Keywords: Secure the IoT, secure boot is the "root of trust" for embedded devices, public key cryptography

APPLICATION NOTE 6005

SECURE THE IOT: PART 2, A SECURE BOOT, THE "ROOT OF TRUST" FOR EMBEDDED DEVICES

By: Yann Loisel, Security Architect and Stephane di Vito, Security Expert, Maxim Integrated

Abstract: This is the second app note in a two-pat series on ways to secure the IoT. This app note concentrates on a secure boot which is, in fact, the "root of trust" and the cornerstone of an electronic device's trustworthiness. It demonstrates how device security can be conveniently implemented and how devices can even be updated in the field. The DeepCover® secure microcontrollers will serve as trust-enabling example devices to secure the IoT.

Introduction

This is our second application note on security issues for the Internet of Things (IoT). Security of electronic devices is a "must have" in today's interconnected world. There is plenty of evidence[1] to show that when the security of a device on the IoT is compromised, you must be cautious, even suspicious of that device and the whole IoT. You most certainly cannot rely on a hacked device for secure data exchange, processing, or storage.

In our Part 1 application note, **Secure the IoT: Part 1, Public Key Cryptography Secures Connected Devices**, we focused on the identification of security risks and argued that the best security is embedded in electronic devices. We emphasized countermeasures, specifically public key-based algorithms.

Now in this follow-up Part 2 application note we concentrate on a secure boot which is, in fact, the "root of trust" and the cornerstone of an electronic device's trustworthiness. Note that this discussion assumes that the reader understands the difference between a private and public key in cryptography. You can quickly refer to our prior application note to find plenty of discussion in a Google® search of the terms. Here we will demonstrate how device security can be conveniently implemented and how devices can even be updated in the field. The DeepCover® secure microcontrollers will serve as examples of trust-enabling devices to secure the IoT.

Root of Trust Starts with Trusted Software

The only solution to protect against attacks that try to breach the casing (i.e., the hardware) of an electronic device is to use a microcontroller that starts executing software from an internal, immutable memory (note that not every microcontroller has this setup and capability). The software stored in the microcontroller is considered inherently trusted (i.e., the root of trust) because it cannot be modified. Such impregnable protection can be achieved using read-only memory (ROM). Alternatively, flash (EEPROM) memory internal to the microcontroller can also be used to store the root-of-trust software, if suitable security exists. Either there is a fuse mechanism to make this flash memory nonmodifiable (as a ROM) once the software has been written into it, or there is a proper authentication mechanism that allows only authorized persons to write the root-of-trust software in flash memory. If this early software can be modified without control, trust cannot be guaranteed. "Early" means that it is the first piece of software executed when the microcontroller is powered on. Hence, the requirement for inherent trustworthiness of this initial software. If this software is trustworthy, then it can be used for verifying the signature of the application before relinquishing the control of the microcontroller. It is like a castle built on strong foundations.

Booting into a Secure State

At power-on, the device's microcontroller starts running the root-of-trust code from a trusted location (e.g., ROM, trusted internal flash). This code's primary task is to start the application code after successful verification of its signature. Verification of the signature (see **Taking Ownership of the Device** below) is done using a public key previously loaded in the microcontroller using Method 1:Self-certification or Method 2:Hierarchical certification discussed in our Part 1 application note. For the reader's convenience, we are using a **Sidebar** below to reproduce that discussion of methods to verify and guarantee the integrity, authenticity, and identity of public keys.

These protected microcontrollers can still be used in their usual manner by developers for writing software, loading and executing the software via JTAG, and debugging. A secure microcontroller does not make development harder.

Releasing the Software and Signing the Code

Once the software is completely finished and tested, and then audited and approved by a certification laboratory or an internal validation authority, it must be released. Releasing software for a secure boot requires one additional, important step: the binary executable code signature. Signing the code, in fact, "seals" the code (it cannot be further modified without those modifications being detected) and authenticates it (the identity of the approver is well established). The code is sealed because, if modified, the associated signature will become invalid–the integrity of the digital signature will no longer be complete. The code is also authenticated because it was signed by a unique, undisclosed, private key guarded jealously by its owner–the persons in charge of signing the code.

Signing the code is an important step for certified software. Once the software has been approved by an external or internal validation authority, it cannot be changed.

Taking Ownership of the Device

Taking ownership of a device is done by personalizing the root of trust in the microcontroller, the immutable code that handles the secure boot. But we also need to load the public code verification key owned by the software approver into the device (see Part 1 application note). Recall that this key is fundamental and shall be trusted.

There are two schemes to personalize the root of trust. One approach uses a small key hierarchy and the other approach has no key. We will now examine both approaches.

In this first approach (**Figure 1**) the device's root of trust already contains a root public "key verification key" (inherently trusted as part of the root of trust). We call it the master root key (MRK). Restated simply, that key is hard-coded in the root of trust, which is used to verify the public code verification key (CVK). (See Method 2 in the **Sidebar**.) As a consequence, the public CVK must be signed prior to being loaded into the microcontroller. For this signature operation, the signing entity is the owner of the root of trust (i.e., the silicon manufacturer who owns the private key matching the public key hardcoded in the root of trust). Once the public CVK has been loaded and accepted by the root of trust, the root of trust's key is not used any more, except for internally rechecking the CVK at each boot to ensure that it has not been modified or corrupted, or for updating the public CVK. The CVK is now used to verify binary executable code. This personalization step has significant potential benefit: it can be executed in an insecure environment because only the correctly signed public key can be loaded into the device. This personalization step, moreover, is not a big hurdle as the same CVK can be deployed on all devices. Note that the CVK can be stored in external, unprotected memory since it is systematically reverified before being used.

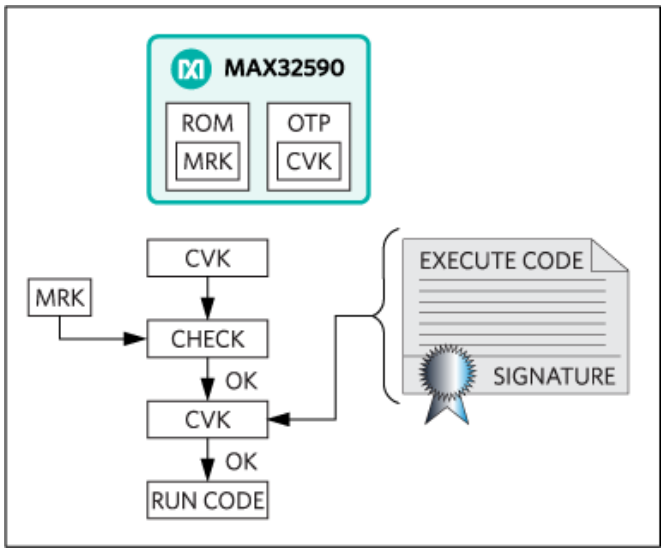

Figure 1. Code verification key (CVK) is verified by the MRK before being used to verify the executable code and then execute it.

A second, simpler approach to personalize the root of trust uses no prior key. Consequently, the public CVK has to be loaded either into internal memory that can be written only by trusted software running from that internal memory, or into nonmodifiable memory such as one-time-programmable (OTP) memory or locked flash (EEPROM) in a secure environment. A trusted environment is needed to ensure that the intended public key is not replaced by a rogue key because the root of trust cannot verify this key. This key also has to be internally protected by a checksum (CRC-32 or hash) to ensure that there is no integrity issue with that key. Alternatively, to save precious OTP space, the key can be stored in unprotected memory, but its checksum value is stored in internal OTP memory.

As described here, one could imagine a multiparty signing scheme where multiple entities (e.g., the software approver and an external validation authority) have to sign the executable code. One could also imagine more complex hierarchies with different code verification keys, more intermediate key-verification keys, and even multiple root keys. The ultimate process used really depends on the context and the security policies required for the application.

During this personalization step, other microcontroller options can be permanently set, for example disabling the JTAG. While useful during development, the JTAG must be disabled on production parts otherwise, the root of trust can be bypassed.

Downloading Application Code During Manufacturing–a Simple and Trusted Process

One step in the manufacturing process consists of loading the previously sealed/signed binary executable code into the device.

A public key scheme has the following unique advantages:

- No diversification is needed.
- No secret is involved.
- The binary executable code is sealed by the software approver and cannot be modified.

The signed binary executable code, therefore, can be loaded anywhere. Only this code will be loadable and executable by the electronic device; other binary executable codes will be rejected.

Using public key cryptography for this process is extremely valuable as no security constraints are imposed upon the manufacturing process.

Deployment, Field Maintenance

Secure-boot-based devices are deployed in the field like others. However, to update the executable code in the field, you must sign it with the software approver's private key and load it into the device by an appropriate means like a local interface or a network link.

If the software approver's code-signing key is compromised, the associated public CVK can also be replaced in the field, provided that the new key is signed by a public-key-verification key previously loaded into the device. (This key-verification key is a "revocation-then-update" key.) Compromising the root key (MRK) is not, however, an option because this root key cannot be replaced in the field. Proper private key management policies do mitigate this risk.

Cryptography Is Not Enough

Assume now that appropriate keys management and good protection practices are followed. Assume too that manufacturing security, trust, and confidence are guaranteed. Now, finally, assume that good cryptography is chosen, like standardized algorithms, long enough keys, high-quality random numbers. Nonetheless, some major security threats still remain and severely expose the device's assets.

The public key is stored in a location of flash memory that is locked, i.e. it cannot be modified any more. If the integrity of the preprogrammed public key or the secure boot is based on a lock mechanism in flash memory or in OTP memory, then the strength of the integrity depends only on the strength of this lock technology. Any attacker who can defeat this technology, can defeat the integrity of the targeted asset itself.

Similarly, a software check of digital signatures should be performed. So, besides compliance with the algorithm, there are several ways to verify the software–some methods robust and others less so. Robustness means resistance to errors (intentional, or not), unexpected problems, mistakes, abnormal environmental conditions (e.g., low temperature, bad powering from power glitches), or corrupted bytes. These constraints are usually addressed and managed by the software and hardware validation of the device.

But robustness also means resistance to specific, deliberate, focused attacks. These malicious attacks can be performed randomly, without any real knowledge of the platform, (i.e., a "black box" attack). Or the attack might come after serious study of the device, (i.e., assaults ranging from "grey" to "white box" attacks and corresponding to the attacker's level of understanding of the platform). In each instance the attacker is looking for weaknesses or limitations that can be converted into attack paths.

Two simple examples illustrate this idea. Our first example involves software good practices, which require you to check the bounds and lengths of inputted data before processing them. The use of static analysis tools, intended to assess the code source quality, is also part of good practices. These actions help developers and auditors to easily improve and guarantee the code quality. It is also recognized that an efficient developer will detect misformatted bunches of bytes. Regrettably, these checks are not regularly implemented in the rush to deliver new software as quickly as possible because, moreover, these "good practice" controls add development time, test and validation time, and code size, and are often considered less critical or even unnecessary compared to basic functioning. Consequently, buffer overflows (**Figure 2**) might emerge after implementing communication protocols, even those considered theoretically secure such as TLS, or memory-to-memory copies like the copy from external NAND flash to RAM before running an application. These buffer overflows are usually well-functioning attacks on the regular operational software.

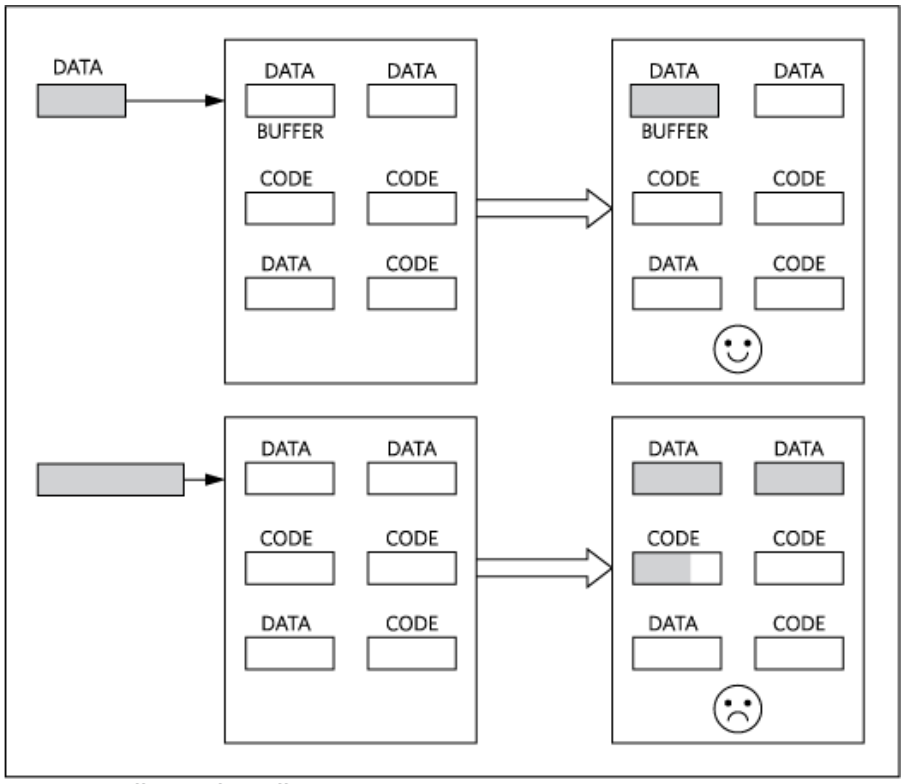

Figure 2. Buffer overflow effects.

Another example involves processes choices and is called a fault attack (**Figure 3**). It is agreed from the above section that a digital signature check is powerful for detecting any integrity/authenticity failure in data/code. In fact, the check can be performed on the bytes where they are stored before they are copied to the application's running memory. But in some other operating scenarios, the byte copy is performed before the check has occurred. This means that these bytes are almost ready to be used/run, even though they are not matching the digital signature. If the attacker is able to skip the check step by triggering a power glitch (Figure 3) or any other kind of small, nondestructive fault, the normal process can be disturbed and skip the check operation, thereby enabling the loaded bytes to be run as a genuine code.

Generally speaking, making a security mechanism only rely on a single implementation mechanism weakens this security and motivates attackers to focus on circumventing the implementation.

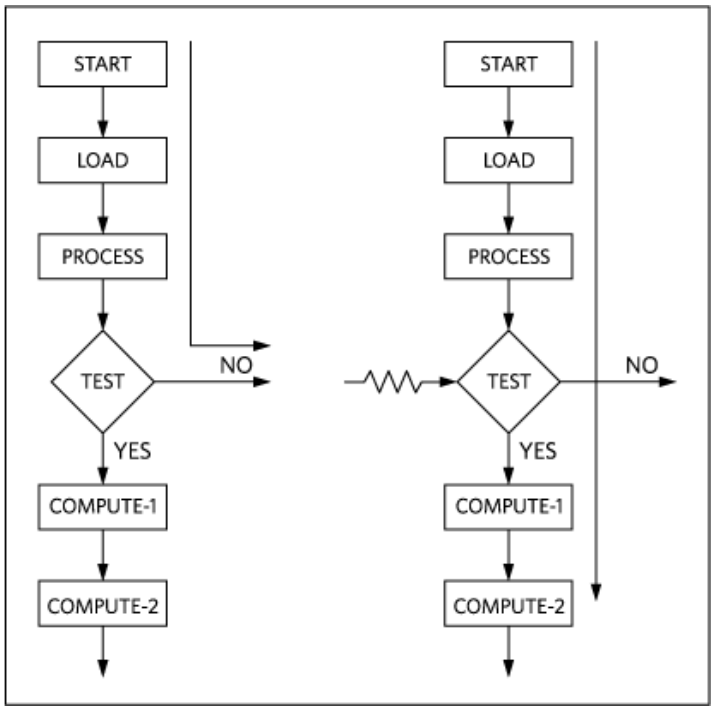

Figure 3. Effect of a fault attack: the fault modifies the normal path of the program by changing the test condition result.

Implement the Best Solution

Today there are secure microcontrollers like the MAX32590 that feature a root of trust containing a preloaded, immutable root key. The root of trust containing the MRK in these secure microcontrollers is either in ROM or internal OTP or internal flash that is locked at the factory. Because the memory technology used to store the key is immutable, the integrity of the MRK is guaranteed. Finally, a checksum is still computed over the key to ensure that no glitch happens before that key is actually used.

To initialize the customized preloaded key, customers submit their public CVK to the manufacturer for signature. The manufacturer signs (i.e., certifies) the customer's public key using their private root key stored in an HSM^2 which is strictly controlled. They then send the signed public key back to the customer. This process is quick and required only once before the software is released for the first time (note that this secure step is not needed during the software development). A customer can then load and replace the manufacturer's key with their own key and download their signed binary executable code.

This process is very flexible because the same key is programmed onto every part, thus making the personalization process very easy. It is even possible for a manufacturer to personalize (i.e., customize) parts with the customer's key before the part is even manufactured–a major reduction of hassle for the customer. This personalization step can also be done by the customer itself. As an interesting shift of liability, this latter key personalization step allows customers to take ownership of the microcontroller themselves. Finally, the ROM code in DeepCover secure microcontrollers (like the MAX32550) allows keys to be revoked and replaced in the field with no loss of trust.

It is very important to note that this key certification process cannot be bypassed. It is not optional. Even secure parts in development enforce these same principles, with a single difference: they can be provided in very limited quantity with test keys activated to lessen the exposure of the customer's key during development or parts evaluation.

It is not enough to design a secure solution for today. The best practical solutions will design for future upgrades too. The most trustworthy security devices like the DeepCover devices support future-proof RSA (up to 2048 bits) and ECC (up to 521 bits) signature schemes. Moreover, PCI PTS labs have audited their code. In addition, hardware accelerators make the

code verification at startup almost invisible so the security protocols do not put any extra burden during the boot process. The digital content digest is computed on the fly as the code is copied from flash to executable RAM. Then the signature verification process takes very little extra time.

The DeepCover devices combine these security mechanisms with other protections, such as JTAG-ICE deactivation which makes code dump, modification, or replacement impossible because only one mechanism can address any security on flexible/programmable parts. All "doors" shall be either closed or keyed to provide full confidence to stakeholders.

Conclusion

We have seen that a secure boot is an inexpensive, but critically important security mechanism that can be used for devices on an IoT or in almost any application where assets are to be protected. The overall development and manufacturing processes remain very simple and straightforward even with this secure boot. The extra steps are only: loading a public code verification key (CVK) in each device, and signing the binary executable code to be loaded into the device.

We have said this before but there is value in saying it again. We cannot allow any security breach of electronic transactions, critical systems such as nuclear plants, or implantable medical devices. A secure boot, the root of trust, is one step to secure the IoT. It makes that trust possible and simple to implement.

Endnotes

1. Higgins, Kelly Jackson, "Smart Meter Hack Shuts Off the Lights," **DarkReading**, 10/1/2014, reports about Spanish smart meters that have been hacked, www.darkreading.com/perimeter/smart-meter-hack-shuts-off-the-lights/d/d-id/1316242. For more information, see Maxim Integrated application note 5537, "Smart Grid Security: Recent History Demonstrates the Dire Need;" also Maxim Integrated application note 5545, "Stuxnet and Other Things that Go Bump in the Night."
2. To assist the customer's production environment, Maxim Integrated provides a turnkey key-management solution using a HSM that meets PCI PTS 4.0 constraints. Go to www.pcisecuritystandards.org/security_standards/documents.php.

Sidebar: Methods to Guarantee a Public Key

Public key *integrity*, *authenticity*, and *identity* must all be guaranteed. This can be done in different ways.

Method 1: Self-certification. The recipient of the digital content receives the public key from the sender in person, or the sender transmits the public key in a way that leaves no doubt about the legitimate origin and ownership of the public key. Then this public key (also called a root key) can be trusted, as long as it is stored where unauthorized persons cannot modify it.

Method 2: Hierarchical certification. In this method, a hierarchy of verifiers guarantees the origin of the public key. Public key infrastructures (PKIs) provide the definitions of such hierarchies. The physical association between a public key and the identity of the key owner is a certificate. Certificates are signed by intermediate entities (i.e., certification authorities) of the PKI hierarchy.

Assume that a person wants to have a certified public key. That person generates a pair of keys and keeps the private key in a safe, hidden place. Then in principle, a certification authority meets this person face-to-face and thoroughly verifies the identity of that person. If authenticated, the identity information (name, organization, address, etc.) is attached to the public key and the resulting document is signed by the certification authority's private key. This permanently binds the identity information to the public key. The resulting signature is attached to the certificate. If any one element among the identity information, the public key value, or the certificate signature is tampered with, then the certified signature becomes invalid. Therefore, the information contained in that certificate cannot be trusted. The certification authority's public key can, in turn, be certified by yet another certification authority.

Certificate validity is verified by using the same cryptographic signature verification scheme as for digital content. The signature verification of the certificate guarantees the integrity and authenticity of the certificate and, consequently, of the information contained in the certificate: the public key and the identity **(Figure A)**.

For the full discussion, see our prior application note, **Secure the IoT: Part 1, Public Key Cryptography Secures Connected Devices**.

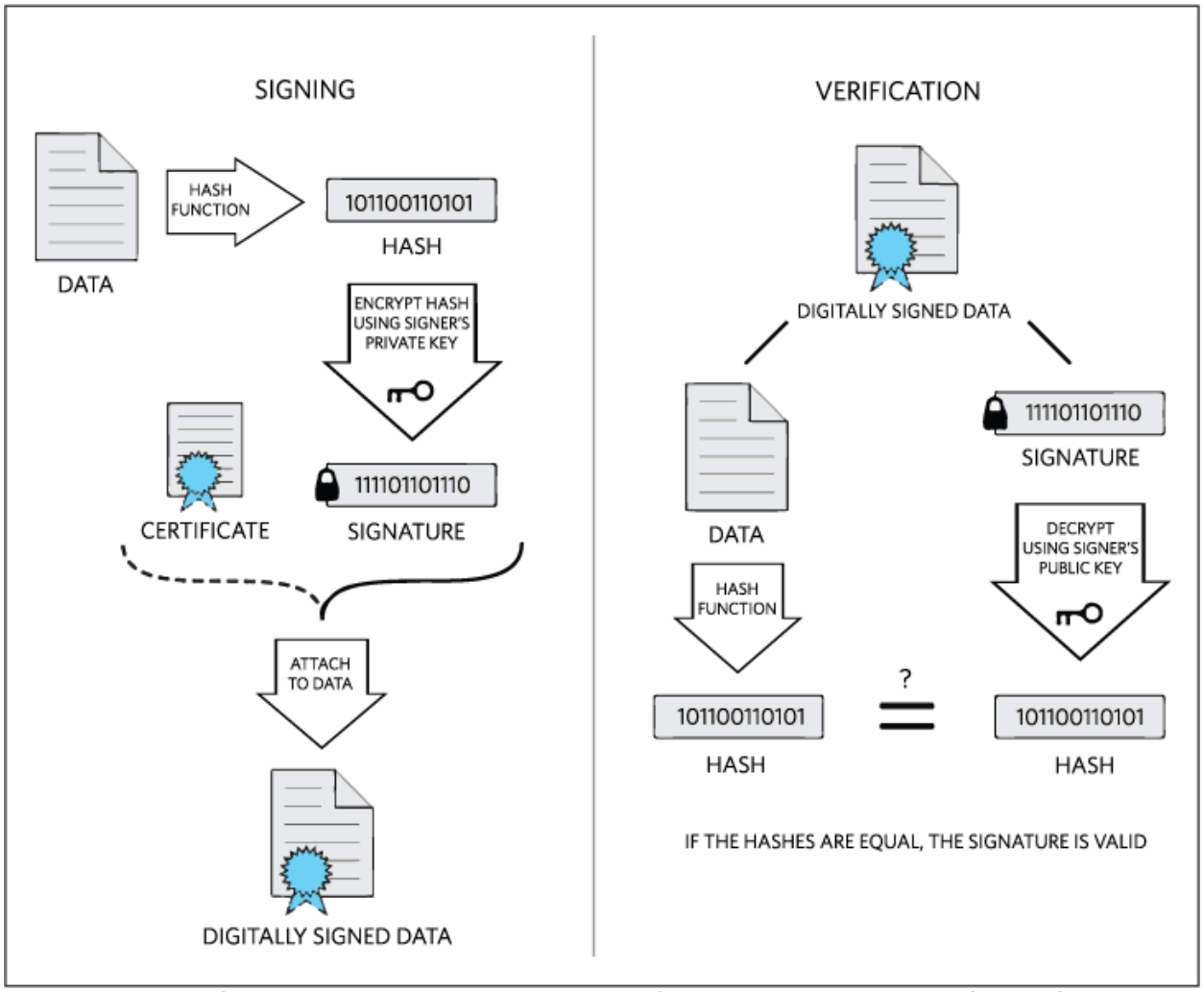

Figure A. A diagram of a digital signature, how it is applied and verified. Illustration Licensed under Creative Commons Attribution-Share Alike 3.0 via Wikimedia Commons, http://commons.wikimedia.org/wiki/File:Digital_Signature_diagram.svg#mediaviewer/File:Digital_Signature_diagram.svg.

A similar version of this article appeared January 11, 2015 on *Embedded*.

Related Parts		
MAX32550	DeepCover Secure Cortex-M3 Flash Microcontroller	Free Samples
MAX32590	DeepCover Secure Microcontroller with ARM926EJ-S Processor Core	Free Samples

More Information
For Technical Support: https://www.maximintegrated.com/en/support
For Samples: https://www.maximintegrated.com/en/samples
Other Questions and Comments: https://www.maximintegrated.com/en/contact

Application Note 6005: https://www.maximintegrated.com/en/an6005
APPLICATION NOTE 6005, AN6005, AN 6005, APP6005, Appnote6005, Appnote 6005