# Firmware Updates for Internet of Things Devices

Brendan Moran, Milosch Meriac, Hannes Tschofenig

Drafts:

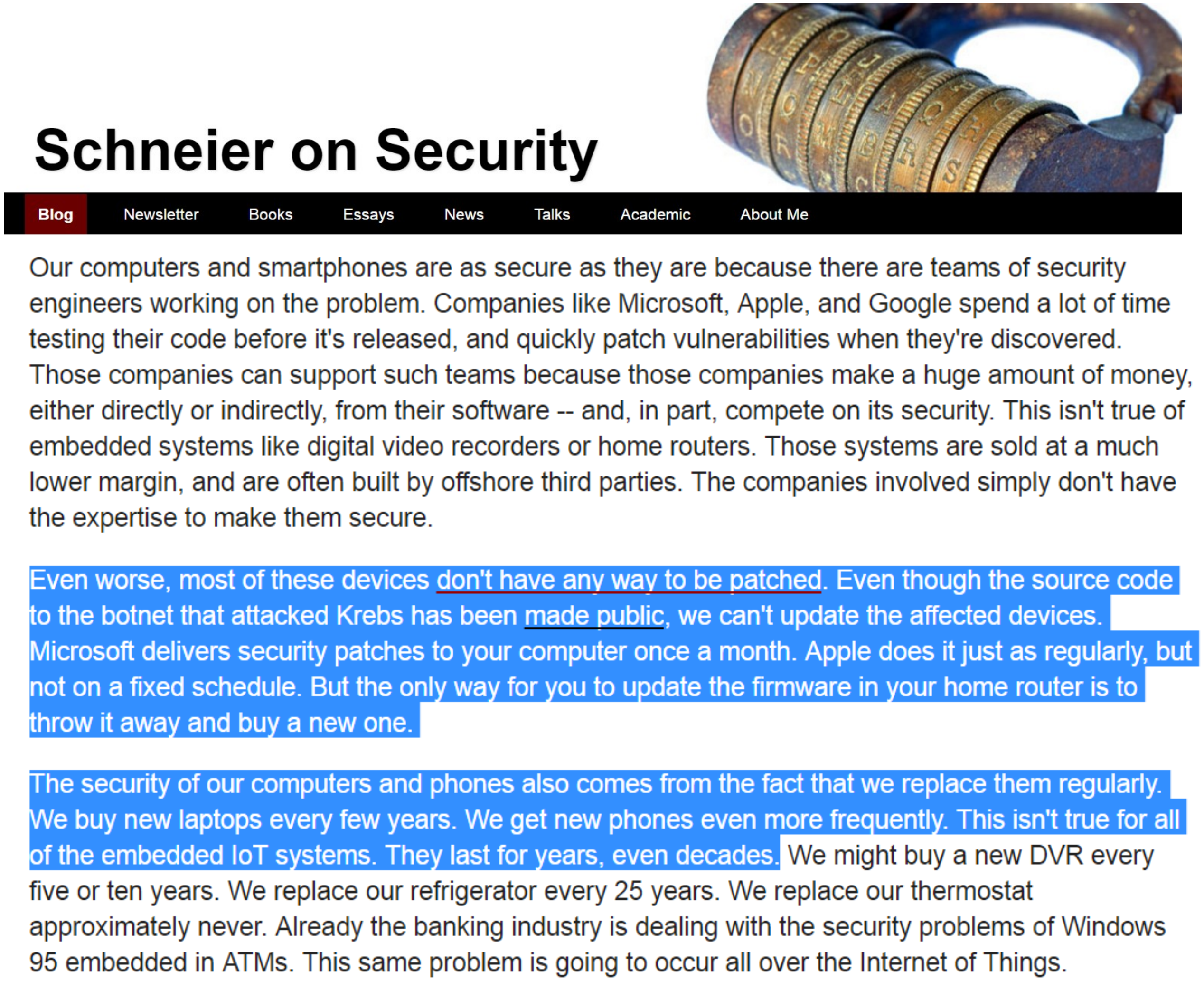
draft-moran-suit-architecture

draft-moran-suit-manifest

## WHY DO WE CARE?

# IoT needs a firmware update mechanism

Schneier in response to DDoS attacks using IP cameras

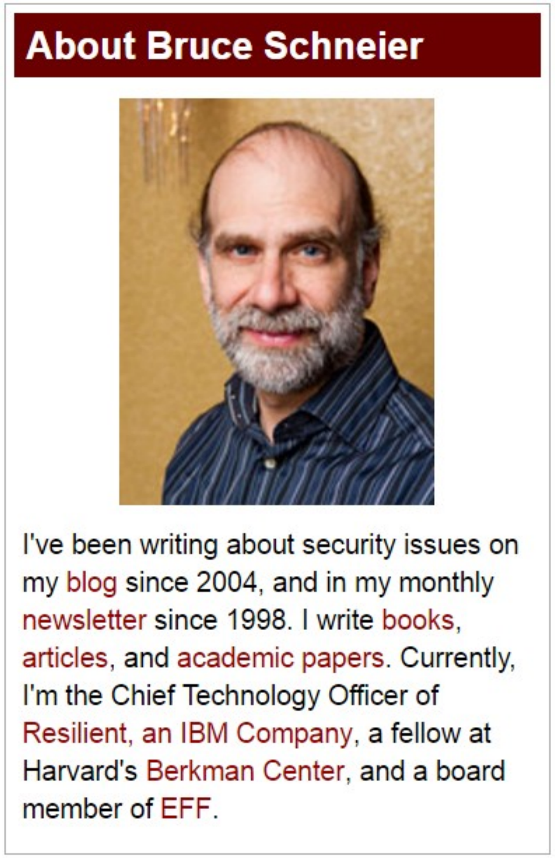


Schneier on Security

Our computers and smartphones are as secure as they are because there are teams of security engineers working on the problem. Companies like Microsoft, Apple, and Google spend a lot of time testing their code before it's released, and quickly patch vulnerabilities when they're discovered. Those companies can support such teams because those companies make a huge amount of money, either directly or indirectly, from their software -- and, in part, compete on its security. This isn't true of embedded systems like digital video recorders or home routers. Those systems are sold at a much lower margin, and are often built by offshore third parties. The companies involved simply don't have the expertise to make them secure.

Even worse, most of these devices don't have any way to be patched. Even though the source code to the botnet that attacked Krebs has been made public, we can't update the affected devices. Microsoft delivers security patches to your computer once a month. Apple does it just as regularly, but not on a fixed schedule. But the only way for you to update the firmware in your home router is to throw it away and buy a new one.

The security of our computers and phones also comes from the fact that we replace them regularly. We buy new laptops every few years. We get new phones even more frequently. This isn't true for all of the embedded IoT systems. They last for years, even decades. We might buy a new DVR every five or ten years. We replace our refrigerator every 25 years. We replace our thermostat approximately never. Already the banking industry is dealing with the security problems of Windows 95 embedded in ATMs. This same problem is going to occur all over the Internet of Things.



About Bruce Schneier

I've been writing about security issues on my blog since 2004, and in my monthly newsletter since 1998. I write books, articles, and academic papers. Currently, I'm the Chief Technology Officer of Resilient, an IBM Company, a fellow at Harvard's Berkman Center, and a board member of EFF.

## SCOPE

# Starting with Low End IoT Devices

<u>Low-end IoT Device</u>

- <u>Cortex M</u> class device
- Class 1 device from [RFC 7228]
- MPU, typically no cache
- High volume, low cost, very energy efficient
- Often run no OS or a small dedicated OS

<u>Not a low-end IoT Device</u>

- Cortex A / Cortex R-based microcontroller
- MMU, cache, DDR RAM
- Sophisticated security features (e.g., TrustZone)
- Use regular OS, such as Linux

If developed solution also works for high-end IoT devices ▭ Great!
If  solution does not work on low end IoT devices ▭ Fail!

## Working Group Scope

- Start small and extend later.
- Forming a working group soon helps to create awareness that the IETF is working on this topic.
  - We need to bring new community to the IETF, in particularly those developing OSs for embedded devices.
- Our preference is on the manifest format. No or little conflict with other, ongoing standardization activities in other SDOs.

## WHY STANDARDS?

# Value of Standards?

**Interoperability**

- IoT device developed by vendor A works with device management environment developed by vendor B
- While OEMs care about their devices they can re-use available tools and services developed by others.
- Server-side tools and IoT device side tools often get developed by different parties.

**Re-Use**

- Availability of best technology knowhow to develop a solution
- Open participation and open standards
- Availability of open source code and well-tested code
- Confidence in the technical solution.

## OUR CONTRIBUTION

## Contribution

- Implemented solution uses ASN.1-based encoding.
  - We are open to other encoding formats but we prefer to have a small number (ideally 1).
  - Integrated into mbed OS.
- Addresses security requirements outlined in architecture and here.
- Focused on asymmetric crypto in the first release.
  - We are planning to contribute a PSK-based solution to address even lower-end devices.
- Backup slides explain complexity of the topic.

## BACKUP

## Firmware update

Consumers rarely update their devices.

Businesses don't want the overhead of updating their devices.

This presents 5 types of problem:

- Device resilience to power failure, network loss, etc. (and associated costs of device replacement)
- Management of the authority to update devices
- Privacy of the updates
- Status monitoring of devices targeted by an update
- Selection of which devices to update

## Firmware update: the ideal

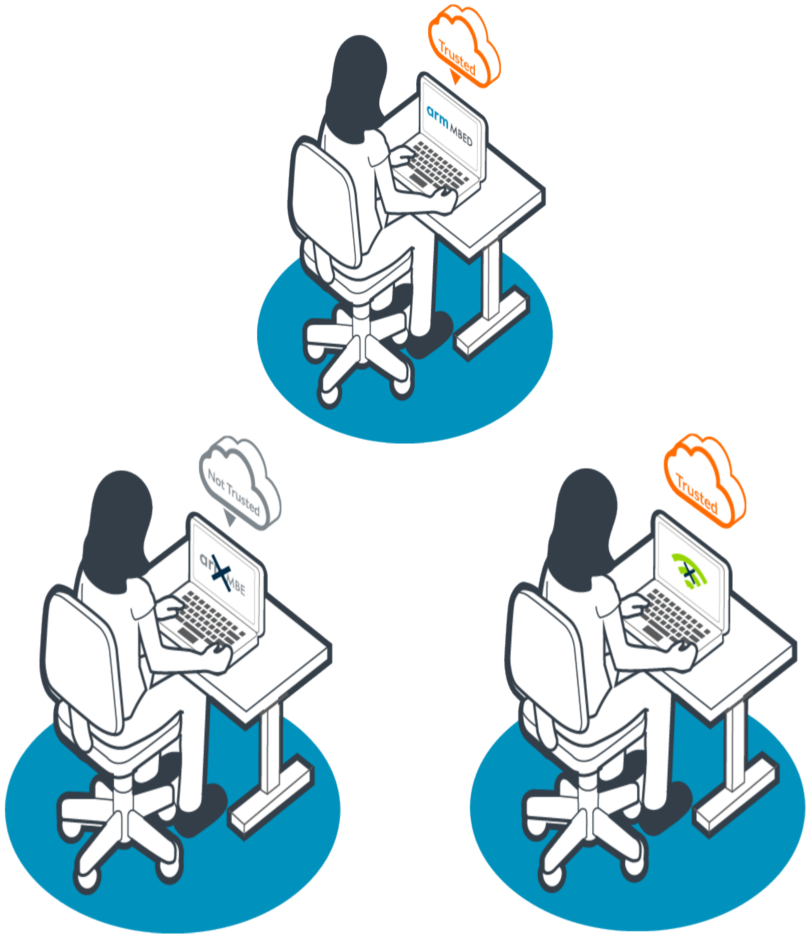
The model of firmware update is simple:

- Devices go through a series of steps:
  - receive a new firmware image from a trusted source
  - install the new firmware image
  - boot into the new image
  - the new image works completely
  - everything works seamlessly

# Firmware update: the real

The reality of firmware update is not so simple. Devices…

- lose network connection
- lose power
- receive firmware that doesn’t work
- receive firmware for the wrong device
- receive firmware from untrusted sources
- suffer flash memory failures on installation
- fail to boot the new firmware
- fail while controlling equipment



## FIRMWARE UPDATE: RESILIENCE

# Firmware update: resilience

Resilience decides how we ensure that a device always works.

To ensure that updates cannot fail, there must be a piece of code that cannot be updated.

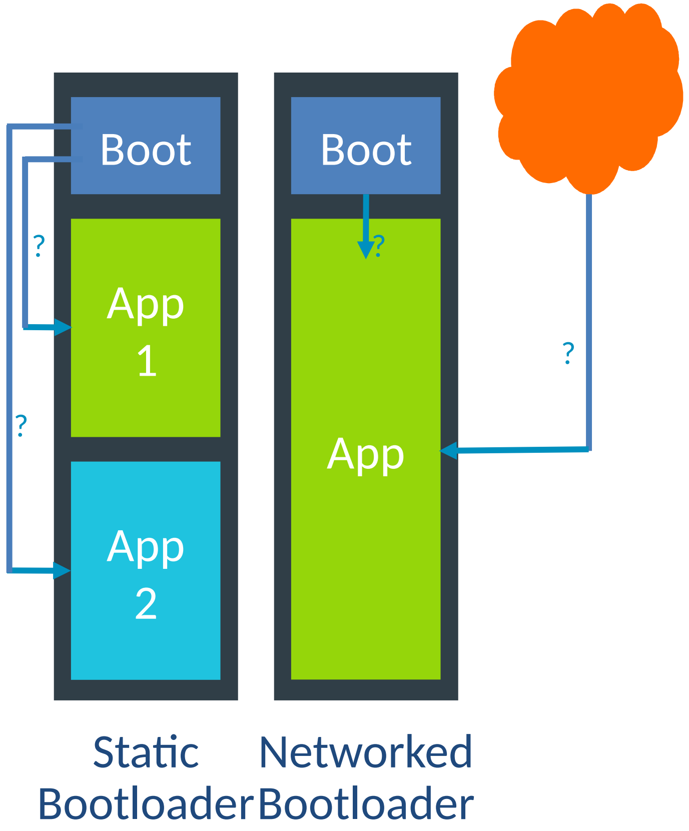
We call this code the bootloader. It ensures that only a valid image is loaded.

Devices need to keep at least two bootable images so that one always works.

# Firmware resilience: what kind of bootloader?

Bootloaders have one key differentiating feature.

- Networked
- Non-networked (or, static)



# Networked bootloader: the ideal

The bootloader contains a network driver, a network stack, and a full update client.

The bootloader:

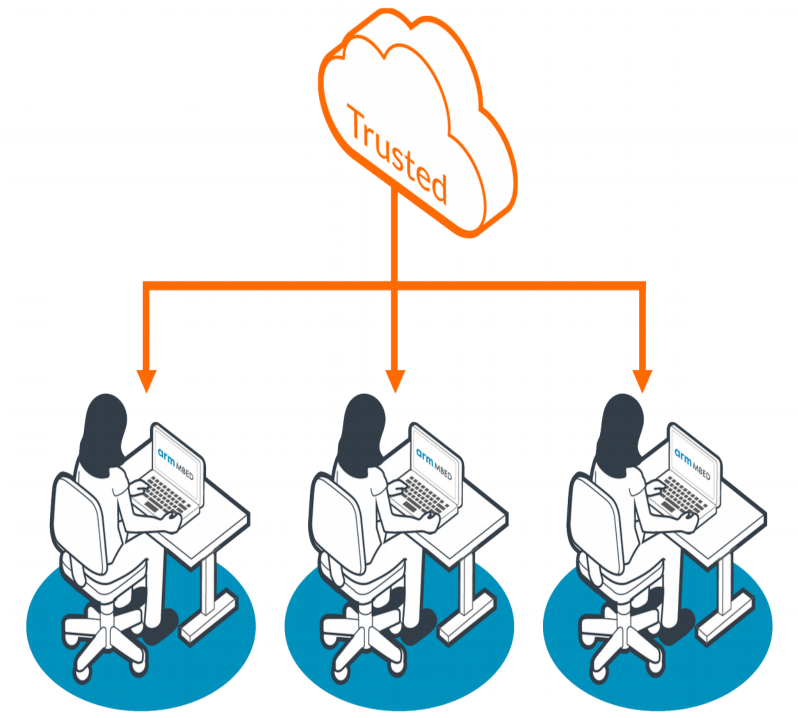
- Connects to an update server
- Downloads an update
- Authenticates the update
- Installs the update
- Hands-over to the new image

# Networked bootloader: the real

Networked bootloaders are large and complex, so the risk is high that they contain bugs.

The application and the bootloader require similar functionality.

The bootloader needs access to authentication data.



# Static bootloader: the ideal

Static bootloaders are very simple and cannot be updated.

- The application contains the update client.
- The update client:
  - downloads a new application image
  - validates the image
  - reboots
- The static bootloader picks which image to boot

# Static bootloader: the real

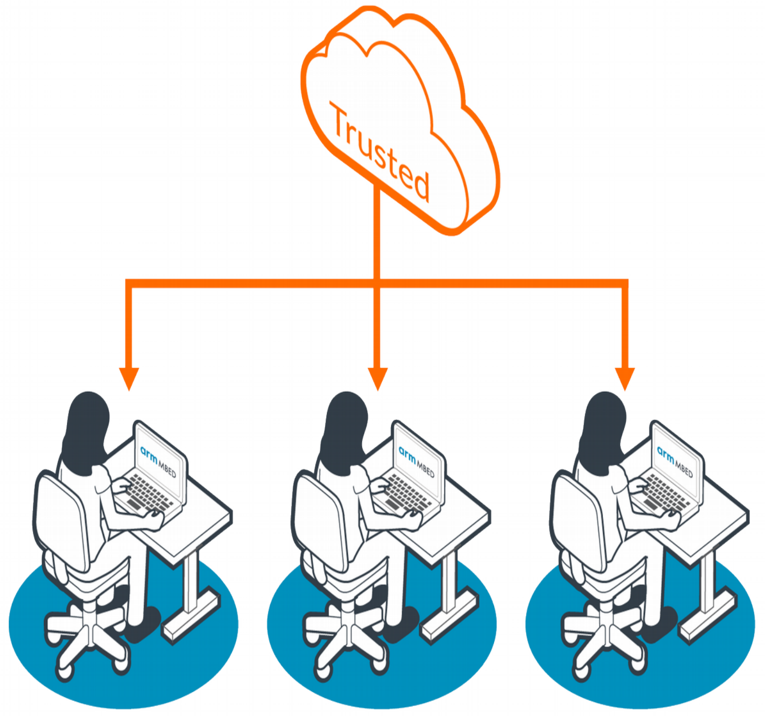
Application images are big.

Static bootloaders still need drivers if off-chip firmware images are used.

Sometimes, updates fail. The bootloader must know how to revert a failed update.

The bootloader cannot be updated. No data or format used by the bootloader can be changed.

The bootloader needs access to authentication data.



# Bootloaders: networked, or static?

Each has its own complexities.

There is no clear-cut answer.

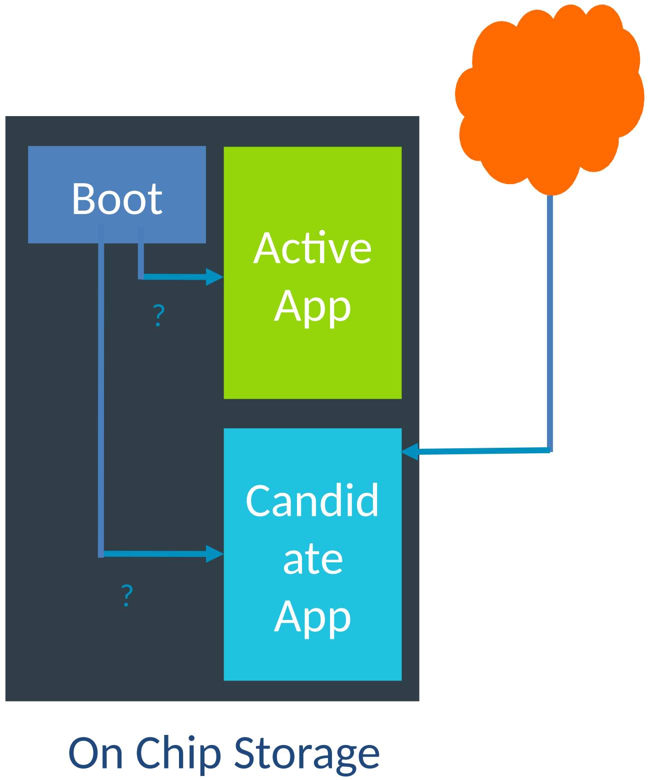
These approaches are not mutually exclusive:

- 2-stage boot
  - Stage-1 static bootloader
  - Stage-2 networked bootloaders
- Recovery image
  - Static bootloader selects the regular image, or a recovery image that contains only the update client

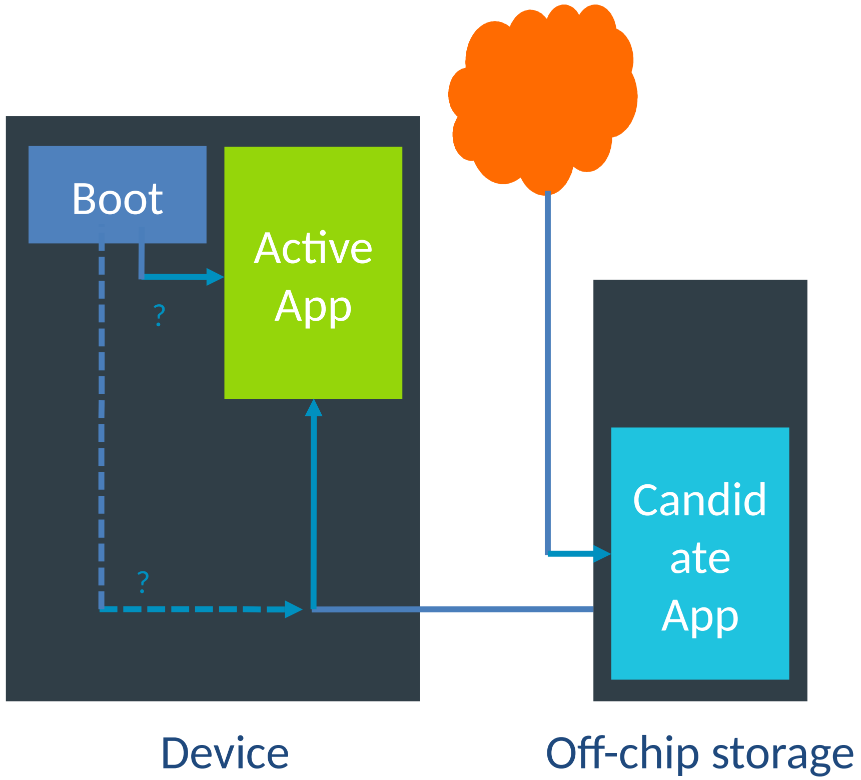
# Firmware resilience: firmware storage

Where should a new image be stored?

## On-chip, in the existing flash



## Off-chip, on an external storage device



## On-chip application image storage: the ideal

The safest place to store an application image is on-chip.

- Any code that can modify the image could equally modify the bootloader and remove any authenticity checks
- Only flash In-Application-Programming drivers are needed, so there are fewer points of failure
- The bootloader can (and should) be very simple

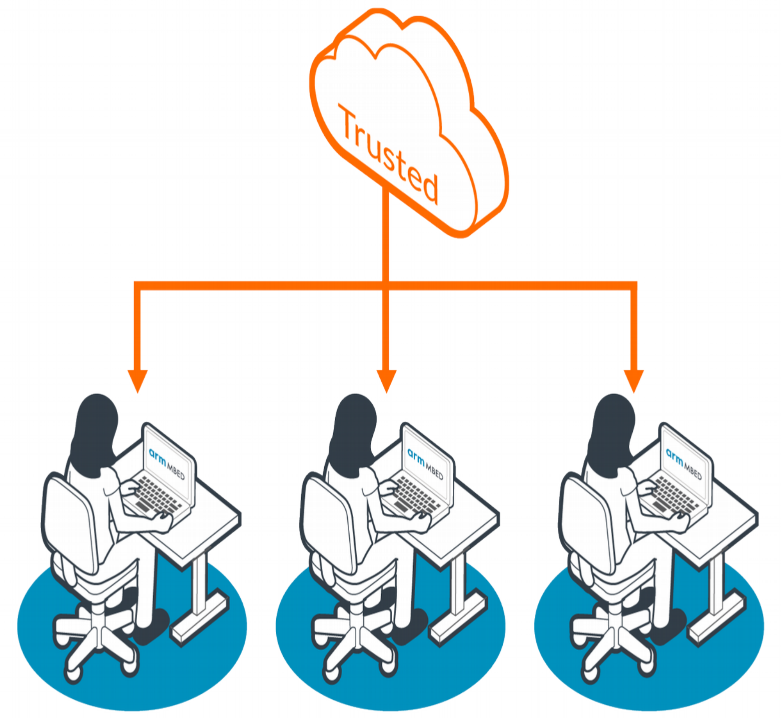
# On-chip application image storage: the real

The application is granted at most half of the remaining flash after the bootloader is installed.

On-chip flash is vastly more expensive than off-chip flash.

Images can be stored at more than one location, so one of these strategies is necessary:

- Copy the candidate image to the active image location
- Execute the candidate in-place



## Off-chip application image storage: the ideal

There's plenty of storage off-chip.

- Off-chip storage is inexpensive and plentiful.
- It allows a device to store many images
- It can be secured using simple cryptographic primitives
- It is simple to access
- It reduces flash cycles on the internal flash

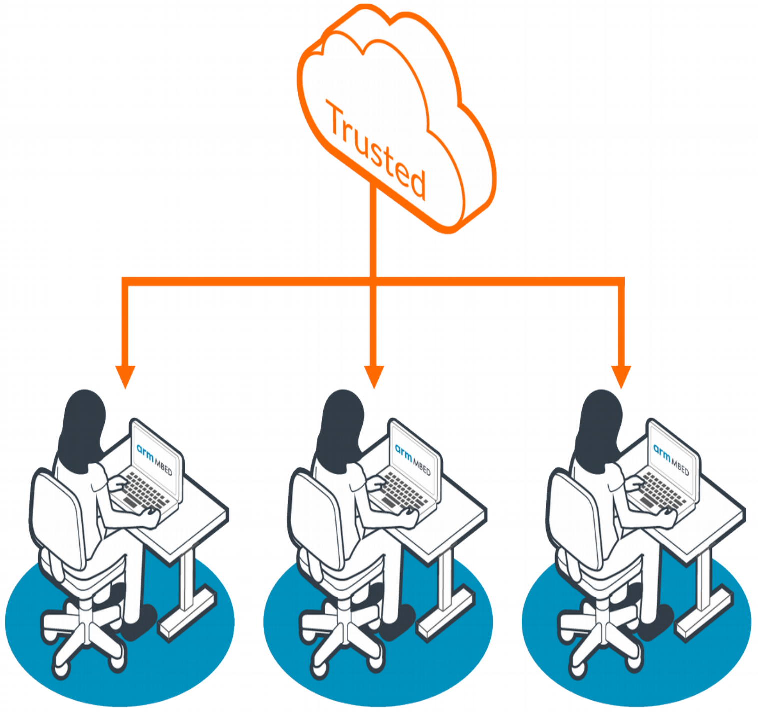
## Off-chip application image storage: the real

Off-chip flash is cheap, but it’s not free.

The security isn’t trivial.

Large storage space has challenges.

- A large off-chip storage requires management of where to store each image
- Image management begins to look like a filesystem
- Filesystems need journaling to withstand power loss
- It appears convenient to use for application purposes as well as update storage

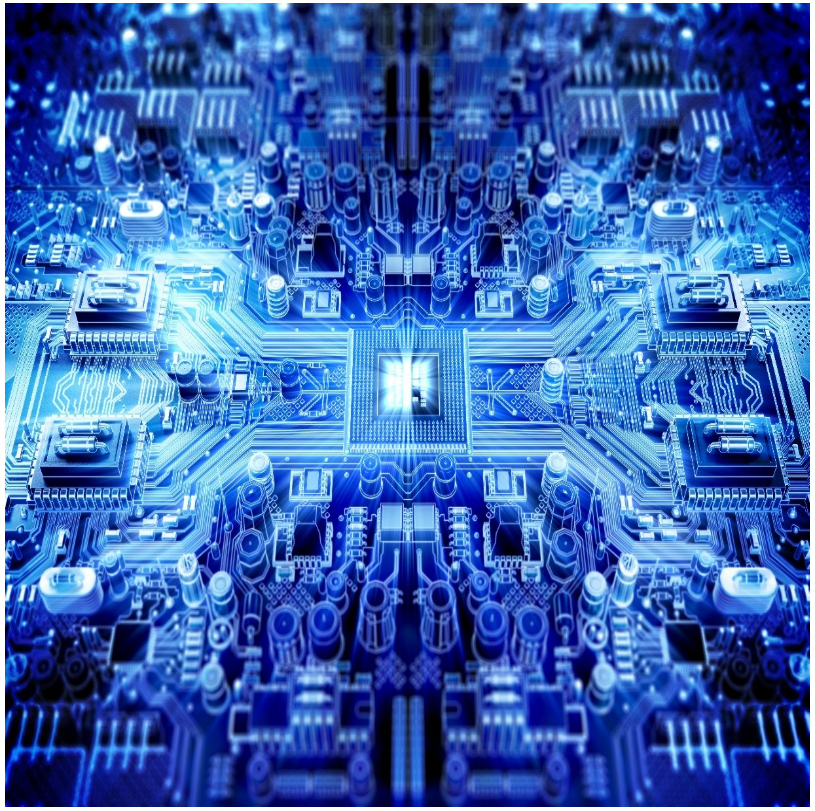


## Application image storage: on- or off-chip?

There are no clear-cut answers.

There are pros and cons for each.

Each design needs to weigh the trade-offs.



## FIRMWARE UPDATE: AUTHORITY

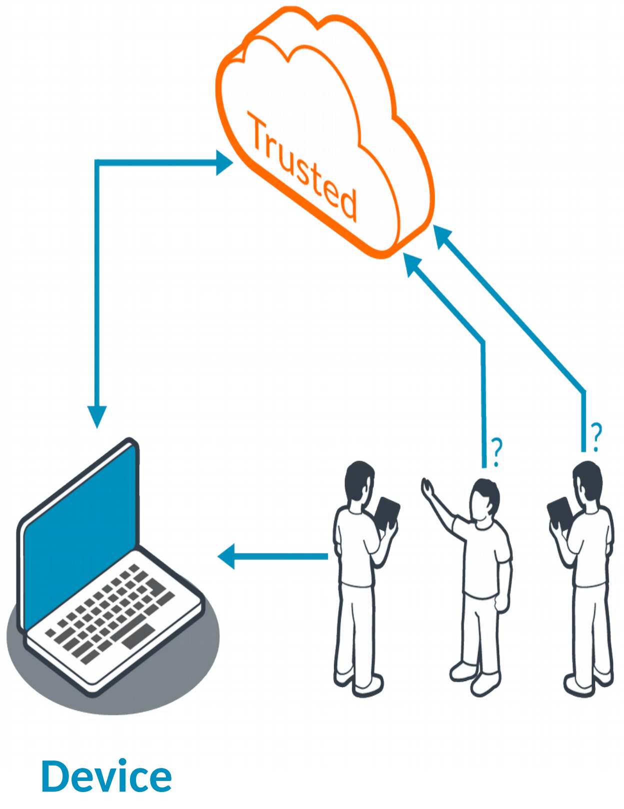
# Firmware update: authority

Authority in firmware update answers several questions:

- Who gets to write firmware?
- Who gets to install it?
- Who decides if it's been tested enough?
- Who decides if it's going to work on this network
- Who chooses when to install it?

There are two immediate options:

- Use TLS with a trusted server
- Use code signing



# Firmware update over TLS

TLS certificate infrastructure does authentication.

We can make the server choose who has what authority.

## Firmware update over TLS: the ideal

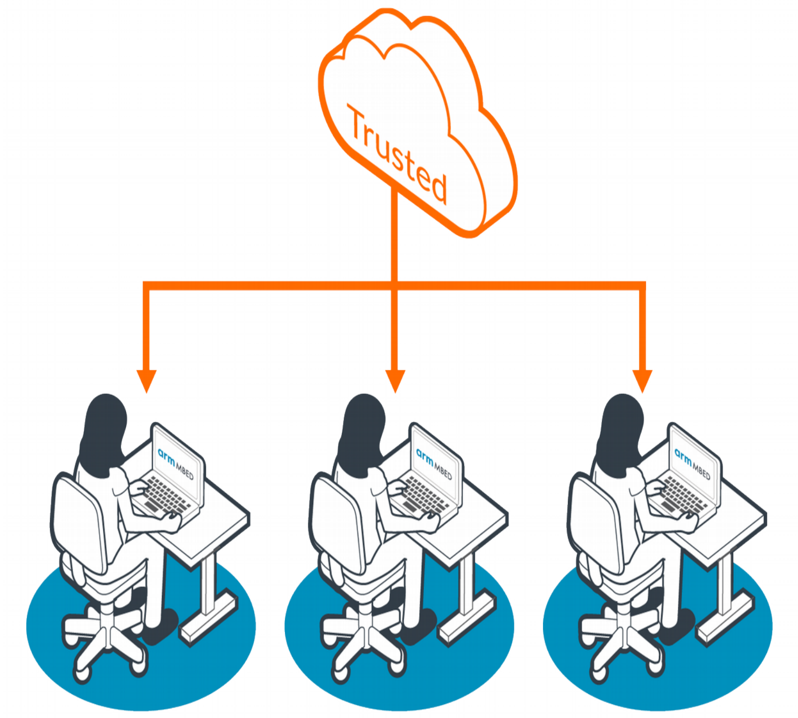
Developers authenticate with the TLS server to start firmware updates.

- Each device trusts the update server completely.
- The update server manages access control.
  - The developer logs in to the update server and uploads a firmware.
  - The update server decides whether or not to send the update, based on the developer's permissions
- Devices only need to trust one set of credentials.

# Firmware update over TLS: the real

A centralized trust system creates a centralized point of failure.

This presents a number of risks.



## Firmware update with code signing

Devices verify the firmware, not the connection.

- The device still trusts a certificate
- The certificate identifies a firmware author

## Firmware update with code signing: the ideal

An author can sign the firmware image before it is distributed.

- The devices trust the developer directly.
- The device verifies the signature of the firmware image before installing it.
- The risks posed by a centralized system are reduced because the author is trusted directly
- The author can perform signing on a very secure machine, such as a Hardware Security Module, which further reduces risk

# Firmware update with code signing: the real

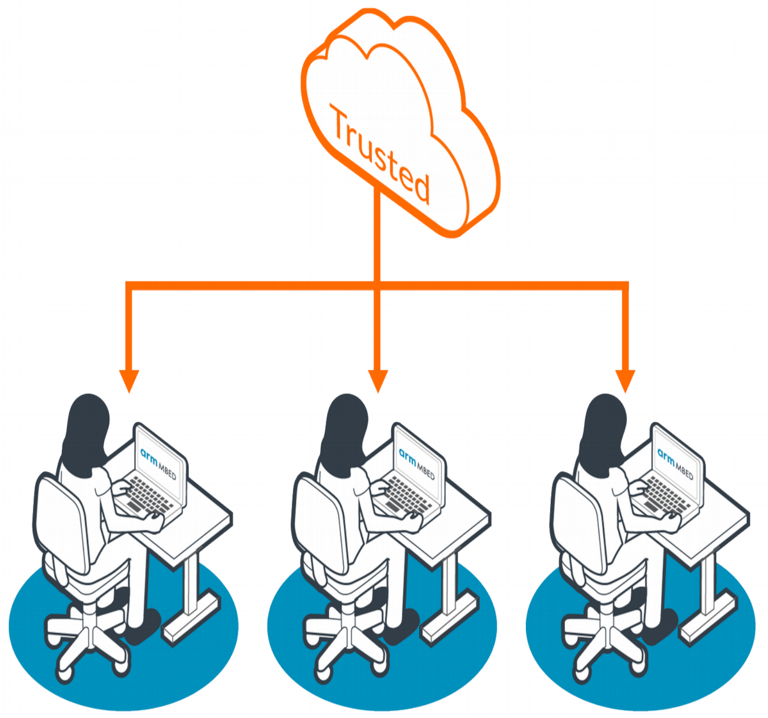
Devices are now responsible for access control.

Firmware authors are now responsible for security.

Devices must perform public key operations for each update.

Devices are exposed to increased risk from old firmware.

Devices must download the whole image before they can check the signature.



## Firmware update: transport security or code signing?

Code signing has significant benefits for security.

- Widely accepted practice in software, driver distribution.
- Signed metadata takes this one step further, offering early validation.
- This still doesn’t prevent a device from downloading the whole image before hash validation.
- Devices need to manage access control.

Transport security offloads the burden of access control.

- Devices aren’t required to handle access rights of individual firmware authors.
- They place the burden of maintaining security on the server.

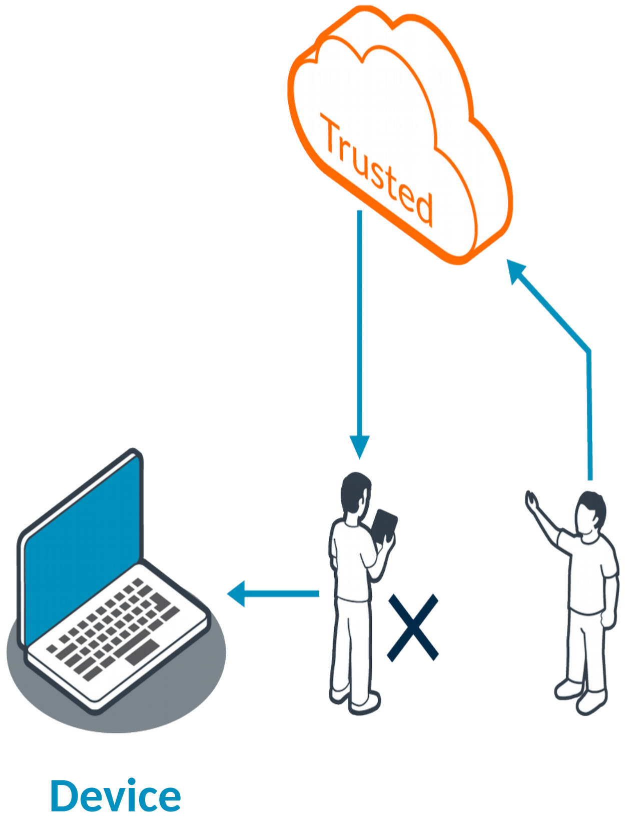
## FIRMWARE UPDATE: PRIVACY

# Firmware update privacy

To avoid exposing the firmware to a third party, it should be encrypted.

There are a number of ways to do this.

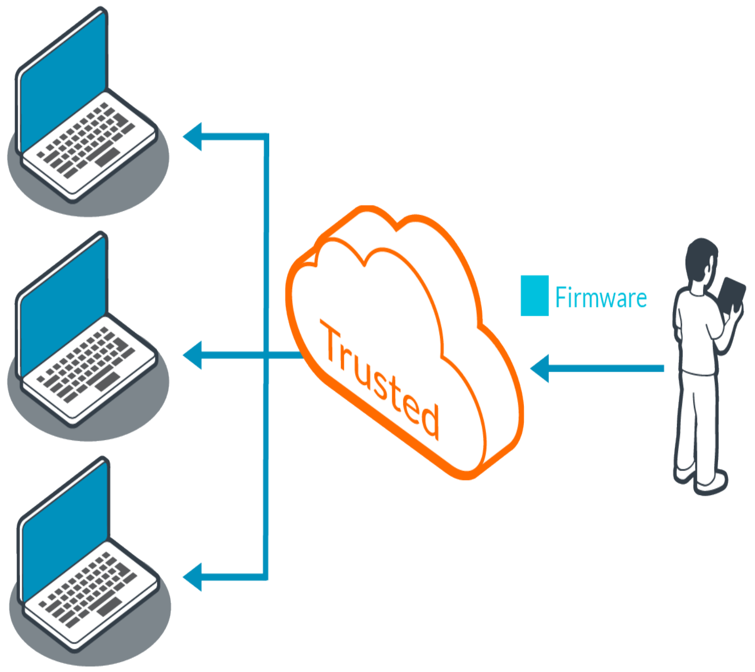
- Use transport security
- Encrypt the firmware image for each device
- Encrypt the firmware image for all devices



# Firmware update with transport security

To ensure that a firmware image is not exposed to a third party, it can be transmitted over a secure transport, such as TLS.

- The author uploads the payload in plaintext to the server.
- The server negotiates a session key with each device.
- The server sends the payload over an unique encrypted link to each device.



# Firmware update with transport security: the ideal

TLS provides adequate security for encrypted firmware distribution.

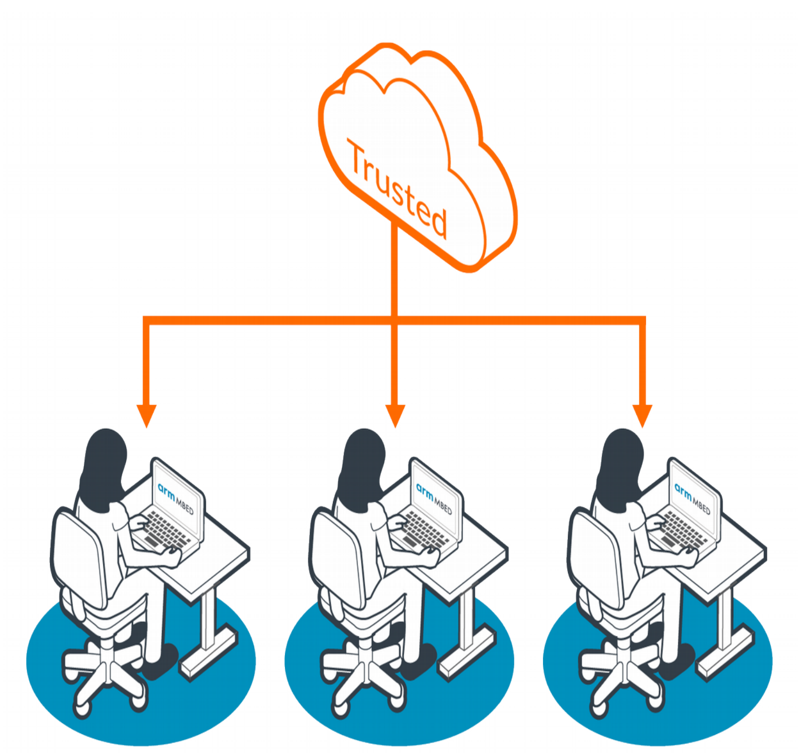
- Modern webservers are more than capable of handling distribution of firmware over TLS, to many devices.
- Existing techniques make this easy.
- The server has granular access control over which devices receive a firmware image.

## Firmware update with transport security: the real

The update server must be managed by a trusted party.

The device must trust the credentials of the update server.

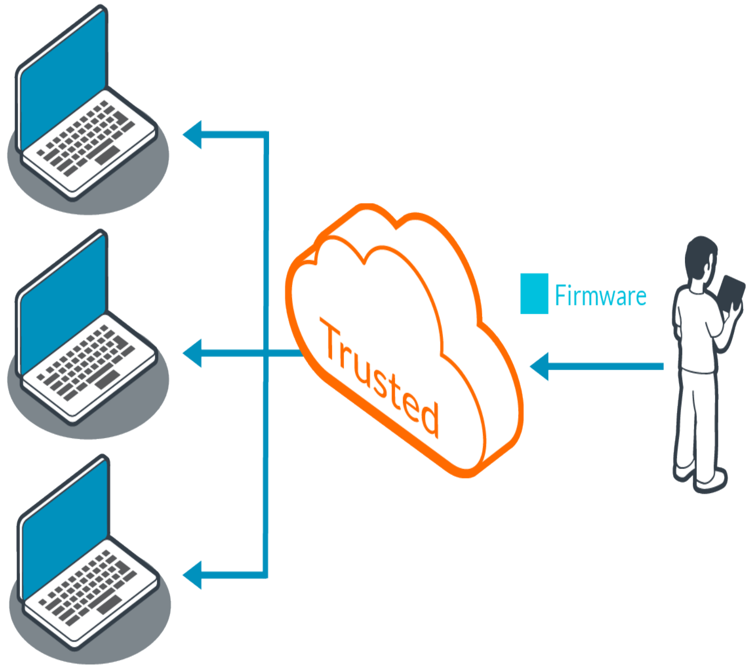
Transport security is not friendly to broadcast or mesh networks.



## Firmware update with per-device encryption

The firmware author encrypts an unique copy of the firmware for every recipient device.

- The firmware author builds a new firmware image
- They encrypt one copy of it for every device
- They upload all of these copies to a distribution service
- Each device downloads its own firmware image and decrypts it



# Firmware update with per-device encryption: the ideal

The firmware's privacy is guaranteed.

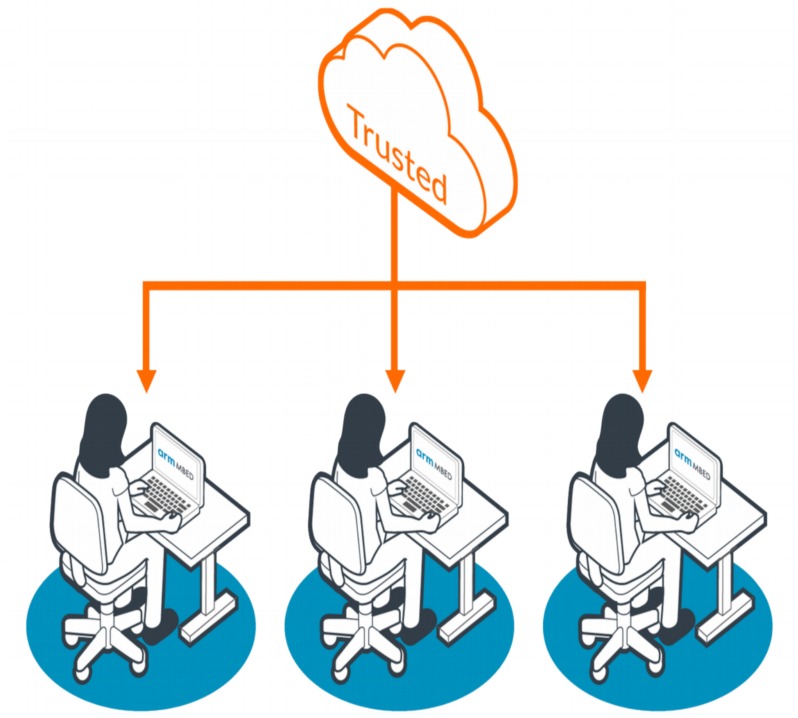
- The firmware author knows each device's encryption key.
- The firmware will not be exposed to the operators of any third-party service.
- No credential negotiation with the server is necessary
- There is no risk of confusing signing and transport credentials

# Firmware update with per-device encryption: the real

Key management is hard.

Per-device encryption doesn't scale.

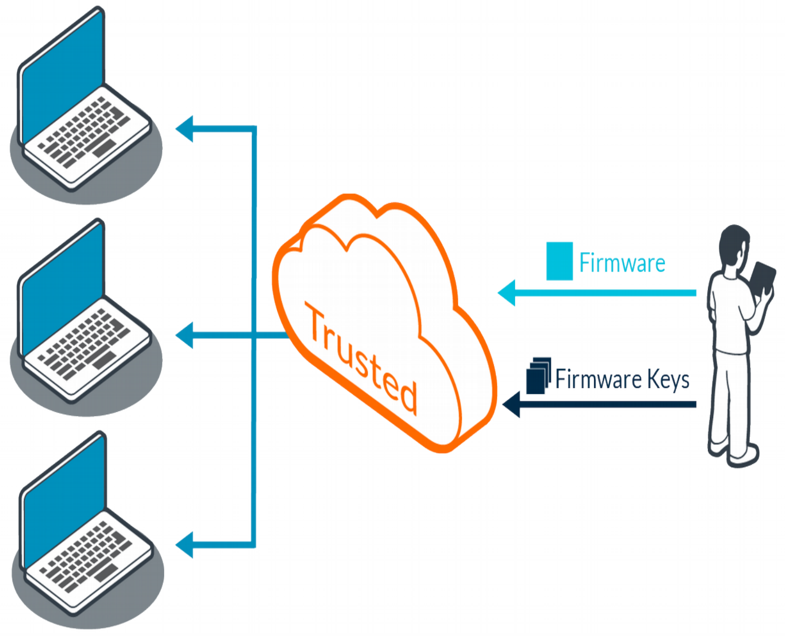
Per-device encryption is not friendly to broadcast or mesh networks.



# Firmware update with single image encryption

A single, encrypted firmware image is distributed.

- Each device also receives a copy of the image decryption key, encrypted using its unique encryption key.
- The device decrypts this with its unique encryption key.
- The device uses the image decryption key to decrypt the image



# Firmware update with single image encryption: the ideal

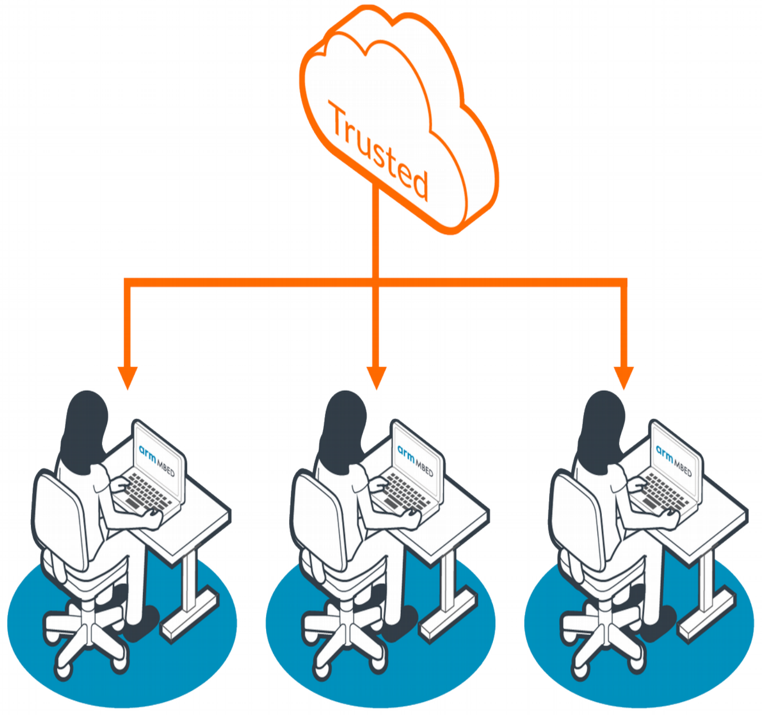
Now the image need only be stored once, and distributed once.

- The image is safe from exposure to a third party.
- The image is only stored once.
- Only the minimum of information necessary for security is uniquely distributed to each device.

# Firmware update with single image encryption: the real

Key management is still hard.

There is an extra step in the update process.



## Firmware update: which kind of privacy?

Single image encryption has a lot of benefits.

- Each option requires some amount of key management.
- Single image encryption is the most scalable.

Exposure of payload contents is of equal risk for each of these solutions.

## FIRMWARE UPDATE: MONITORING

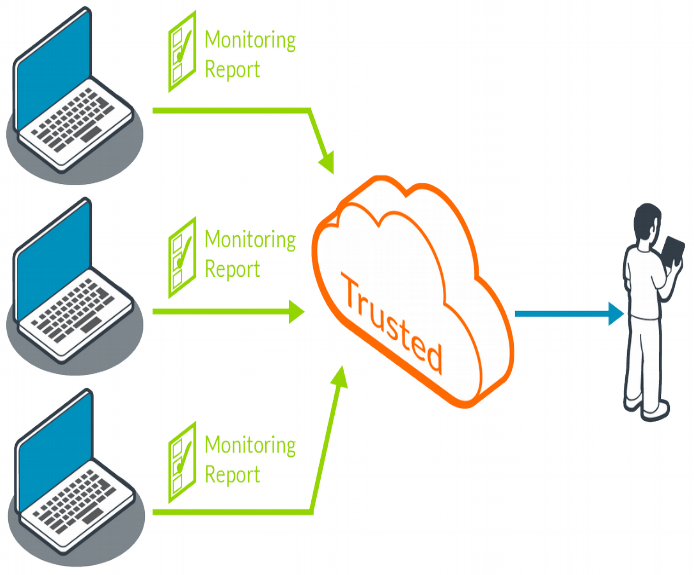
# Firmware update monitoring

Devices need to report:

- Their current firmware version
- The status of the last update
- Progress of the current download
- The types of payloads accepted
- The version of metadata accepted

Each device reports its invariant information when it connects to the monitoring server.

Devices report variant information when it changes.



# Firmware update monitoring: the ideal

The monitoring server aggregates the information from devices and presents it to the user who is managing the update.

- Users can see, at a glance, the status of all their devices.
- Errors are highlighted and corrective actions are suggested to the user.

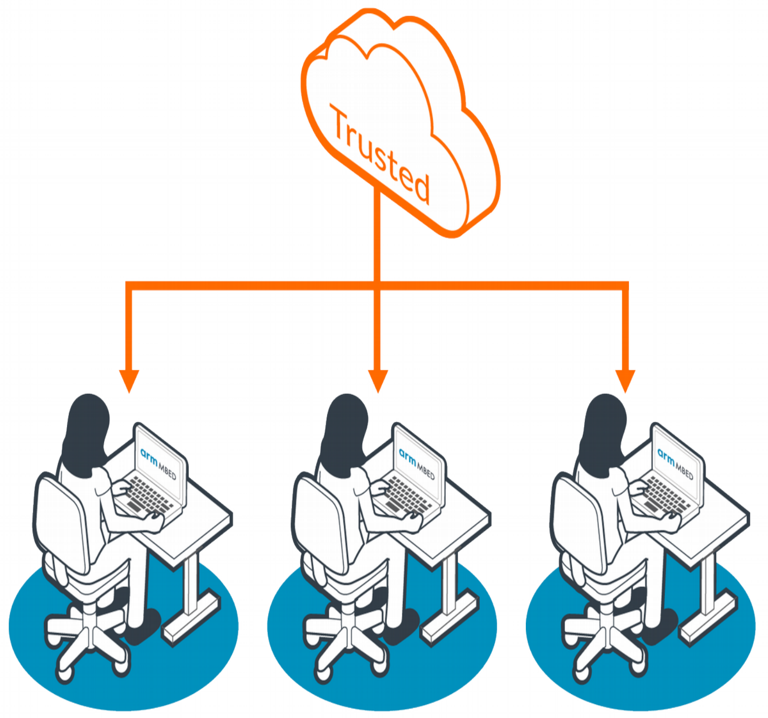
# Firmware update monitoring: the real

The data collected scales with the number of active devices.

The devices need to decide which errors are recoverable and which are not.

Errors which are not recoverable must be communicated to the operator.

Decisions are made based on monitoring reports.

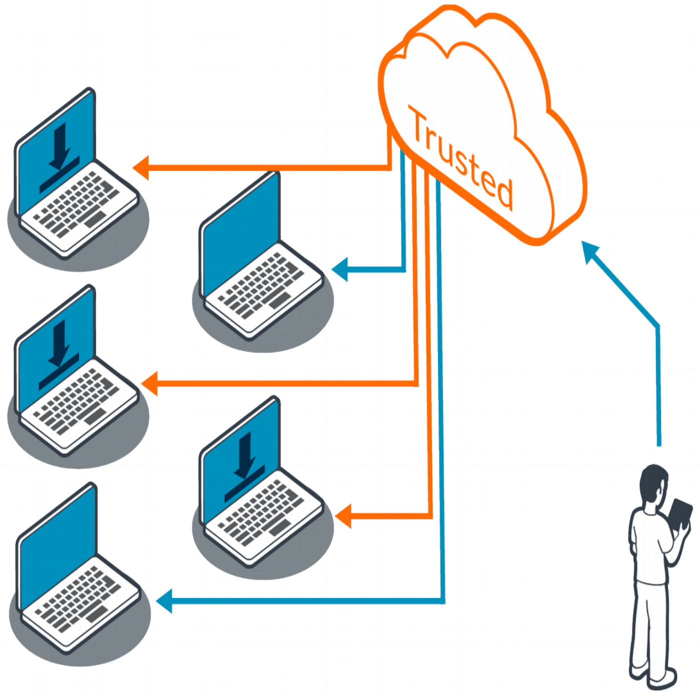


## FIRMWARE UPDATE:
## TARGETING

## Targeting firmware update

Only the correct devices should be targeted for a firmware update.

- Operators select the devices that they want to be targeted with updates
- Each update is only delivered to a targeted device



# Targeting firmware update: the ideal

The operator can select a group of devices.

- They can be confident that all relevant devices are targeted
- They can select devices by a variety of parameters, such as: Vendor & Model, Current firmware version, Owner, Geographic location
- The operator can instruct the system to update some or all devices automatically when the vendor publishes new firmware
- When a device comes online for the first time, it is automatically updated
- The operator can select a phased roll-out to minimize risk

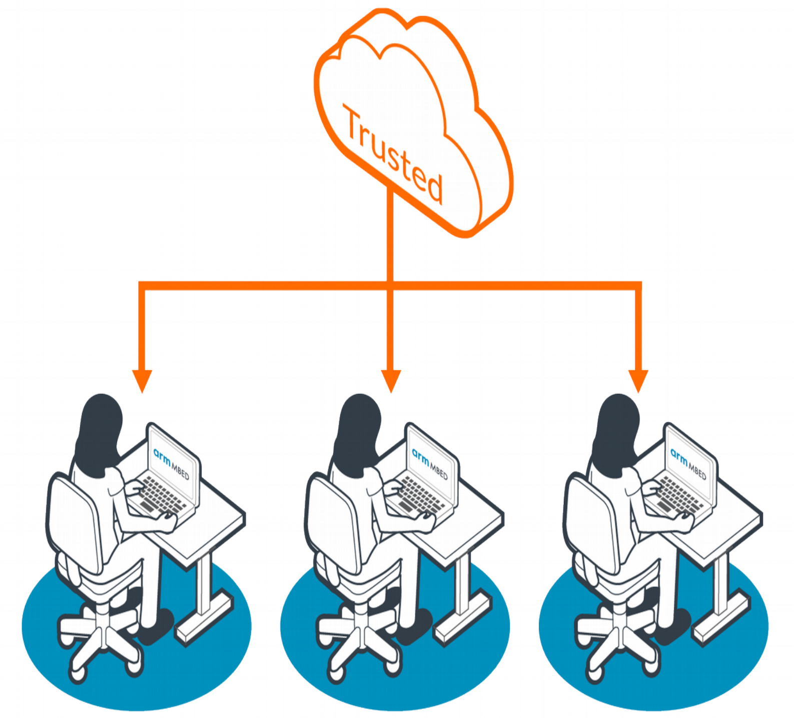
## Targeting firmware update: the real

The operator can’t target an offline device.

First-time-update has authority problems.

Targeting devices is either too precise or too imprecise.

If devices must coordinate, a phased roll-out could cause device interactions to break.



## SUMMARY

## Building firmware update

It seems like it should be easy.

There are many subtle ways that things can go wrong in five areas:

- Device resilience to power failure, network loss, etc.
- Management of the authority to update devices
- Privacy of the updates
- Status monitoring of devices targeted by an update
- Selection of which devices to update

Each of these issues requires careful considerations and tradeoffs to be made.

For some issues, there are no clear answers.