Live Kernel Patching

DSU applied to the Linux Kernel

Matthias G. Eckermann
Senior Product Manager
mge@suse.com

SLAC 2015
2015-06-24 13:35 UTC

Why live patching?

Why live patching?

- Huge cost of downtime:
  - Hourly cost >$100K for 95% Enterprises – ITIC
  - $250K - $350K for a day in a worldwide manufacturing firm - TechTarget
- The goal is clear: **reduce planned downtime.**

Why Live Patching?

Change Management

Common tiers of change management

1. Incident response
   "We are down, actively exploited …"

SUSE

Why Live Patching?

Change Management

Common tiers of change management

1. Incident response

   “We are down, actively exploited …”

2. Emergency change

   “We could go down, are vulnerable …”

SUSE

Why Live Patching?

Change Management

Common tiers of change management

1. Incident response
   "We are down, actively exploited …"

2. Emergency change
   "We could go down, are vulnerable …"

3. Scheduled change
   "Time is not critical, we keep safe"

SUSE

Why Live Patching?

Change Management

Common tiers of change management

1. Incident response
   "We are down, actively exploited …"

2. Emergency change
   "We could go down, are vulnerable …"

3. Scheduled change
   "Time is not critical, we keep safe"

} Kernel Live Patching (1–2)

SUSE

Barcelona Supercomputing Centre

Mare Nostrum supercomputer

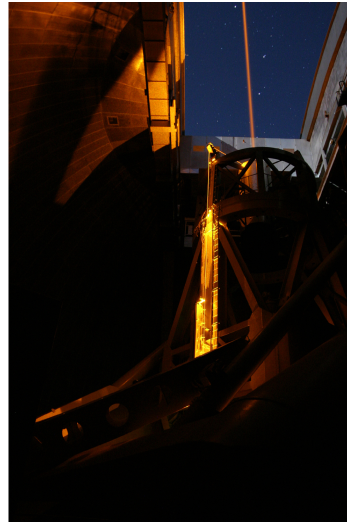
© BSC

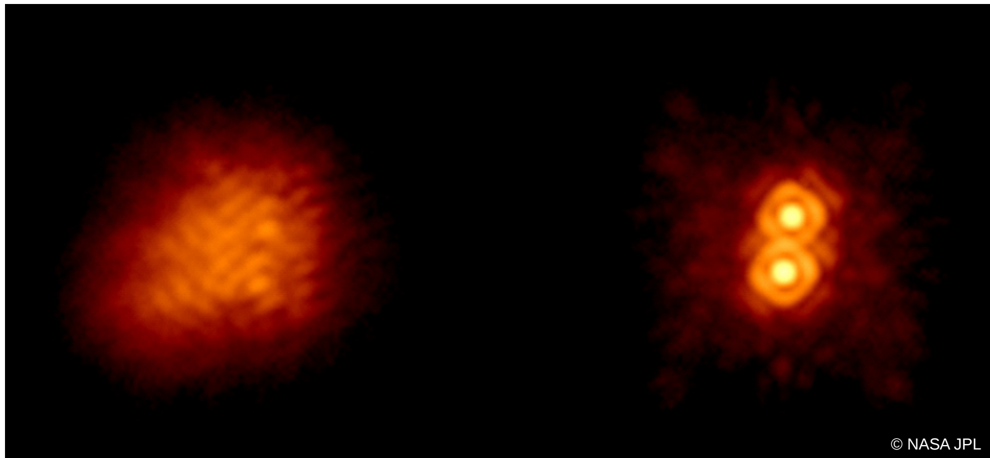
- 50k Sandy Bridge cores
- The most beautiful supercomputer in the world
- Terabytes of data
- Reboot?

SUSE

NASA JPL

Hale telescope PALM-3000 Adaptive optics





- 5m telescope with adaptive optics on Mount Palomar
- Avoid atmospheric blurring in Real Time
- Control 3888 segments of a deformable mirror with a latency <250 µs
- Reboot?

SAP HANA

In-memory database and analytics engine

© HP

HP DL980 w/ 12 TB RAM

- 4-16 TB of RAM
- All operations done in memory
- Disk used for journalling
- Active-Passive HA
- Failover measured in seconds
- Reboot?

SUSE

History of DSU

1943: Manhattan project – punchcards

- IBM punchcard automatic calculators were used to crunch the numbers
- A month before the Trinity nuclear device test, the question was: “What will the yield be, how much energy will be released?”
- The calculation would normally take three months to complete – recalculating any batches with errors
- Multiple colored punch cards introduced to fix errors in calculations while the calculator was running

Image in public domain

- Trinity test site, 16ms after initiation

Modern history of DSU: C language

- DSU: Dynamic Software Updates
  - the goal is to be able to fix bugs and add features
  - either by changing some functions
  - or replacing the whole program

- Let's focus only on C
  - the Linux kernel is (mostly) in C
  - all the major techniques were developed for C
  - C most closely matches the system ABI

1991-1993: PoDUS (University of Michigan)

- The first DSU to work on C in Berkeley Unix
- Uses binary *overwriting* of code *segments*
- The first to include *Activeness Safety*
  - functions are only changed when not running or on stack
- No state format changes allowed

Segment overwriting

Activeness Safety

1994: Deepak Gupta's DSU (IIT)

- Proved that the safety of applying an update is undecidable in general.
  - by reduction to the halting problem
- Replaces whole program with a new version
- Introduces *State Transfer*
  - no state transformation yet, no state format changes allowed

Whole program replacement

State Transfer

1998: Erlang (Ericsson)

- Not C, an own language, with DSU built-in
- Replacing functions on the fly
- Relies on the programmer for safety
- The first *commercially deployed* DSU
  - widely deployed in telecommunications systems

Commercially deployed

2006: Ginseng (U of Maryland, U of Cambridge, ETH Zurich)

- Introducing automated *patch generation*
- Uses *function indirection* and *lazy migration*
- Introducing *type safety*
  - Decides which functions to call based on matching data types

Patch generation

Type safety

Function indirection

Lazy migration

2008: Ksplice (MIT, Oracle)

- First to patch *Linux kernel*
- Stops kernel execution for activeness check
  - restarts and tries again later when active
- Uses jumps patched into functions for redirection
  - solves the call by pointer problem

Commercially deployed

Kernel patching

Activeness safety

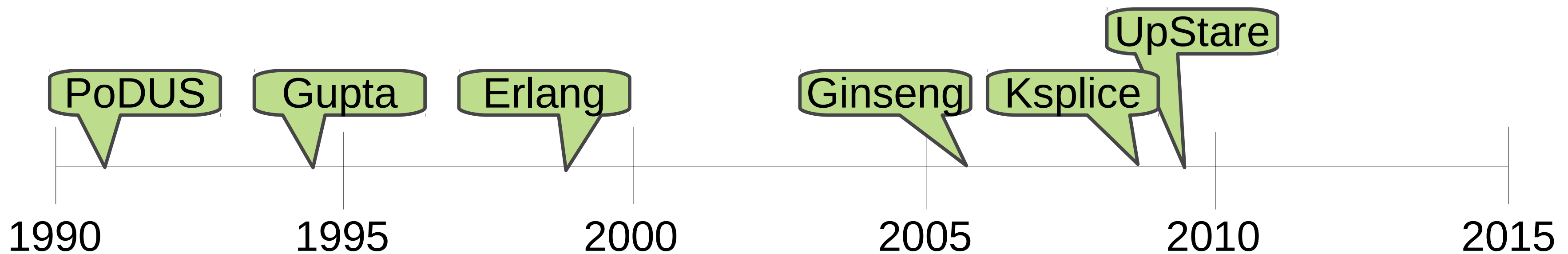
Binary patching

2009: UpStare **(Arizona State University)**

- Introduces *Stack reconstruction*
  - rebuilds stacks to match the new software
- *Immediate patching*
  - no Activeness safety required

Stack reconstruction

Immediate patching

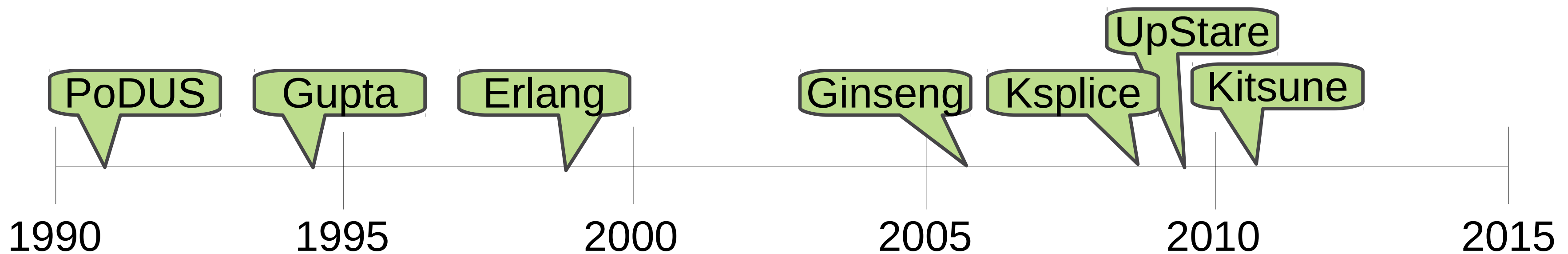


2011: Kitsune and Ekiden (University of Maryland)

- Introduces *State transformation*
  - transforms state to match the new software
- Uses controlled updating with *safe points*

State transformation

Safe points



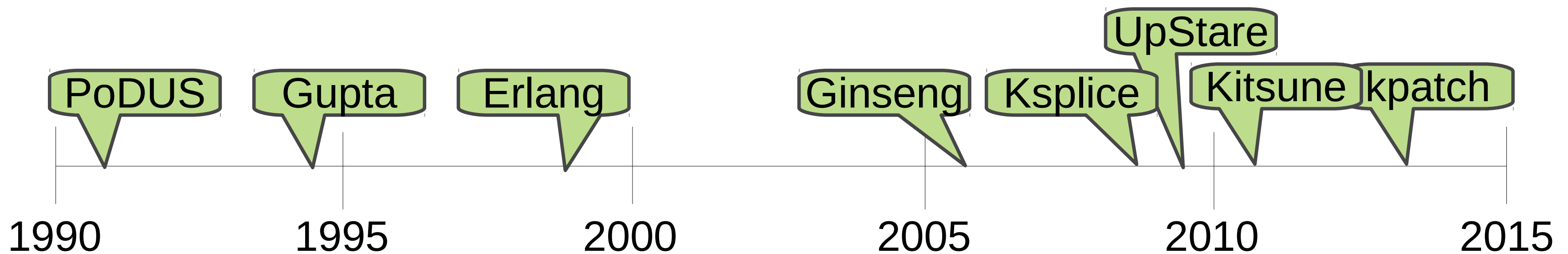
2014: kpatch (Red Hat)

- *Linux kernel* patching
- Originally uses the same consistency model as ksplice

Kernel patching

Activeness safety

Binary patching



2014: kGraft (SUSE)

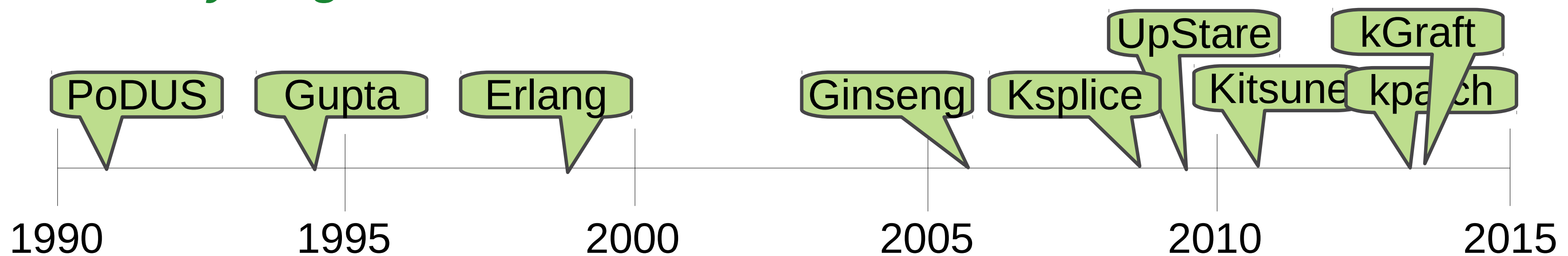
- *Linux kernel* patching
- *Immediate* patching with *lazy migration*
  - *Function type safety*
- *Commercially deployed*

Commercially deployed

Linux kernel

Immediate

Lazy migration



kGraft

kGraft goals

- Applying limited scope fixes to the *Linux kernel*
  - security, stability and corruption fixes
- Require *minimal changes* to the source code
  - no changes outside the kGraft engine itself
- Have no runtime *performance* impact
  - full speed of execution
- *No interruption* of applications while patching
  - full speed of execution
- Allow *full review* of patch source code
  - for accountability and security purposes

Patch Lifecycle

More Details

- Build
  - Identify changed function set
  - Expand set based on inlining and IPA/SRA compiler decisions
  - Extract functions from built image (or source code)
  - Create/adapt framework kernel module source code
  - Build kernel module
- Load
  - `insmod`
- Run
  - Address redirection using ftrace
  - Lazy per-thread migration

SUSE

Call Redirection

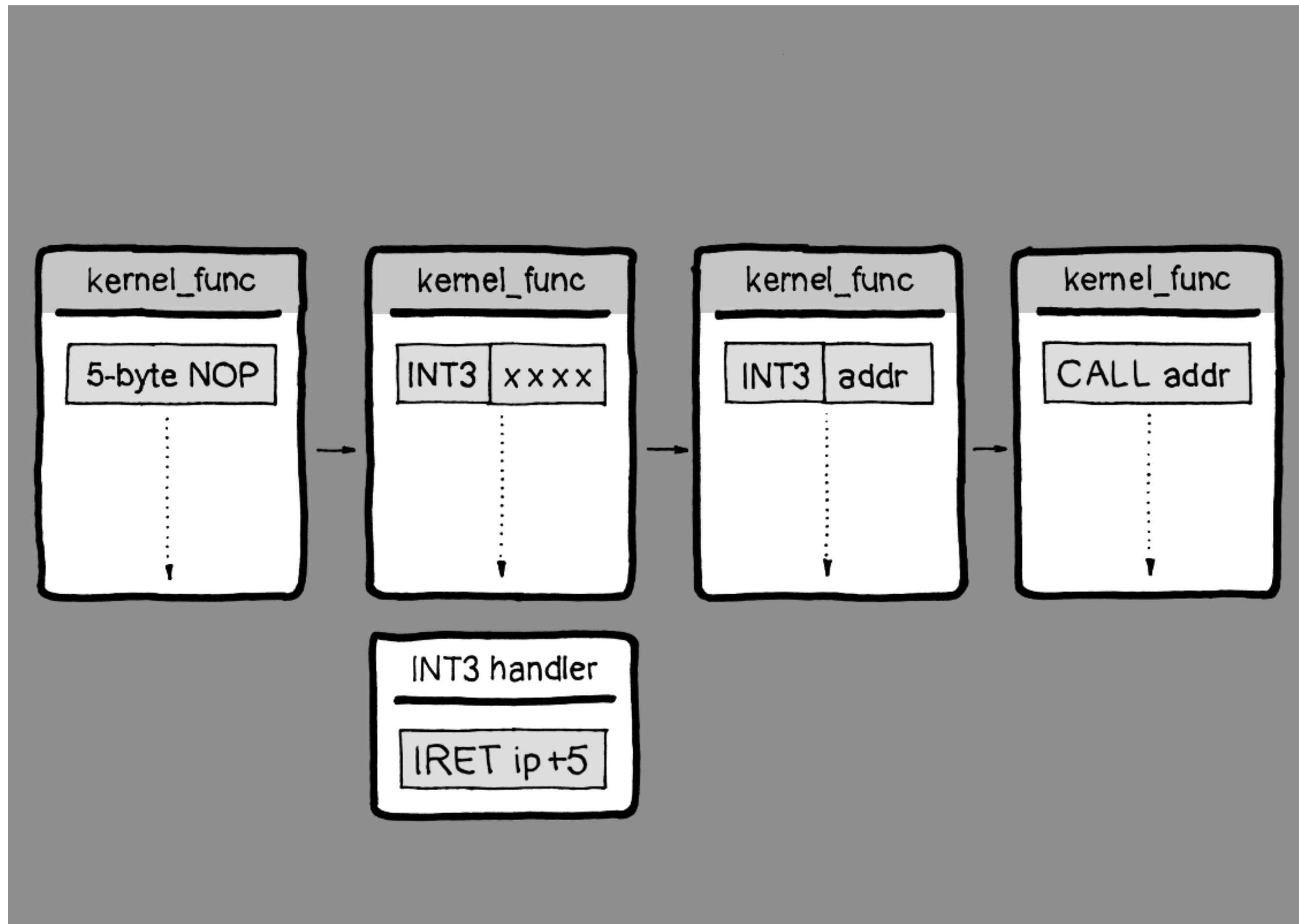
How Does It Work

- Use of **ftrace** framework
  - `gcc -pg` is used to generate calls to `_fentry_()` at the beginning of every function
  - ftrace replaces each of these calls with **`NOP`** during boot, removing runtime overhead
  - When a tracer registers with ftrace, the **`NOP`** is runtime patched to a **`CALL`** again
  - kGraft uses a tracer, too, but then asks ftrace to change the return address to the new function
  - And that's it, call is redirected

SUSE

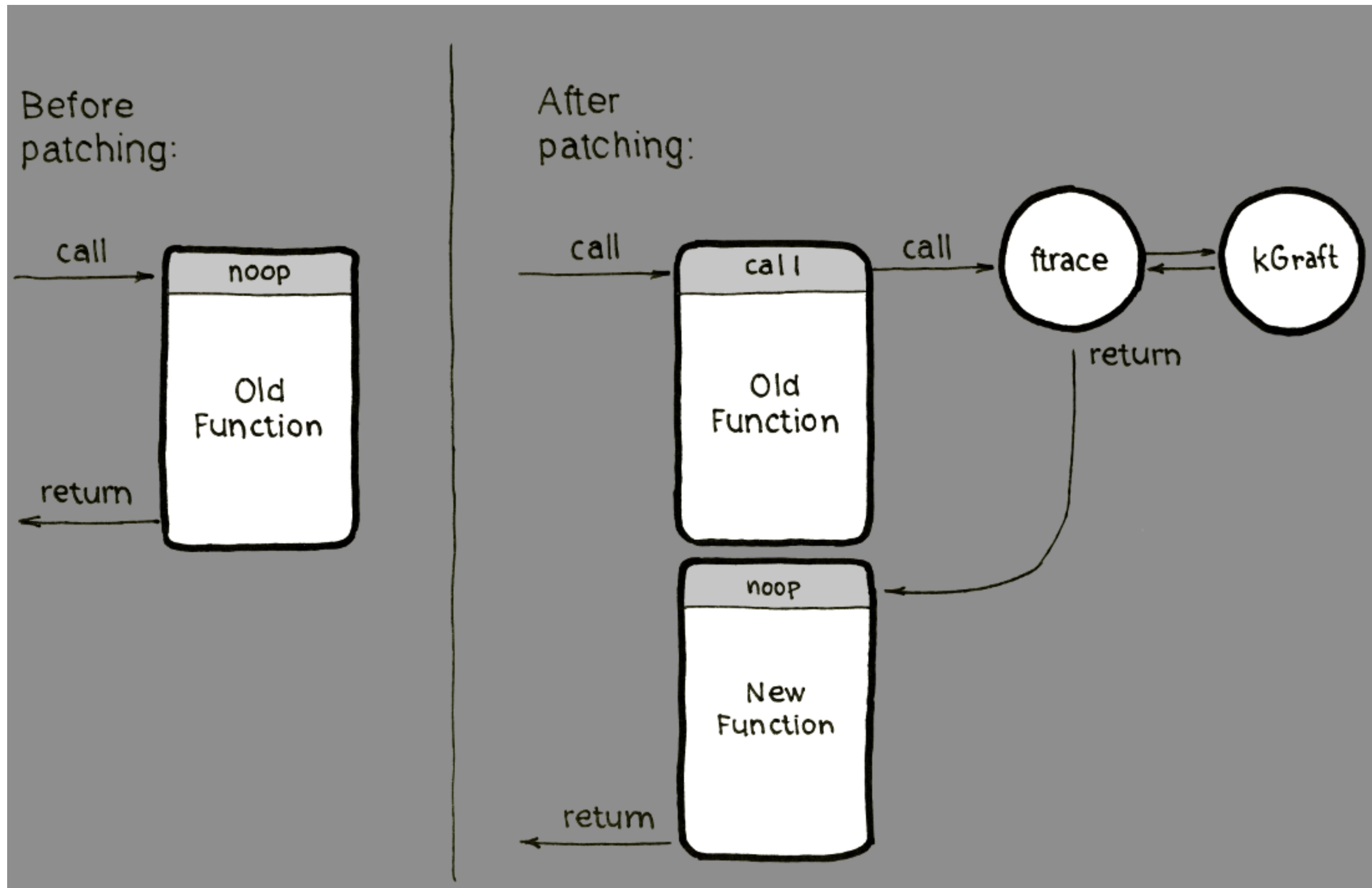
Call redirection

ftrace: SMP-safe code modification



Call Redirection

ftrace: return address modification mechanism



Call Redirection

The Final Hurdle

- Changing a single function is easy
  - since ftrace patches at runtime, you just flip the switch
- What if a patch contains multiple functions that *depend* on each other?
  - Number of arguments changes
  - Types of arguments change
  - Return type change
  - Or semantics change
- We need a **consistency model**
  - *Lazy migration* enforcing *function type safety*

SUSE

Consistency Models

Ksplice Consistency Model

Making a Clean Cut

- Ksplice uses *Activeness safety*
- First `stop_kernel();`
  - that stops all CPUs completely, including all applications
- Then, check all stacks, whether any thread is stopped within a patched function
- If yes, resume kernel and try again later
  - and hope it'll be better next time
- If not, flip the switch on all functions and resume the kernel
- The system may be stopped for 10-40ms typical

- Also implemented in the first version of kpatch

SUSE

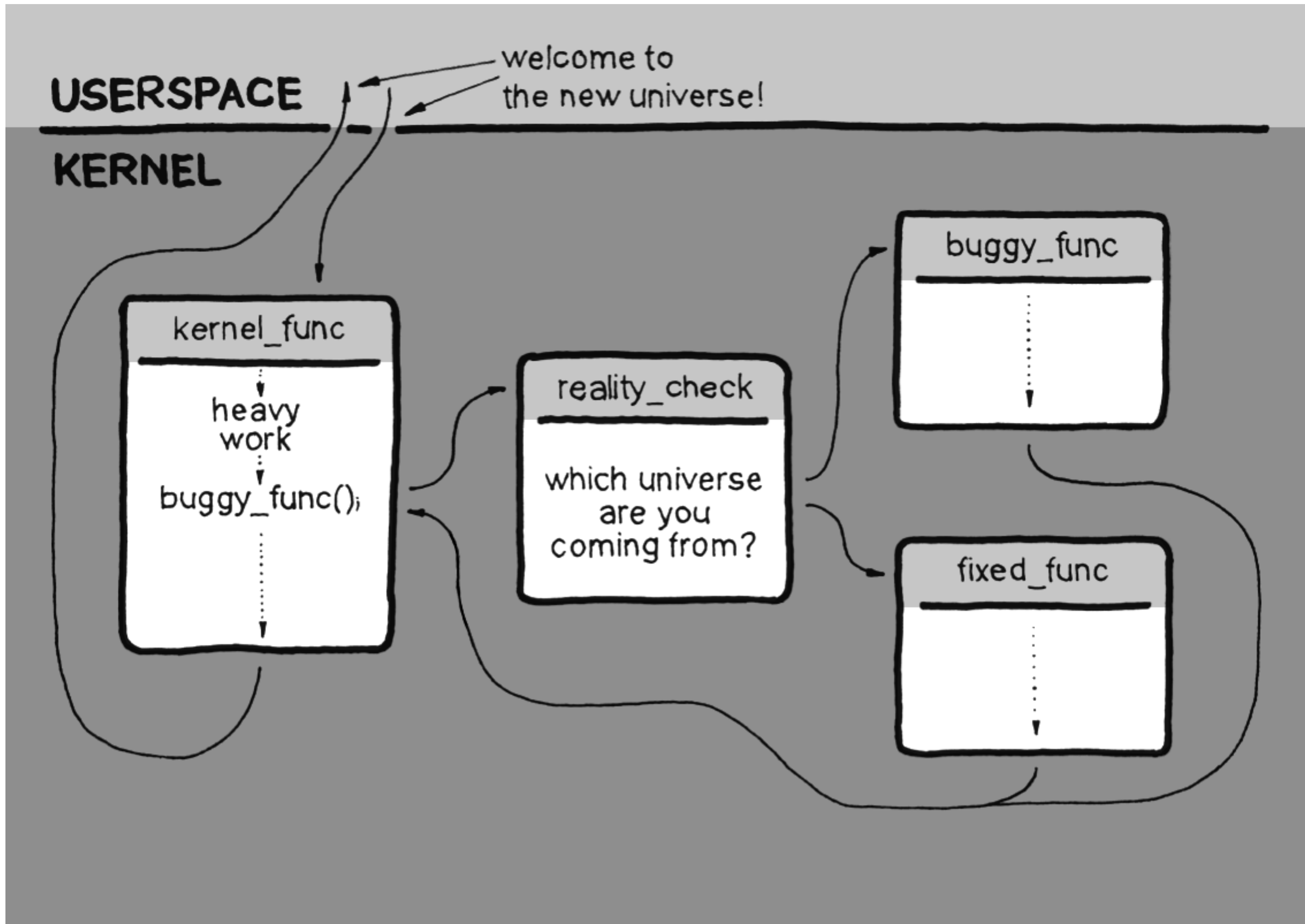
kGraft Consistency Model

Keeping Threads Intact

- We want to avoid calling a new function from old and vice versa: *Function type safety*
- Execution threads in kernel are of four types
  - interrupts (initiated by hardware, non-nesting)
  - user threads (enter kernel through SYSCALL)
  - kernel threads (infinite sleeping loops in kernel)
  - idle tasks (active when there is nothing else to do)
- We want to make sure a thread calls either all old functions or all new
- And we can migrate them one by one to 'new' as they enter/exit execution
- No stopping for anybody

SUSE

kGraft Consistency Model



SUSE

kGraft Consistency Model

Complications

- How about eternal sleepers?
  - like `getty` on a console 10
  - They'll never exit the kernel
  - They'll never be migrated to 'new'
  - They'll block completion of the patching process forever
- #1 Wake them up
  - sending a *fake signal* (SIGKGRAFT)
  - the signal exits the syscall and transparently restarts it
- #2 Just ignore them
  - once they wake up to do *anything,* they'll be migrated to 'new'
  - so they're not a security risk

SUSE

Upstream … or the battle for the best consistency model

- Ideas' exchange between engineers from
  - Hitachi
  - Red Hat
  - SUSE

- Original consistency models
  - kpatch (original = ksplice): "leave-set" + "switch-kernel"
  - kGraft (original): "leave-kernel" + "switch-thread"
  - kpatch (proposed): "leave-set" + "switch-thread"
  - kGraft (currently): "leave-kernel+signal"

- Proposed
  - leave-kernel+signal/switch-thread (kgraft)
  - leave-set/switch-thread model (kpatch-new)

Demo

Your questions?

Thank you.

Thanks

Vojtěch Pavlík & Team
Director SUSE Labs

Hannes Kühnemund
Product Manager SUSE Linux Enterprise Live Patching

Corporate Headquarters
Maxfeldstrasse 5
90409 Nuremberg
Germany

+49 911 740 53 0 (Worldwide)
www.suse.com

Join us on:
www.opensuse.org

Unpublished Work of SUSE LLC. All Rights Reserved.

This work is an unpublished work and contains confidential, proprietary and trade secret information of SUSE LLC. Access to this work is restricted to SUSE employees who have a need to know to perform tasks within the scope of their assignments. No part of this work may be practiced, performed, copied, distributed, revised, modified, translated, abridged, condensed, expanded, collected, or adapted without the prior written consent of SUSE.
Any use or exploitation of this work without authorization could subject the perpetrator to criminal and civil liability.

General Disclaimer

This document is not to be construed as a promise by any participating company to develop, deliver, or market a product. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. SUSE makes no representations or warranties with respect to the contents of this document, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. The development, release, and timing of features or functionality described for SUSE products remains at the sole discretion of SUSE. Further, SUSE reserves the right to revise this document and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes. All SUSE marks referenced in this presentation are trademarks or registered trademarks of Novell, Inc. in the United States and other countries. All third-party trademarks are the property of their respective owners.