Security Target for Secure Thingz Secure Boot Manager

Secure Thingz Ltd

Version	Change date	Author	Notes
1	2019-09-12	Amyas Phillips	Complete first draft
2	2019-09-13	Amyas Phillips	Internal STz review
3	2019-09-27	Amyas Phillips	Address feedback from evaluator
4	2019-11-15	Amyas Phillips	Address feedback from evaluator
5	2019-12-03	Amyas Phillips	Revised vulnerability analysis
6	2019-12-11	Amyas Phillips	Minor corrections
7	2019-12-17	Amyas Phillips	Minor corrections

1 Introduction

The Security Target describes the Platform (in this chapter) and the exact security properties of the Platform that are evaluated under [SESIP] (in chapter “Security requirements and implementation”) and that a potential consumer can rely upon the product upholding if they fulfill the objectives for the environment (in chapter “Security Objectives for the operational environment”).

SESIP v1.3 is used in this document.

1.1 ST reference

See title page.

1.2 Platform reference

TOE name	Secure Thingz Secure Boot Manager
TOE version	1.30
TOE identification	1.30
TOE Type	Secure bootloader for microcontrollers used in IoT applications

1.3 Included guidance documents

The following documents are included with the platform:

Reference	**Name**	**Version**
[Manual]	Embedded Trust User Guide	2

1.4 Platform functional overview and description

The Target of Evaluation (TOE) consists of a bootloader for microcontrollers (MCUs) called the Secure Boot Manager (SBM).

The TOE is intended to be used by an embedded C developer who integrates it into an Internet of Things (IoT) Product, along with an IoT Application and other parts of an IoT Platform.

In SESIP terms, the TOE is a component of the IoT Platform. The TOE scope is depicted in Figure **1**. The blue block is within the evaluated scope and the grey blocks are outside of the evaluated scope. Out of scope are the IoT Application, the other components of the IoT Platform including the RTOS and the microcontroller, the integrated development environment (IDE) used to configure and build the TOE, and the programming system used to provision the TOE onto the microcontroller.

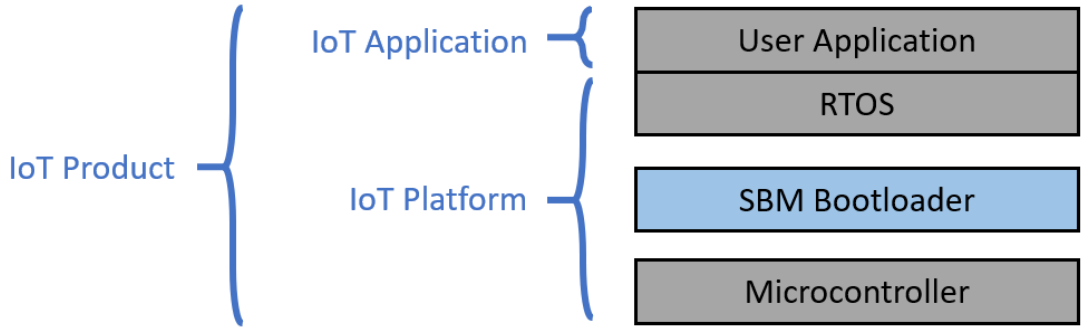

Figure 1 **TOE scope (in blue) in terms of SESIP**

The TOE is a specialised application that is run when the microcontroller starts, before the user application, to which it provides several security benefits. The main security features of the TOE are as follows:

- **Configuration of hardware security features**. The TOE ensures that hardware security features of each supported microcontroller are used effectively, for example to write-protect the TOE, set debug lock features, store unique device keys, and set lifecycle states.
- **Provisionable key store**. The TOE accesses a data structure in Flash memory called the Provisioned Data Block (PDB). This data structure includes device private keys, and trust anchor certificates, as specified by the composite developer. The PDB is created, personalised and programmed onto each device by a compatible provisioning system in the factory, providing each device with a unique cryptographic identity. Where suitable hardware features are available the TOE sets access permissions so that data in the PDB cannot be modified by the user application, before passing control to that application. Where hardware features for secure key generation and storage such as a PUF or silicon root of trust (ROT) are provided, device secret keys are generated and stored there instead of in the PDB.
- **Anti-cloning.** Multiple features protect firmware from being cloned onto other devices. Excepting during development, the TOE permanently activates debug lock features of the host microcontroller to prevent external readout or modification of the internal firmware. The device hardware ID is used to seed a hash of the provisioned data that is included in the TOE at provisioning time, allowing the TOE to verify that it is not running on a cloned device. The device hardware ID can also be incorporated into device identity certificates included in the provisioned data, allowing the user application to verify that it is not running on a cloned device.
- **Secure boot**. The TOE receives the program pointer immediately on device reset. The TOE checks that a signature over the main application image is present and validates it against a trust anchor stored in the PDB, proving the authenticity of the main application image. Finally, the TOE passes the program pointer to the user application. Using microcontroller-specific features, the TOE write-protects itself to

ensure that a compromised user application cannot persist itself by also compromising the trusted bootloader.

- **Firmware update**. The TOE includes an installer, which determines whether a more recent valid user application image is available, and if so, installs that before executing it. New images must be linked for a predetermined location in memory known as the active slot, packaged with version metadata, signed by a party trusted by the TOE, distributed to target devices, and stored in a predetermined location in memory known as the update slot by another process, usually the user application. The active slot is always located in internal Flash memory in execute-in-place (XIP) devices, aligned on erase sectors. The update slot can be located in internal Flash memory, also aligned on erase sectors, or it can be in external serial Flash. By having the installer copy new images into a fixed location before executing them, reliance on position-independent code (PIC), runtime linking, multiple versions of images linked for different locations, or knowledge of slot state outside of the IoT device is avoided. Rollbacks can be accomplished only by repackaging an older image with a newer version number and distributing that for installation.

The TOE is located at the reset vector of the microcontroller (see Figure **2**) so that on start it can perform a sequence of operations before passing control to the user application.

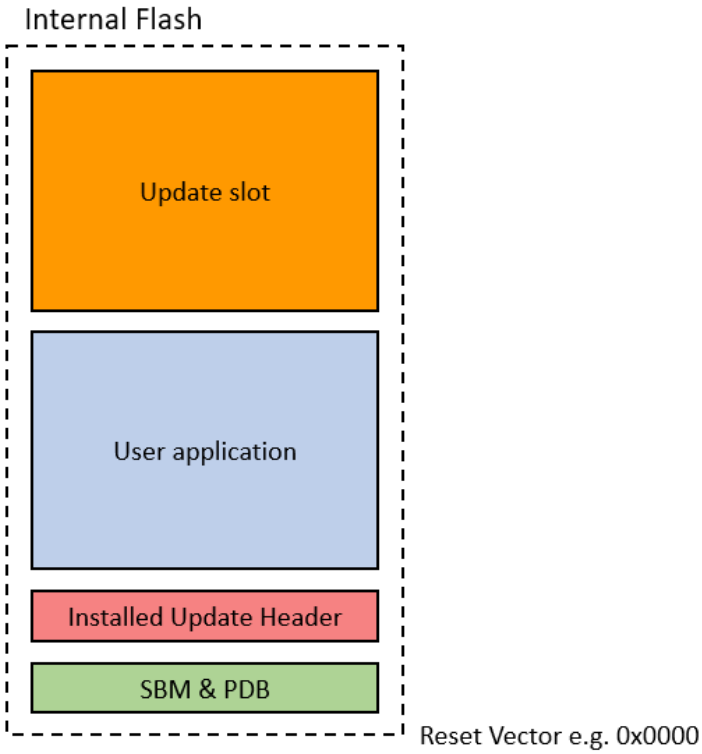

Figure 2 **Example flash memory layout of MCU devices using the TOE (the SBM bootloader).**

While the TOE must be located at the reset vector, the location of other slots is configurable, subject to hardware constraints and the SBM features enabled by the developer:

- The update slot may be located in external Flash.

- The user application is always located in internal Flash in the active slot and must be linked for that location.
- The Installed Update Header slot is used by the TOE to store metadata received with the user application image, including version information and a signature.
- All slots in internal Flash are aligned on Flash erase sectors so that they and/or their neighboring slots can be independently updated.

Knowledge of the memory map is built into the TOE. The TOE makes use of a default memory map and other hardware-specific features for each supported microcontroller. Supported microcontrollers are listed in Table 1.

Vendor	MCU family	MCU variant (excluding package variants)
Microchip	SAML11	SAML11E16A
NXP	K65	MK65FN2M0VMI18
NXP	K66	MK66FN2M0VMD18
Renesas	RX65N	R5F565NEHDFC
ST	STM32F412	STM32F412ZGT6
ST	STM32F777	STM32F777ZIT6
ST	STM32L4S5	STM32L4S5ZIT6
ST	STM32F405	STM32F405VGT6
ST	STM32F407	STM32F407VG
ST	STM32H7	STM32H753XI
ST	STM32H7	STM32H753ZI
ST	STM32L4	STM32L475VG

Table 1 Supported microcontrollers

The TOE is provided to embedded developers as C source code in an IAR Embedded Workbench (a popular IDE) project that is generated by an IDE plugin called Embedded Trust. The project includes all source code plus configuration files for the selected microcontroller. This allows the TOE source code to be inspected as part of any security audit.

IAR Embedded Workbench and the Embedded Trust plugin are distributed in Windows installers downloadable from https://iar.com. Once installed, the Embedded Trust release notes and user guide and example projects are accessible directly via the Help menu. Example projects are accessible via Help / Information Centre / Example Projects / Embedded Trust / Getting Started. A whitepaper “Getting started with IoT security” is accessible via Help / Information Centre / Integrated solutions / Additional downloads and support files.

When the IDE builds the TOE project, the Embedded Trust plugin also creates a set of instructions for a factory provisioning system provided as another component of the Embedded Trust product, describing how the PDB is to be constructed and programmed onto each device along with the built TOE image. The PDB in each device contains a unique secret identity key, identity certificate chain, and firmware trust anchors. Developers can configure the identity certificate chain and other details of the provisioning process in the graphical user interface (GUI) when creating a new bootloader project.

The workflow for preparing and provisioning the TOE onto microcontrollers in IoT devices is summarized in Figure 3.

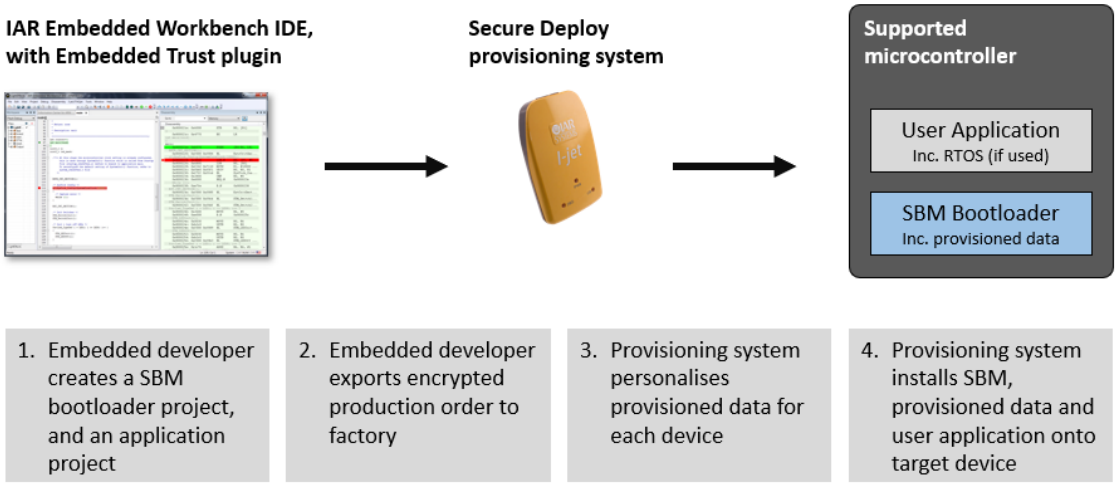

Figure 3: Overview of how the TOE is prepared, and provisioned onto microcontrollers

Several optional developer-specific or board-specific functions in the TOE are provided in the source code, where they are called at specific points in the start-up sequence. These functions are called “OEM API functions” and must be implemented by the

developer. Developers are instructed not to modify bootloader code or behavior outside of these functions because doing so will lead to unsupported behavior.

The OEM API functions provide log outputs that can be routed to a serial line or a log file, a handler for boot failures, and a tamper-detection handler. By default, they are disabled. They are enabled by setting conditional compilation flags in the TOE source code – flags which are in turn set by options selected in the IDE by the developer when creating the bootloader project. These selections also set a number of other conditional compilation flags that can be used to disable certain features of the TOE for convenience during development.

Developers create a separate project for the user application, where they have a free hand. The TOE exposes a set of APIs called the “Application APIs” to the user application as wrapper functions in a library provided for inclusion in that application. The wrapper functions map arguments into and results out of buffers whose addresses and sizes are passed into a function in the TOE called the SBM Access Method, a pointer to which is located at a fixed memory location in the TOE, set using #defines in both the TOE (the bootloader) and the user application C code. These Application APIs expose functions for:

- Signing, validation and key derivation
- Checking the contents of the update slot and initiating an update
- Checking the metadata of the installed application
- Checking the status of PDB slots
- Retrieving certificates from the PDB
- Checking the version of the TOE

When the Embedded Workbench IDE builds the user application the Embedded Trust plugin packages the resulting binary image in a structure called a software update (SWUP) file. The

SWUP includes metadata about the package's image including its version number, a signature created using the developer's secret firmware signing key, and information allowing the target devices to decrypt the firmware image.

The TOE installer will install only valid SWUPs found in the update slot. SWUPs may be delivered into the update slot over the air to deployed IoT devices via a running previous version of the user application or programmed directly into the update slot on the production line in the factory at the same time as the TOE is provisioned.

2 Security Objectives for the operational environment

In order for the platform to fulfill its security requirements, the operational environment (technical or procedural) must fulfil the following objectives.

- Only authorized and trustworthy personnel should have access to the TOE development environment (as described in [Manual] section “Mastering”).

3 Security requirements and implementation

3.1 Security Assurance Requirements

The claimed assurance requirements package is: **SESIP1** as defined in [SESIP].

3.1.1 Flaw Reporting Procedure (ALC_FLR.2)

In accordance with the requirement for a flaw reporting procedure (ALC_FLR.2), including a process to give generate any needed update and distribute it, the developer has defined the following procedure:

Flaws can be reported by email at securityalert@securethingz.com or by web form at https://www.securethingz.com/contact/ . These channels are monitored by the SecureThingz customer service team. Bug reports including security flaws are passed to the Embedded Trust development team. A fix is developed and merged into the current release candidate version of the TOE, where it forms part of the next release of the Embedded Trust product, of which the TOE is a part. Embedded Trust releases are typically several months apart. For severe issues an updated version of the current release is made available as soon as the fix is available.

Customers are notified by email when new releases of Embedded Trust are made. Known issues and new fixes are described in the release notes of each release. The reporter of the issue is updated on progress via IAR customer support.

When new releases of the TOE are made customers can update their factory production lines to provision the newer release onto newly manufactured IoT devices. They cannot update the TOE on devices that have already been provisioned with a previous version, for two reasons:

1. As a bootloader, a function of the TOE is to verify and install updated user application images. It cannot update itself, because in a XIP microcontroller that would require replacing the code it is currently executing – something that cannot be accomplished reliably.
2. As a keystore, a function of the TOE is to provide the host device with an immutable identity, which by definition must not change. On microcontrollers where the root identity secrets are stored as part of the TOE, the TOE cannot be updated without issuing a new identity to the device.

To minimize the risk of serious security flaws in the TOE leading to compromise of the IoT device, its codebase is kept as small and stable as possible.

3.1.2 Vulnerability Survey (AVA_VAN.1)

AVA_VAN.1 requires that the TOE demonstrate an attack potential rating of at least 16. We do so in two steps. First, we review known vulnerabilities in the TOE and its components. Second, we systematically review potential attacks on the TOE. At each step, we show that no attacks with attack potential less than 16 can be identified.

3.1.2.1 Survey of known vulnerabilities

Table 2 lists software components used in the TOE, including third-party libraries. Details of each are provided, including how vulnerability notices are monitored.

Name of software component	Version	Supports specific microcontrollers ?	Maintainer	Where is it obtained?	How are announcements and new releases monitored?
Secure Boot Manager	1.30	See Table 1	Secure Thingz	https://www.iar.com/embedded_trust/	We invite vulnerability reports directly, per section 3.1.1 Flaw Reporting Procedure (ALC_FLR.2), and monitor public channels including https://cve.mitre.org/cve/
microecc	1.0 (commit 3345d50)	n/a	Ken MacKay	https://github.com/kmackay/micro-ecc	We are notified of new releases by GitHub and monitor them for vulnerability announcements
SHA256	RFC4634	n/a	IETF	https://datatracker.ietf.org/doc/rfc4634/	We monitor IETF - Announce mailing list for vulnerability announcements about draft-eastlake-sha2 or draft-eastlake-sha2b
libtomcrypt	1.17	n/a	libtom team	https://github.com/libtom/libtomcrypt	We are notified of new releases by GitHub and monitor them for vulnerability announcements
jansson	2.12	n/a	Petri Lehtinen	https://github.com/akheron/jansson	We monitor https://groups.google.com/forum/#!forum/jansson-users and new releases on GitHub for vulnerability announcements.
Atmel SAML11 Series Device Support	1.0.109	SAML11E16A	Microchip	http://packs.download.atmel.com/	We subscribe to Microchip PCNs for SAML11 at https://www.microchip.com/pcn
MCUXpresso SDK for Kinetis Freedom K66F	2.50	MK65FN2M0VMI18, MK66FN2M0VMD18	NXP	https://mcuxpresso.nxp.com/en/welcome	We subscribe to product announcements from NXP and monitor NXP user forums at https://community.nxp.com/community/mcuxpresso/mcuxpresso-sdk for vulnerability announcements

Name of software component	Version	Supports specific microcontrollers ?	Maintainer	Where is it obtained?	How are announcements and new releases monitored?
STM32CubeF7	1.15.0	STM32F777ZIT6	ST Micro	https://www.st.com/en/embedded-software/stm32cubef7.html	We subscribe to product announcements from ST and monitor ST user forums at https://community.st.com/ for vulnerability announcements.
STM32CubeF4	1.24	STM32F405VGT6, STM32F407VG, STM32F412ZGT6	ST Micro	https://www.st.com/en/embedded-software/stm32cubef4.html	We subscribe to product announcements from ST and monitor ST user forums at https://community.st.com/ for vulnerability announcements.
STM32CubeL4	1.14.0	STM32L4S5ZIT6	ST Micro	https://www.st.com/en/embedded-software/stm32cubel4.html	We subscribe to product announcements from ST and monitor ST user forums at https://community.st.com/ for vulnerability announcements.
STM32CubeH7	1.50	STM32H753XI, STM32H753ZI	ST Micro	https://www.st.com/en/embedded-software/stm32cubeh7.html	We subscribe to product announcements from ST and monitor ST user forums at https://community.st.com/ for vulnerability announcements.
Trusted Secure IP Driver	1.05	R5F565NEHDFC	Renesas	https://www.renesas.com/eu/en/products/software-tools/software-os-middleware-driver/security-crypto/trusted-secure-ip-driver.html	Renesas account managers contact us with any vulnerability announcements.
RX Family RX Driver Package	1.13	R5F565NEHDFC	Renesas	https://www.renesas.com/eu/en/products/software-tools/software-os-middleware-driver/software-package/rx-driver-package.html	We subscribe to product announcements through Renesas MyPages and https://www.renesas.com/us/en/support/pcnsearch.html and monitor them for security vulnerabilities.

Table 2 First- and third-party software used in the TOE, and associated channels which are monitored for security vulnerability announcements.

A search of the public Common Vulnerabilities and Exposures[1] (CVE) database for known vulnerabilities in these components shows that no applicable vulnerabilities have been reported.

Two CVEs do exist against libtomcrypt 1.17 but are not applicable to the TOE:

i) CVE-2019-17362 reports a vulnerability in the der_decode_utf8_string function. This function is not present in the TOE.

ii) CVE-2018-12437 reports that under certain conditions a vulnerability exists in the ECDSA signature function. The applicable conditions are execution of libtomcrypt at the same time as attacker code on a microprocessor with memory caches. The TOE is deployed only in MCU-based systems that do not feature memory caches.

3.1.2.2 Survey of potential vulnerabilities

This survey uses the method of attack trees[2] to develop a tree of possible attack chains on the TOE, using information about its design and implementation. Well-known methods of attacking secure embedded systems have been considered at each node in the attack tree. Where appropriate they have been identified as potential attacks. The document [AM] provides an enumeration of well-known methods of attack.

Note that the survey considers only attacks on the TOE. Attacks on a composite system may be able to proceed to the attackers' goals by other means.

Attack trees are constructed by working backwards from the attackers' goals:

Goal A: **Extract software IP**

A1. Modify the TOE so that it does not disable debug interfaces on first boot
- A.1.1. Alter TOE in supply chain or on-chip before first boot

A2. Extract firmware decryption group private key
- A.2.1. Probe or image internal Flash memory to obtain firmware decryption key
- A.2.2. Use simple power analysis (SPA) / differential power analysis (DPA) during firmware decryption operations
- A.2.3. Use higher-order DPA during firmware decryption operations
- A.2.4. Use electromagnetic emissions analysis (EMA) during firmware decryption operations
- A.2.5. Use fault analysis (FA) during firmware decryption operations

A3. Exploit bugs or undocumented features in the TOE to externally influence it during boot to expose firmware decryption group private key or proprietary firmware

A4. Exploit factory test modes to access provisioned data

Goal B: **Take over a device (execute attacker's software)**

B1. Have TOE boot malware placed in the active slot
- B.1.1. [AND] Re-enable debug access
  - B.1.1.1. Inject faults to re-enable debug access

[1] https://cve.mitre.org/cve/search_cve_list.html
[2] https://www.schneier.com/academic/archives/1999/12/attack_trees.html

B.1.1.2. Exploit bugs or undocumented features in the TOE to externally influence it during boot to re-enable debug access
B.1.2. [AND] Disable TOE validation of the active application during boot
B.1.2.1. Inject faults to disable validation of the active application during boot
B.1.2.2. Modify the running user application to write changes to the installed TOE
B.1.2.2.1. Provoke a buffer overflow in the user application to execute specially crafted code to write changes to the TOE
B.1.2.2.2. Compromise unvalidated software loaded outside the validated boot chain to write changes to the TOE
B.1.2.3. Modify the running TOE to write changes to its own code
B.1.2.3.1. Send a malformed SWUP to cause undocumented behaviour in the TOE such that the attacker can execute code to write changes to the TOE
B.1.2.4. Probe the TOE in internal Flash to write changes to its code
B.1.2.5. See [Alter TOE in supply chain or on-chip before first boot]
B.1.2.6. Exploit bugs or undocumented features in the TOE to circumvent validation of the user application at boot
B2. Have TOE install invalid SWUP containing malware
B.2.1. [AND] Inject faults during authentication of invalid SWUP
B.2.2. [AND] Deliver invalid SWUP to target device
B3. Have TOE install stale but validly signed SWUPs containing old firmware with known vulnerabilities

Goal C: **Masquerade as an authentic device**
C1. Exfiltrate device's private identity key using attacker's software
C.1.1. See [Take over the device (execute attacker's software)]
C.1.2. Provoke a buffer overflow in the user application to execute specially crafted code to exfiltrate the device identity key
C2. Use simple SPA/DPA during elliptic curve digital signal algorithm (ECDSA) signature operations to obtain device identity key
C3. Use higher-order DPA during ECDSA signature operations to obtain device identity key
C4. Use EMA during ECDSA signature operations to obtain device identity key
C5. Modify random number generator (RNG) behaviour during ECDSA operations to obtain device identity key
C6. Probe or image the device private key in internal Flash memory
C7. Exploit bugs or undocumented features in the TOE to cause it to expose device's private key externally during boot
C8. See [Exploit factory test modes to access provisioned data]
C9. Use FA during ECDSA signature operations to obtain device identity key

Goal D: **Manufacture counterfeit devices**
D1. Install firmware image on unauthorised devices
D.1.1. [AND] See [Extract software IP]
D.1.2. [AND] Run software on unauthorized devices

Goal E: **Remotely disable a deployed device**
E1. Send a malformed SWUP such that the TOE installs a non-functional user application
E.1.1. See [Take over the device (execute attacker's software)]

E2. Modify the TOE to always invalidate the active and update slots

E.2.1. See [Modify the running user application to write changes to the installed TOE] and [Modify the running TOE to write changes to its own code]

Leaf nodes in the attack tree describe attacks on the TOE. Next, we discuss the feasibility of these attacks.

Attack	Feasibility
A.1.1. Alter TOE in supply chain or on-chip before first boot	Supply-chain security is out of scope of the TOE. Users are advised to ensure they have an authentic copy of IAR Embedded Workbench with Embedded Trust plugin, and to employ a secure provisioning system to deliver their project from development onto devices. Users are advised to ensure the security of provisioned information on-chip before first boot by operating a secure production environment, by utilizing a secure firmware programming interface (where available), and/or by disabling the debug interface using the programmer immediately after initial programming.
A.2.1. Probe or image internal Flash memory to obtain firmware decryption key **B.1.2.4. Probe the TOE in internal Flash to write changes to its code** **C6. Probe or image the device private key in internal Flash memory**	Resistance to probing attacks is in a feature of the microcontroller platform and out of scope of the TOE. On devices, boards or microcontrollers fitted with latching tamper detection circuits, the tamper detection state line can be checked at each boot by the OEM API function *oem_return_to_provisioned_state_now*, which will cause the TOE to erase the active and update slots if evaluated to true. The composite designer must implement or specify a tamper detection mesh to use this feature.
A.2.2. Use simple SPA/DPA during firmware decryption operations **A.2.3. Use higher-order DPA during firmware decryption operations**	An attacker with the ability to have the TOE decrypt SWUPs while monitoring power or emissions side channels may attempt to extract the firmware decryption group private key, hence the ephemeral symmetric firmware encryption key, and thus the software IP. The TOE employs a crypto library (micro-ecc) that implements anti-SPA/DPA measures to frustrate this. These measures are detailed in [TJERAND]. Further, the TOE severely restricts the number of operations an attacker can trace by checking that new

<table>
<tr><td></td><td>SWUPs are validly signed and declare a higher software version number, before decrypting them.</td></tr>
<tr><td>A.2.4. Use EMA during firmware decryption operations

C4. Use EMA during ECDSA signature operations to obtain device identity key</td><td>Electromagnetic emissions may leak information, at whole-chip or die level. The TOE is not designed with a design goal of preventing information leakage through near-field or far-field electromagnetic emissions, and other than by minimizing the use of private keys provides no protection against attacks utilizing such side-channels.

The TOE effectively limits the number of usages of the firmware decryption group private key by checking that new SWUPs are validly signed and declare a higher software version number, before decrypting them.

The device identity key is used to encrypt a random hash challenge during each transport layer security (TLS) protocol handshake. An attacker with physical access can attempt an EMA side-channel analysis on it by stimulating multiple TLS connections. The TOE does not limit the rate of TLS connections, but the composite designer may be able to do so at system design level.

To frustrate EMA attacks more completely users must rely on other parts of the platform, in particular the MCU hardware, and on device-level design features such as cans, potting and tamper meshes.

EMA attacks, especially at die level, require expertise and equipment that at attack rating 22 puts them beyond the reach of a basic attack potential:

<table>
<tr><th></th><th colspan="2">Identification phase</th><th colspan="2">Exploitation phase</th></tr>
<tr><td>Elapsed time</td><td><1 week</td><td>2</td><td><1 day</td><td>3</td></tr>
<tr><td>Expertise</td><td>Expert</td><td>5</td><td>Expert</td><td>4</td></tr>
<tr><td>Knowledge of the TOE</td><td>Public</td><td>0</td><td>Public</td><td>0</td></tr>
<tr><td>Access to the TOE</td><td><10 samples</td><td>0</td><td><10 samples</td><td>0</td></tr>
<tr><td>Equipment</td><td>Specialised</td><td>3</td><td>Specialised</td><td>4</td></tr>
<tr><td>Open samples</td><td>Public</td><td>0</td><td>Public</td><td>0</td></tr>
<tr><td>PHASE TOTALS</td><td></td><td>10</td><td></td><td>11</td></tr>
<tr><td>ATTACK RATING</td><td colspan="4">21</td></tr>
</table>
</td></tr>
<tr><td>A.2.5. Use FA during firmware decryption operations

C9. Use FA during ECDSA signature operations to obtain device identity key</td><td>A specialization of fault injection attacks, FA attacks seek to inject bit errors into cryptographic operations in tamper-proof microcontrollers, in such a way as to leak data about secret keys.

This kind of attack requires expertise and equipment beyond the reach of a basic attack potential:</td></tr>
</table>

	Identification phase		**Exploitation phase**	
Elapsed time	<1 week	2	<1 day	3
Expertise	Expert	5	Expert	4
Knowledge of the TOE	Public	0	Public	0
Access to the TOE	<10 samples	0	<10 samples	0
Equipment	Specialised	3	Specialised	4
Open samples	Public	0	Public	0
PHASE TOTALS		10		11
ATTACK RATING	**21**			

A3. Exploit bugs or undocumented features in the TOE to externally influence it during boot to expose firmware decryption group private key or proprietary firmware	The only external input the TOE should accept during boot is a SWUP stored in external memory. That SWUP is authenticated before use. Correct and robust behavior in processing that SWUP is ensured using good development practices.
A4. Exploit factory test modes to access provisioned data	No factory test modes are implemented in the TOE. Factory test modes implemented in the user application have the same access to secret material as the user application, potentially approaching equivalence to full debug access. Implementors of such modes must take care to permanently disable them before deployment, or else ensure they do not expose secret material.
B.1.1.1. Inject faults to re-enable debug access **B.1.1.2. Exploit bugs or undocumented features in the TOE to externally influence it during boot to re-enable debug access**	The TOE irreversibly disables debug and programming interfaces, preventing attackers with physical access to a target MCU's debug ports from obtaining provisioned data, user data, or software IP. Irreversible debug lockdown is a required feature on all microcontrollers supported by the TOE. Once set, no software attack can reverse it. The resistance of the MCU's debug lock feature to fault injection attacks is out of scope of the TOE.
B.1.2.1. Inject faults to disable validation of the active application during boot **B.2.1. [AND] Inject faults during authentication of invalid SWUP**	An attacker may inject faults during TOE execution by means of clock or voltage glitches. By causing execution to skip instructions, this technique can be used to bypass validations to, for example, install untrusted code, or to change lifecycle state. It can also be used to weaken cryptographic functions, potentially allowing the extraction of private keys. This kind of attack requires expertise and equipment that at attack rating 22 puts it beyond the reach of a basic attack potential:

		Identification phase		Exploitation phase	
	Elapsed time	<1 week	2	<1 day	3
	Expertise	Expert	5	Expert	4
	Knowledge of the TOE	Public	0	Public	0
	Access to the TOE	<10 samples	0	<10 samples	0
	Equipment	Specialised	3	Specialised	4
	Open samples	Public	0	Public	0
	PHASE TOTALS		10		11
	ATTACK RATING	**21**			
B.1.2.2.2. Provoke a buffer overflow in the user application to execute specially crafted code to write changes to the TOE **C.1.2. Provoke a buffer overflow in the user application to execute specially crafted code to exfiltrate the device identity key**	A corrupted user application may try to enable malware to become permanently resident on the device by disabling boot-time validation of the user application by the TOE. To prevent modification by a corrupted user application, the TOE must be protected from internal write operations. Use can be made of an immutable first-stage bootloader in ROM if available. This must be capable of calculating a hash over the TOE and verifying it is the same as an immutably stored hash, or the same as that obtained by decrypting a signature over the TOE using an immutably-stored public key. Use can also be made of write-protection features provided by the MCU. The TOE automatically write-protects itself on first boot on all supported ST microcontrollers, with the sole exception of the SAM-L11 where it is expected that the composite developer will isolate user applications using TrustZone-M trusted execution environment (TEE) technology. This is an example of a third option to protect device secrets from corrupt user applications: using application isolation mechanisms. MCUs are available with a range of isolation mechanisms, from none, through OSes exploiting privileged modes and memory protection units, to integrated secure elements and TEE partitioning with trusted supervisory firmware. It is up to the composite developer to configure their chosen isolation mechanism to prevent writes to the TOE Flash and direct reads of provisioned data by user applications. If none of these options is available, composite developers rely on attackers being unable to find ways to compromise running software on target devices. Executing buffer overflow attacks on Harvard architecture execute-from-flash MCUs is a difficult task that as well as a suitable buffer overflow vulnerability				

	requires knowledge of at least the firmware binary image and ideally the source code, where a Flash write function must be available. Confidence in this approach can be increased using firewalls and robust API testing.
B.1.2.2.1. Compromise unvalidated software loaded outside the validated boot chain to write changes to the TOE	Although the TOE validates the next application in the boot sequence, nothing in the TOE prevents that application from loading further applications, potentially including untrusted applications. Composite developers must take care to write-protect the TOE, employ an immutable ROM bootloader, or apply appropriate application isolation techniques in such situations.
B.1.2.3.1. Send a malformed SWUP to cause undocumented behaviour in the TOE such that the attacker can execute code to write changes to the TOE	An attacker may try to compromise the TOE directly by sending a malformed SWUP. The SBM validates each SWUP's signature before doing any further processing, so an attacker would have to compromise the composite developers' firmware repository, firmware signing key or firmware signing procedures to attempt this. This is outside the scope of the TOE.
B.1.2.6. Exploit bugs or undocumented features in the TOE to circumvent validation of the user application at boot	The TOE should always validate the user application at boot. Its correct and robust behavior is ensured using good development practices.
B.2.2. [AND] Deliver invalid SWUP to target device	The TOE can process new SWUPs placed in internal or external Flash memory. SWUPs are expected to be delivered via external processes. Because the TOE validates the integrity, freshness and authenticity of each SWUP those processes are not required to be secure. For instance, new SWUPs could be placed on a removable SD card. Nevertheless, use of secure channels to deliver SWUPs does no harm and further reduces opportunities for attack.
B3. Have TOE install stale but validly signed SWUPs containing old firmware with known vulnerabilities	Downgrade attacks where a validly signed but old SWUP with known vulnerabilities is delivered to target devices to enable further attacks are prevented by having the TOE check that new SWUPs declare higher application version number than the currently-installed SWUP before installing them.
C2. Use simple SPA/DPA during ECDSA signature operations to obtain device identity key **C3. Use higher-order DPA during ECDSA signature**	A device's identity key is used to encrypt a random hash challenge during each TLS handshake, so an attacker with physical access can attempt a side-channel analysis on it by stimulating one or more TLS connections. The TOE employs a crypto library (micro-ecc) that

operations to obtain device identity key	implements anti-SPA/DPA measures to frustrate this. These measures are detailed in [TJERAND].
C5. Modify RNG behaviour during ECDSA operations to obtain device identity key	High quality random numbers are essential to many cryptographic operations. The only such operation implemented in the TOE is ECDSA signature. An attacker in possession of an ECDSA signature and the random number used in its creation can recover the device's secret identity key. Thus, the randomness of the RNG is critical. As part of the physical MCU platform the RNG is outside the scope of the TOE, but all MCUs supported by the TOE are equipped with high-quality RNGs, whose behaviour is not trivial to alter while retaining other MCU functionality.
C7. Exploit bugs or undocumented features in the TOE to cause it to expose device's private key externally during boot	The TOE contains no functions that expose devices' private keys, internally or externally. Its correct and robust behavior is ensured using good development practices.
D.1.2. [AND] Run software on unauthorized devices	The TOE performs a check at boot time, that the provisioned data includes a hash seeded with the device hardware ID. A firmware image that has been extracted from an authentic device, for example by sniffing it from a serial programming line in the factory, will fail this check on any other device and the TOE will terminate. A counterfeiter would have to reverse-engineer and modify the firmware image to disable this check. If the counterfeit devices are to connect to a genuine web service, a further check can be implemented at the genuine web service that the device certificates are valid. This is however outside the scope of the TOE.

The following security practices are employed in development of the TOE:

1. Traceability. Code commits are linked back to requirements management. This helps ensure only documented features enter the codebase.
2. Code review. All code commits undergo review for quality and function before merging. This helps prevent bugs or undocumented features from entering the codebase.
3. Unit testing. All functions are subject to unit tests that run automatically on each merge request. Unit tests are themselves subject to code review, including by dedicated testers, for thoroughness. Unit tests include valid and invalid inputs. This helps ensure robust and stable behaviour.

4. API testing. All application and OEM APIs are subject to API tests including malformed requests that run automatically on each merge request. API tests are themselves subject to code review, including by dedicated testers, for thoroughness. This helps ensure robust and stable API behaviour.

3.2 Security Functional Requirements

The platform fulfils the following security functional requirements:

3.2.1 Identification of platform type

The platform provides a unique identification of the platform type, including all its parts and their versions.

Self-assessment:

1. The developer can identify the version of the TOE they are working with in the IAR Embedded Workbench IDE, where the Embedded Trust plugin in adds an "Embedded Trust Release Notes" option to the "Help" menu. Selecting the option shows the release notes in the system web browser. The release notes identify the installed version of the Embedded Trust plugin. Embedded Trust uses semantic versioning. The TOE is versioned along with the Embedded Trust plugin.

   This is tested by selecting the "Embedded Trust Release Notes" option in the "Help" menu in IAR Embedded Workbench, and verifying that the system web browser loads release notes including the correct Embedded Trust version number.

2. The user application can identify the version of the TOE installed using the Application API *STZ_getSBMInformation*. This API returns the version number of the TOE. This version number is optionally defined by the developer by setting the conditional compilation flag *SBM_REPORT_SBM_VERSION* and defining the string *SBM_VERSION_ID*. The developer is given control of this to allow them to increment the version number for their bootloader project when they change their implementation of functions called by the OEM APIs. To keep track of which version of the TOE is installed on which IoT devices, the developer must keep production records showing what version of the TOE was installed onto each device. Alternatively they can have connected devices report *SBM-VERSION_ID* to a central service, and maintain a separate record of which version of the TOE is associated with each *SBM_VERSION_ID*.

   This is tested by enabling *SBM_REPORT_SBM_VERSION* and defining *SBM_VERSION_ID* and verifying that this produces a TOE build that reports the correct version number to the user application via the *STZ_getSBMInformation* API.

~~3.2.2 Secure update of platform~~

~~The platform can be updated to a newer version in the field such that the integrity, authenticity and confidentiality of the platform is maintained.~~

3.2.3 Identification of individual platform

The platform provides a unique identification of that specific instantiation of the platform, including all its parts and their versions.

Self-assessment:

As a provisionable key store the TOE is provisioned with a unique secret identity key (256 bit ECC key using curve NIST P-256) and a corresponding certificate (PEM format X.509 containing the corresponding 256 bit ECC public key, with a SHA-256 hash). The certificate is issued by a CA installed on the factory provisioning system and is installed onto the device with the complete chain of CA certificates back to the root. This certificate provides each individual device with a unique cryptographic identity, verifiable by challenging the device to prove possession of the corresponding private key by signing a piece of data provided by the challenging party. The IETF's TLS specifications (IETF RFC 5246 and IETF RFC 8446) for transport-layer security implement such a challenge-response mechanism and are widely implemented in embedded Internet protocol libraries.

This is tested by provisioning the TOE onto a target microcontroller using the Embedded Trust provisioning system and having a test user application report the installed identity certificates and prove possession of the corresponding private key.

3.2.4 Genuine platform instantiation

The platform provides an attestation of the "Identification of platform type" and "Identification of individual platform", in a way that cannot be cloned or changed without detection.

Self-assessment:

Composite developers can guarantee that end users receive only genuine devices, not clones or counterfeits, by having remote Internet services or the user application or both check the TOE for a validly signed device identity certificate. Such certificates are only issued by the factory provisioning system provided as a component of the Embedded Trust product. This system signs device identity certificates using a CA key generated by the composite developer and installed via secure channels into a secure provisioning system located on the authorised production line. Only Embedded Trust provisioning systems specifically authorised by the composite developer receive this CA key. Without it, no counterfeiter can issue a valid device identity certificate.

Validly certified devices cannot be cloned either, because their private identity keys are never exposed either before or after being provisioned onto each device.

This is tested by provisioning the TOE onto a target microcontroller using the Embedded Trust provisioning system and having a test user application report the installed identity certificate chain and verifying that the composite developer's production CA certificate is present in that chain.

3.2.5 Attested secure state of platform

The platform provides an attestation of the state of the platform, such that it can be determined that the platform is in a secure state.

Self-assessment:

End users and remote services can check that the device's identity certificate chain includes the CA certificate of a composite developer that they trust. If such a certificate is present, and the composite developer is trusted to have securely provisioned the TOE, then the relying parties can trust that the connecting device is genuine and has executed a secure boot process, and consequently is running authentic and integral software, that is reporting correct information about its state.

Because this implementation relies on SFRs "Genuine platform instantiation" and "Secure initialization of platform", its validation is derived from the validation of both those SFRs.

3.2.6 Factory reset of platform

The platform can be reset to the state in which it exists when the composite product embedding the platform is delivered to the user, before any personal user data, user credentials, or user configuration is present on the platform.

Self-assessment:

After production-time provisioning of the TOE and user application, the IoT device is ready for operation. During operation, storage of user data is managed by the user application - the TOE does not store user data. The TOE always remains in the state it was originally after production.

This is verified by reviewing the code of TOE functions serving application APIs, to ensure that no data originating in the user application is stored by the TOE.

3.2.7 Secure install of application

The application can be installed in the field such that the integrity, authenticity and confidentiality of the application is maintained.

Self-assessment:

The composite developer can install a first SWUP file to the update slot on devices in the field by calling a serial loader from the TOE. This requires setting the conditional compilation flag *SBM_INCLUDE_LOADER* so that the TOE calls the OEM API function *sbm_serial_loader*, which the composite developer must implement. The TOE will then validate the SWUP and install the user application into the active slot before passing execution to it.

Validation consists of a check that the SWUP is signed by the composite developer. The signature is generated by the Embedded Trust plugin in the IAR Embedded Workbench IDE when the composite developer exports the SWUP. The signature is generated using ECDSA with NIST curve P-256. The signature is verified by the TOE using the same algorithm, the necessary trust anchor certificate being part of its provisioned data. This check verifies both authenticity and integrity of the SWUP.

Application firmware binaries packaged in SWUPs are encrypted using a 128-bit AES key in GCM mode. This key is derived by both parties using ECIES key agreement as described in section 3.2.9 part 2. The key pair used in the ECIES procedure by the Embedded Trust plugin is ephemeral. A new key pair is generated for each SWUP export operation. The key pair used in the ECIES procedure by the TOE is part of its provisioned data and is known as an

Update Group Key pair because it may be provisioned onto a group of devices of the same underlying hardware platform, which will receive the same SWUP file.

This is tested by building the TOE with a test implementation of the *sbm_serial_loader* function that reports on a serial output that it has been called.

3.2.8 Secure update of application

The application can be updated to a newer version in the field such that the integrity, authenticity and confidentiality of the application is maintained.

Self-assessment:

The composite developer can load an updated version of the application into the update slots of selected devices in the field by implementing an over-the-air SWUP distribution mechanism in all over-the-air updatable versions of the application. Once a SWUP has been downloaded into the update slot it will be validated and decrypted into the active slot by the TOE on next reset. Reset can be triggered by the running version of the application.

Validation steps include a check that the SWUP is signed by the composite developer as detailed in section 3.2.7, and that its version number is higher than that of the currently installed application. Version numbers are set by the composite developer at project build time and consist of one to three integers, each from 0 to 255, separated by periods, for example: 9.4.4. Decryption also proceeds as detailed in section 3.2.7.

This is tested by enabling debug access in a test build of the TOE and writing both valid and invalid SWUP files into the test device's update slot, verifying that on reset only valid SWUPs are installed.

3.2.9 Cryptographic operation

The platform provides the application with:

1. **ECDSA digital signature generation and verification** functions per NIST FIPS 186-4 Digital Signature Standard (DSS) section 6.4, using curve P-256 (i.e. a key length of 256 bits) per section D.1.2.3 of the same document. These functions are made available in the Application API as *STZ_signUsingKey* and *STZ_verifyUsingKey*.

   Signature generation is tested by verifying that If the Application API *STZ_signUsingKey* is called with a 256b hash and an index to a 256b ECC private key as arguments, an encrypted version of that hash is returned, decryptable using the public part of that key and the NIST P-256 curve.

   Signature verification is tested by verifying that if the Application API *STZ_verifyUsingKey* is called with a 256b encrypted hash and a 256b ECC public key as arguments, the decrypted hash is returned.

2. **ECIES key agreement** per NIST SP800 56Ar3 Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography, sections 6.2.2.2 and 5.7.1.2, excepting that the shared secret is returned directly from the ECDH process and not put through a key derivation function. Key agreement also uses curve P-256

(i.e. 256 bit private and public keys), per section D.1.2.3 of NIST FIPS 186-4 Digital Signature Standard (DSS). This function is made available in the Application API as *STZ_generateSharedSecret*.

This is tested by verifying that if the Application API *STZ_generateSharedSecret* is called with an ephemeral ECC NIST P-256 public key and an index to an ECC NIST P-256 private key as arguments, an ephemeral 256b secret is returned.

4 Mapping and sufficiency rationales

4.1 SESIP1 sufficiency

Assurance Class	Assurance Families	Covered by	Rationale
ASE: Security Target evaluation	ASE_INT.1 ST Introduction	Section "Introduction" and "Title"	The ST reference is in the Title, the TOE reference in the "Platform reference", the TOE overview and description in "Platform functional overview and description".
	ASE_OBJ.1 Security requirements for the operational environment	Section "Security Objectives for the operational environment"	The objectives for the operational environment in "Security Objectives for the operational environment" refers to the guidance documents.
	ASE_REQ.3 Listed Security requirements	Section " Security Functional Requirements".	All SFRs in this ST are taken from [SESIP]. "Identification of platform type" is included. Exclusion of "~~Secure update of platform~~" is addressed in section" Flaw Reporting Procedure (ALC_FLR.2)"
	ASE_TSS.1 TOE Summary Specification	Section "Security requirements	All SFRs are listed per definition, and

		and implementation”	for each SFR the implementation and verification is defined in Security Functional Requirements.
ALC: Life-cycle support	ALC_FLR.2 Flaw reporting procedures	Section “Flaw Reporting Procedure (ALC_FLR.2)”	The flaw reporting and remediation procedure is described.
AVA_VAN.1	AVA_VAN.1 Vulnerability survey	Section “Vulnerability Survey (AVA_VAN.1)”	The vulnerability survey and associated test results are described.

5 References

[SESIP] Security Evaluation Scheme for IoT Platforms, version 1.3

[AM] Application of Attack Potential to Smartcards and Similar Devices , version 3.0

[TJERAND] Comparative Study of ECC Libraries for Embedded Devices, Tjerand S, COSADE 2019