Beyond the Demo Kernel: Mapping BSPs to Mainline

Alessandro Carminati - Red Hat

東京 2025
LINUX
PLUMBERS CONFERENCE
TOKYO, JAPAN / DEC. 11-13, 2025

Agenda

- **Understanding the BSP**
  - BSPs and Their Lifecycle (Motivation)
  - Understanding Your Starting Point
- **You Didn't Write This BSP... but Now It's Your Problem**
  - Understanding a BSP When You Didn't Write It
  - Build & ABI Analysis: What's Alive and What's Dead
  - Separating Signal from Noise
- **Turning Findings into Action**
  - From Analysis to Action: Upstreaming, OOT Diagnostics, and BSP Reality Check
  - Keeping a BSP Healthy (Even During Upstreaming)
- Conclusions

Before we excavate, we need to know where we're standing.

What a BSP Really Is (and What It Isn't)

- A snapshot of hardware enablement at T=0
- Optimized for board bring-up, not for kernel evolution
- A mixture of upstream-ready code and vendor shortcuts
- A frozen view of dependencies that will quickly diverge from mainline
- A starting point, not an architectural statement

The BSP Lifecycle Problem

- Shipped early, frozen early: vendor forks stall on old LTS releases
- Subsystem evolution upstream never flows back (regmaps, PHYs, DMA, PM domains...)
- Incomplete upstreaming leaves gaps: half-upstreamed SoCs, orphaned drivers
- Board bring-up shortcuts become long-term technical debt
- Growing divergence: economically infeasible to rebase, test, or maintain

Why identify the BSP content at all?

- Upstreaming is the sustainable endpoint: long-term stability, shared maintenance
- But maintainers often inherit BSPs with no history or rationale
- First step is always the same: understand what the BSP actually contains
- Large vendor drops mix real hardware enablement with noise
- Reducing the BSP to its essential components serves both maintainers and upstreamers

How to identify upstream support using docs only (bindings)

- Device Tree bindings encode IP families: fallback compatibles signal lineage
- Start with upstream DT bindings: the catalog of supported IP & compatibles; check **Documentation/devicetree/bindings/**
- Peripheral IP is usually upstream (Synopsys UARTs, Cadence I2C/SPI, PrimeCell timers)
- But the SoC integration (glue) may not be: clocks, resets, pinmux, wrappers, MFD parents
- Frameworks exist upstream (clocks, resets, genpd, PHYs), but vendors wire them differently
- Only the SoC-specific blocks and glue are truly unique

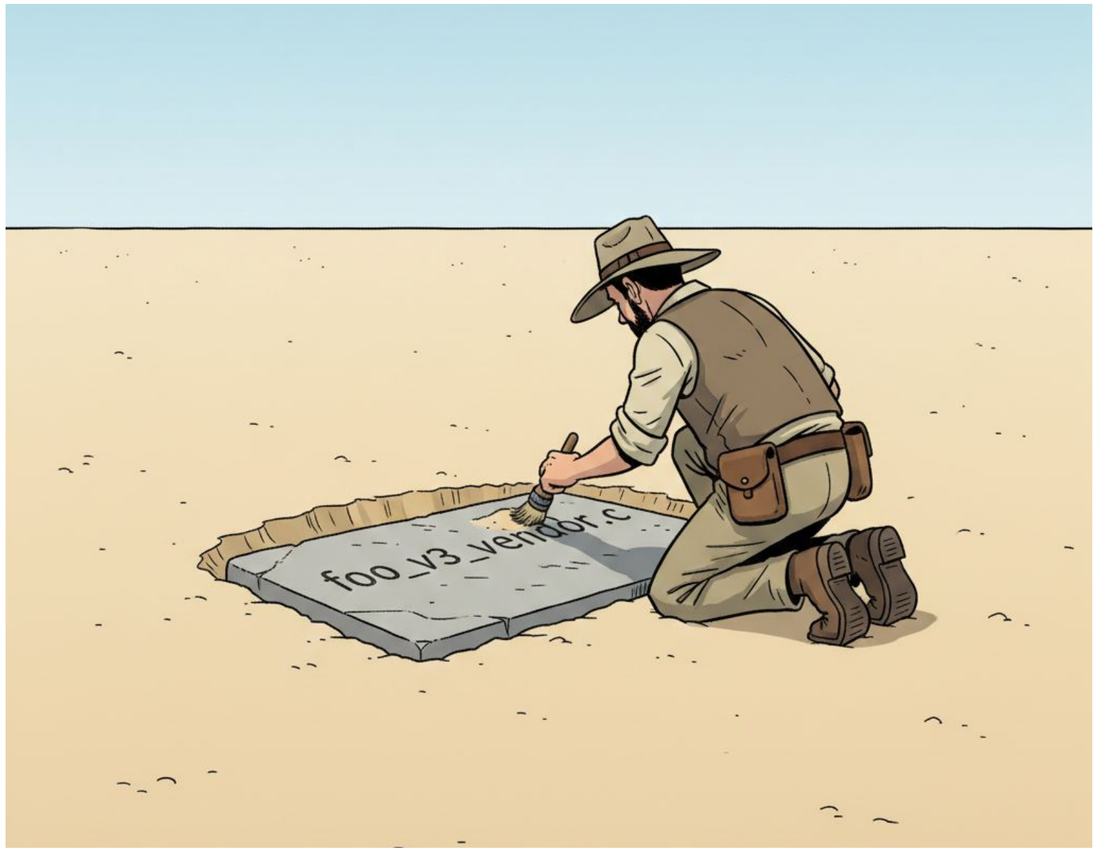
Spotting BSP-Only Subsystems Early: Drivers, Headers, Kconfigs

- Look for vendor-only subsystems that have no upstream counterpart
- Identify drivers absent in mainline (folders, Makefile entries, naming patterns)
- Check for patched or duplicated frameworks in
  - **drivers/**
  - **include/**
  - **lib/**
- Examine BSP-specific Kconfig options that introduce new features or quirks
- Spot non-standard headers (custom APIs, private abstractions, duplicated helpers)

Why Shallow Analysis Isn't Enough: The Partial Upstream Overlap Problem

- Same IP blocks appear under vendor-specific names, obscuring upstream lineage
- Mainline may support related SoCs, but not this exact revision or integration
- Small differences matter: offsets, IRQs, glue logic, missing or added features
- Many vendor compatibles predate dt-schema conventions
- Schema validation is noisy: errors don't indicate uniqueness, only divergence
- Superficial checks rarely reveal which hardware is shared vs. genuinely new

You Didn’t Write This BSP... but Now It’s Your Problem



- **Understanding the BSP**
  - BSPs and Their Lifecycle (Motivation)
  - Understanding Your Starting Point
- **You Didn’t Write This BSP... but Now It’s Your Problem**
  - Understanding a BSP When You Didn’t Write It
  - Build & ABI Analysis: What’s Alive and What’s Dead
  - Separating Signal from Noise
- **Turning Findings into Action**
  - From Analysis to Action: Upstreaming, OOT Diagnostics, and BSP Reality Check
  - Keeping a BSP Healthy (Even During Upstreaming)
- Conclusions

Step one: accept that things won’t look the way upstream would expect.

Inheriting a BSP With Zero Knowledge

- BSPs arrive with no history, no rationale, and no documentation
- They cover entire SoC families, not the board you care about
- You don't know which pieces matter and which are vendor leftovers
- Backports, rebases, and vendor patches obscure what changed
- Before you can maintain or upstream anything, you must reconstruct the starting point

Minimal File-Set Extraction: What Does Your Board Actually Use?

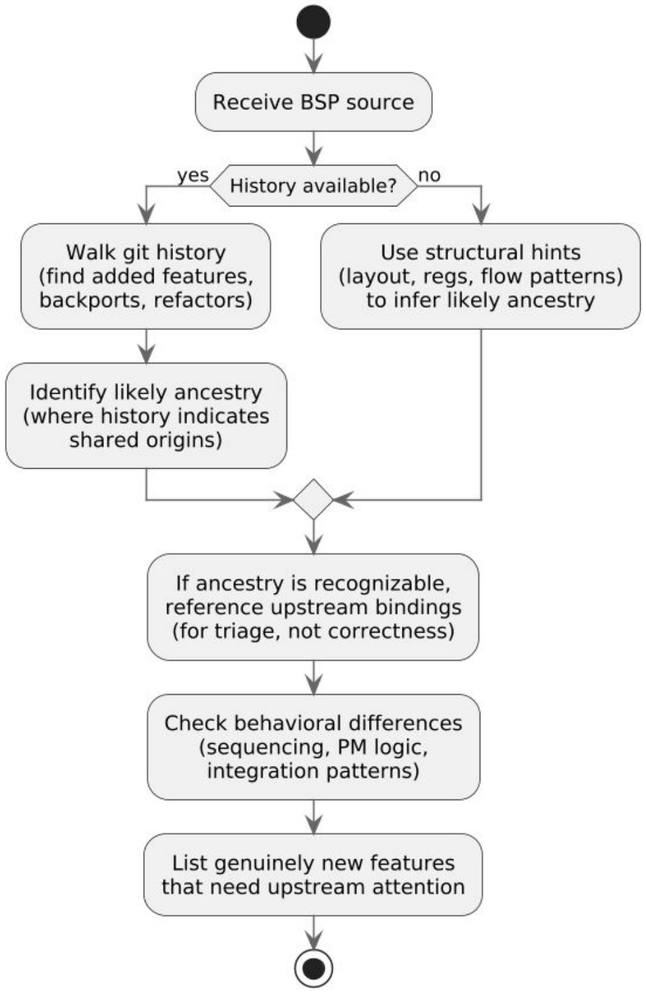
- Start from what actually runs: modules loaded, built-ins used, DT nodes enabled
- Use build artifacts to filter noise: DTB → drivers, symbol → objects, Kconfig → features
- Most BSP code isn't used by the board
- Focus on the board, not the SoC family
- **Result:** a smaller BSP footprint, with fewer stale or unmaintained code paths

Mapping Dependencies: From Modules & DTBs Back to Source

- Use DTB and runtime info to trace active drivers
- Map loaded modules and built-ins back to their source files
- Follow dependencies: includes, helpers, subsystem glue
- Identify what parts of the tree each active block actually pulls in
- **Result:** a verified, dependency-aware view of what the board uses

Fingerprints, Backports, and Refactors: Distinguishing Real Features

- Two starting points: BSPs **with** history vs. BSPs **without**
- **With** history: use git evolution to see what diverged or was added
- **Without** history: use structural hints (layout, register patterns, behavior) to identify ancestry
- When hints suggest similarity to an upstream family, use that family's bindings as a reference (triage only, not correctness)
- Similarity ≠ equivalence: integration and errata still dominate
- **Result:** separate true new features from inherited or refactored code



Comparing Images, DTBs, and Modules Across BSP and Mainline

- Find what's actually used: modules, DTBs, built-ins
- Compare BSP nodes to the nearest upstream binding or framework model
- Check integration expectations: clocks, resets, IRQs, power domains
- DTB + probe behavior expose wiring mismatches and downstream quirks
- **Result:** identify unused code and real points of divergence from upstream

When the Tool Lies

- Static similarity ≠ functional equivalence
- dt-schema warnings often reflect legitimate vendor integration choices
- Some "messy" BSP code exists because of silicon errata and timing quirks
- Hardware behavior differences don't appear in diffs (clocking, resets, read-after-write needs)
- Backports may look identical but miss later upstream fixes
- Clean upstream drivers may misbehave on this silicon revision
- Automation narrows the search space, humans validate correctness

Exported Symbols: The Boundaries That Reveal Everything

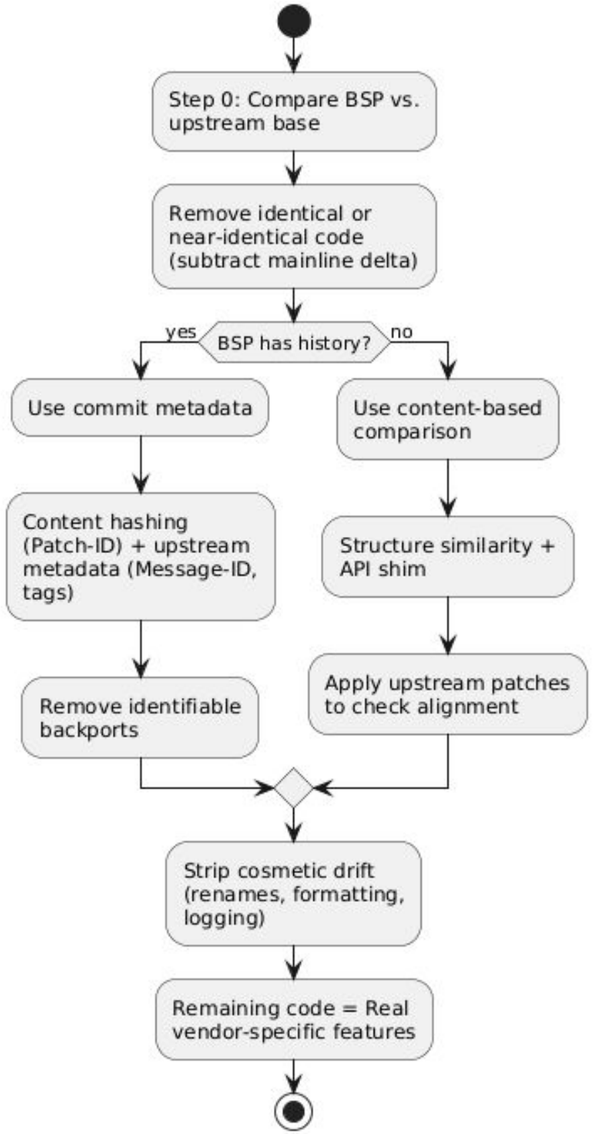
- Symbols define the true kernel ABI: the contract all drivers must follow
- Missing or renamed exports reveal BSP-only APIs and layering violations
- Extra exports expose private helpers turned into quasi-public interfaces
- Symbol mismatches predict OOT driver breakage and upstream integration pain
- **Early outcome:** if a BSP driver cannot build OOT against upstream, the BSP has drifted

A Real Example: Symbol Drift in the Wild

```
$ diff -u linux-6.6.65/aarch64/Module.symvers bsp/aarch64/Module.symvers
--- linux-6.6.65/aarch64/Module.symvers 2025-11-29 16:40:38.392738590 +0100
+++ bsp/aarch64/Module.symvers  2025-11-29 16:30:22.388914614 +0100
[...]
@@ -7294,6 +7361,18 @@
 0x00000000     dma_resv_test_signaled  vmlinux EXPORT_SYMBOL_GPL
 0x00000000     dma_resv_describe       vmlinux EXPORT_SYMBOL_GPL
+0x00000000     dma_heap_buffer_alloc   vmlinux EXPORT_SYMBOL_GPL
+0x00000000     dma_heap_get_name       vmlinux EXPORT_SYMBOL_GPL
+0x00000000     dma_heap_add    vmlinux EXPORT_SYMBOL_GPL
+0x00000000     dma_heap_find   vmlinux EXPORT_SYMBOL_GPL
+0x00000000     dma_heap_put    vmlinux EXPORT_SYMBOL_GPL
+0x00000000     cma_heap_add    vmlinux EXPORT_SYMBOL_GPL
[...]
```

Identifying Vendor Backports Masquerading as Features

- Step 0: subtract upstream, isolate the BSP delta
- With history: remove anything before the fork point + backports (hash, tags, patch-ID, Message-ID)
- Without history: subtract via content similarity + API-shim detection
- Strip cosmetic drift (renames, formatting, log noise)
- **Outcome:** only real vendor-specific behavior survives



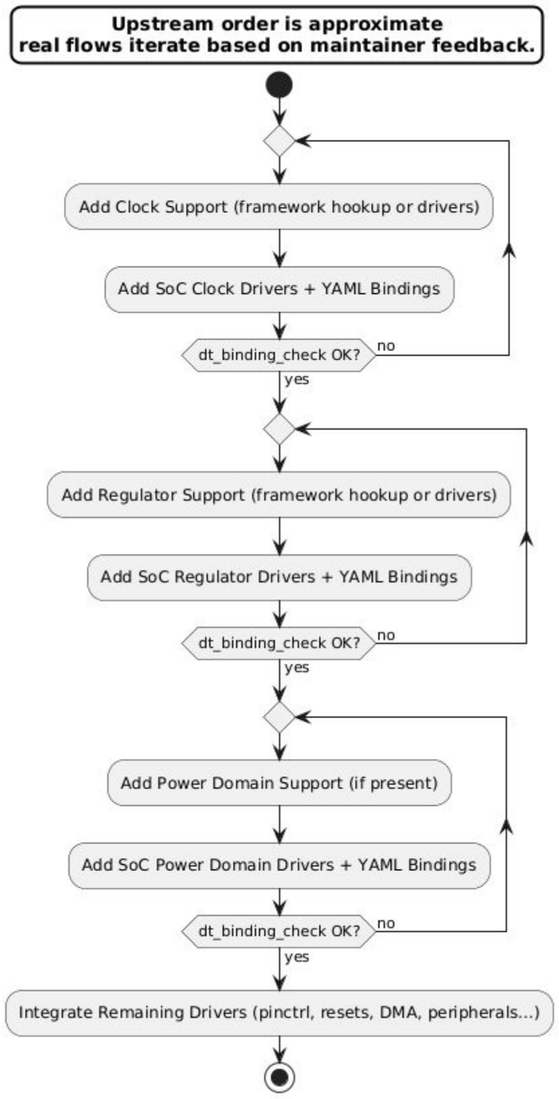
Turning Findings into Action

- **Understanding the BSP**
  - BSPs and Their Lifecycle (Motivation)
  - Understanding Your Starting Point
- **You Didn't Write This BSP... but Now It's Your Problem**
  - Understanding a BSP When You Didn't Write It
  - Build & ABI Analysis: What's Alive and What's Dead
  - Separating Signal from Noise
- **Turning Findings into Action**
  - From Analysis to Action: Upstreaming, OOT Diagnostics, and BSP Reality Check
  - Keeping a BSP Healthy (Even During Upstreaming)
- Conclusions

We can rebuild it. Preferably with fewer patches.

Prioritizing What to Upstream First

- **Start with foundational frameworks:** clocks, regulators, power domains, resets, interconnects
- Introduce DTS bindings early, validate them continuously (dt_binding_check iterations are normal)
- Split features by subsystem; avoid cross-tree coupling unless coordinated
- Follow subsystem-specific **MAINTAINERS** and **Documentation/process/** guidance when structuring series



Using Out-of-Tree Builds as a Boundary Check (Not a Compatibility Test)

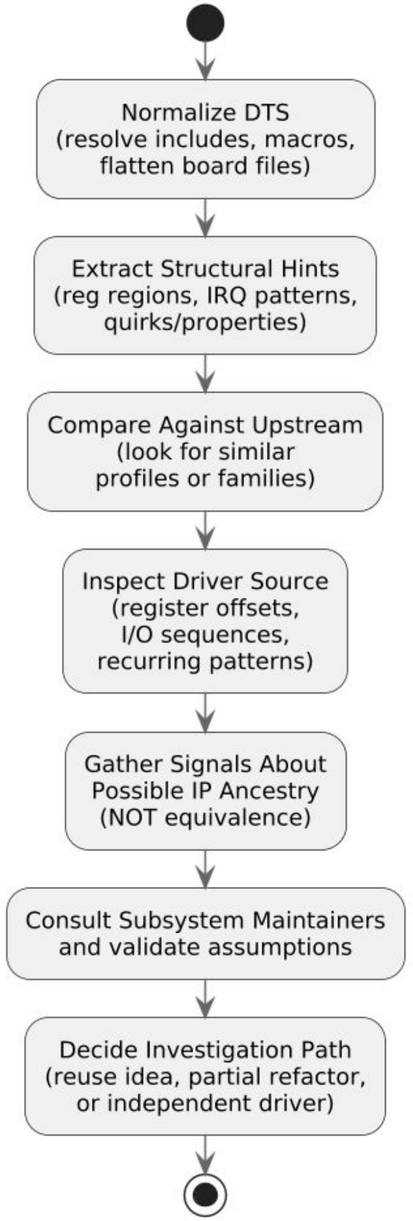
- **OOT builds are noisy:** most errors come from refactoring, churn, header changes
- **But the type of breakage often exposes:**
  - private APIs the BSP relied on
  - layering violations
  - accidental ABI coupling
  - missing subsystem abstractions
- OOT builds don't validate correctness, they highlight assumptions
- CI + OOT catches drift early, before upstreaming work is derailed
- **Value:** OOT builds reveal what is not upstream-ready yet

When OOT Fails: Reading the Signals

- **Build errors:** private headers, vendor-only helpers
- **Link errors:** reliance on internal or unstable symbols
- **Type/layout mismatches:** implicit dependency on kernel version internals
- **Probe-order failures (if it loads):** integration drift in clocks/resets/power
- **Results:**
  - These signals guide what needs cleanup before upstreaming
  - OOT failures reveal boundary issues, not hardware issues

Consolidating With Upstream Drivers: Handling Overlaps & Redundancies

- Spot potential IP lineage using structural hints
  - register regions
  - IRQ patterns
  - Recurring quirks
- Treat fingerprints as heuristic only
  - similar offsets → possible ancestry
  - silicon IDs rarely authoritative
- Integration differences always matter
  - wrappers, fabrics, errata
- **Final validation requires subsystem review**

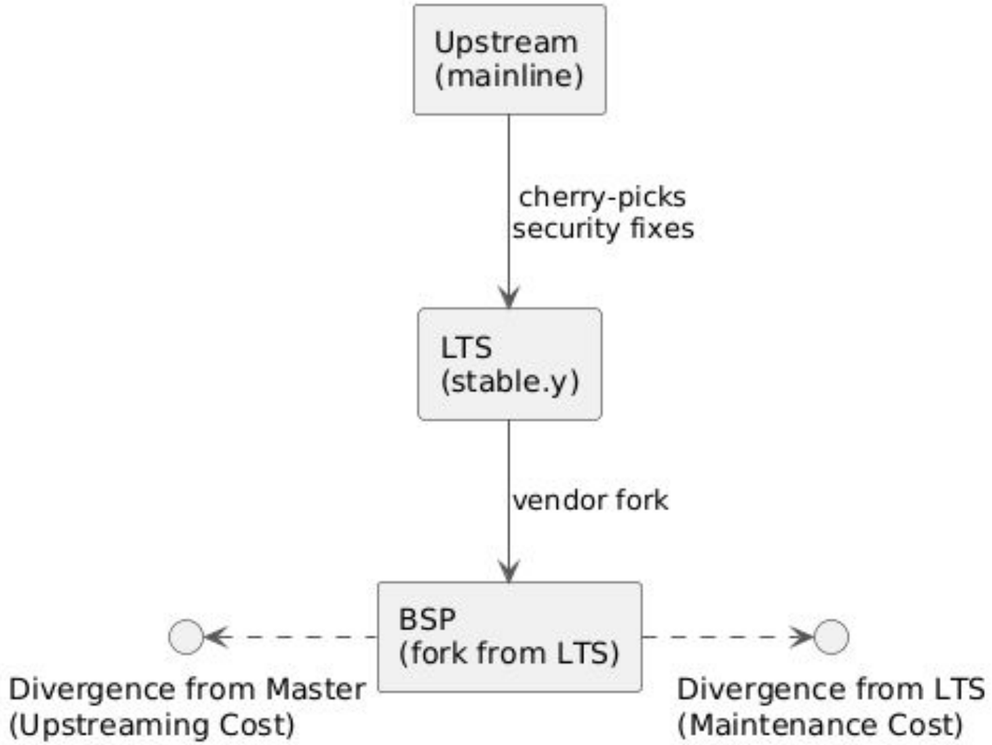


Upstream Patch Strategy: From Overlap to Submission

- Show evidence that the upstream driver covers the hardware (bindings, register map, integration notes)
- Isolate the true vendor deltas; refactor upstream only where needed
- Keep the consolidation patch small and mechanical
- Retire redundant BSP drivers after acceptance
- Use a clear cover letter to explain the consolidation and its scope

Two Divergences, Two Goals

- BSP vs. LTS: divergence that determines how hard it is to apply security fixes and stable updates
- BSP vs. Mainline: divergence that determines how hard upstreaming will be later
- These are separate problems: one affects maintainability, the other affects sustainability
- Keeping both small requires different strategies



Continuous Visibility: Maintain the LTS patchability

- Keep BSP deltas manageable while upstreaming progresses
- Automate diffing (source / DTB / configs) to spot drift early
- CI helps detect where BSP patches diverge from LTS
- Small deltas → easier LTS updates → fewer surprises
- This is survival during the transition, not a replacement for upstream

Avoiding the “Throwaway BSP” Trap

- Keep BSP delta small enough to survive the transition to upstream
- Upstream reusable parts early, before they fossilize
- Separate real silicon needs from temporary bring-up hacks
- Treat the BSP as a temporary bootstrap, not a parallel kernel
- Make every patch something Future-You can live with
- Survival now → lower cost when you upstream later

Key Takeaways & Roadmap for a Healthy BSP

- These methods surface signals, they do not establish truth
- Structural hints narrow the search space; they never imply equivalence
- Artifact drift points at questions, not answers
- Integration and errata always dominate behavior
- The only sustainable path is upstreaming

Questions?
質問?

The End
終了