

LPC 2026 · PRAGUE · OCTOBER 7

A loadable crypto module for FIPS certification

Linux Plumbers Conference 2026

Jay Wang

Amazon Linux · AWS



© 2026, Amazon Web Services, Inc. or its affiliates. All rights reserved.



Today's discussion

The problem - FIPS validation and our proposal

Design details - Making built-in crypto loadable

How to use it - Reusing and refreshing certified crypto

Going forward - Upstream direction

Summary

The FIPS validation problem



© 2026, Amazon Web Services, Inc. or its affiliates. All rights reserved.

What is FIPS?

Federal Information Processing Standards

Standards published by NIST for federal information systems

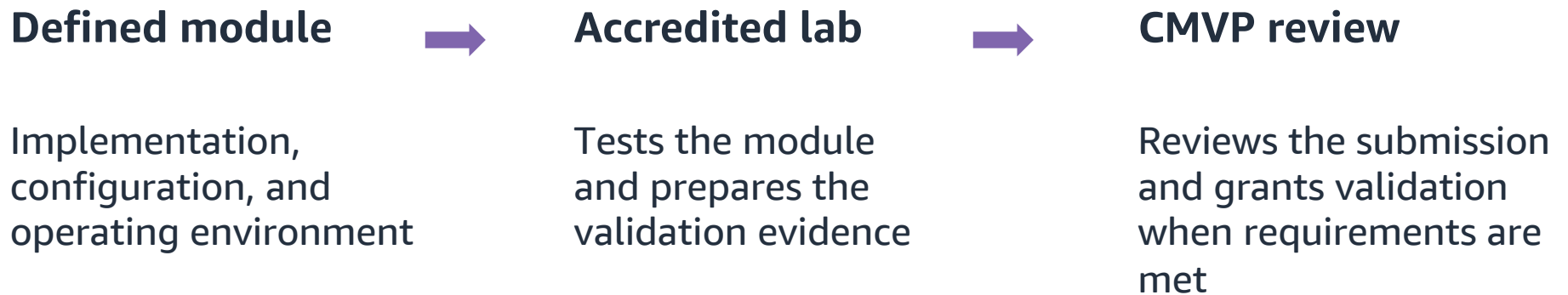
FIPS 140-3

Security requirements for cryptographic implementation

Federal agencies and customers with related obligations need validated cryptography.



FIPS 140-3 validation



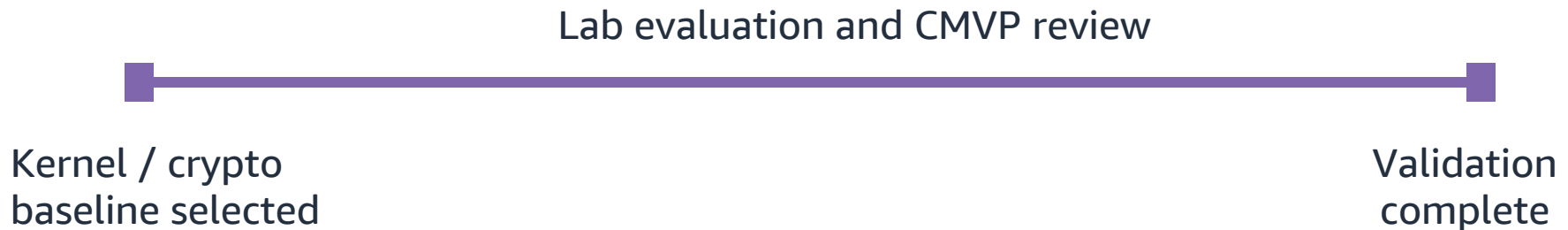
Algorithms, self-tests, integrity, interfaces, and operating conditions all matter.



Problem 1: full validation takes time

12–18 months

Full validation cycles reported in the project publications



Customers awaiting this validation remain on an earlier validated baseline. Timing is reported experience.



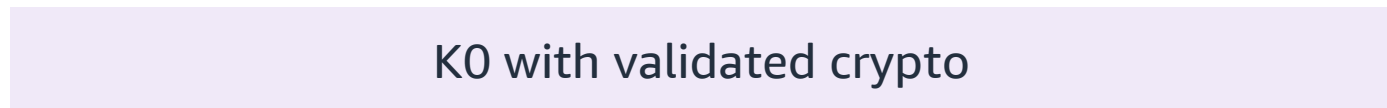
The bigger problem: certification cannot be reused

Available
kernels



Fixes and new feature support continue

Customer
baseline



Users stay on K0 while newer kernels await validation,
even when only non-crypto code changes.

If a new full validation is required, the wait can be another 12–18 months. The review depends on the change.

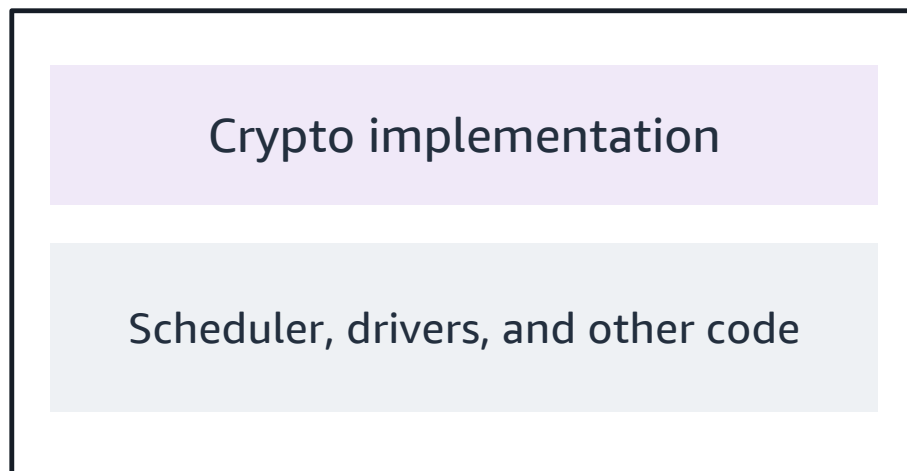


Why isn't it reusable?
Certification at the entire kernel level



Built-in crypto couples validation to the kernel image

One kernel image



A non-crypto change still changes the kernel image containing crypto.

Carrying validation forward becomes harder, changes to kernel triggers a re-certification

The customer's dilemma

Stay on the older kernel
with validated cryptography, or

Adopt a newer uncertified kernel before
its validation path is ready



Our proposal

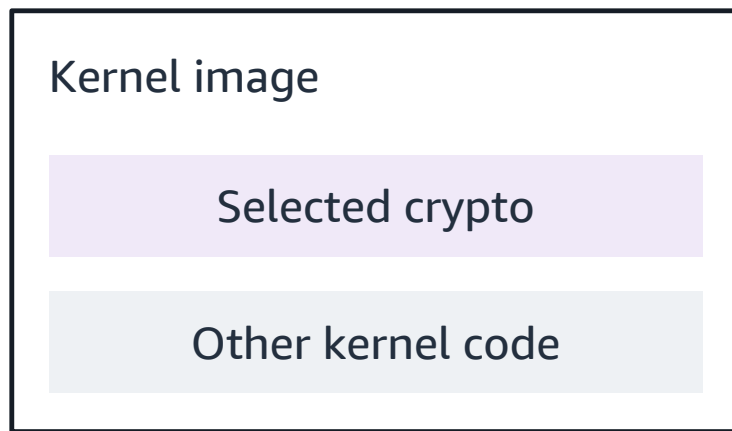
A separate, reusable module for kernel cryptography

Keep kernel updates moving while managing crypto separately

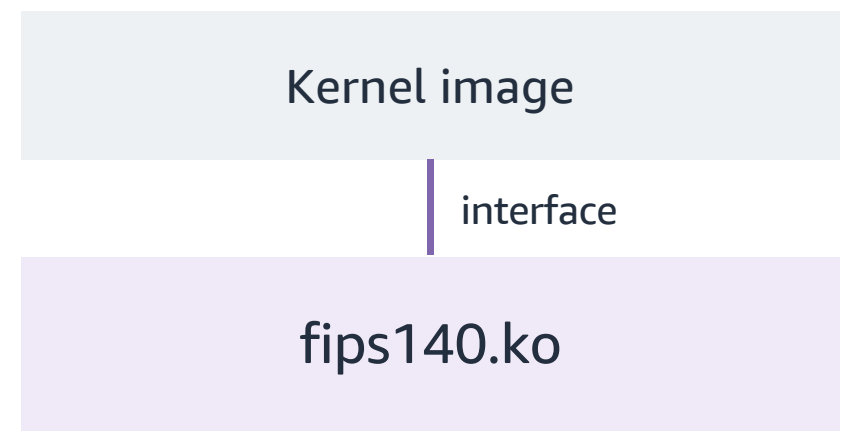


A separate module for kernel cryptography

Built-in crypto



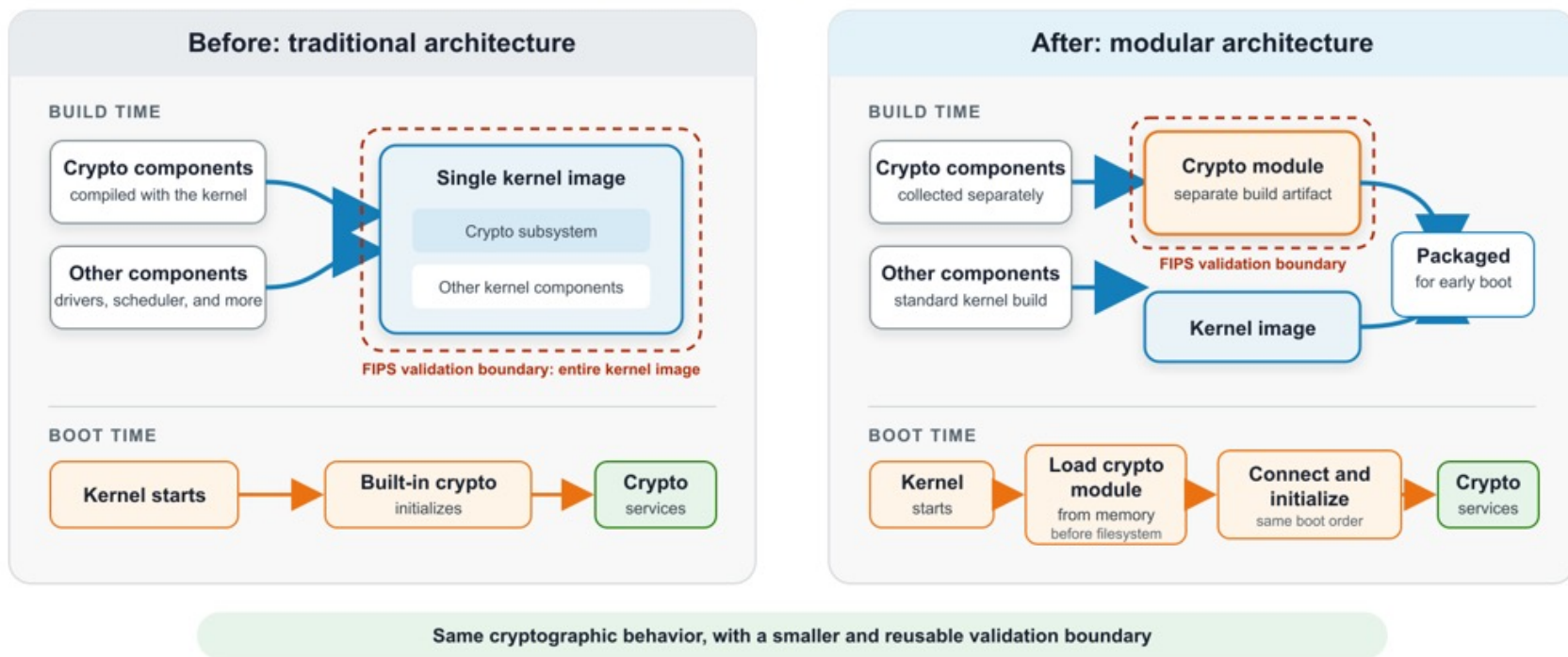
Loadable crypto



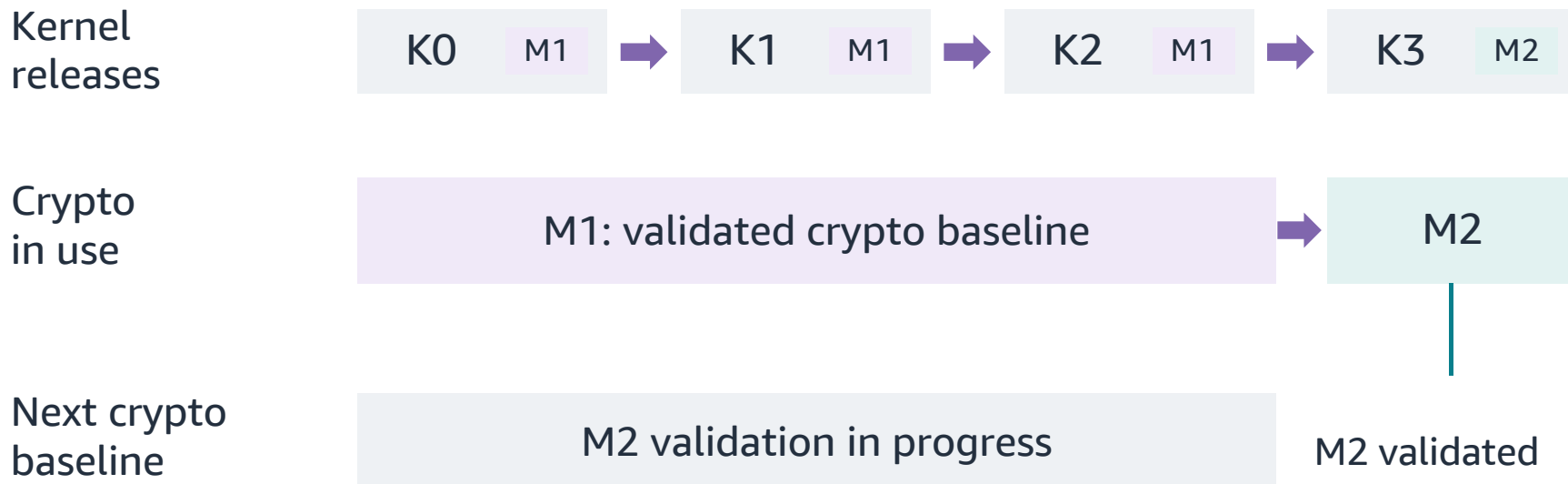
The crypto implementation has its own artifact. Kernel updates can preserve the selected crypto baseline.

Separate at build time, integrated at boot

From built-in cryptography to a modular architecture



Newer kernels with a maintained crypto baseline



Reuse requires compatibility and permitted use. M1 bridges the review of M2, within its approved lifecycle.

Related work

Related approaches

User-space crypto libraries with a small validation boundary

Android GKI 5.10 FIPS kernel module

Oracle Linux loadable crypto module prototype

This work

Designed for maintainability, with support for arbitrary kernel config to facilitate use across Linux distros.

First production deployment of this approach in a Linux distribution offered by a cloud provider, to the best of our knowledge.

Design differences and related work: [LWN.net \[4\]](#)



© 2026, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Feature design

Making built-in crypto loadable



© 2026, Amazon Web Services, Inc. or its affiliates. All rights reserved.

A kernel feature for crypto modularization

`CONFIG_CRYPTO_FIPS140_EXTMOD=y`

Same cryptographic algorithms and implementation

Move selected built-in crypto into fips140.ko.
Change how it is built, packaged, and loaded.
Keep the cryptographic algorithms unchanged.

First shipped in Amazon Linux with kernel 6.18. Also adopted by other OSs.



Source code and design references

[1] Production branch: amazon-6.18.y/mainline

<https://github.com/amazonlinux/linux/tree/amazon-6.18.y/mainline>

[2] Development branch: fips-kernel-module (tracks newer stable kernels)

<https://github.com/amazonlinux/linux/tree/fips-kernel-module>

[3] Detailed implementation documentation

<https://github.com/amazonlinux/linux/blob/fips-kernel-module/crypto/fips140/README>

[4] LWN.net: design and implementation

<https://lwn.net/Articles/1073759/>

[5] AWS Compute Blog: Amazon Linux deployment

<https://aws.amazon.com/blogs/compute/introducing-modularized-kernel-cryptography-in-amazon-linux/>



Design question

Why can't we simply use
the existing kernel module system?

Three obstacles: build integration, reverse symbol resolution,
and early boot with correct initialization



Obstacle 1: no rule to collect built-in crypto objects

Scattered build rules

Hundreds of source files span crypto/ and architecture directories.

Each directory owns its configuration rules and compilation settings.

A separate module Makefile does not scale

Copying the build rules (e.g., flag) creates a second build description to keep in sync.

Replacing the original Makefiles would break normal built-in and modular builds when feature disabled.

We need to gather selected built-in objects across directories while preserving their existing build rules.



Build solution: reuse the existing Makefiles in place

New build rule: crypto-objs-y

```
- obj-$(CONFIG_F00) += foo.o  
  
+ crypto-objs-$(CONFIG_F00) += foo.o
```

Illustrative rule. Local flags and dependencies stay in place.

Collect selected built-in objects

Recursively gather them into fips140.ko.

Keep crypto already configured as ``=m`` remain in the separate module form (e.g., `aes.ko` if `CONFIG_CRYPTOAES =m`).

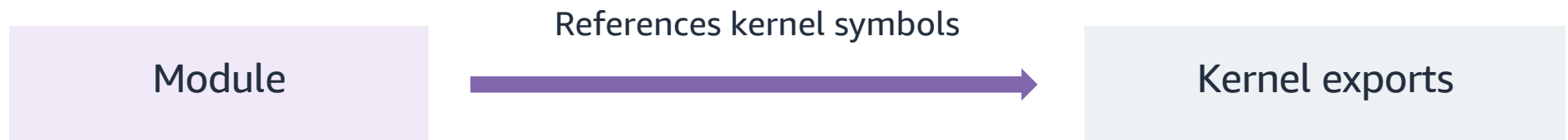
Validation scope covers `fips140.ko` and the selected separate crypto modules.

Feature disabled: preserve the normal built-in and modular build behavior.



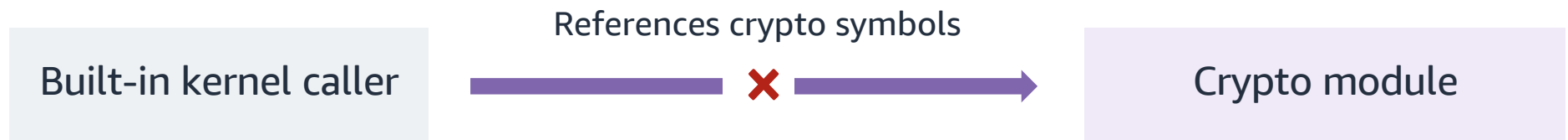
Obstacle 2: the kernel needs symbols from the module

Ordinary module loading, one-way direction



The loader resolves these references when the module loads.

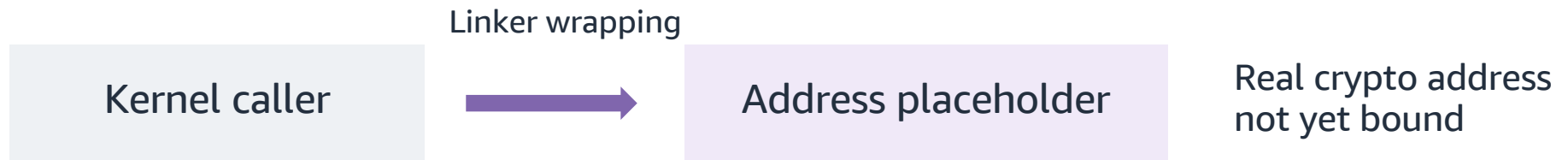
After moving built-in crypto into a module, reverse direction



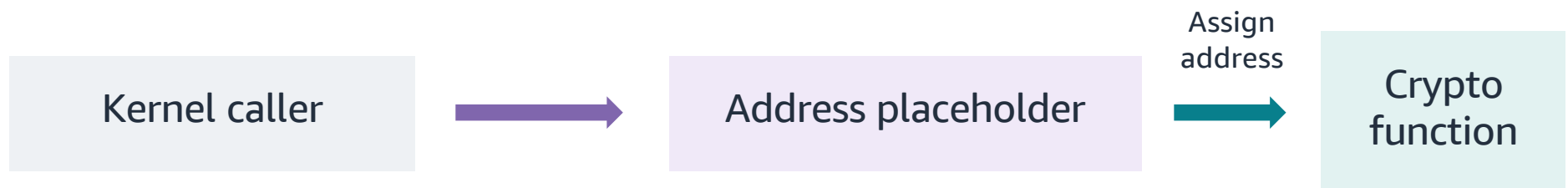
Crypto definitions are absent when the main kernel links. The kernel build fails.

Symbol solution: link now, connect at boot

1. Link time: give the linker a blank target



2. Boot time: bind the real implementation

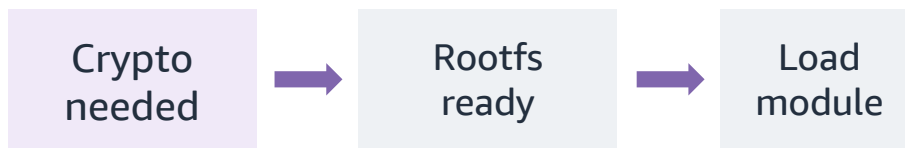


The loader fills in the address. Pointers become read-only after initialization.

Obstacle 3: early availability and initialization order

A module on disk arrives too late

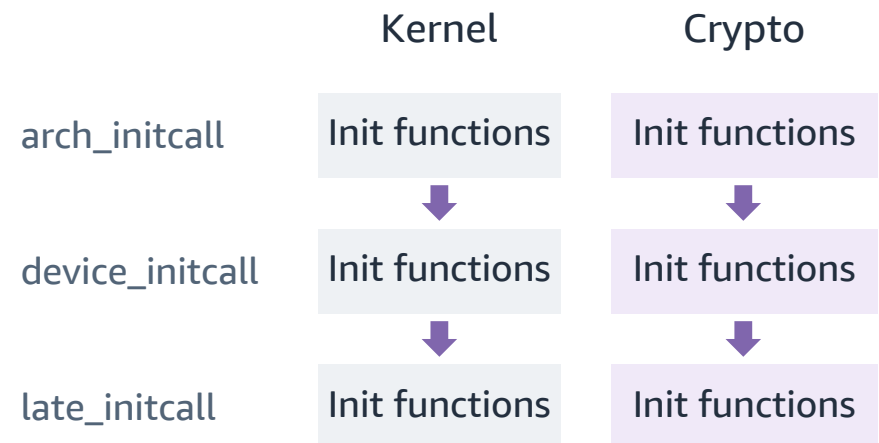
Boot progresses



Too late

Early kernel users need crypto before the filesystem loading path is ready.

Initialization follows boot phases



Illustrative boot phases and dependencies.
Batching crypto init can violate this order.

Boot solution: load early and preserve initcall order

In order to	What happens	Why it matters
Load early	Embed the module back into vmlinux as data	Available in memory before the filesystem
Preserve initcall order	Synchronize module and kernel initcall phases	Dependencies run in the required order

The module participates in the boot sequence instead of running every initializer at once.



Design choices that reduce/eliminate developer overhead

Keep local build rules

Reuse existing flags and configuration.

Automatically support arbitrary kernel configurations.

Automate the interface

Automatically redirect callers through address placeholders at build time to support reverse symbol resolution.

Preserve boot semantics

Coordinate initialization with the kernel.

Keep callers using the crypto services.

Kernel and crypto developers can work almost as if the modularization feature did not exist.



Low overhead

Average boot-time overhead

<1%

Runtime crypto overhead

<1%

Binary size

<1 MB

Memory footprint

<1 MB

Measured using the Amazon Linux kernel configuration.



© 2026, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Reusing a certified module across kernel releases



Two ways to reuse crypto

Reuse the source

Rebuild unchanged crypto source.

Maintain source-level API compatibility.

Reuse the binary

Keep the exact module binary.

Preserve its runtime ABI, including data layouts and calling conventions.

Recommended: source reuse
Generally requires less compatibility work.



Which source code do we reuse?

Kernel source tree

Modularization feature

For example, scripts/Makefile.build

← Decouples crypto from the kernel without changing crypto algorithm implementations.

Selected crypto source

For example, crypto/algapi.c

← **The distribution owns the crypto baseline**

Scope: choose the validation boundary and track the source, build and module artifact.

Changes: customize crypto as needed and assess the validation impact of those changes.

Lifecycle: maintain API compatibility, plan crypto refreshes and security updates.

Remaining kernel source

Other kernel code

Which source code do we reuse?

K0 source tree

K0 modularization feature
For example, scripts/Makefile.build

K0 selected crypto source
For example, crypto/algapi.c

K0 remaining kernel source
Other kernel code

Update normally*



Reuse unchanged



Update normally



K1 source tree

K1 modularization feature
Integration for the newer kernel

Same K0 crypto source
Preserved implementation baseline

K1 remaining kernel source
Newer kernel code

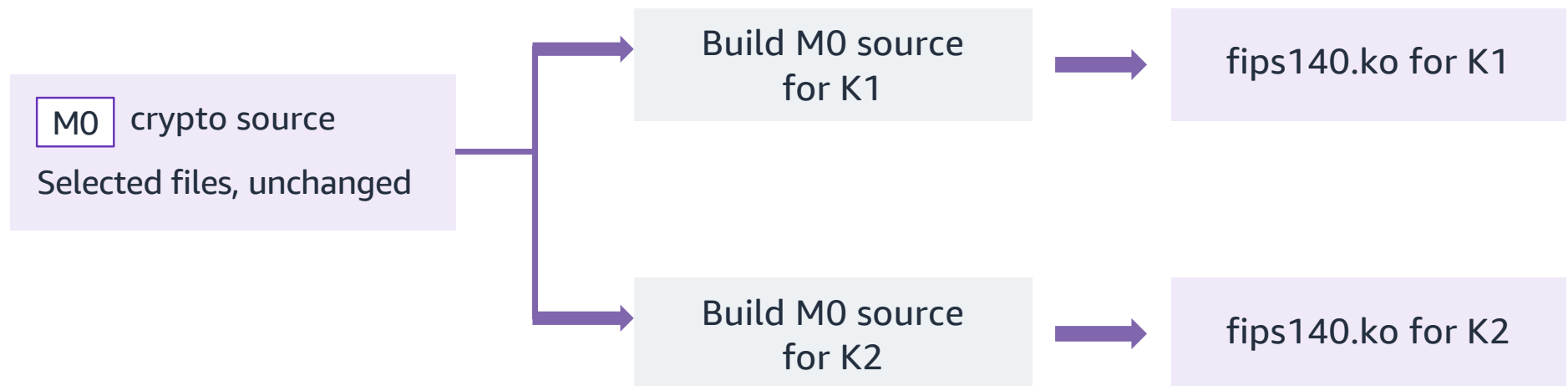
* The modularization feature decouples crypto from the kernel without changing crypto algorithm implementations.



Rebuilding the same crypto source for newer kernels

Preserve the crypto implementation

Feature patches and build configuration follow each target kernel



Patch the main kernel as needed to preserve the crypto source API.

Can support day-one availability under certain validation conditions.

Refreshing the certified module



When to refresh the certified crypto

Common reasons

Security fixes in certified crypto

Better crypto performance

Growing cost of maintaining kernel compatibility

Not a reason to refresh: non-certified crypto can evolve separately through extensibility, with changes kept outside the validated boundary.



Reuse bridges the next validation cycle

Milestone	Kernel releases	Crypto lifecycle
M1 is validated	Ship a compatible kernel with M1	Maintain M1 for approved use
M2 is under review	Continue compatible kernel updates	Develop and validate M2
M2 is validated	Move releases to M2	Retire M1 according to policy

M1 and M2 denote crypto baselines.



Going forward Upstream direction



Revisiting the kernel source tree

K0 source tree

K0 module feature
Build, linking, early loading

K0 crypto source
e.g. crypto/algapi.c

K0 remaining source
Other kernel code

Update normally



Reuse unchanged



Update normally



K1 source tree

K1 module feature
Integration for K1

Same K0 crypto source
Preserved crypto baseline

K1 remaining source
Newer kernel code



Proposed upstream infrastructure

Shared build, linking and boot support



Crypto algorithms remain unchanged

Summary - Key takeaways

The work

A separate crypto module enables reuse and reduces repeated FIPS validation work.

Current status

Deployed in Amazon Linux 2023 with kernel 6.18 and later.

Also adopted by several other operating systems.

Release strategy

Source reuse. Module refresh as needed

Upstream direction

Share the mechanism. Distributions choose the crypto scope and manage their releases.



Thank you!

Jay Wang

wanjay@amazon.com

Questions?

References in previous slides

LWN.net, AWS Blog, implementation guide and source branches

© 2026, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Acknowledgments

Amazon engineering and non-engineering teams for their work and collaboration.

Developers from other distributions, including Vegard Nossum (Oracle), for technical discussion and feedback.
vegard.nossum@oracle.com

Contributors to earlier attempts and ongoing related work.



Backup slides



Source code and design references

Production branch: amazon-6.18.y/mainline

<https://github.com/amazonlinux/linux/tree/amazon-6.18.y/mainline>

Development branch: fips-kernel-module (tracks newer stable kernels)

<https://github.com/amazonlinux/linux/tree/fips-kernel-module>

Detailed implementation documentation

<https://github.com/amazonlinux/linux/blob/fips-kernel-module/crypto/fips140/README>

LWN.net: design and implementation

<https://lwn.net/Articles/1073759/>

AWS Compute Blog: Amazon Linux deployment

<https://aws.amazon.com/blogs/compute/introducing-modularized-kernel-cryptography-in-amazon-linux/>

