td>
</tr>
<tr>
<td>unused variable</td>
<td>563</td>
</tr>
<tr>
<td>uninitialized variable</td>
<td>457</td>
</tr>
<tr>
<td>incompatible types</td>
<td>843</td>
</tr>
<tr>
<td>type errors</td>
<td>5</td>
</tr>
<tr>
<td>undefined symbol</td>
<td>-</td>
</tr>
<tr>
<td>undeclared identifier</td>
<td>-</td>
</tr>
<tr>
<td>logic errors</td>
<td>3</td>
</tr>
<tr>
<td>behavior violation</td>
<td>440</td>
</tr>
</tbody>
</table>

(© Andrzej Wasowski, IT University of Copenhagen 21)
Lesson 3

Other researchers will be glad if you help them avoiding dirt.

You will help research quality in your field.
AGENDA

- (variability) Bugs in the **Linux** kernel
- Bugs in the **Robot Operating System (ROS)**
Software engineering for robotics
Why is it so hard?

- Programs for robots are not input → output mappings
- Intelligence, planning, mapping, vision, proximity, kinematics
- Operating under uncertainty and lack of predictability
- Huge diversity: simple robots to very complex autonomous robots, difficult to generalize (no one-size fits all)
- HW abstractions easily fall short
- Complex systems made of many components parallel, distributed
- Reliability and safety requirements
- Complex vendor market (OEMs, component providers, integrators, end users)
... an **open-source**, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.
Does ROS matter?
Is ROS the OSS platform for robotics?

- In 2016: **1M+ unique page views** a month at wiki.ros.org
- Yearly interest growth 21%
- Biggest reception in USA and China (comparable share)
- In July 2016: **300K+ visits** to answers.ros.org (your support channel)
- 17 new questions a day, **21K+ questions answered**
- More than **100K+ unique IP addresses** downloading ROS packages in July 2016
- **4400 papers citing** the ROS report

Yes! Finally a pretty bug from ROS!

Context: argument processing code in main

Fix indexing beyond end of array #167

This ought to crash the node immediately, shouldn’t it?

- Access out of bounds!
- **Wait!** the element outside is zero, so NULL
- A printf-like fun gets NULL, accesses to crash!
- **Wait!** ROS_FATAL is a macro (call-by-name) → might not access
- Nested macros expand to ::ROS::CONSOLE::PRINT, several calls reach vsnprintf that accesses NULL and crashes
- **Wait!** In glibc vsnprintf typesets "(null)" for a NULL string, exits safely
- So no major bug! Just a misprinted error message.

https://github.com/ros/geometry2/pull/167/commits/1957a441092d0bd90f115924f54831657ac6161f

© Andrzej Wąsowski, IT University of Copenhagen
The first thing a ROS node `main` does is to **initialize the framework** (line 37)

- Includes **processing of framework arguments** (usually quite a few, never zero)
- Arguments `argc` and `argv` are passed by reference
- Processed arguments are removed
  - `argv` is **re-sorted** so that only non-framework arguments are in front
  - `argc` is **decreased** accordingly
- What does it mean? **There was never any access out of bounds!**
  (this is just a minor formatting error on a failing execution)

```
int main(int argc, char ** argv)
{
    //Initialize ROS
    ros::init(argc, argv,"static_transform_publisher", ros::init_options::AnonymousName);
    tf2_ros::StaticTransformBroadcaster broadcaster;
```

[https://github.com/ros/geometry2/pull/167/commits/1957a441092d0bd90f115924f54831657ac6161f](https://github.com/ros/geometry2/pull/167/commits/1957a441092d0bd90f115924f54831657ac6161f)
Lesson 4: Reproduce!

Andrzej Wąsowski @AndrzejWasowski · Mar 6

When you are a bug researcher, never claim that you understood what the bug really is, before you managed to actually reproduce the failure and see it yourself. The bug *is* more nuanced than you think it is. [credits to @zhoulaifu]
A dataset of 200 bugs in the Robot Operating System for BugZoo

- 119 commits
- 4 branches
- 0 releases
- 5 contributors

- care-o-bot: Updated Dockerfile and BugZoo files to use one-fork-many-branch model
- confidential: moved confidential bugs
- doc: Docs should go in 'doc' dir.
- geometry2: Updated Dockerfile and BugZoo files to use one-fork-many-branch model
- kobuki: Added kobuki b166c93 (#49)
- mavros: Removed L1 and L2, and made L3 the norm (fixes #17)
Lesson 1

Don’t drive your research by problems that are abstract (remote) for you

Lesson 2: Hunting bugs in software
Is for tough warriors
Not for those of faint heart
It’s not (only) about λ-calculus or the tiny term grammar in Fig. 2, your paper

Lesson 3

Other researchers will be glad if you help them avoiding dirt.
You will help research quality in your field.

Lesson 4: Reproduce!

Andrzej Wąsowski @AndrzejWasowski · Mar 6
When you are a bug researcher, never claim that you understood what the bug really is, before you managed to actually reproduce the failure and see it yourself. The bug "is" more nuanced than you think it is. [credits to @zhoululu]
Part 2: Bug Hunting
Is for tough warriors,
But even warriors need the right weapons
Real Problems will take you to interesting work

Sometimes not where you planned to go
Problem → Solution
Problem ← Understanding
Problem → solution: EBA bug finder
Problem ← understanding: Why variability bugs appear?
Problem ↔ research methods: How to collect bug data?
Another Bug: double lock

```c
void inode_get_rsv_space(struct inode *inode) {
    if (*) return;
    spin_lock(&inode->i_lock); // 2nd lock
    spin_unlock(&inode->i_lock);
}

void add_dquot_ref(struct inode *inode) {
    spin_lock(&inode->i_lock); // 1st lock
    if (*) {
        spin_unlock(&inode->i_lock);
        return;
    }
    inode_get_rsv_space(inode); // call
    spin_unlock(&inode->i_lock);
}
```

- Coccinelle matches patterns over traces
- Supports CPP, efficient,
- **Integrated** into the kernel build system
- Intra procedural, unaware of aliasing

Welcome to EBA!

Effect-Based Analyzer

```c
void inode_get_rsv_space(struct inode *inode) {
    if (*)(*) return;
    spin_lock(&inode->i_lock); // 2nd lock
    spin_unlock(&inode->i_lock);
}

void add_dquot_ref(struct inode *inode) {
    spin_lock(&inode->i_lock); // 1st lock
    if (*)(*) {
        spin_unlock(&inode->i_lock);
        return;
    }
    inode_get_rsv_space(inode); // call
    spin_unlock(&inode->i_lock);
}
```

- For this to work we need to know about **effects**
- And about memory objects (to detect **aliasing**)

See [http://iagoabal.eu/eba](http://iagoabal.eu/eba)
Shape term: $\text{ref}_{\rho'} \text{ptr ref}_\rho \perp$

Polymorphic in regions and shapes (abstraction)
Syntax

Used to show the type system

\[
\begin{align*}
\text{l-value expressions} \quad L & : \quad x \mid f \mid *E \\
\text{r-value expressions} \quad E & : \quad n \mid E_1 + E_2 \mid \text{if} \ (E_0) \ E_1 \ \text{else} \ E_2 \mid (T)E \\
& \quad \mid \ \text{new} \ x : T = E_1; E_2 \mid \ !L \mid \ &\&L \mid \ L_1 := E_2; E_3 \\
& \quad \mid \ \text{fun} \ T \ f(T_1 \ x_1, \cdots, T_n \ x_n) = E_1; E_2 \mid \ L_0(E_1, \cdots, E_n)
\end{align*}
\]

- We use a specialized language only for presentation
- The type system is formalized and implemented for CIL
- Entire C is translatable to CIL
Inference of Shapes & Effects

\[ \Gamma \vdash E : \text{Env} \times \text{Exp} \times \text{Shape} \times \text{Effect} \]

[FETCH]
\[ \begin{align*} 
\Gamma & \vdash_L L : \text{ref}_\rho Z & \& & \varphi \\
\Gamma & \vdash_E !L : Z \& \varphi \cup \{\text{read}_\rho\} 
\end{align*} \]

ASSIGN]
\[ \begin{align*} 
\Gamma & \vdash_L L : \text{ref}_\rho Z & \& & \varphi_1 \\
\Gamma & \vdash_E E_1 : Z \& \varphi_2 \\
\Gamma & \vdash_E E_2 : Z' \& \varphi_3 \\
\Gamma & \vdash_E L := E_1; E_2 : Z' \& \varphi_1 \cup \varphi_2 \cup \{\text{write}_\rho\} \cup \varphi_3 
\end{align*} \]

- Formalized and implemented for entire C
- Including spec. of selected kernel functions, e.g:

```
spin_lock:  ref_\rho_1 \text{ ptr ref}_\rho_2 \zeta \xrightarrow{\text{lock}_\rho_2} \perp
spin_unlock: ref_\rho_1 \text{ ptr ref}_\rho_2 \zeta \xrightarrow{\text{unlock}_\rho_2} \perp
```
Bug Pattern Definitions

- Formalize **bug patterns in CTL** with nominals over effects
- A simple **reachability checker** finds paths matching a formula
- E.g. **double lock** = lock, then take same lock again without unlocking

\[ \top \text{EU} (\text{lock}_\rho \land \text{EX} (\neg\text{unlock}_\rho \text{EU lock}_\rho)) \]

\[
\text{double free} \quad \top \text{EU} (\text{free}_\rho \land \text{EX} (\neg\text{alloc}_\rho \text{EU free}_\rho))
\]

\[
\text{memory leak} \quad \top \text{EU} (\text{alloc}_\rho \land \text{EX EG} \neg\text{free}_\rho)
\]

\[
\text{use before initialization} \quad \neg\text{init}_\rho \text{EU use}_\rho
\]

© Andrzej Wasowski, IT University of Copenhagen
Does this work?

Precision (false positives), new bugs

- **Nine thousand files** in drivers analyzed (you do get dirty!)
- **9 reports for 9K lines** is not a lot of noise
- Each reported **bug classified** as either a true or a false positive.
- Still a lot of work to **filter out false positives** (you get dirty!)
- You talk to devs: they want you to **fix bugs**! (you may get dirty!)
- We right now have **14 new bugs** and >=5 **fixed** in the Linux kernel project (some in the main tree already)

<table>
<thead>
<tr>
<th></th>
<th>EBA</th>
<th>SMATCH</th>
<th>COCCINELLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugs found</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>False positives</td>
<td>5</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>TIME (minutes)</td>
<td>23</td>
<td>16</td>
<td>2</td>
</tr>
</tbody>
</table>
Does this work?

Experimental Evaluation, **time** in seconds, **recall** on historical bugs

<table>
<thead>
<tr>
<th>hash ID</th>
<th>depth</th>
<th>E</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1173ff0</td>
<td>0</td>
<td>0.6</td>
<td>1.3</td>
<td>0.1</td>
</tr>
<tr>
<td>149a051</td>
<td>0</td>
<td>0.7</td>
<td>0.6</td>
<td>0.3</td>
</tr>
<tr>
<td>16da4b1</td>
<td>0</td>
<td>0.4</td>
<td>0.8</td>
<td>0.1</td>
</tr>
<tr>
<td>344e3c7</td>
<td>0</td>
<td>0.7</td>
<td>1.3</td>
<td>0.1</td>
</tr>
<tr>
<td>2904207</td>
<td>0</td>
<td>5.8</td>
<td>2.0</td>
<td>2.8</td>
</tr>
<tr>
<td>59a1264</td>
<td>0</td>
<td>0.2</td>
<td>0.6</td>
<td>0.1</td>
</tr>
<tr>
<td>5ad8b7d</td>
<td>0</td>
<td>0.6</td>
<td>3.4</td>
<td>0.1</td>
</tr>
<tr>
<td>8860168</td>
<td>0</td>
<td>0.7</td>
<td>1.0</td>
<td>0.1</td>
</tr>
<tr>
<td>a7eef88</td>
<td>0</td>
<td>0.6</td>
<td>1.2</td>
<td>0.2</td>
</tr>
<tr>
<td>b838396</td>
<td>0</td>
<td>3.3</td>
<td>2.8</td>
<td>1.1</td>
</tr>
<tr>
<td>ca9fe15</td>
<td>0</td>
<td>0.4</td>
<td>0.7</td>
<td>1.8</td>
</tr>
<tr>
<td>e1db4ce</td>
<td>0</td>
<td>0.4</td>
<td>1.1</td>
<td>0.2</td>
</tr>
<tr>
<td>e50fb58</td>
<td>0</td>
<td>0.5</td>
<td>0.9</td>
<td>0.1</td>
</tr>
<tr>
<td>023160b</td>
<td>0</td>
<td>1.0</td>
<td>2.6</td>
<td>0.1</td>
</tr>
<tr>
<td>09dc3cf</td>
<td>0</td>
<td>1.2</td>
<td>1.4</td>
<td>0.1</td>
</tr>
<tr>
<td>0adb237</td>
<td>0</td>
<td>1.1</td>
<td>1.5</td>
<td>0.2</td>
</tr>
<tr>
<td>0e6f989</td>
<td>0</td>
<td>0.4</td>
<td>1.0</td>
<td>0.3</td>
</tr>
</tbody>
</table>

(17 historical bugs, intra-proc, double-lock, in Linux kernel, biased against EBA)

<table>
<thead>
<tr>
<th>Bug</th>
<th>depth</th>
<th>E</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>00dff7</td>
<td>2</td>
<td>5.0</td>
<td>1.5</td>
<td>0.1</td>
</tr>
<tr>
<td>5c51543</td>
<td>2</td>
<td>2.3</td>
<td>1.5</td>
<td>0.3</td>
</tr>
<tr>
<td>b383141</td>
<td>2</td>
<td>0.4</td>
<td>2.9</td>
<td>0.3</td>
</tr>
<tr>
<td>1c81557</td>
<td>1</td>
<td>5.0</td>
<td>1.9</td>
<td>0.6</td>
</tr>
<tr>
<td>328be39</td>
<td>1</td>
<td>8.9</td>
<td>1.7</td>
<td>0.2</td>
</tr>
<tr>
<td>5a276fa</td>
<td>1</td>
<td>0.9</td>
<td>1.2</td>
<td>0.2</td>
</tr>
<tr>
<td>80ed72</td>
<td>1</td>
<td>6.9</td>
<td>2.1</td>
<td>0.7</td>
</tr>
<tr>
<td>872c782</td>
<td>1</td>
<td>1.7</td>
<td>2.8</td>
<td>1.9</td>
</tr>
<tr>
<td>d7e9711</td>
<td>1</td>
<td>21</td>
<td>1.3</td>
<td>2.7</td>
</tr>
</tbody>
</table>

(9 historical bugs, inter-proc, double-lock, in kernel, biased against EBA)

EBA
An effective bug finder for C

EBA is a prototype tool to find non-trivial resource manipulation bugs in C programs, at compile-time, and super-fast.

In its few months of existence, EBA has found several double-lock bugs in Linux 4.7–4.10 releases (i.e. in code that has passed code reviews). All the following bugs are caught by EBA in a matter of seconds:

- HSI: cmt_speech: Fix double spin_lock
- usb: gadget: pch_udc: reorder spin_[un]lock to avoid deadlock
- ath10k: fix deadlock while processing rx_in_ord_ind
- net: ethernet: t: cpdfma: fix lookup in cpdfma_ctxr_destroy()
- libceph: ceph_build_auth() doesn’t need ceph_auth_build_hello()
- [PATCH] Fix: scsi: megaraid: reduce the scope of pending-list lock to avoid double lock
- iommu/vt-d: Fix dead-locks in disable_dmarm_iommu() path
- Re: Potential double-lock BUG in drivers/tty/serial/sh-scl.c (Linux 4.9)
- Potential deadlock BUG in drivers/net/wireless/st/cw1200/sta.c (Linux 4.9)
- Potential deadlock BUG in Linux 4.9 drivers/dma/coh901318.c
- [PATCH] [media] pctv452e: fix double lock bug
- Potential double-lock BUG in drivers/infiniband/core/umem_odp.c (Linux 4.9-rc7)
- dmaengine: p330: fix double lock
- cros_ec: Fix deadlock when EC is not responsive at probe
We are hiring!
After all, lambdas, types, and model checking are useful for solving real problems. (but we wouldn’t know without trying on real problems)
AGENDA

- Problem → solution: EBA bug finder
- Problem ← understanding: Why variability bugs appear?
- Problem ↔ research methods: How to collect bug data?
What do we find in the variability bugs?

A quick extract

- Bugs appear in **unanticipated configurations**
  - The programmer did not think about other configurations
  - Essentially all 98 VBDB bugs appear to be such...

- Bugs may **involve non-locally defined features** (defined in other subsystems)
  - 30 out of 43 Linux bugs in VBDB have this feature

- The interviewed developer: cross-cutting features are a frequent source of problems; Developers are often **experts only in a particular subsystem**.

“Code cluttered with `ifdefs` is difficult to read and maintain. Don’t do it. Instead put your `ifdefs` in a header, and conditionally define ‘static inline’ functions or macros, which are used in the code.” [submitting patches]
An interesting pattern with negative conditions
(a presence conditions for a bug to trigger)

<table>
<thead>
<tr>
<th>some-enabled</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>21</td>
</tr>
<tr>
<td>a ∧ b</td>
<td>21</td>
</tr>
<tr>
<td>a ∧ b ∧ c</td>
<td>6</td>
</tr>
<tr>
<td>a ∧ b ∧ c ∧ d ∧ e</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>some-enabled-one-disabled</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬a</td>
<td>20</td>
</tr>
<tr>
<td>a ∧ ¬b</td>
<td>20</td>
</tr>
<tr>
<td>incl.: (a ∨ a') ∧ ¬b</td>
<td></td>
</tr>
<tr>
<td>a ∧ b ∧ ¬c</td>
<td>4</td>
</tr>
<tr>
<td>a ∧ b ∧ c ∧ d ∧ ¬e</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>other configurations</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬a ∧ ¬b</td>
<td>1</td>
</tr>
<tr>
<td>a ∧ ¬b ∧ ¬c</td>
<td>2</td>
</tr>
<tr>
<td>a ∧ ¬b ∧ ¬c ∧ ¬d ∧ ¬e</td>
<td></td>
</tr>
</tbody>
</table>

In Linux commit 60e233a5660 function `add_uevent_var` with `HOTPLUG` disabled overflows a buffer.

Originally we spinned this for testing and sampling.

**Second thought:** isn’t this a symptom of forgetting to “mentally” enable/disable a feature?

The kernel developer: you hardly can think of more than 5 involved configs (features) when coding, debugging or profiling.
What are the cognitive challenges?
This program contains a simple coding bug

How would you debug it?

Where is the bug?

What configurations contain the bug?

```java
import java.util.Random;

public class Http {
    String subject = null;
    int totalLength = 600;
    final int HTTP_UNAUTHORIZED = 401;
    final int HTTP_NOT_IMPLEMENTED = 501;
    String REQUEST_GET = "GET";

    public void sendHeaders(int responseNum) {
        int buf = 0;
        buf = totalLength - responseNum;
        subject = "response header";
        if (subject.isEmpty())
            subject = "Void response";
        System.out.println("Done");
    }

    private void handleIncoming(String requestType) {
        boolean http_unauthorized = new Random().nextBoolean();
        if (http_unauthorized)
            sendHeaders(HTTP_UNAUTHORIZED);
        if (!requestType.equals(REQUEST_GET))
            sendHeaders(HTTP_NOT_IMPLEMENTED);
    }

    public static void main(String[] args) {
        Http http = new Http();
        http.handleIncoming("POST");
    }
}
```
Controller Experiment I

RQs
How does the degree of variability affect ...
- ... the time of bug finding?
- ... the accuracy of bug finding?

Cross-over Design
- Programs: 3 programs from 3 systems
  - Linux, open source, 12MLOC/13K features
  - Busy Box, open source, 204KLOC/600 features
  - Best Lap, Commercial game, 15 KLOC
- Subjects: N=69 [31 × Msc+32 × Phd+6 × post-doc]
- Task: find the bug
- Metrics: time and accuracy
- Small, fit on screen—no scrolling (25-35 LOC)
- Bugs: uninitialized var, null ptr deref, assert violation
- Deactivate features in NO/LO version, keep the bug

Jean Melo, Claus Brabrand, Andrzej Wasowski.
How does the degree of variability affect bug finding? ICSE 2016
Your guess?

How does the time change from P-NO via LO to HI?
Bug Finding Time Increases Linearly with |F|

Variance of the time is amplified by variability

- Variability makes debugging harder but not terribly so
- Might explain why FOSD/SPLE works !!!
- Humans “reason family-based” (at least until |F|=3)
- We’ve got some innate ability for meta-programming!

I tried to keep all different paths in mind, but it was especially difficult with multiple colors [HI]

- Obvious consequence of the former, but meaningful
- Variability amplifies differences in bug finding competences

How does the degree of variability affect bug finding? ICSE 2016
Finding Variability Bugs is Easy
Linking them to configurations is harder

- Most developers correctly identify bugs regardless of the variability degree
- Many fail to identify the of erroneous configurations; give too few or too many (!)
- Precision decreases with increasing variability degree
- We can expect this to be harder in presence of constraints (feature models)
- Speculation: devs don’t think about configurability all the time; Afterthought, reversed staging
- We likely make the same mistakes when coding (vbdb!) as when debugging;
What’s in your head when you work with variability?
Variability through the eyes of the programmer. ICPC 2017
Controlled Experiment II

- **2 buggy programs**
  - 1 derived from Busy Box, 1 derived from Best Lap
  - Same programs as before, but using #ifdefs not colors
  - We wanted to see whether people look at #ifdefs

- **Same two bugs**
  - Null pointer dereference, assertion violation
  - Both bugs in HI (3 features) and NO (0 features) versions

- **N=20 subjects**, 7 BSc, 1 MSc, 7 PhD, 5 post-doc

- **Task**: What is the bug? where is the bug? and in which configurations it appears? **while we track your gaze**

- **Latin square**: subject solves two tasks order on different programs (randomized order and assignments)

- **No time limit** (effectively 4–12 minutes per task)

Analyzing the Eye Tracker Data

Eye tracker gives a **fixation sequence**: triples $(x, y, t)$ of locations and time stamps.
Observation: Variability appears to increase debugging time of the areas of the program that contain variability.

- **Time doubles** from no to hi for both programs
- Consistent with the previous study, but now for **#ifdefs not colors**
- **Heatmaps similar** (KL divergence), but there is a small shift

<table>
<thead>
<tr>
<th>area of interest</th>
<th>variability without</th>
<th>variability with</th>
<th>increase factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-9 fields</td>
<td>25 s</td>
<td>58 s</td>
<td>2.2 x</td>
</tr>
<tr>
<td>12-21 sendHeaders</td>
<td>83 s</td>
<td>120 s</td>
<td>1.9 x</td>
</tr>
<tr>
<td>23-33 handleIncoming</td>
<td>56 s</td>
<td>98 s</td>
<td>1.8 x</td>
</tr>
<tr>
<td>35-38 main</td>
<td>8.2 s</td>
<td>5.3 s</td>
<td>0.7 x</td>
</tr>
<tr>
<td>Σ: all four areas</td>
<td>153 s</td>
<td>281 s</td>
<td>1.8 x</td>
</tr>
</tbody>
</table>

**Observation**: Time increases for fragments without variability in proximity of code fragments that do contain variability.
Variability Intensifies Eye Movements (or Confusion)

Variability appears to increase the number of gaze transitions between definitions-usages for fields and call-returns for methods.

Could alternative code organization help?

(a) Without variability.

(b) With variability.

Fig. 10: Average number of gaze transitions (eye switches) between the different elements of program P.
Problem → solution: **EBA bug finder**

Problem ← understanding: **Why variability bugs appear?**

Problem ↔ research methods: **How to collect bug data?**
We need realistic bug benchmarks

Rule 1

- Most interesting questions about programs are undecidable.
- More theoretical misbehaviours than what occurs in real systems
- Bugs not caused by meticulously contrived computations and circumstances.
- But by simple misconceptions: omissions, misspellings, confusion, miscommunication, misunderstandings, misusage of a library, or simply lack of information about the intended behavior of the system
- Human cognition functions that determine the errors
- Historical bugs approximate problems introduced by human cognition
- Need benchmarks reflecting real problems to guide research
Selection of bugs should be unbiased

Rule 2

- Avoid **sampling bias**.
- You can limit the **bug category** (ROS bugs, Linux bugs, variability bugs, concurrency bugs, etc.)
- But do **question correctness of sampling within the category**
- Do the bugs collected represent **anything more than your collection**
- Using a particular **tool introduces bias**
Reproducible bugs, reproducible benchmarks!

Rule 3

- It should be possible for another researcher **to recreate a reasonably similar benchmark** by following your method
- **Each bug** should be reproducible
- Recall what are the **risks** of misunderstanding the bug
- **Hard** to achieve for flaky non-testable bugs (concurrency!), bugs relying on hardware that you don’t have etc. For instance:
  - Robotics is diverse: actuators, sensors, control, distribution, communication, planning, simulation and visualization, diagnostics, perception (incl. object and collision detection), HRI (human-robot interaction), SLaM (Simultaneous Localization and Mapping), artificial intelligence
  - **Specialist skills required!**
  - Robotics software depends on hardware and a physical environment.
  - hardware might be unavailable, intermittent environmental conditions irreproducible
- Consequently, not all historical bugs will be reproducible
Restoration of buggy version of the system code

Rule 4

- Get it from the code repository (git)
- Use the **time of reporting or fixing** the bug (travel in git history)
  - The bug has likely been fixed since it was reported which means that it can be **only reproduced on an older snapshot**
  - ROS is a **moving target** with ever changing properties which means that the newest ROS version is likely to prevent reproduction in a **repeatable** manner.

- The problem is however not only getting the file with the bug at the right time. **You need the entire system source code.**
- in Robust, we obtain the **entire source distribution** of ROS from a given point in time for each bug (compute only dependencies of the buggy package; credits: rosinstall generator)
- Really irritating to hit **more than one bug** in this snapshot (for instance a build problem prevents reproducing a dynamic problem)
Restoration of historical development ecosystem

Rule 5

- Compilers and interpreters (for all the languages), runtime library, build system, operating system and all dependent distribution libraries.
- In ROBUST we use docker containers (one per bug) in which the environment for a bug is re-established.
- We use bugzoo to manage the containers uniformly.
- Code repositories and branches disappear.
- In ROBUST we fork all involved repositories (for the source of buggy packages).
  We store all dependencies in a docker container and store the container on dockerhub.
- We keep a redundant copy at the university.

https://github.com/squareslab/BugZoo
Facilitate automatic test reproduction

Rule 6

- In the fork we develop a test case (bug witness, regression test)
- Make available both in the textbfix and the broken branch

<table>
<thead>
<tr>
<th>buggy code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C )</td>
</tr>
<tr>
<td>(code ( C ), with the bug)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>test case, ( \varphi ):</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C \neq \phi )</td>
</tr>
<tr>
<td>(code ( C ) fails test case ( \varphi ))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fixed code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C' = \text{fix}(C') )</td>
</tr>
<tr>
<td>(code ( C' ), without the bug)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>test case, ( \varphi ):</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C' = \phi )</td>
</tr>
<tr>
<td>(code ( C' ) passes test case ( \varphi ))</td>
</tr>
</tbody>
</table>

- We provide scripts to manipulate the container state: build, test, fix/unfix
- The test case has to be non-intrusive
Add the test-case non-invasively

Rule 7

- The test case **contaminates** the original historic source
- Because we require realism, the **invasion shall be minimal**
- Modifying existing code should be avoided (but often impossible)
- The legacy code often exhibits **bugs outside the testable surfaces**
- We use a number of **patterns** to minimize the invasion:
  - Inject short **assertions** (if the property cannot be tested on an output of a function),
  - **Determinize** control-flow (if the bug is not reproducible with decent probability),
  - **Mock hardware** components with software, etc.
Bugs are deeply embedded in **intricate** functionality, architecture, and other idiosyncratic aspects of the subject system.

This creates a very **high entry barrier for researchers** and **inhibits usefulness** of a benchmark.

Meta-data lets the users **understand the problem fast**

ROBUST and VBDB record detailed meta-data as **human-readable descriptions** to facilitate this usage
Benchmark design rules

1. We need realistic benchmarks
2. Selection of bugs for a benchmark should be unbiased
3. Make benchmarks reproducible, and reproduce bugs in them
4. Restore historical system source code
5. Restore historical development ecosystem
6. Facilitate automatic test reproduction
7. Add the regression test possibly non-invasively
8. Document context meta-data
Problem → solution: EBA bug finder
Problem ← understanding: Why variability bugs appear?
Problem ↔ research methods: How to collect bug data?
Donald E. Stokes. *Pasteur’s Quadrant: Basic Science and Technological Innovation.*