Understanding RP2350's security features

Colophon

© 2022-2025 Raspberry Pi Ltd

This documentation is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International (CC BY-ND).

Release	1
Build date	17/11/2025
Build version	b029b8c12d39

Legal disclaimer notice

Document version history

Release	Date	Description
1	20 Nov 2025	Initial release

Scope of document

This document applies to the following Raspberry Pi products:

Microcontroller Boards

Pico	Pico2
	✓

RP-series Microcontrollers

RP2040	RP2350		RP2354	
-	A	B	A	B
	✓	✓	✓	✓

Introduction

RP2350 is the latest addition to Raspberry Pi Ltd's RP-series microcontroller range. One of RP2350's main improvements over its predecessor, RP2040, is its advanced security features. This document outlines these features and provides walk-throughs and guides on how to use them effectively.

Security on RP2350 is entirely optional; if your use case does not require the security measures described in this document, there is no need to enable them. It should also be noted that once the security features are turned on, they cannot be turned off.

Why might you need to implement security?

There are many reasons why you may need to secure a RP2350-based device — these range from keeping commercial secrets secret to making sure your product adheres to local laws (e.g. the EU's GDPR and CRA regulations).

You may also wish to ensure that the software running on your RP2350 chip cannot be altered to behave in an unauthorised manner, like when an IoT device is commandeered to make DDoS attacks over its network or a security camera is subverted to send its images to an untrusted third party. The security features detailed in this document will help protect your RP2350 chip against these types of attack.

Types of attack

There are many ways to attack a device in order to break its security. We'll cover a few of them here.

Remote access

Remote access attacks are possible if an attacker can connect to the device via a communication network of some sort, e.g. Wi-Fi, Ethernet, or Bluetooth. Successful remote access attacks require the software on the device to be badly secured or faulty in some way, which is why using signed software is preferable.

If your device can be updated in the field — which is becoming more and more common and, in some jurisdictions, is a legal requirement — it should only be able to be updated with the correct software. Requiring software to be signed ensures this will always be the case.

You may also wish to prevent exfiltration of data from the device; this is where encrypted binaries can be of use. Even if an attacker manages to extract data from the device through a security hole, that data will be encrypted, and therefore secure.

Physical access

If an attacker has physical access to the device, there are many more vulnerabilities to consider.

Debugger access is an obvious security risk, so all mechanisms for debugging should be disabled on released products.

In an effort to bypass the signing process, an attacker might physically remove the flash device from the product, then multiplex that chip with one of their own. If they time the multiplexing right, the device will initially select the original flash chip to pass the signature check before switching to the attacker's flash and running the attack code. This means you cannot trust any code that is stored in a device's flash memory if an attacker has physical access to the device.

Glitching

Glitching, also known as fault injection, is a form of hardware attack in which a device's operating conditions are manipulated to cause unexpected behaviour, potentially bypassing security measures. Some examples include:

Voltage Glitching
: The attacker briefly disrupts the power supply to the target device, causing a momentary drop or increase in voltage. This can make the processor skip instructions or corrupt data.

Clock Glitching
: The attacker injects short, incorrect signals into the device's clock line, disrupting the timing of operations. This can confuse the circuit into skipping security checks or storing incorrect data.

Electromagnetic (EM) Fault Injection
An attacker blasts a chip with a focused electromagnetic pulse, inducing localized transient voltages within the chip to alter its behaviour.

Laser Fault Injection
Using a focused laser beam (ionizing radiation) to physically alter the state of individual transistors or logic gates on a decapsulated chip, allowing for extremely precise bit flips or instruction replacements.

Temperature Variation
Operating a device outside its specified temperature range (e.g., overheating or cooling) can induce faults and lead to security breaches.

Side-channel analysis

Side-channel analysis (SCA) is a hacking method used to circumvent system security by analysing leaked information. Leaks can be exposed in various ways, through the analysis of things like chip timing, power consumption, or electromagnetic emissions.

SCA is typically used to reverse-engineer or weaken important cryptographic operations, allowing the attacker to gain access to secret key material.

2024 Hacking Challenge

In late 2024, Raspberry Pi Ltd opened a hacking challenge to encourage users to find gaps in RP2350's security model. Following a number of successful attacks, Raspberry Pi Ltd updated the RP2350 chip to the newer A3 stepping, fixing the majority of the issues found. Further updates have now been released in the A4 stepping, including fixes for the security vulnerabilities discovered in RP2350's boot ROM.

2025 Hacking Challenge

In mid-2025, Raspberry Pi Ltd launched a second hacking challenge, focusing on practical side-channel attacks to extract the AES key material used by our SCA-hardened AES implementation.

At the time of publication, this second hacking challenge is still in progress. Read the official challenge page for more information: https://github.com/raspberrypi/rp2350_hacking_challenge_2

Further reading

This whitepaper should be read alongside the RP2350 Datasheet — particularly 'Chapter 5: Boot ROM', and 'Chapter 10: Security' (which provides detailed technical information on the security features) — as well as the SDK documentation, specifically 'Chapter 4: Signing and encrypting', which includes information on the functions Raspberry Pi Ltd provides for implementing security features in user products.

These documents can be found in Raspberry Pi Ltd's Product Information Portal (PIP): https://pip.raspberrypi.com/

The security features

There are three main purposes to the security features on RP2350:

1. Preventing unauthorised code from running on the device
2. Preventing unauthorised reading of user code and data
3. Isolating trusted and untrusted software from one another

Point 1 is referred to as **secure boot**.

Point 2 is referred to as **encrypted boot**.

Point 3 enables the enforcement of internal security boundaries, ensuring that if part of an application is compromised, access to critical hardware remains restricted.

On top of these three points, there are also extra hardware layers to help protect against various attack vectors at the hardware level.

One-time programmable (OTP) memory

Before looking at security in more detail, it's worth describing the function of one-time programmable (OTP) memory. OTP memory is the underlying storage mechanism on which much of RP2350's security relies. As the name implies, OTP memory can only be programmed once, as it uses an irreversible physical process to change zero bits to one.

Each RP2350 chip has 8 kB of OTP memory, and this can be protected at a 128-byte granularity using either hard or soft locking. Hard locking permanently revokes (by using an OTP bit and/or page keys) read or write access for secure or non-secure code; soft locking revokes permissions up until the next time the OTP block is reset. Resetting the OTP block (and hence the soft locks) also resets the processors, which ensures that soft locks cannot be bypassed.

OTP memory is used to store the permission flags that enable secure boot, as well as encryption keys, signing key hashes, and so on. Permissions also specify what can be accessed by the boot ROM when the device is in BOOTSEL mode.

When a decryption process uses an encryption key, it can then soft-lock that key so that nothing else can subsequently read it.

See section 10.8. OTP in the RP2350 Datasheet for more details.

Secure boot

In short, secure boot means that only the code you want to run on the device is allowed to run. This is achieved by signing the binaries with a key and enabling the secure boot feature (via OTP settings), which ensures only binaries signed with the correct signing key will be executed.

> RP2350's boot ROM uses a cryptographic signature to distinguish authentic binaries from inauthentic ones. A signature is a hash of the binary, signed with the user's private key. You can include signatures in binary images compiled for RP2350-based devices. Signatures use the SHA-256 hash algorithm and secp256k1 ECDSA elliptic curve cipher to authenticate binaries.
>
> — RP2350 Datasheet, Section 10.1.1 Secure boot

RP2350 can also disallow downgrading to older software versions through version numbering. This prevents bad actors from reinstalling older versions of signed software that may have security issues.

Encrypted boot

Encrypting the binaries running on a device ensures that any attempt to extract its code only recovers encrypted data, protecting the code from examination.

Encrypted boot stores decryption keys in OTP memory, using soft locks to protect them from being read at later boot stages (including secure code). RP2350 supports loading encrypted binaries from external flash into SRAM, where they can decrypt their own contents in-place. Loading the encrypted binary into SRAM and keeping it there avoids attacks based on physical access to the flash.

The Raspberry Pi Pico–series C/C++ SDK and `picotool` have features you can use to sign and encrypt binary payloads, append a decryption stage, and then re-sign the entire result.

Our AES implementation has been hardened against SCA and glitching attacks.

Note

The open-source decryption stage is not encrypted, but it is signed. Since it is not built into the boot ROM, this stage can be updated to protect the device against new intrusion techniques.

See section 10.1.2. Encrypted boot in the RP2350 Datasheet for more details.

Processor level security

The standard Arm security features present at the processor level are:

- Support for the Armv8-M Security Extension
- 8× security attribution unit (SAU) regions
- 8× Secure memory protection unit (MPU) regions
- 8× Non-secure memory protection unit (MPU) regions

See section 10.2 Processor security features (Arm) in the RP2350 Datasheet for more details.

Note

The RISC-V cores are not secured, so securing a RP2350 disables those cores to prevent them from being used as attack vectors.

Isolating trusted/untrusted software

The Cortex-M33 processors contain hardware that separates two execution contexts, known as Secure and Non-secure, and enforces a number of invariants between them, such as:

- Non-secure code cannot access Secure memory
- The Secure context cannot execute code in Non-secure memory
- Non-secure code cannot directly access peripherals managed by Secure code
- Non-secure code cannot prevent Secure interrupts from being serviced

This means that, at the hardware level, you can limit code that accesses the hardware to only those applications that need access.

RP2350 also applies this level of partitioning to things like direct memory access (DMA), so that non-secure code cannot use DMA to access secure hardware.

Access for each bus endpoint is controlled through different registers. These ACCESSCTRL registers can be read by any code (for enumeration purposes), but writing is strictly controlled and can be locked down further using the LOCK register.

Peripherals/memory that can be access-controlled:

- GPIO
- Boot ROM
- XIP
- SRAM
- SYSINFO
- XIP_AUX (DMA FIFOs)
- SHA-256
- POWMAN
- True random number generator
- Clock control registers
- XOSC
- ROSC
- SYSCFG

- PLLs
- Tick generators
- Watchdog
- XIP control registers
- QMI
- CoreSight trace DMA FIFO
- CoreSight self-hosted debug window

See section 10.6.2. Bus access control in the RP2350 Datasheet for more details.

Even if your application is secure, you should isolate it from using any hardware that it does not need to use. If your application does become compromised, any further compromise to the rest of the system will be greatly reduced if that application is unable to access other peripherals.

Note

At present, there is no API for helping configure access control in the Raspberry Pi Pico–series C/C++ SDK, so whilst this capability is present in the hardware, any access control will need to be done at the register level.

Other protection features

Glitch detector

A glitch attack is where an attacker deliberately manipulates the system clock or supply voltage in order to block or skip an instruction execution, perhaps in some critical part of an authentication routine.

RP2350 contains four glitch-detection circuits spread over the silicon, and these are triggered based on their register setup. When they do trigger, the system is reset.

Note

Glitch detectors cannot catch all possible glitch attacks. At slower clock speeds, they become less effective, making attacks during reduced-speed stages like boot ROM execution more likely to succeed.

Glitch detectors are disabled by default. They can be enabled in code, but it is recommended that they be enabled in OTP instead, as this safeguards the chip from intrusion before the processor has had a chance to enable the detectors.

See section 10.9 Glitch Detector in the RP2350 Datasheet for more details.

Redundancy coprocessor (RCP)

See section 3.6.3 Redundancy coprocessor (RCP) in the RP2350 Datasheet for more details.

The redundancy coprocessor (RCP) has a number of hardware features that can help secure the device. Using these features requires the insertion of the RCP's instructions into the software being run. For example, RP2350's built-in boot ROM makes extensive use of the RCP to ensure it is not compromised. The features that the RCP provides are fully detailed in the datasheet; in brief, these are:

- Stack canaries
  - Ensure that the exit from a function matches the entry
  - Ensure the stack was not overwritten during execution of the function
- Sequence counters
  - Ensure that code is executed in the correct order
  - Ensure that code has no missing steps
- Validation of boolean values
- Validation of 32-bit integers
- Halting of all processors when a panic condition is detected

Each RCP instruction can be executed with a pseudorandom delay (0–127 cycles). Although the intention is to make it more difficult to time injection attacks, having non-zero delays can, in practice, actually make this easier, so it is advisable that you do not use this delay functionality. See section Appendix E, RP2350-E3 in the RP2350 Datasheet for more details.

The RCP is implemented as a coprocessor attached to the Arm cores. To make it easier to use, the Raspberry Pi Pico–series C/C++ SDK contains a set of macro definitions to access the RCP instructions. If at any point one of these instructions detects an invalid state, the RCP will halt all cores — effectively, the device stops functioning.

It is down to the user to insert these RCP instructions into their code; they are not automatically added during compilation. They are used extensively in the RP2350 boot ROM.

Brownout detector

Whilst not strictly a security mechanism, system security may be enhanced when the brownout detector is enabled, as this can prevent erratic behaviour and potential data corruption during power fluctuations. The brownout detector is not as fine-grained as the glitch detector, but it can be used to detect power-supply tampering during system operation.

See section 7.6.2 Brownout detection (BOD) in the RP2350 Datasheet for more details.

OTP compromise: vulnerability to focused ion beam (FIB) imaging

A modern imaging technique called passive voltage contrast (PVC), which uses a focused ion beam device, has been known to expose data from OTP memory. This has been covered fully in the RP2350 Datasheet, but it is worth mentioning here, along with the appropriate mitigations.

> RP2350's OTP memory is designed to store boot key fingerprints and boot decryption keys. The ability to protect encrypted content in external flash storage depends on the ability to protect the OTP contents from unauthorised or external reads. OTP memory uses antifuse bit cells, which store data as a charge, similar to a flash bit cell. They do not make use of a physical structural change like traditional fuse cells do. This makes them resistant to many imaging techniques, such as optical and scanning electron microscopy. However, antifuse cells can be imaged through a novel technique called passive voltage contrast (PVC), which uses a focused ion beam (FIB) device.
>
> For more information on passive voltage contrast imaging, read this whitepaper by IOActive: https://www.ioactive.com/wp-content/uploads/2025/01/IOActive-RP2350HackingChallenge.pdf
>
> This process involves decapsulating the die. Therefore, physical access to the device is a strict requirement, and there is a moderate chance of destroying the die without being able to recover its OTP contents.
>
> — RP2350 Datasheet, Section 13.8 Imaging Vulnerability

Best practices to minimise susceptibility to OTP imaging include:

- Provisioning unique keys per device, rather than sharing secrets across a fleet of devices; if one device is compromised, not all are compromised
- Using chaff (dummy data added to the payload) to make imaging more difficult

Due to the way in which the OTP memory on RP2350 works, bits come in pairs, which are known as bit cells. In each 64-row OTP page, rows i and 32 + i share the same bit cells. The particulars of the PVC technique make it difficult to distinguish which of the two bits within a bit cell is set. However, if one bit in each pair is known to be zero — for example, a key stored at the bottom of an otherwise blank page — then the data can be trivially read from the PVC image. However, the presence of unknown data in both bits frustrates these attempts. This loophole can be exploited by storing data redundantly in the top and bottom half of each page. Specifically:

- Store actual data in each row i from 0 to 31
- Store the 24-bit bitwise complement of those values in each row 32 + i

The bitwise operations specified here are on the entire 24-bit raw row contents, including the ECC bit pattern. An alternative technique is to store a random value in row 32 + i and the XOR of that random value with the desired data value in row i. This is advantageous from a power side-channel perspective because it avoids reading the secret value directly from OTP; the example RP2350 encrypted bootloader uses a similar technique with a 4-way XOR. However, the bitwise complement technique described above is recommended for pairwise chaff. This is the same as the XOR technique, with a fixed XOR pattern of 0xffffff.

What does all this security mean?

There is a chain of trust that runs through any security system — whether that’s making sure that what you are running is correct by controlling access to peripherals and memory, or ensuring your software does not have security holes that can be exploited. RP2350’s various security features implement this chain of trust right up to the execution of end-user software; RP2350’s built-in security measures cannot help if the applications running on the chip are inherently insecure (for example, leaving open network ports, having bad password security, etc.).

The number of features you choose to incorporate into your software stack depends on your security requirements. For many customers, simply implementing secure boot provides sufficient security for their use case. If your device does not use access-controlled features, there is no need to apply access control to them; in fact, it may be prudent to disable them instead.

Implementing security

This section should be read alongside the Raspberry Pi Pico–series C/C++ SDK documentation, particularly 'Chapter 4: Signing and encrypting', from which this content is taken.

Creating a signed and packaged binary

To create a signed and packaged SRAM binary, add the following lines to your `CMakeLists.txt` file, before the `pico_add_extra_outputs(<TARGET>)` line:

Code

```
pico_set_binary_type(<TARGET> no_flash)
pico_package_uf2_output(<TARGET> 0x10000000)
pico_sign_binary(<TARGET> /path/to/private.pem)
```

`pico_set_binary_type` and `pico_package_uf2_output` are explained in section 4.1.1 of the Raspberry Pi Pico–series C/C++ SDK documentation. `pico_sign_binary` signs the binary using the private key.

Tip

An SRAM binary is used because you should never execute secure code from flash. This is because an attacker with physical access can swap out the flash after the signature check and run their own code. With a packaged SRAM binary, the `bootrom` copies the binary from flash to SRAM before performing the signature check, so switching out the flash has no effect.

To sign and package all the binaries in your project (for example, in a project with multiple binaries for each device in a system), add the following lines after `pico_sdk_init()` :

Code

```
set(PICO_DEFAULT_BINARY_TYPE no_flash)
set_property(GLOBAL PROPERTY PICOTOOL_UF2_PACKAGE_ADDR 0x10000000)
set_property(GLOBAL PROPERTY PICOTOOL_SIGFILE /path/to/private.pem)
set_property(GLOBAL PROPERTY PICOTOOL_SIGN_OUTPUT TRUE)
```

This sets the default properties globally for all targets in the CMake project, whereas the CMake functions only set the properties for the specified target. Setting a property for a specific target overrides the property set globally.

Generating keys and encrypting binaries

Before encrypting, you must generate an AES key and an initialisation vector (IV) salt. If you are using a Linux or macOS-based computer with a cryptographically secure random number generator, the AES key can be generated using `dd` :

Code

```
dd if=/dev/urandom of=privateaes.bin bs=1 count=32
```

On Windows, AES keys can be generated using Powershell 7:

Code

```
[byte[]] $(Get-SecureRandom -Maximum 256 -Count 32) | Set-Content privateaes.bin -AsByteStream
```

You will also need a per-device IV salt. This should be a 16-byte file of random data, which can be generated similarly:

Code

```
dd if=/dev/urandom of=ivsalt.bin bs=1 count=16
```

Warning

Keep the `privateaes.bin` and `ivsalt.bin` files for each device safe and secure. If you don't keep these files secure, you won't be able to encrypt any new data for that device without writing a new key and IV salt to the OTP memory.

To create the default encrypted binary — which fits into 512 kB of SRAM with the default decrypting bootloader running from scratch SRAM 8–9 — add the following lines to your `CMakeLists.txt` file, before the `pico_add_extra_outputs(<TARGET>)` line:

Code

```
pico_set_binary_type(<TARGET> no_flash)
pico_package_uf2_output(<TARGET> 0x10000000)
pico_sign_binary(<TARGET> /path/to/private.pem)
pico_encrypt_binary(<TARGET> /path/to/privateaes.bin /path/to/ivsalt.bin EMBED)
```

The first three lines are explained in section 4.1.2 of the Raspberry Pi Pico-series C/C++ SDK documentation. The `pico_encrypt_binary` command encrypts the binary using the private AES key and salts the IV with the per-device IV salt; `EMBED` specifies that the default decryption stage should be embedded in the encrypted binary to create a self-decrypting binary. To implement this for all the binaries in your project, add the following lines just after `pico_sdk_init()` :

Code

```
set(PICO_DEFAULT_BINARY_TYPE no_flash)
set_property(GLOBAL PROPERTY PICOTOOL_UF2_PACKAGE_ADDR 0x10000000)
set_property(GLOBAL PROPERTY PICOTOOL_SIGFILE /path/to/private.pem)
set_property(GLOBAL PROPERTY PICOTOOL_SIGN_OUTPUT TRUE)
set_property(GLOBAL PROPERTY PICOTOOL_AESFILE /path/to/privateaes.bin)
set_property(GLOBAL PROPERTY PICOTOOL_IVFILE /path/to/ivsalt.bin)
set_property(GLOBAL PROPERTY PICOTOOL_EMBED_DECRYPTION TRUE)
```

Similarly to signing, these lines will output OTP JSON files for each target in the build directory named .otp.json, which can be used to write the AES key and IV salt to your device as well as the secure boot setup.

Enabling secure boot

Warning

This procedure is irreversible. Before programming, ensure that you are using the right public key, correctly hashed. `picotool` supports the programming of keys from standard PEM files into OTP, performing the fingerprint hashing automatically. Programming the wrong key will make it impossible to run code on your device.

See section 10.5 Secure boot enable procedure in the RP2350 Datasheet for more details.

To enable secure boot:

1. Program at least one public key fingerprint into OTP, starting at BOOTKEY0_0
2. Mark programmed keys as valid by programming BOOT_FLAGS1.KEY_VALID
3. Optionally [1], mark unused keys as invalid by programming BOOT_FLAGS1.KEY_INVALID — this is recommended to prevent malicious actors from installing their own boot keys at a later date
   - KEY_INVALID takes precedence over KEY_VALID, which prevents more keys from being added later
   - Program KEY_INVALID with additional bits to revoke keys at a later time
4. Disable debugging by programming CRIT1.DEBUG_DISABLE, CRIT1.SECURE_DEBUG_DISABLE, or installing a debug key (See section 3.5.9.2 Debug Keys in the RP2350 Datasheet for more details.)
5. Optionally, enable the glitch detector by programming CRIT1.GLITCH_DETECTOR_ENABLE and setting the desired sensitivity in CRIT1.GLITCH_DETECTOR_SENS. See section 10.9 Glitch detector in the RP2350 Datasheet for more details.
6. Disable unused boot options such as USB and UART boot in BOOT_FLAGS0
7. Enable secure boot by programming CRIT1.SECURE_BOOT_ENABLE

Note

[1] If your application does not have its own update procedure, but you anticipate needing to update the software in the future, you may wish to leave one of the unused keys active.

General security advice

Per-device keys

It is advisable to provision unique encryption keys for each device, rather than sharing secrets across a fleet. This will ensure that if one device is compromised, the rest remain secure.

Lock down debug and console interfaces

RP2350’s debug interface is enabled by default. When moving a device from development to release, do not forget to disable any debug and console interfaces.

Updates

You should always try to maintain an update capability within your software. If a compromise is found, it is important to be able to upload new software with fixes. In fact, in many parts of the world, it is becoming a legal requirement to allow software to upgrade in the field (see the Cybersecurity Resilience Act, the RED-DA directives, etc.). These updates can be delivered over the air (OTA) or in person.

If you do implement updates, you should take care to ensure that the anti-rollback facility is enabled. This feature, which is part of the boot ROM firmware on the device, uses version checking to ensure that older, possibly-less-secure versions of software cannot be reinstalled — even if they are signed correctly.

Version numbering

Make sure your version-numbering scheme is straightforward to use. Threat assessment is much simpler when your software version is easy to identify.

Conclusion

The RP2350 microcontroller offers a compelling combination of high performance and enhanced security features, making it a strong contender for a wide range of applications — especially those where data integrity and system security are paramount. While its open-source nature does foster innovation and community support, it is crucial that developers leverage the built-in security mechanisms effectively. This includes employing secure boot, hardware acceleration for cryptography, and secure storage to protect sensitive data and intellectual property.

RP2350's architecture, with its separate processing cores and robust memory protection, provides a solid foundation for building secure IoT devices and embedded systems. Ultimately, the security of any device powered by RP2350 hinges on its developers doing their due diligence to implement these features, ensuring the final product can withstand evolving cyber threats.

This whitepaper has briefly outlined the security features available on RP2350 as well as the procedures needed to enable them. It is strongly recommended that the reader refers to the RP2350 Datasheet and the Raspberry Pi Pico–series C/C++ SDK for further details.

Contact Details for more information

Please contact applications@raspberrypi.com if you have any queries about this whitepaper.

Web: www.raspberrypi.com

Raspberry Pi