SUPPLYCHAIN SECURITYCON @ THE LINUX FOUNDATION OPEN SOURCE SUMMIT NORTH AMERICA

TPMs, Merkle Trees and TEEs:

Enhancing SLSA with Hardware Assisted Build Environment Verification

Marcela Melara (Intel Labs) & Chad Kimes (GitHub)

github.com/chkimes

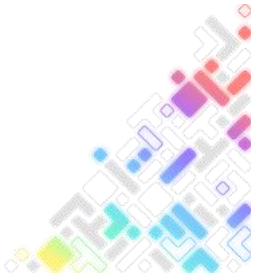
- Principal Software Engineer at GitHub Actions
  - Primary focus on GitHub Hosted Runners
- Member of the Actions Security team
- Interests
  - Supply chain security
  - Building for scalability
  - Distributed systems

@marcelamelara

- Research Scientist at Intel Labs
  - Software supply chain security research lead
- OpenSSF TAC member & in-toto maintainer
- Interests:
  - Distributed systems security
  - OS-level security
  - Hardware-assisted security

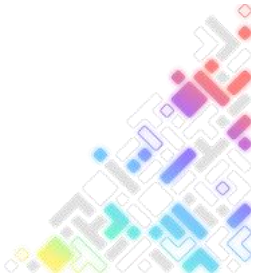
Talk Overview

- Status Quo: SLSA & CI/CD Builds
- The Problem: Validating Build Environments
- The nitty gritty crypto and hardware stuff
  - TPMs
  - *-verity and Merkle trees
  - TEEs
- SLSA Enhancements
- Final Words



Recap - Provenance, Attestations, Signing

- SLSA Provenance: A detailed description of the build process
- in-toto attestations: Standard data format for attestations about any aspect of the SW supply chain
- Sigstore: Identity management service and public transparency log



What We Get from Provenance, Attestations, Signing

- SLSA Provenance: A detailed description of the build process
- in-toto attestations: Standard data format for attestations about any aspect of the SW supply chain
- Sigstore: Identity management service and public transparency log

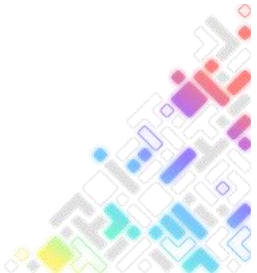
Cryptographic, verifiable link between a software artifact and the build process that created it

SLSA Refresher: Build Track

Track/Level	Requirements	Focus
Build L0	(none)	(n/a)
Build L1	Provenance showing how the package was built	Mistakes, documentation
Build L2	Signed provenance, generated by a hosted build platform	Tampering after the build
Build L3	Hardened build platform	Tampering during the build

SLSA Guiding Principle: “Trust platforms, verify artifacts”

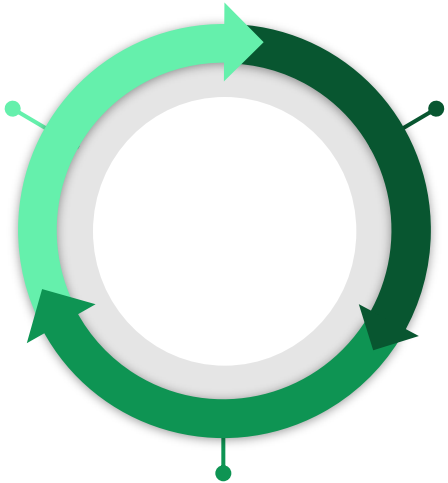
- Trusted computing bases are needed in practice
- Platforms have economic incentives to correctly implement SLSA



So, why do build platforms need more hardening?

High-value Target

May handle secrets & signing
Arbitrary code execution by design



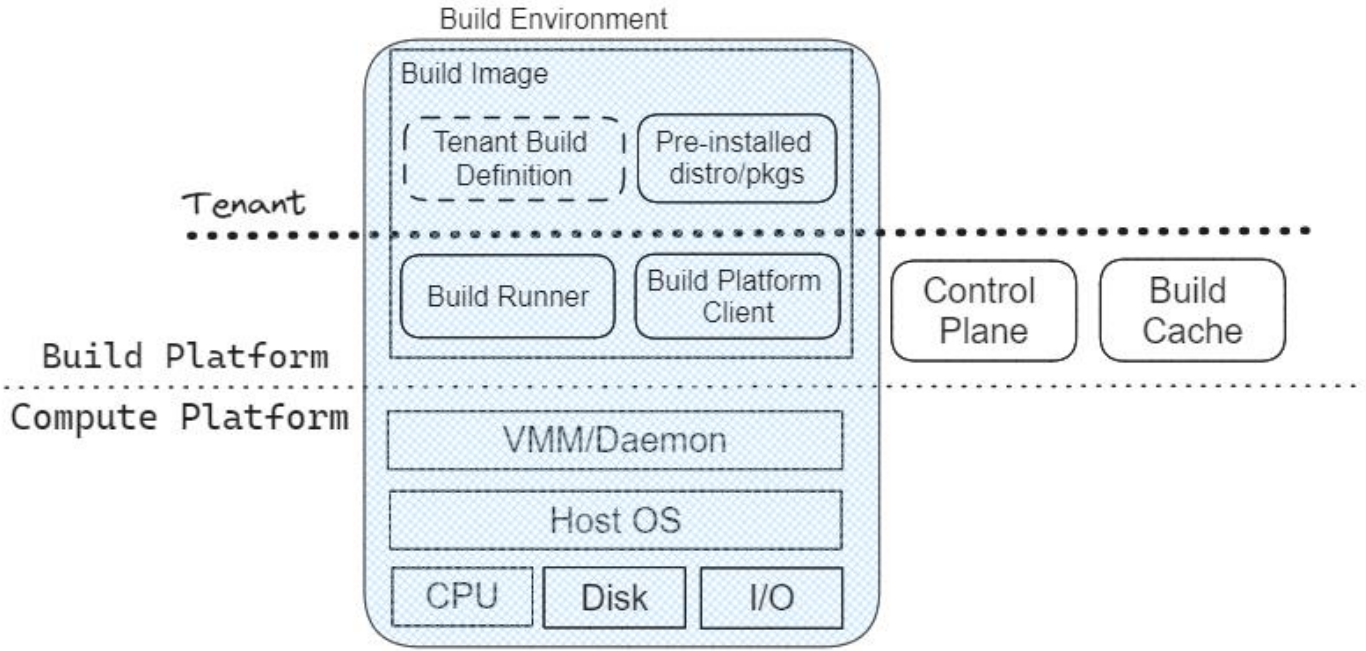
Attack Amplification

Often large user bases
CI users have many customers, too

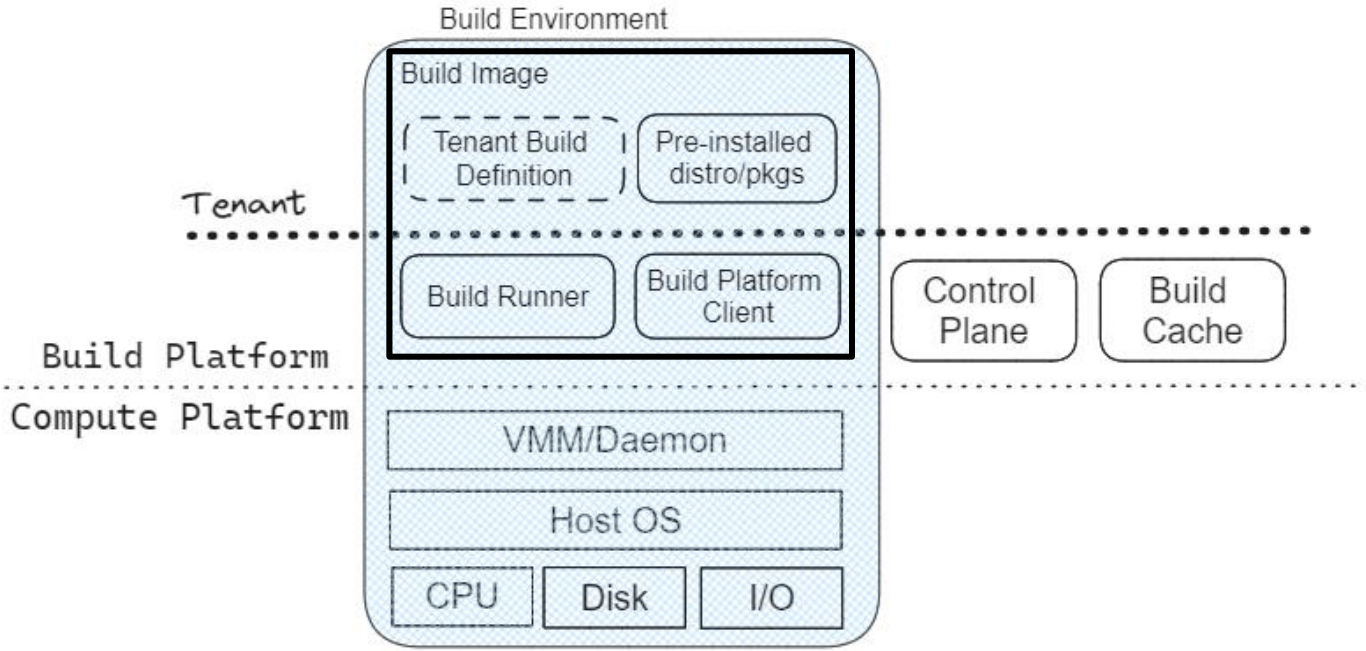
Very Complex System

Many actors and components
Hard to audit

The Many Components of a Build Environment



The Many Components of a Build Environment

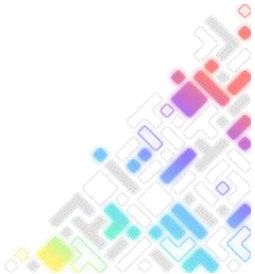


The Problem: Unquestioning Trust in the Build Platform

01	Trust to create a “good” build image	Tampering with image generation
02	Trust to supply the “right” environment	Tampering with image distribution
03	Trust to “protect” your executing build	Tampering with running environment

Our Approach: “Prefer attestations over inferences”

- SLSA Guiding Principle: Capture explicit evidence about artifacts
- Build platform consists of cooperating software layers!
- Leverage existing mechanisms to attest to build environment configuration



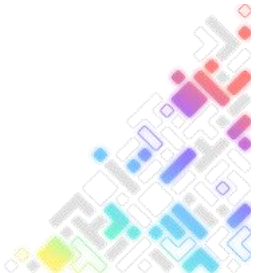
How do we trust, but verify a build platform?

01	Trust to create a “good” build image	Publish build image SLSA Provenance
02	Trust to supply the “right” environment	Check environment vs. image SLSA Provenance
03	Trust to “protect” your executing build	Check execution context vs. build environment

How it could work

(aka the nitty gritty hardware/crypto stuff)

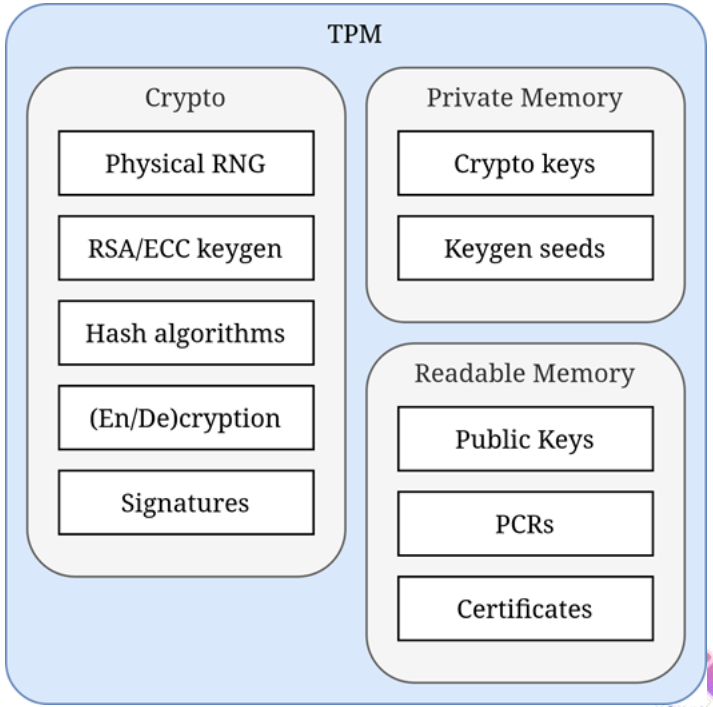
- Trusted Platform Modules (TPM)
- Measured Boot
- Integrity Checking
- Confidential Computing



TPM Recap

- Defined by a 1200-page Trusted Computing Group specification
- Private key inaccessibly burned into the silicon
- Public key readable using API
- Optionally, manufacturer signed certificate readable from non-volatile memory (NVRAM)

- **What to know:** TPM key pair represents identity, certificate shows the key pair is a “real TPM”



TPM Recap - Platform Configuration Registers

- TPMs have banks 24+ “hash registers”
  - ~16 usable by the OS
- Each register stores a single hash-length value
- Updated with “extend” operation
- Extending the same data into a PCR in the same order always produces the same result

Register	Value
PCR0	0xE5FA44F2B31C1FB553B6021E7360D07D5D91FF5E
PCR1	0x7448D8798A4380162D4B56F9B452E2F6F9E24E7A
PCR2	0xA3DB5C13FF90A36963278C6A39E4EE3C22E2A436
PCR3	0x9C6B057A2B9D96A4067A749EE3B3B0158D390CF1
PCR4	0x5D9474C0309B7CA09A182D888F73B37A8FE1362C
…	…

TPM Recap - Platform Configuration Registers

- Extend operation

  HASH(Value || HASH(Data))

- SHA1 Example
  - Initial Value: `0x0000000000000000000000000000000000000000`
  - Data: `0xDECAFBAD`
  - HASH(Data): `0x4D0FD7CFF63740134DD8AAE8A13819BF1C4707AA`
  - New Value:

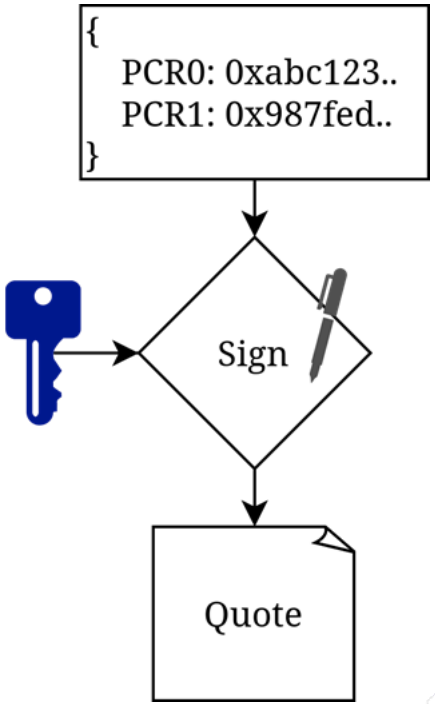
    `HASH(0x0000000000000000000000000000000000000000 4D0FD7CFF63740134DD8AAE8A13819BF1C4707AA)`
  - New Value: `0x70716824C3CFEB65BDDF24064C5F91F932AD79C3`

- **What to know:** Similar to hash for a git commit, PCR values are a hash representing all the data used to generate them

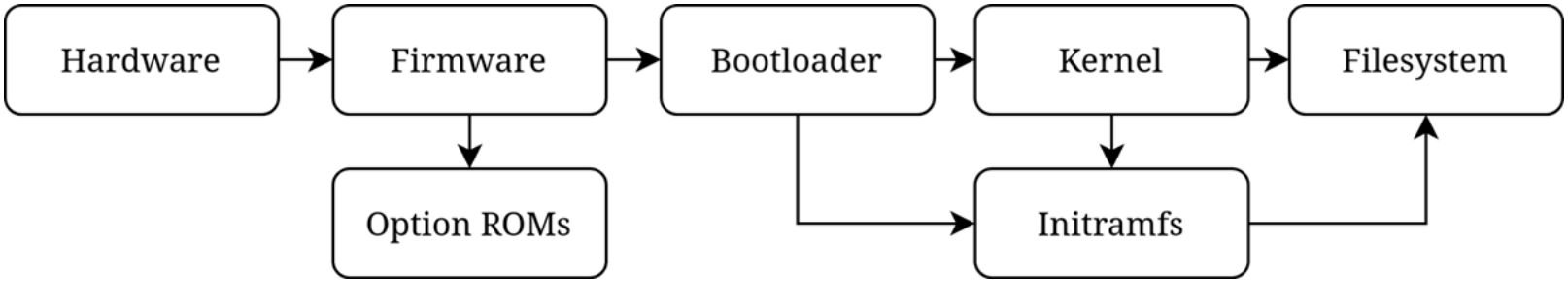
TPM Recap - Quoting

- TPMs expose a “quote” operation
- Quote signs PCR values using a TPM private key
- Verifiers can validate the PCR values using the TPM public key

- **What to know:** Quoting can cryptographically prove the state of PCR registers in a TPM

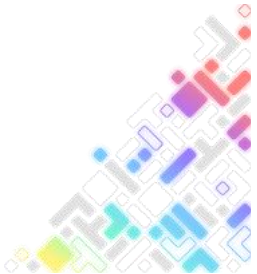


TPM Recap - UEFI Boot Sequence

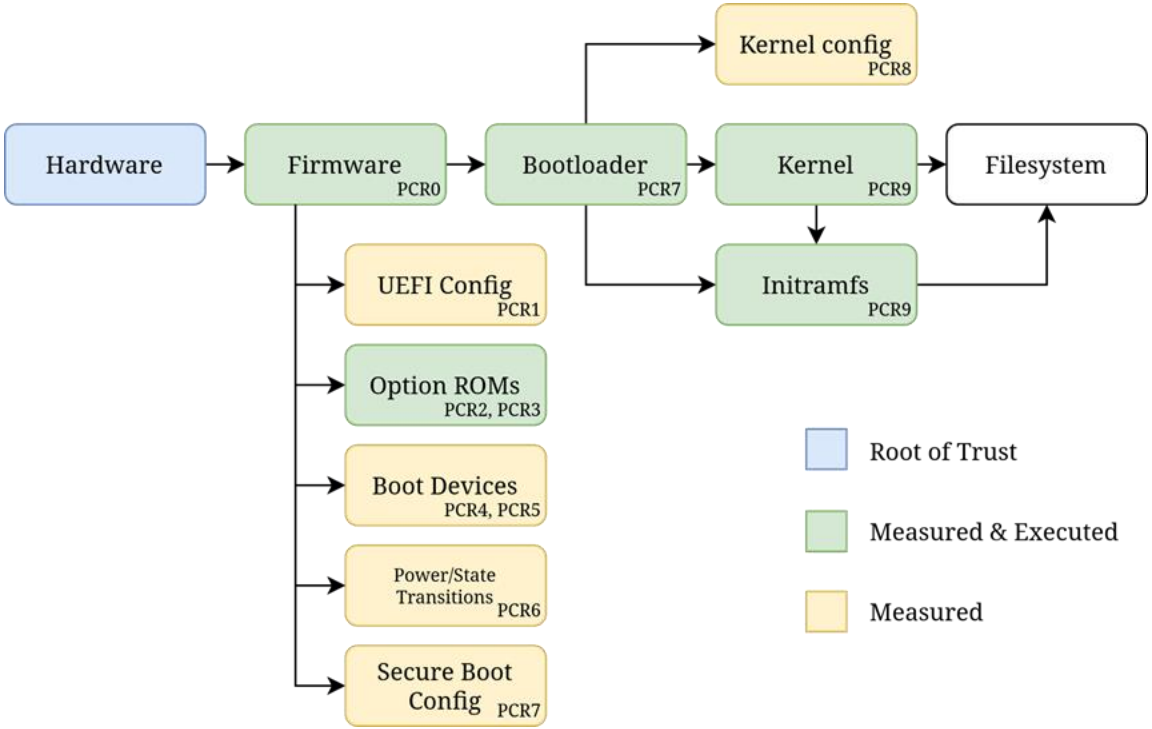


TPM Recap - Measured Boot

- Each step in the boot chain “measures” the next steps before execution
- Measurements are extended into specific PCRs
- Boot events written into a log

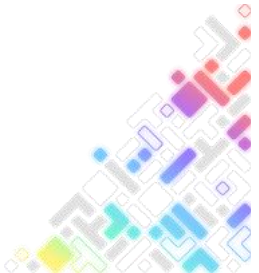


TPM Recap - Measured Boot for UEFI



TPM Recap - Measured Boot

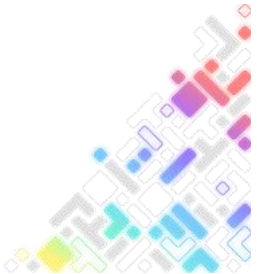
- **What to know:** the same boot chain results in the same PCRs
- **What to know:** PCR values can be reconstructed remotely using an event log



TPM Recap - Measured Boot

- Measure boot chain into PCRs
- Quote PCRs
- Validate quote with event log and TPM public key

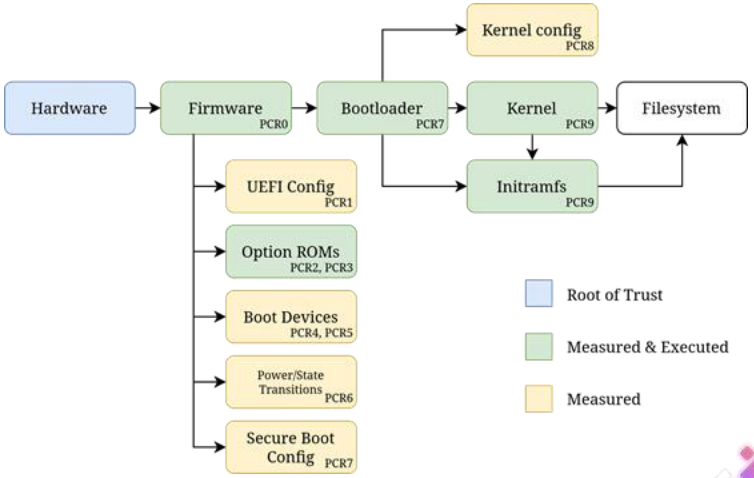
- **What to know:** *Assuming each boot layer is secure*, a third party can validate certain aspects of system state by following the boot chain



TPM Recap - Measured Boot

- Most measured boot systems do not attempt to measure the root file system
- FS is generally expected to be mutable

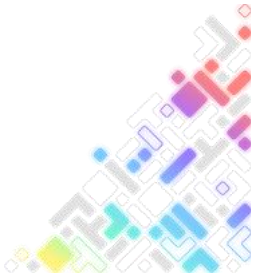
- Counter-example: Android, iOS



What’s special about CI?

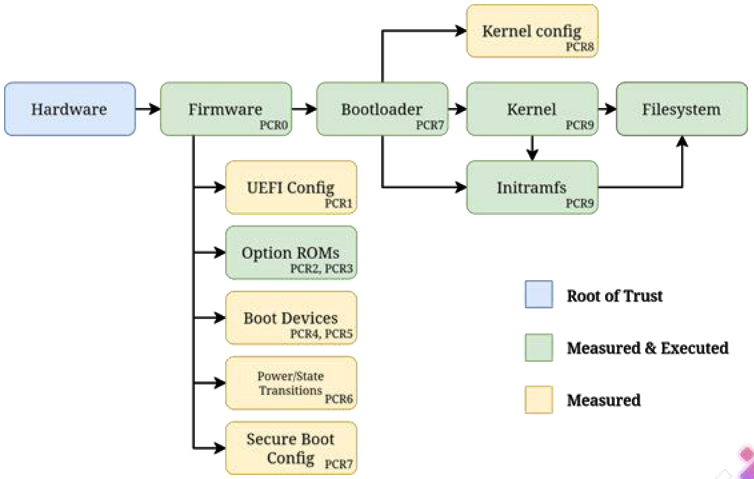
- In an ephemeral CI service, we always expect the disk image content to be exactly the same

- **How can we extend validation to the entire filesystem?**



File System Integrity

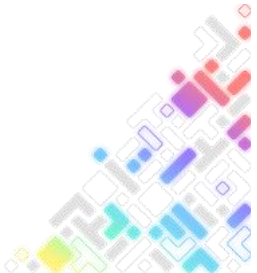
- Hook the initramfs
- Measure the filesystem
- Extend into unused/less used PCR



File System Integrity - Hashing

- Easy option: hash the block device

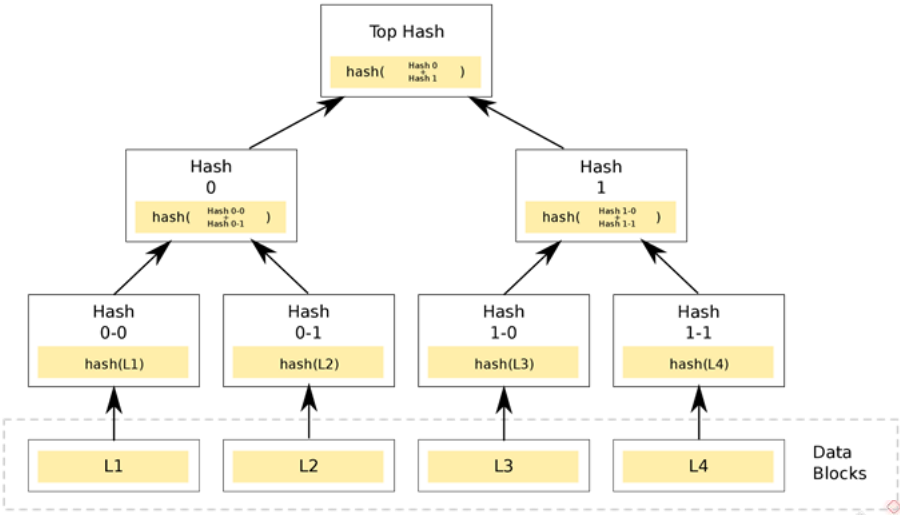
- Pros
  - Simple
  - Same approach as other measured boot layers
- Cons
  - Slow, potentially prohibitively



File System Integrity - *-verity

- dm-verity / fs-verity
- Merkle tree of hashes
  - dm-verity: hash block devices
  - fs-verity: hash files
- Recursively hash to a single root hash

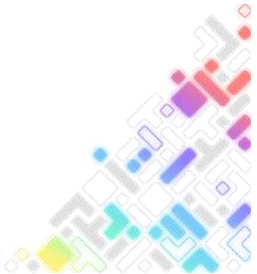
- Minimal verification perf overhead
  - log(n) verification cost
  - Heavily cached
  - CPU bound vs. latency bound disk reads



File System Integrity - *-verity

- Measure the *configuration* into PCR

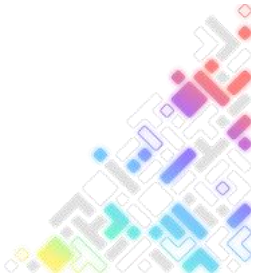
- Pros
  - Minimal upfront cost
- Cons
  - Additional complexity
  - Verity devices must be read-only
    - overlayfs
    - device-mapper snapshots



TPMs & Merkle Trees Summary

- Remote validation
- File system measurement
- Tech already exists

- **What to know:** We can generate VM image provenance and validate it on use



How do we trust, but verify a build platform?

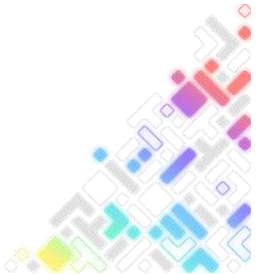
01	Trust to create a “good” build image	Publish build image SLSA Provenance
02	Trust to supply the “right” environment	Check environment vs. image SLSA Provenance
03	Trust to “protect” your executing build	Check execution context vs. build environment

TPMs + *verity

Binding Build Execution to Environment, Verifiably

- Idea: Include unique build execution ID in environment attestation pre-exec
- Goal: Detect if build environment has been tampered with during a particular build execution

- Good news! TEE hardware is designed for this use case



TEEs (Trusted Execution Environments), A Primer

- Capable of running entire programs, unlike TPMs
- Support measurement and quoting operations over (portions of) program memory to enable code & data integrity checking
- Typically support program data encryption

- **What to know:** Program data, *such as a unique build execution ID*, can be bound the TEE by including it in the TEE measurement.

Examples: ARM TrustZone, AWS Nitro Enclaves, Intel SGX, AMD SEV-SNP

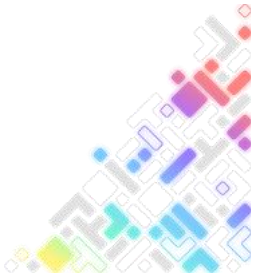
How CPU-Based TEEs Work

- Program memory measurement, extend, and quoting are implemented via dedicated CPU instructions
- Some: Hardware-encrypt memory for protection of code & data in use
- Note: Still need *verity for FS integrity

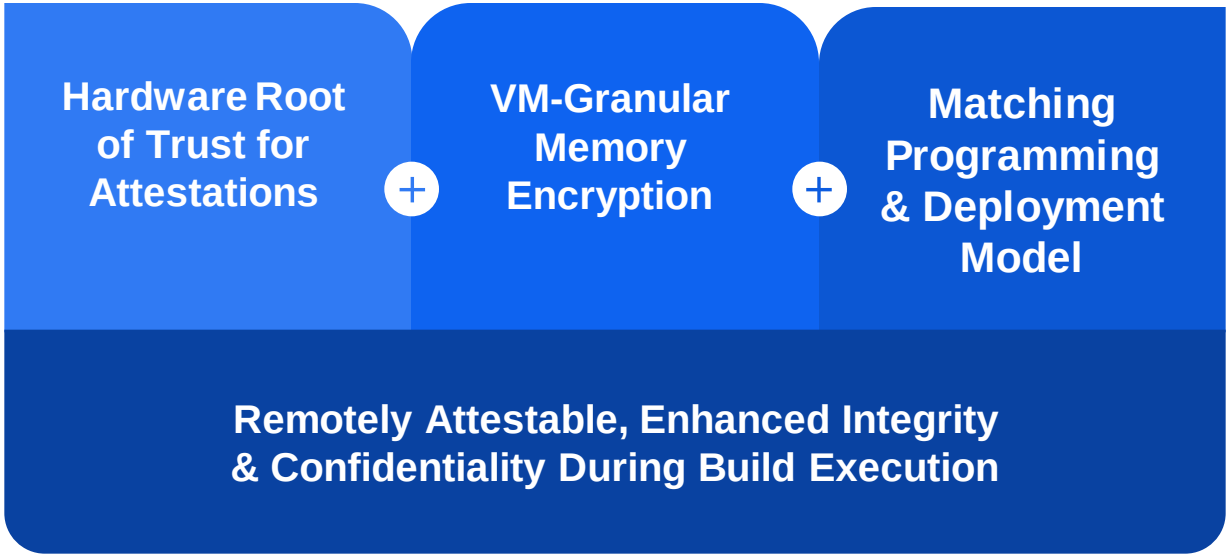
- **What to know:** Confidential Computing is a hardware-based TEE that supports attestation between remote parties (per CCC).

Examples: Intel SGX, AMD SEV-SNP, Intel TDX, ARM CCA

Source: https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/Common-Terminology-for-Confidential-Computing.pdf



Why Confidential VMs for Builds?



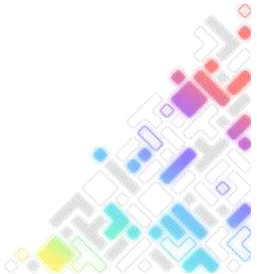
How do we trust, but verify a build platform?

01	Trust to create a “good” build image	Publish build image SLSA Provenance	
02	Trust to supply the “right” environment	Check environment vs. image SLSA Provenance	TPMs + *verity 
03	Trust to “protect” your executing build	Check execution context vs. build environment	Confidential Computing 

Coming back to SLSA: Proposed Enhancements

New Build Platform Requirements:

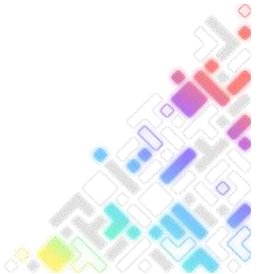
1) Build images have an associated SLSA Provenance attestation
2) The boot chain and disk image of build images are measured and HW-attested
3) Each unique build execution ID is bound to the build env via HW attestation
4) Attestations are verified before passing control to the build execution



Coming back to SLSA: Proposed Enhancements

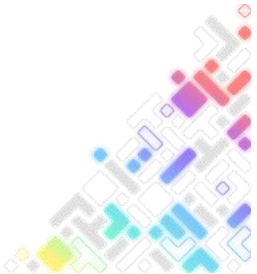
New Build Platform Requirements:

1) Build images have an associated SLSA Provenance attestation
2) The boot chain and disk image of build images are measured and HW-attested
3) Each unique build execution ID is bound to the build env via HW attestation
4) **Attestations are verified before passing control to the build execution**



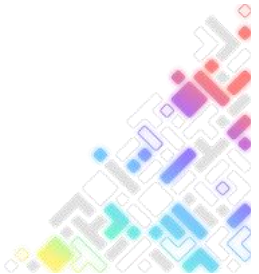
New Software Producer SLSA Requirements

1) Must run a build on a platform that meets the hardware-attested requirements
2) Should verify the build platform's attestation(s) prior to a build
    a) Provide proof of verification



New Software Producer SLSA Requirements

1) Must run a build on a platform that meets the hardware-attested requirements
2) Should **verify the build platform's attestation(s) prior to a build**
    a) Provide proof of verification



Takeaways:
Hardware-Assisted Build Environment Verification

Why — Build platforms are not immune to security compromises.

What — Hardware-based attested technologies increase the default security level.

How — SLSA is an open standard framework for enhancing build integrity.

Find More Info

SLSA Workstream:
Hardware Attested Platforms

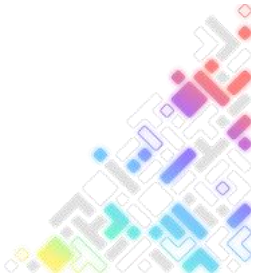
Check out our 30-min demo!
Apr 18 @ OpenSSF Booth 10:30am PT

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

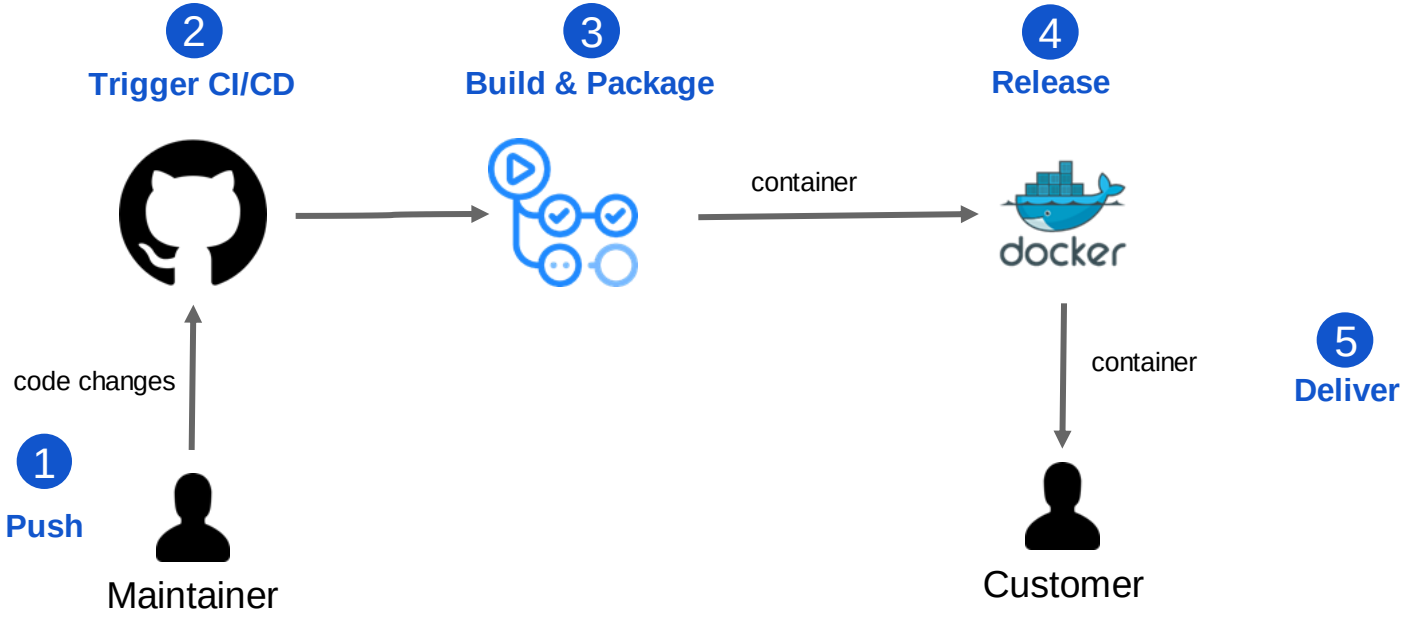


THE LINUX FOUNDATION

OPEN SOURCE SUMMIT

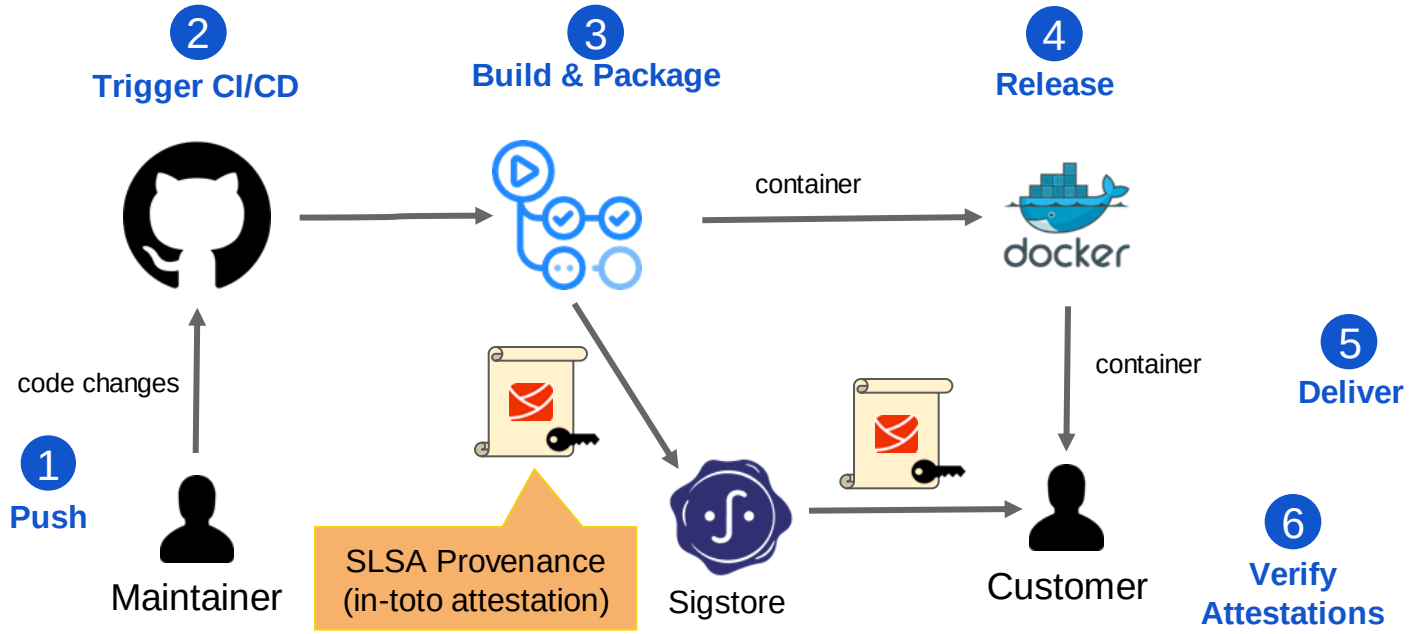
NORTH AMERICA

A Typical Software Supply Chain



Do I trust the software?

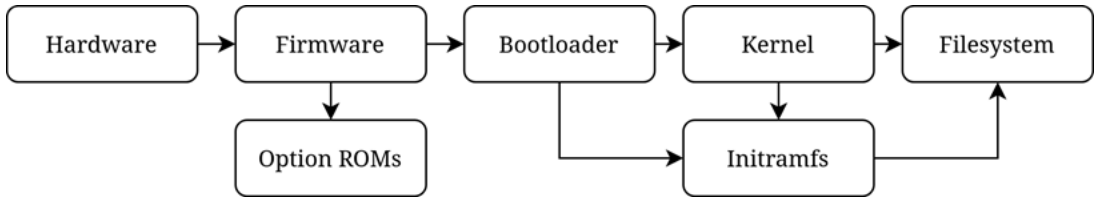
Trust State of the Art: Provenance, Attestations, Signing



Aside - What's an initramfs?

- "Initial ram filesystem"
- Commonly used for loading a minimal set of functionality required to set up your root filesystem

- Example: Disk encryption
  - something has to load unencrypted in order to decrypt and load the rest of the drive



TEEs (Trusted Execution Environments), A Primer

Compute environment (HW and/or SW) that provides:

Examples: ARM TrustZone, AWS Nitro Enclaves, Intel SGX, AMD SEV-SNP

Source: https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/Common-Terminology-for-Confidential-Computing.pdf