September 2016 – BalCCon 2k16

Kernel Exploitation and Hardening
Why we could have nice things!
(using Split Kernel)

Anil Kurmus
kur@zurich.ibm.com
IBM Research - Zurich

Outline

1. **Background**
   Hardening
   Kernel Vulnerabilities
   Kernel Hardening
2. **Split Kernel**
   Overview
   Design
   Implementation
   Evaluation

A Program

```
F(A,B):
C ← A+B
C ← C*C
RET C
```

A Hardened Program

```
F(A,B):
C ← A+B
Are we doing OK?
C ← C*C
Are we doing OK?
RET C
```

What Makes A Good Hardening Feature?

Asking the right “are we doing OK” question:

- Mitigates many likely vulnerabilities (security)
- Taking few resources to answer (performance)
- Retrofitting and configuration is easy (usability)
- Not breaking the program (correctness)

Hardening Feature Examples

- SSP, Heap hardening, Format hardening, ...
- CFI, CPI, SafeStack, Softbound+CETS, ...

- **All mitigate some vulnerabilities**
- **All have some performance overhead**

➡ **Is hardening worth it?**

Is Hardening worth it?

Is Hardening worth it?

Easy-to-answer cases first?

A wall clock

Although...

The Great Seal Bug

An IoT Wall Clock

Let's recap

Cases that require hardening, cases that don't

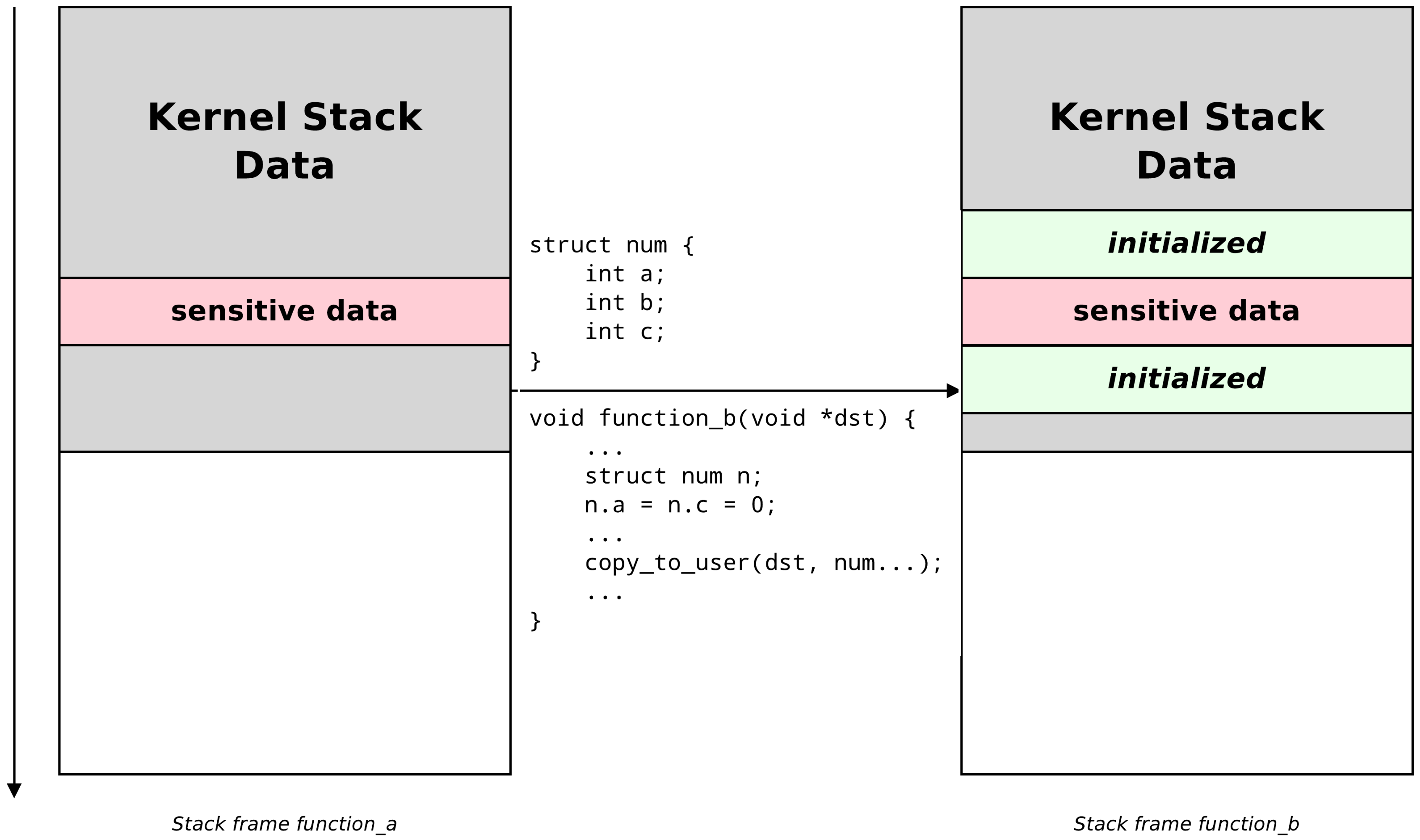
- Hard to tell at the development phase
- Better if configurable during deployment

Often a security vs. performance trade-off

Kernel Vulnerabilities

- A long standing issue...
  - USAF Study [Anderson 1972]
- … relevant to this day in practice
  - Windows kernel TrueType Font parsing (Duqu)
  - iOS jailbreaks (e.g., PEGASUS recently)
  - Linux SCTP remote exploit (sgrakkyu)
  - And many more!

Stack infoleak



Kernel stack clearance

- Zeroing the kernel stack at each syscall (PAX STACKLEAK)
  - Can be expensive, does not prevent all vulnerabilities
- Zeroing after each stack allocation
  - Even more expensive, but mitigates all stack missing initialization vulnerabilities
- Do we always have to pay this cost?

Example: OpenSSH

- The OpenSSH daemon is privilege separated

  [Provos et al., Sec'03]

  - The main daemon runs as root
  - A sandboxed process handles session establishment
- An attacker gaining code execution in the sandboxed process can escape via a kernel exploit

  → kernel hardening is beneficial
- An attacker gaining code execution in the main process has full access

  → kernel hardening causes unnecessary overhead

The best of both worlds

What if one could select kernel hardening at runtime, at no cost, in a granular way?

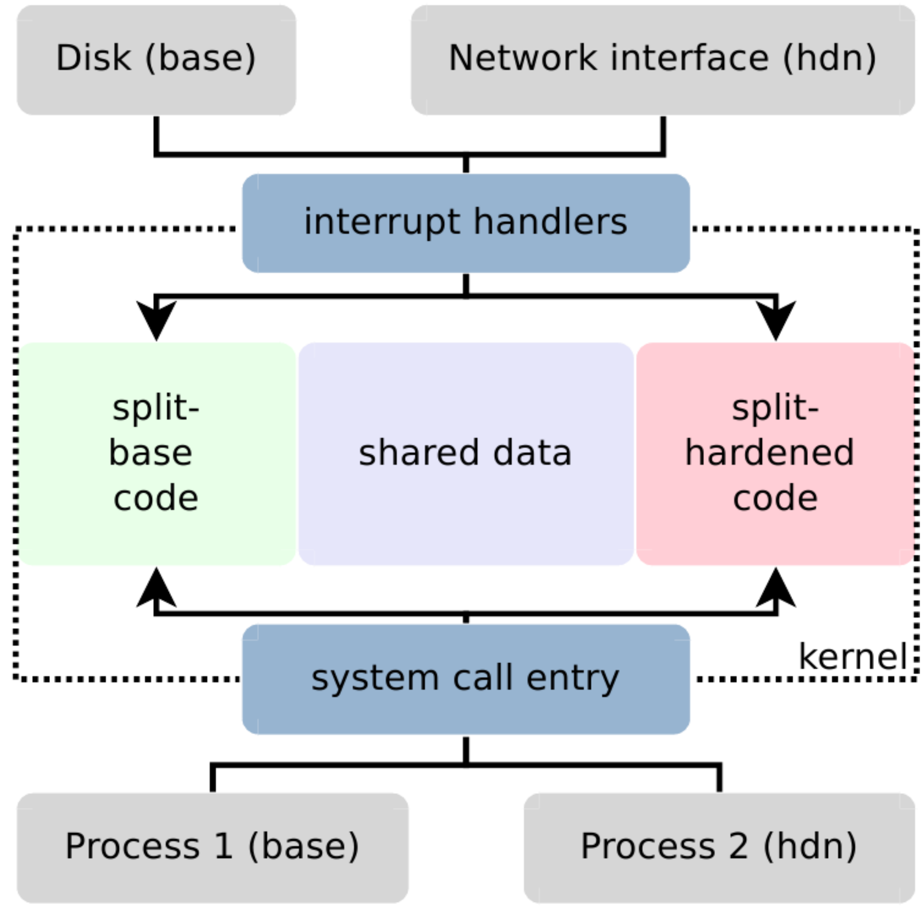
Split Kernel

Kurmus & Zippel, ACM CCS'14

http://static.securegoose.org/papers/ccs14.pdf

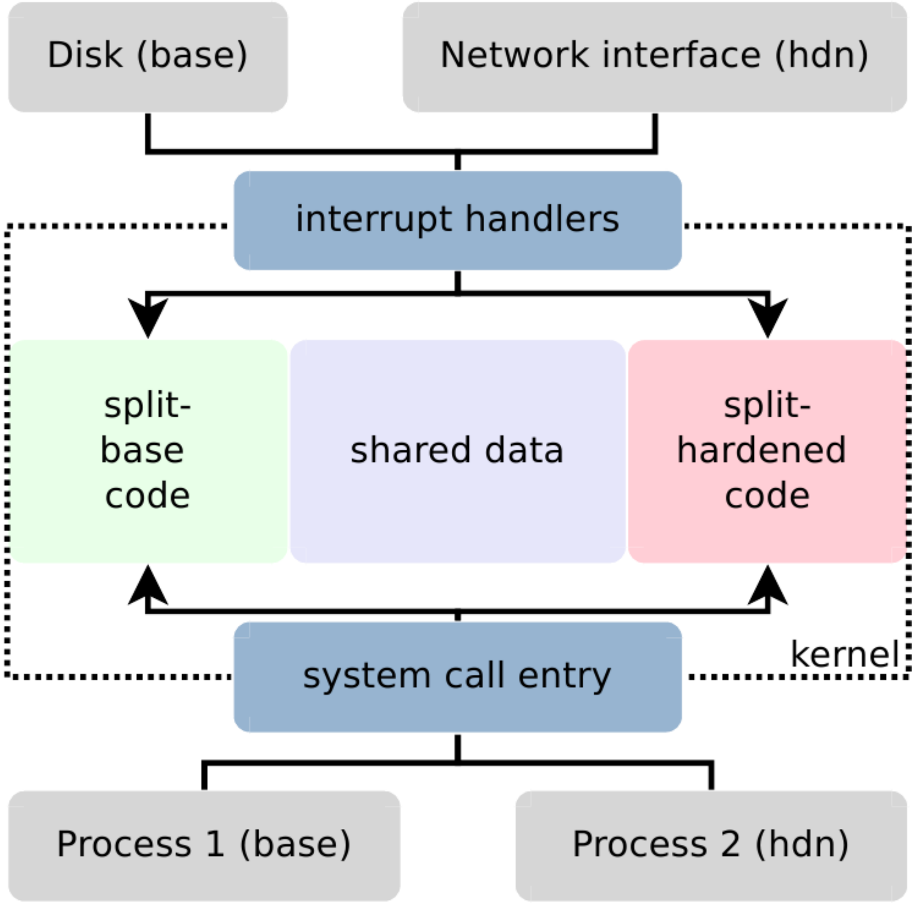
Overview

- Build kernel with and without hardening
- Chose at run-time whether to run in hardened mode
- Shared data enables switching safely between the two sets of kernel functions



Two modes, One kernel

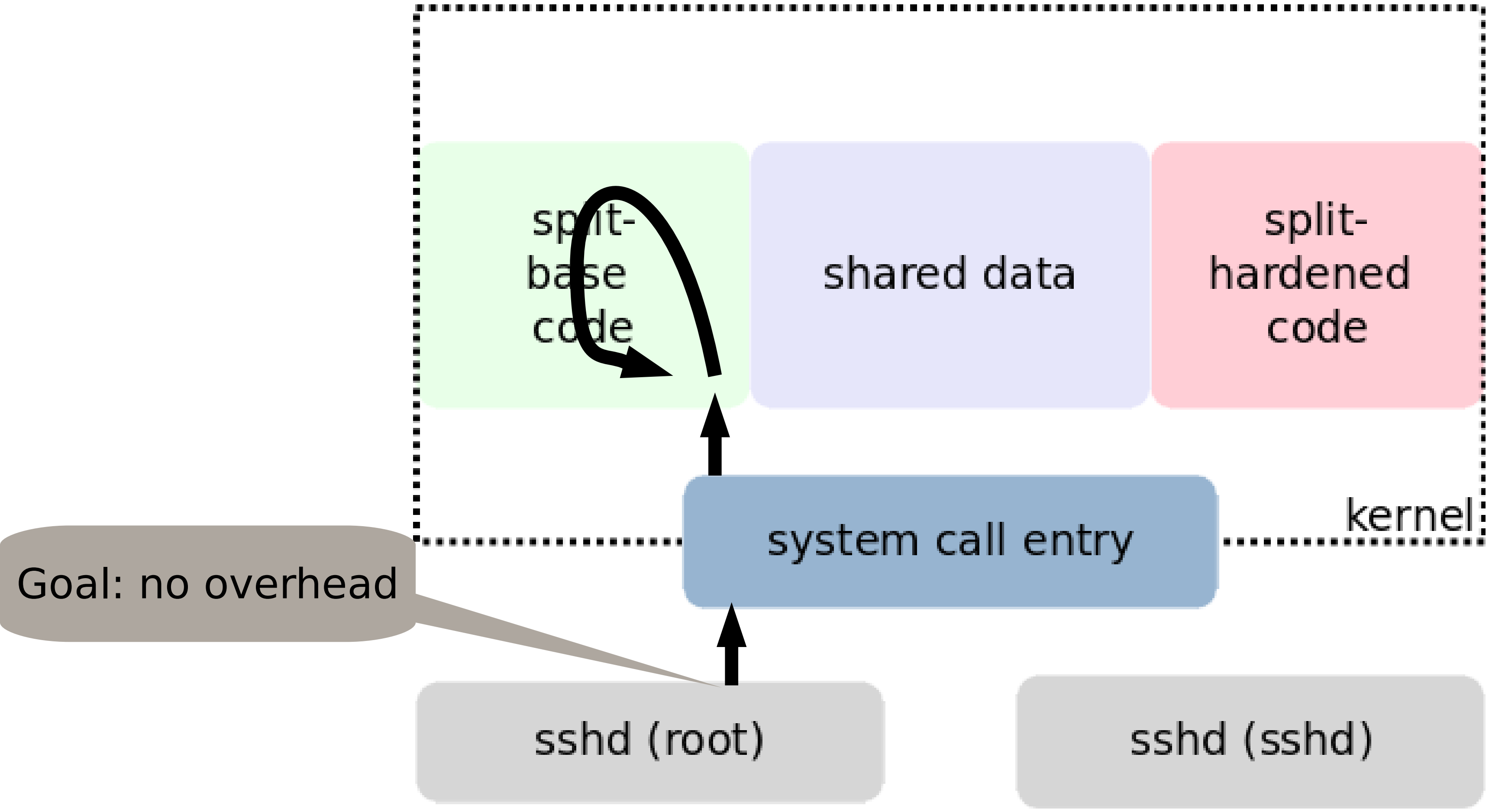
- If compromised in any of the two modes, compromised in both
- But: the kernel is less likely to be compromised in split-hardened mode



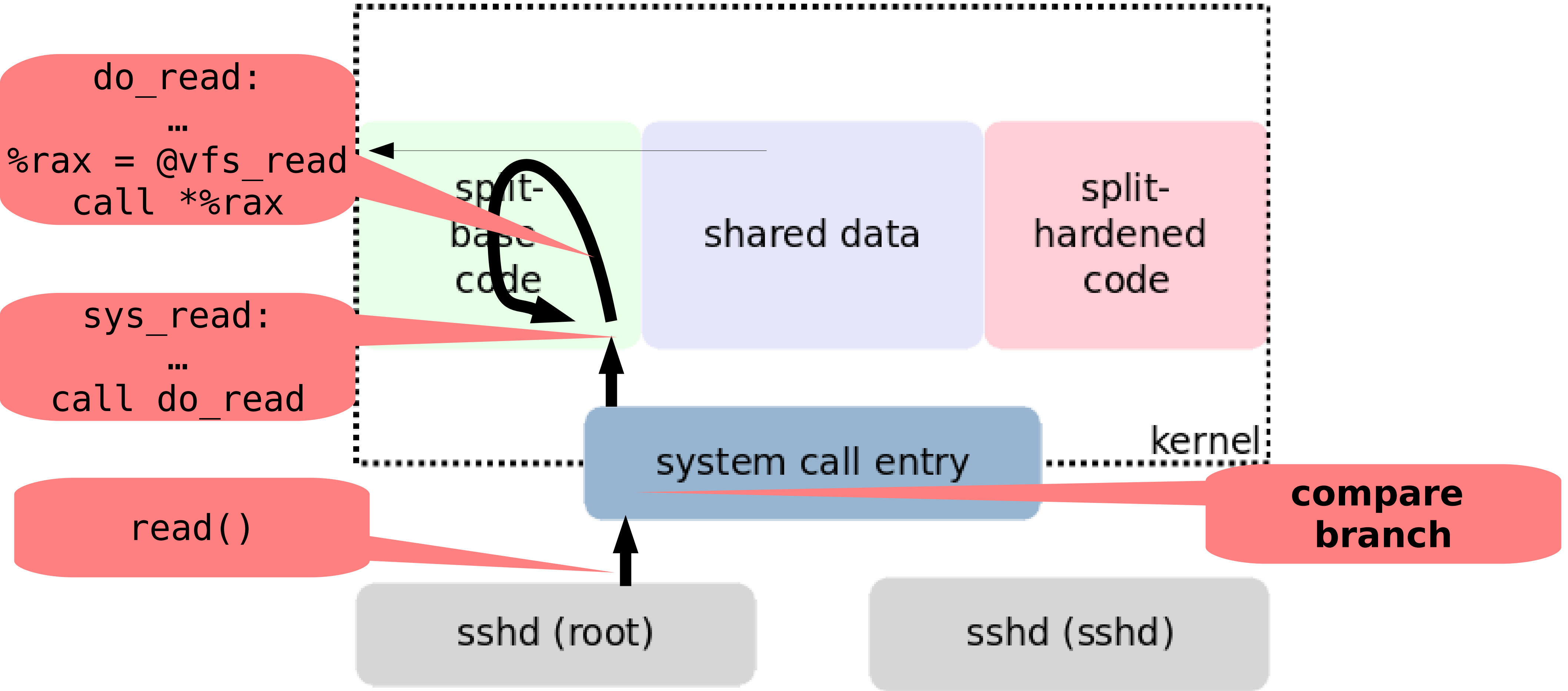
Design goals

- No split-base overhead
  → minimal changes to split-base code
- No control flow from split-hardened to split-base
  → need to instrument the code
- Run-time configurability
  → binding processes, users, interrupts, ...
- Maintainability
  → limit changes to the Linux kernel

Example: OpenSSH + Split Kernel

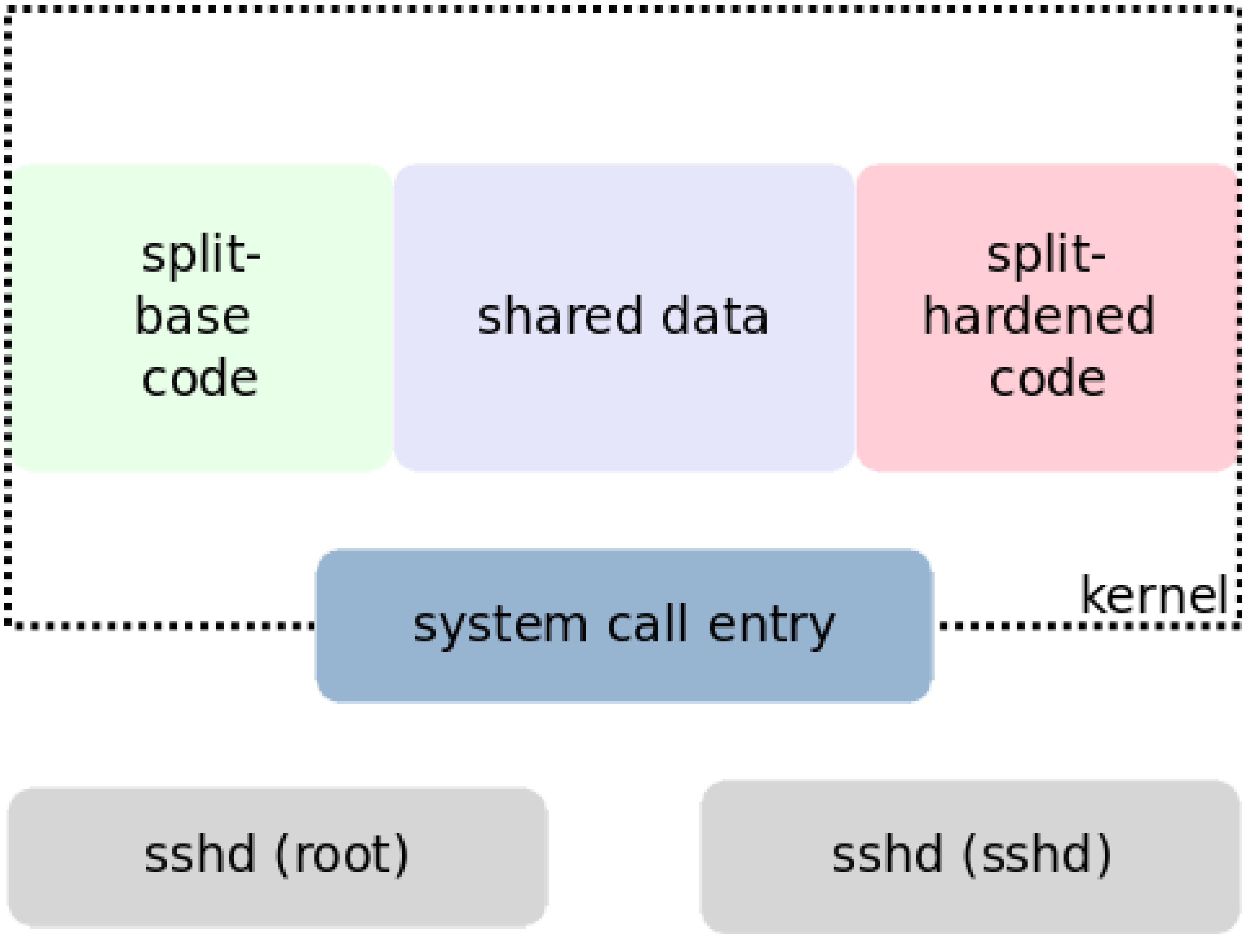


Low overhead split-base



Low split-base overhead by design

Split-hardened mode



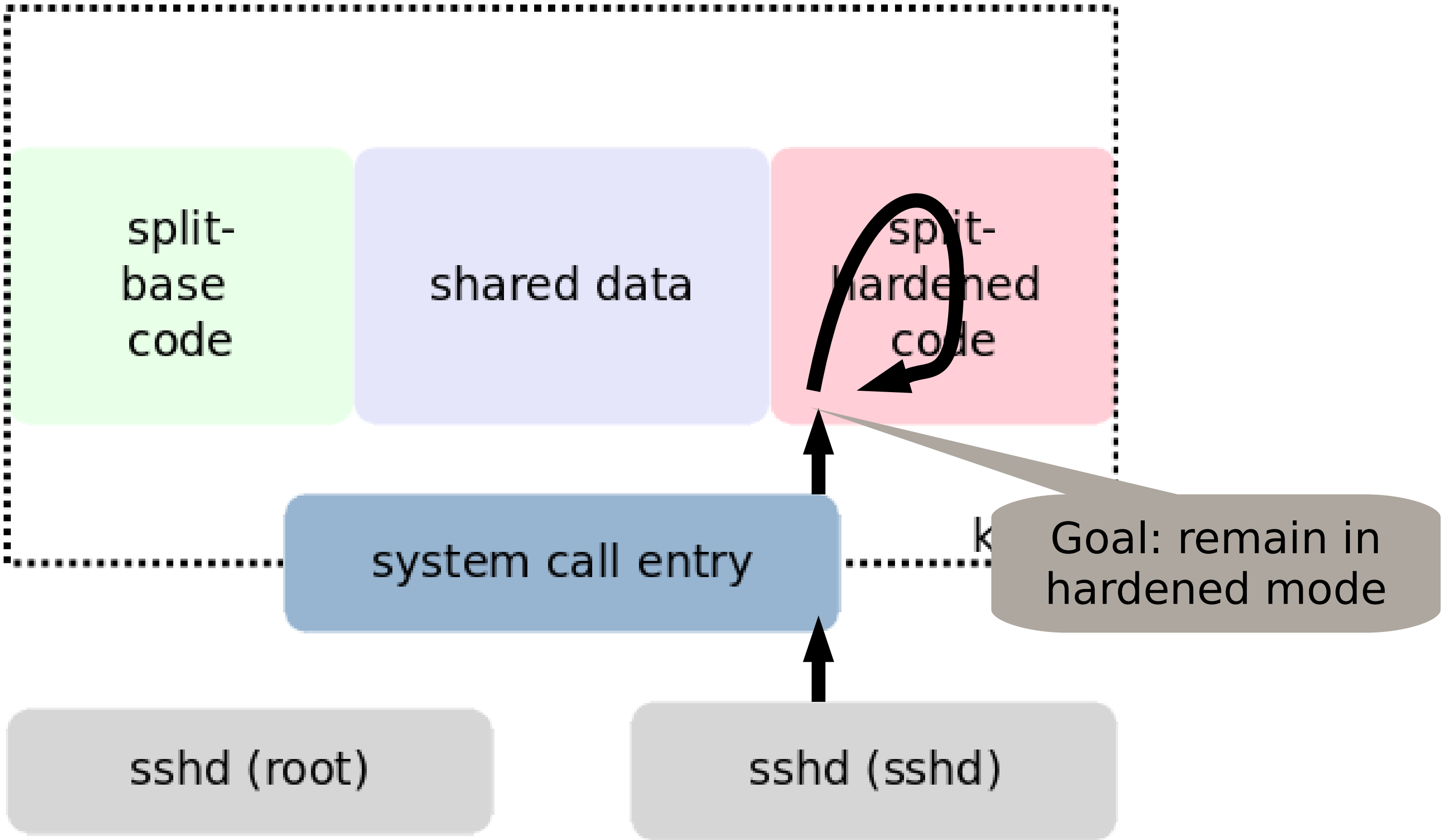
Binding sandboxed sshd to split-hardened mode

- Execute during boot scripts:

  id -u sshd >> /sys/kernel/split/hdn_uid_list

- No application source changes
- Similar: binding individual processes, interrupts

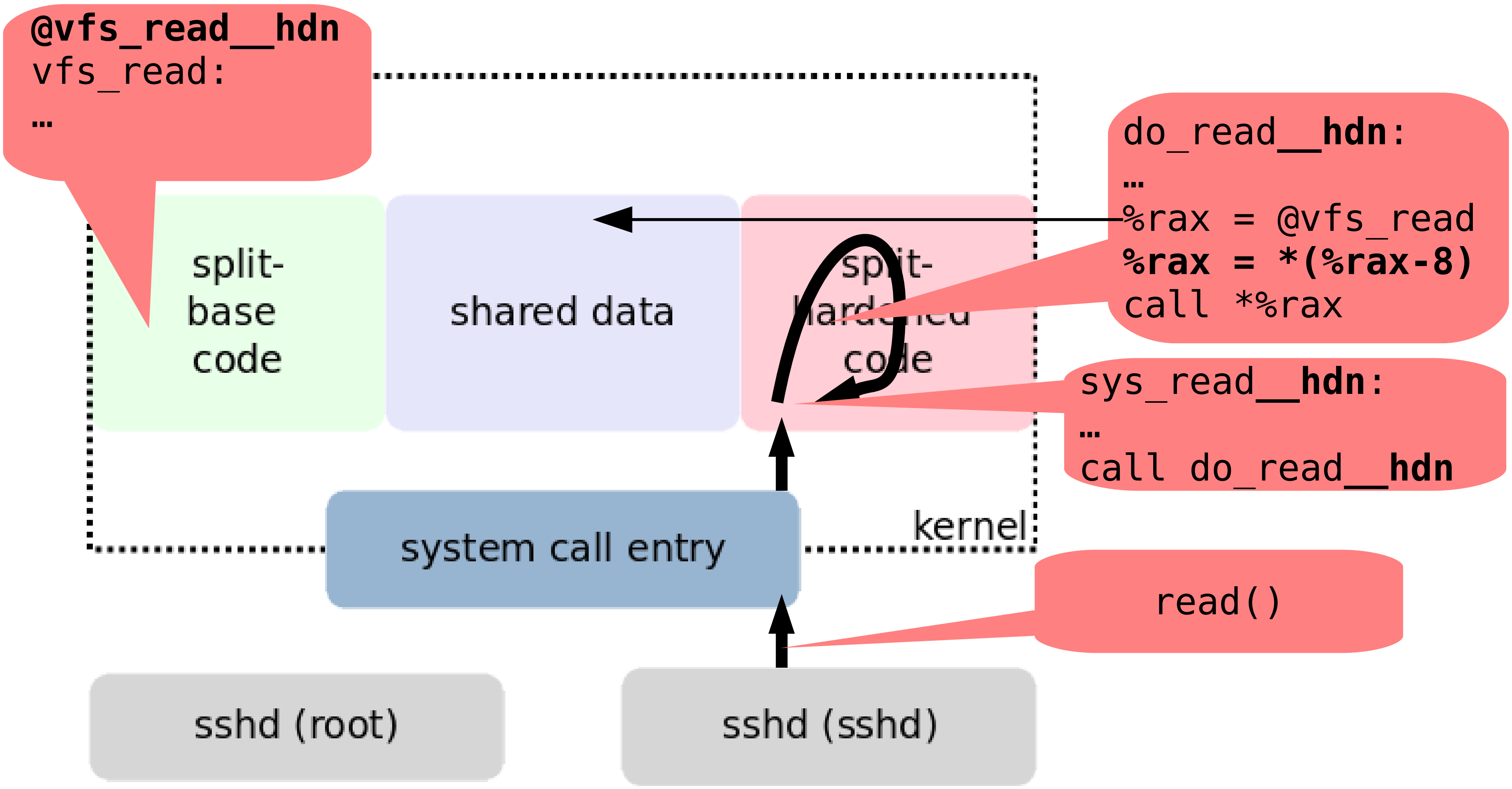
Split-hardened mode



Remaining in hardened mode

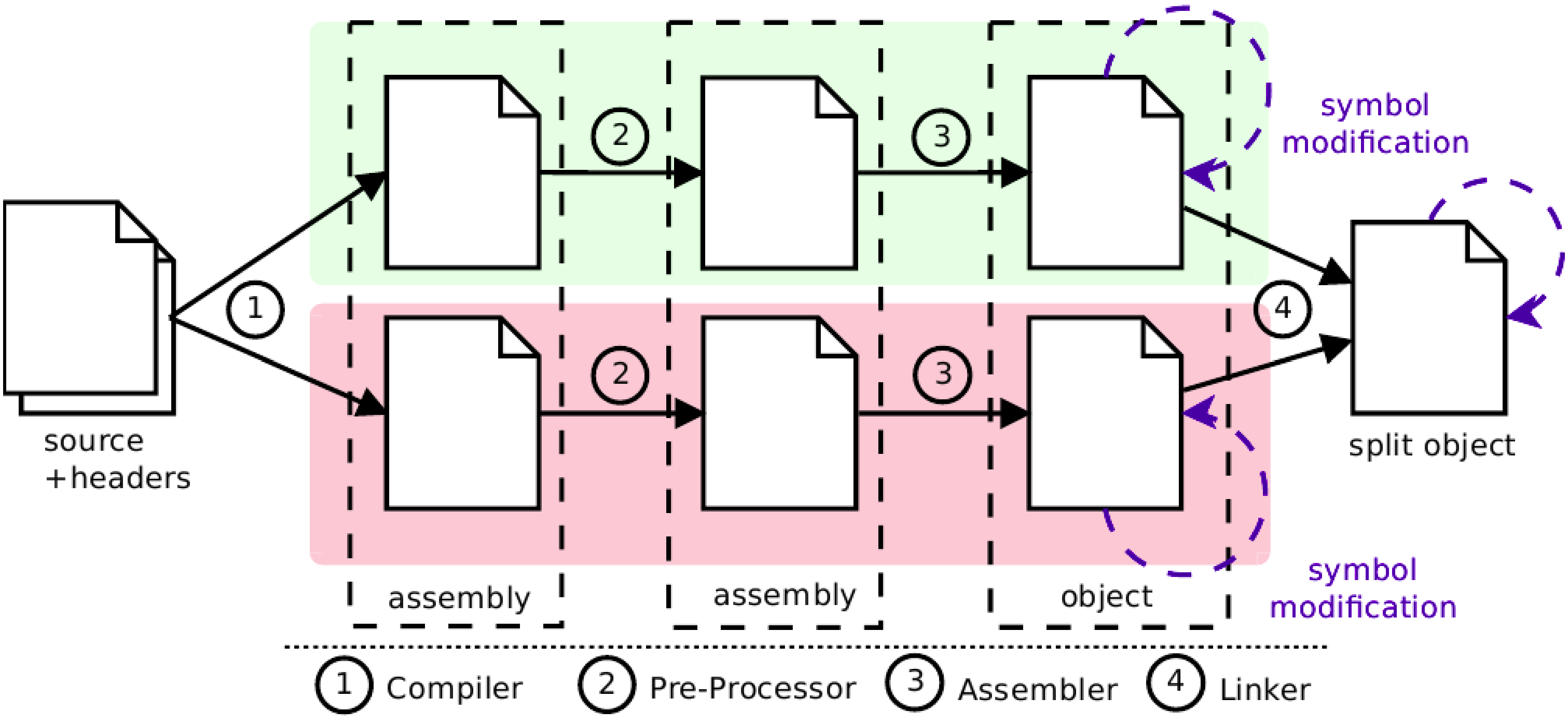
- Idea: instrument every base function
  - Problem: creates overhead in split-base mode
  - Better to instrument at the call source
- Split kernel modifies or instruments split-hardened function calls
  - Direct calls: modified at build-time
  - Indirect calls: statically instrumented to use *alternative function address*

Example: OpenSSH



Efficient indirect call instrumentation

Split Builds



Implementation highlights

- Split builds use ELF symbol weakening to ensure data sharing
- Kernel hardening mechanisms implemented:
  - Kernel stack exhaustion prevention
  - Kernel stack clearance
  - Kernel function pointer protection
- Code segregation to prevent split-base performance regressions from spatial locality
- x86-64 and MIPS32 (OpenWrt) port
- Full loadable kernel module support

Implementation

Split Builds: ~600 Lines of code

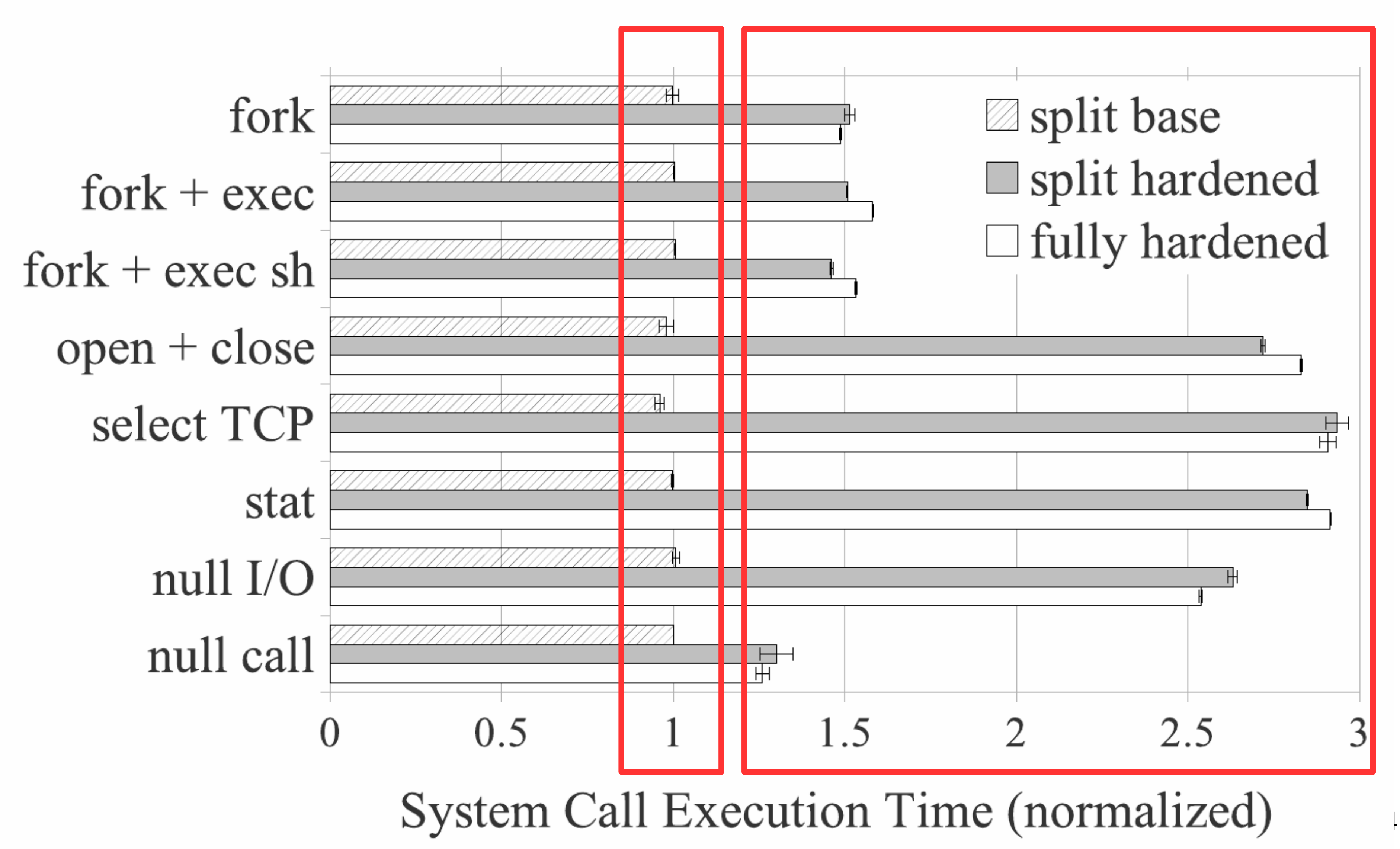
Kernel modifications: ~500 lines of code (+300)

Hardening mechanisms + preprocessor: ~800 lines of code

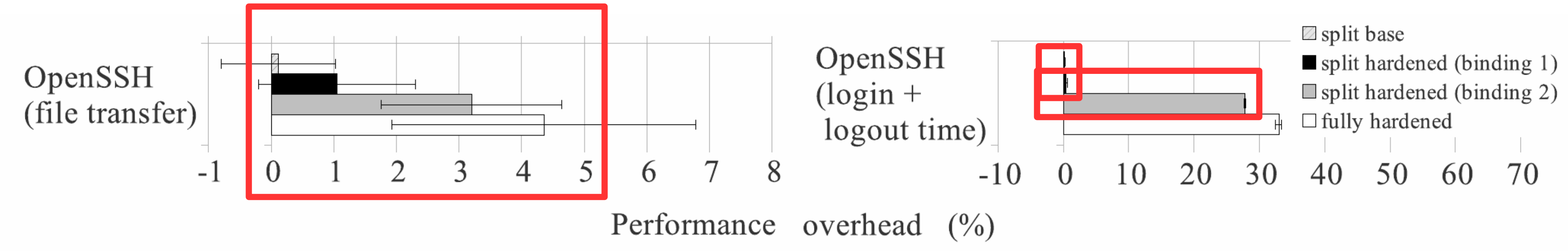
Demo

Evaluation

Micro benchmarks



Macro benchmarks: OpenSSH



- Kernel performance insensitive workloads
  - “Opportunistic hardening” possible
- Split-hardened binding 1 is much faster
  - Reason: 1K syscalls vs. 100K syscalls, network polling thread
- Choosing between binding 1 and 2 (w/ interrupt binding)
  - Depending on perceived attack surface and performance gain

Conclusion

Split Kernel may be used...

- By sysadmins, developers, packagers:
  - to tailor the kernel to security and performance needs.
- By kernel maintainers:
  - to cut down on “performance vs. security” debates.
- By researchers and kernel developers:
  - to argue that a powerful, yet slow, kernel hardening mechanism is practical

We could have nice things!

Contact: kur@zurich.ibm.com

IBM Research - Zurich