TUTORIALS 7274

CRYPTOGRAPHY: IS A HARDWARE OR SOFTWARE IMPLEMENTATION MORE EFFECTIVE?

By: Zia Sardar, Stewart Merkel, Aaron Arellano

Abstract: This tutorial is part of a series that is designed to provide a quick study guide in cryptography for a product development engineer. Each segment takes an engineering rather than theoretical approach on the topic. In this installment, you'll learn the difference between hardware and software implementations of cryptographic solutions and get insights on some common applications. A similar version of this tutorial originally appeared on May 27, 2020 on Electronic Design.

Comparing Two Cryptographic Approaches

Modern cryptographic algorithms can be implemented using dedicated cryptographic hardware or software running on general-purpose hardware. For various reasons, dedicated cryptographic hardware provides a better solution for most applications. **Table 1** shows a list of reasons hardware-based cryptographic solutions are more desirable.

Table 1. Hardware vs. Software Cryptography Comparison

Hardware-Based Cryptography	Software-Based Cryptography
1. Uses dedicated hardware thus much faster to execute.	1. Uses shared hardware thus slower to execute.

Hardware-Based Cryptography	Software-Based Cryptography
2. Not dependent on the operating system. Supported by dedicated software for operating the hardware.	2. Dependent on the security levels and features of the operating system and supported software.
3. Can use factory provisioning and securely store keys and other data in dedicated secure memory locations.	3. No dedicated secure memory locations available. Thus, susceptible to stealing or manipulation of keys and data.
4. Maxim's hardware implementations have protections built in against reverse engineering such as PUF (ChipDNA).	4. Software implementations can be easier to reverse engineer.
5. In a hardware system, special care is taken to hide and protect the vital information such as private keys to make it much more difficult to access.	5. In a general-purpose system where software cryptography is implemented, there are more ways to snoop and access to vital information. An example would be intercepting the private key in transit within the computer's system.

Applications in Cryptography Today

Understanding Secure Boot and Secure Download

IoT devices based on embedded hardware are woven into our everyday lives:

- Home devices such as WiFi cameras, thermostats, and smoke detectors
- Medical devices
- Wearables, including fitness trackers and smart watches
- Industrial machines such as robotic arms in factories

Almost all these devices (see **Figure 1**) contain boot firmware or downloadable data that access the internet, so they can be vulnerable to security threats. Boot firmware—the brains of the device—is essentially saved in nonvolatile memory inside the device. This software is updated periodically to correct and enhance certain features, from adding a new intruder detection algorithm for a WiFi camera or enabling better positioning of a weld from the angle of an industrial robotic arm.

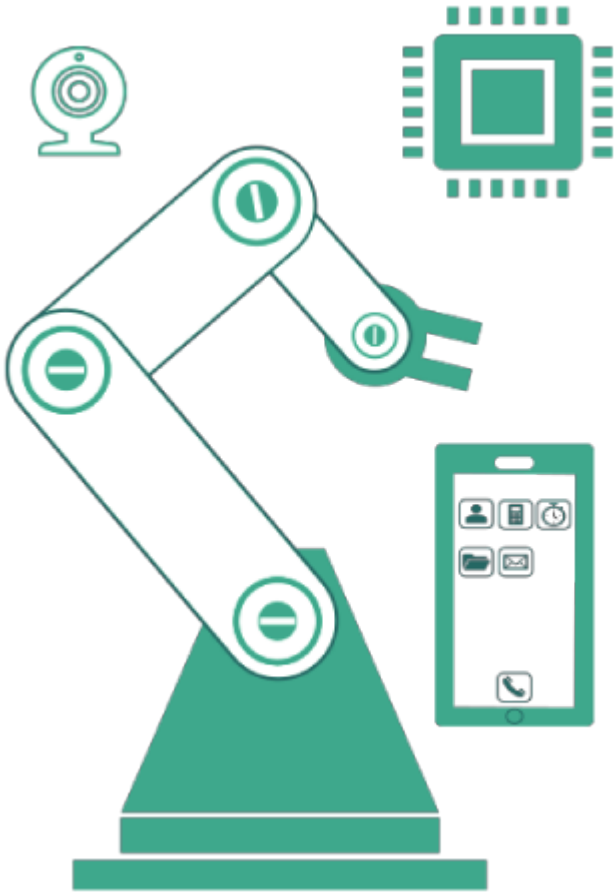

Figure 1. IoT devices such as a robotic arm in a factory have embedded hardware that could pose a security risk.

In this tutorial, we will cover all the necessary steps needed to securely boot as well as upload new firmware in a connected device.

Importance of Protecting IoT Device Firmware or Data

Because IoT devices must be trustworthy, the device firmware and critical data must be verified to be genuine. In a perfect world, boot firmware and configuration data would be locked down at the factory. In reality, customers have come to expect firmware updates and reconfiguration to be available over the internet. Unfortunately, this creates an opening for malicious actors to use these network interfaces as a conduit for malware. If someone gains control of an IoT device, they may take control of the device for malicious purposes. For this reason, any code that purports to come from an authorized source must be authenticated before it's allowed to be used.

An attacker may deliver malware to an IoT device by various means (**Figure 2**):

- Through physical access to the device, an attacker can introduce malware by means of a physical connection (such as USB, Ethernet, etc.).

- Operating systems often exhibit vulnerabilities that are closed as they are discovered by means of a patch. An attacker can introduce malware by accessing an unpatched system.
- Frequently, IoT devices will contact update servers to determine if updated firmware or configuration data is available. An attacker may intercept the DNS request and redirect the IoT device to a malicious source that hosts the malware or corrupt configuration data.
- The authentic website may be misconfigured in such a way to allow an attacker to take control of the website and replace authentic firmware with one that contains the attacker's malware.

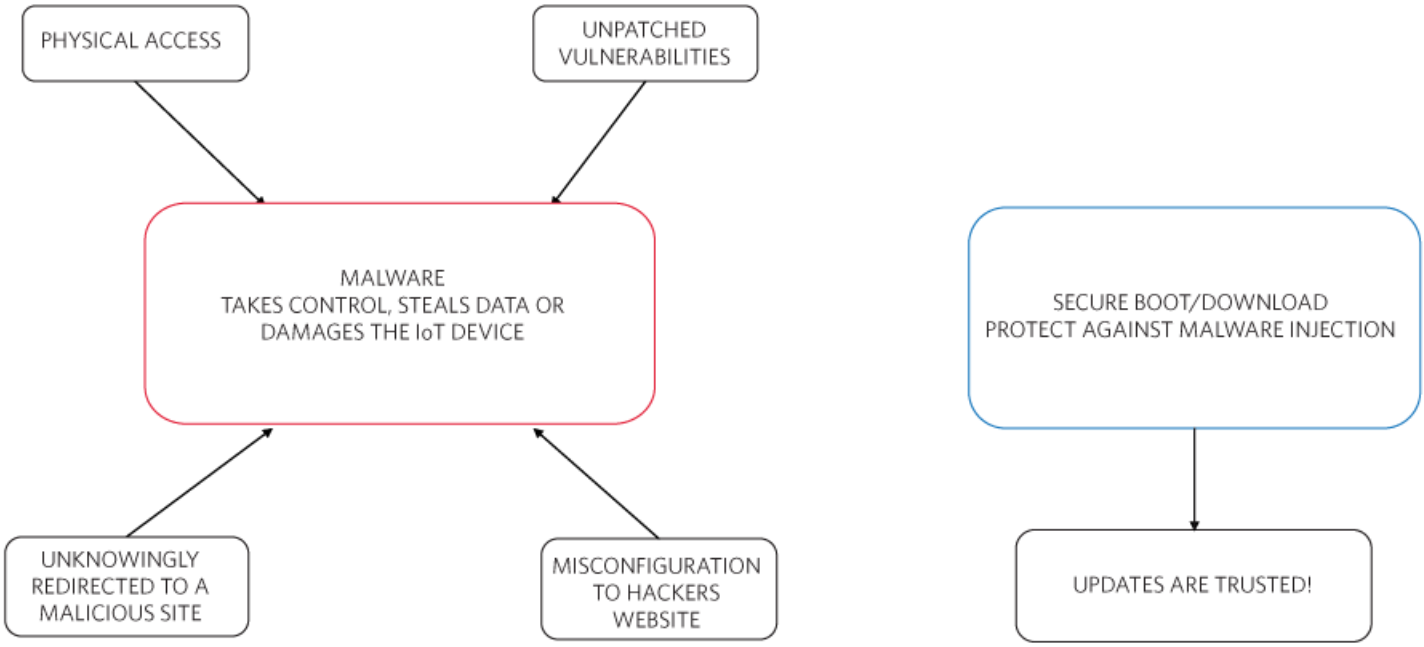

Figure 2. Attackers can infiltrate an unprotected IoT device vs. a secured IoT device.

Secure boot and secure download can help prevent infiltration and protect against malware injection. This means the IoT device can trust the updates being received from the command/control center. If a command/control center wants to fully trust the IoT device, an additional step of authenticating the IoT device's data needs to occur.

Authentication and Integrity of the Firmware

Authentication and integrity can provide a way to:

- Ensure that the targeted embedded device runs only authorized firmware or configuration data.
- Confirm that the data is trusted and not subsequently modified.
- Allow cryptography to be used to prove that data is both authentic and has integrity.
- Utilize cryptographic digital signatures, like a seal or manual signature at the bottom of a letter.

With authentication and integrity, the firmware and configuration data are loaded during the manufacturing phase and all subsequent updates are digitally signed. This way, the digital signature enables trust during the device's entire lifetime. The following features of digital signature are paramount to providing security.

- The digital signature used must be computed by a cryptographic algorithm.
- To bring the highest level of security, the algorithms need to be public and well proven.

For our secure solution, we'll examine asymmetric cryptographic algorithms, specifically, the FIBS 186 ECDSA.

Asymmetric Cryptography Applied to Secure Boot/Download

Asymmetric cryptography uses a public/private key pair for algorithm computations (**Figure 3**).

- The start of any key pair generation includes selecting a random number to be used as the private key.
- The random number is inputted into the key generator and the computation begins outputting a public key.
- The public key is made public (it can be distributed freely to all without any security risk).
- However, the private key is critical information that must be kept confidential.

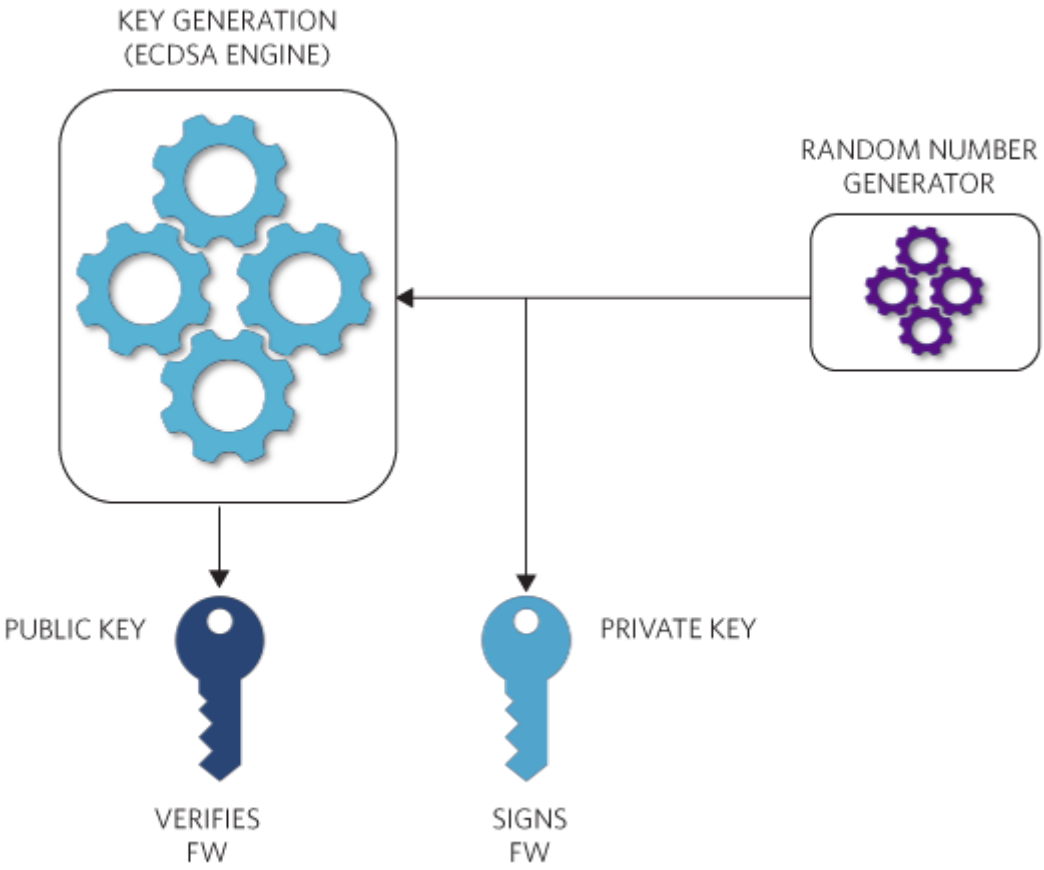

Figure 3. Asymmetric cryptography includes ECDSA key generation.

The fundamental principles of secure download in asymmetric cryptography are:

- The firmware developer uses the private key for signing.

- An embedded device (or an IoT device) uses the public key for verification.

What are the advantages of asymmetric key cryptography?

- No private key is stored on the embedded device.
- There is no way an attacker can retrieve the private key.
- The algorithm chosen (ECDSA) makes it mathematically infeasible to derive the private key from the public key.

Let's now look at an example of what must occur at an R&D facility that utilizes asymmetric key cryptography.

- We start with complete firmware.
- The firmware must be put through a SHA-256 multi-block hash computation.
- The private key and hash are inputted into the ECDSA signing algorithm. The output is a unique signature that could have only been signed by a private key.
- Combine our firmware with the signature and send it out upon request for field usage.

Figure 4 illustrates these points in greater detail.

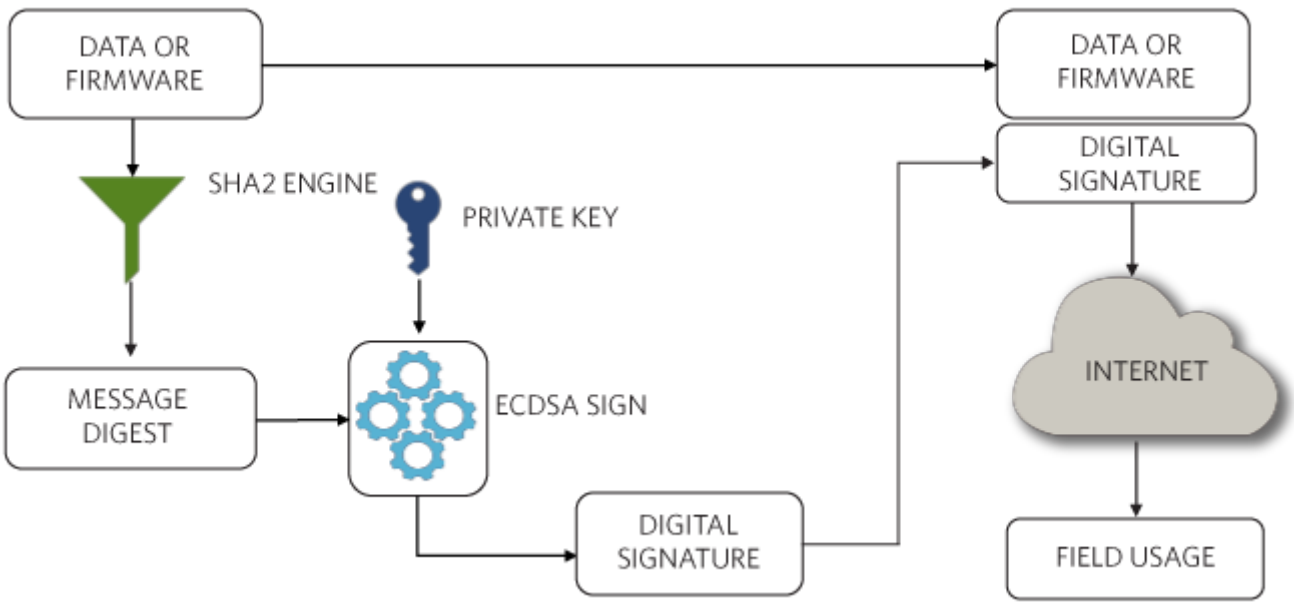

Figure 4. Asymmetric cryptography digitally signs a set of data or firmware.

Now let’s examine what occurs during field usage.

- The embedded device receives the firmware and signature.
- The firmware will go through a SHA-256 multi-block hash computation.
- Our embedded device will already contain the public key created during the key generation at the R&D facility.

- The signature and the other ingredients will then be used as input for the ECDSA verify.
- The result from the ECDSA verify will determine if the firmware can be used by the embedded device.
- If the result is a PASS, then the embedded device accepts the firmware that has both authenticity and integrity.
- If the result is a FAIL, then the firmware is rejected.

Watch this video, “Security Short Subjects: Secure Firmware Download for Embedded Systems,” to learn more about how firmware can be securely downloaded to a remote system.

Secure Boot and Secure Download Using DS28C36

For embedded devices that do not have a secure microcontroller with the computational capacity to perform the required calculations to verify the authenticity and integrity of downloaded firmware or data, DS28C36 DeepCover® secure authenticator is a cost-effective hardware-based IC solution (**Figure 5**).

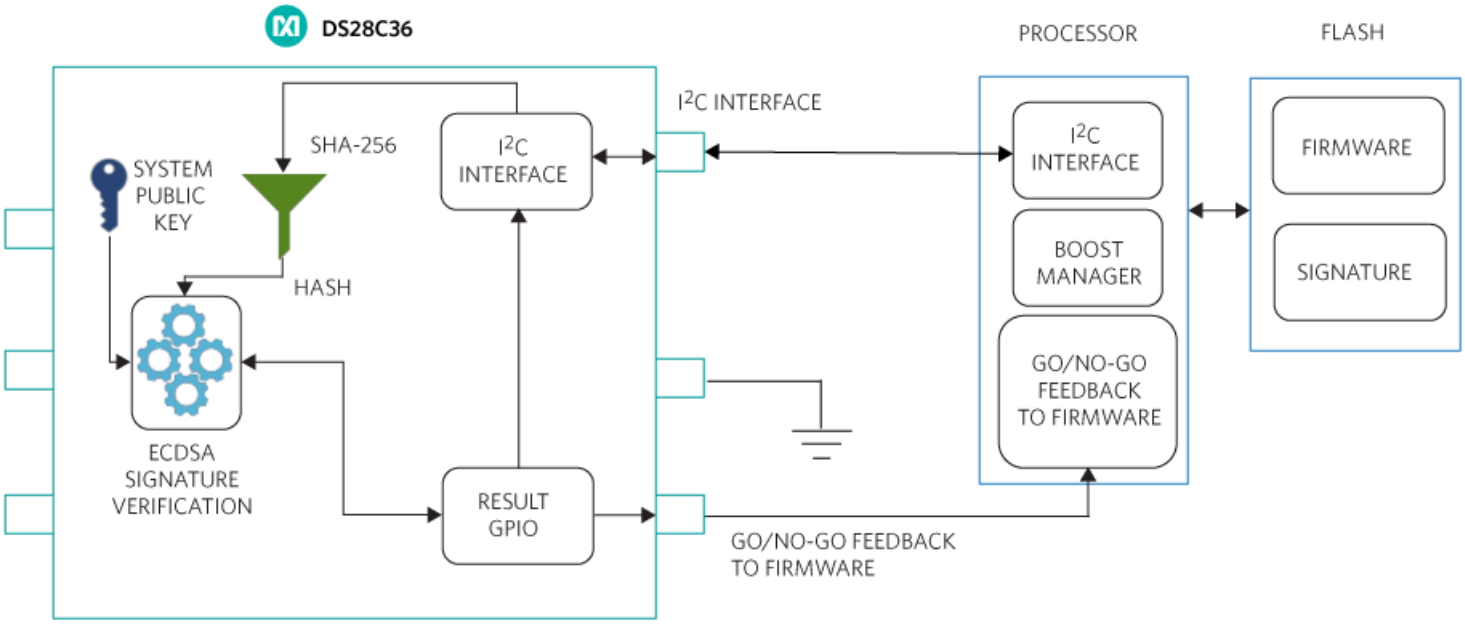

Figure 5. Secure boot and secure download in a cost-effective, hardware-based solution using the DS28C36.

Steps for secure boot and secure download:

1. As previously discussed, a system public-private key pair for the secure boot or secure download function is established at the R&D facility. The system private key of this pair is used to sign firmware or data that ultimately is verified by the DS28C36 embedded in the end system. This system private key never leaves the controlled development environment. The system public key of this pair is installed in the DS28C36 in a key register location that has an "authority key" attribute, which is a configurable setting in the DS28C36.
2. The system private key is used to compute the digital signature of the firmware or data.

3. The DS28C36 with the preprogrammed system public key is located on the interface to the host processor.
4. When firmware is required to be run by the processor, it is first retrieved by the processor boot manager and delivered to the DS28C36 in sequential 64-byte blocks to compute a SHA-256 hash.
5. After the DS28C36 completes the SHA-256 hash computation, the processor delivers the ECDSA signature of the firmware or data that was computed in the development environment and appended to the file.
6. After the DS28C36 receives the ECDSA signature, the processor sends commands to use the preinstalled system public key to perform a signature verification.
7. If the DS28C36 verifies the signature, a pass result parameter byte and a GPIO pin set to logic 0 is delivered to the processor. The status of this pin and parameter byte result acts as a go/no-go result to the processor to run the now known trusted firmware or data update.
8. In addition, if the command/control center would like to trust the DS28C36 an extra ECDSA signature engine is optionally available.

In summary, we have shown a proven security solution for secure boot or secure download using the DS28C36 that addresses threats to IoT devices. This secure authenticator IC offloads the heavy computational math involved to prove both authenticity and integrity of firmware or data updates.

For more information about Maxim’s secure boot and secure download solutions and services, please see the following:

- DS28C36 I^2C Interface DeepCover Secure Authenticator
- DS28E36 1-Wire Interface DeepCover Secure Authenticator

Go to the Security Lab tool to execute this sequence example or using Maxim’s other additional hardware labs.

Bidirectional Authentication to Protect Your IP

Bidirectional (or mutual) authentication is an important part of secure communication. Both parties of communication should be certain that their counterpart can be trusted. This can be accomplished by proving possession of private information. This information can be shared between the parties, or kept completely private, as long as the ability exists to prove possession.

Symmetric authentication systems require information to be shared among all participants in a given communication. This information is usually called a "secret.’ A secret is a piece of information known only to those who need it. The secret is used in concert with a symmetric authentication algorithm such as SHA, along with other data shared between participants. The ability to generate a matching signature on both sides of communication proves possession of the secret.

Asymmetric authentication systems (like ECDSA) employ hidden information that is not shared between parties (known as a ‘private key’) but is used to produce information that can be known to the public (known as a ‘public key’). Proper use of the public key proves possession of the private key because the private key is needed to unlock a message locked by the public key and vice versa.

Recipient Authentication

To authenticate a recipient device in a sender-recipient configuration, a piece of random data (also known as a “challenge”) is sent to a recipient. Along with any shared data between the devices, the challenge is run through a signing operation with a secret or private key to produce a “response” signature. The response signature can be verified by the sender because the sender is in possession of the shared secret, or a public key that corresponds to the recipient's private key. The general flow of this process is shown in **Figure 6**.

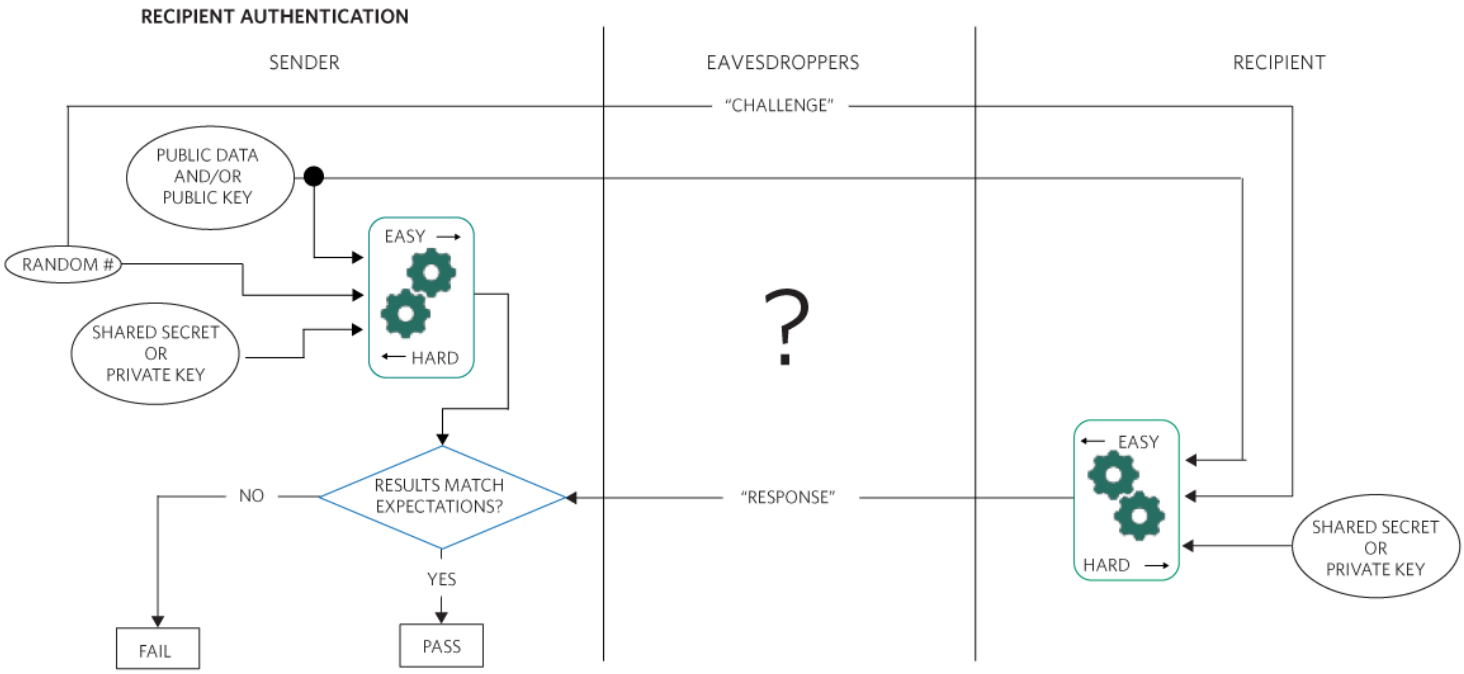

Figure 6. Recipient device authentication in a sender-recipient system.

Authentication generally depends upon algorithms that produce signatures that prove possession of a participant’s hidden information but make it difficult to discover the information itself. These are known as 1-way functions. SHA and ECDSA are examples of such algorithms.

Sender Authentication

In order to prove all parties can be trusted, the sender must also need to prove authenticity to the recipient. An example of this process is shown below in the form of an authenticated write.

In **Figure 7**, the sender is writing new data into a recipient device. However, to complete the write, the recipient must verify authenticity of the information by requiring the sender to produce a signature based on that information as well as the sender's hidden data (secret or private key). By using either a shared secret or the public key corresponding to the sender's private key, the recipient can verify that the signature is authentic.

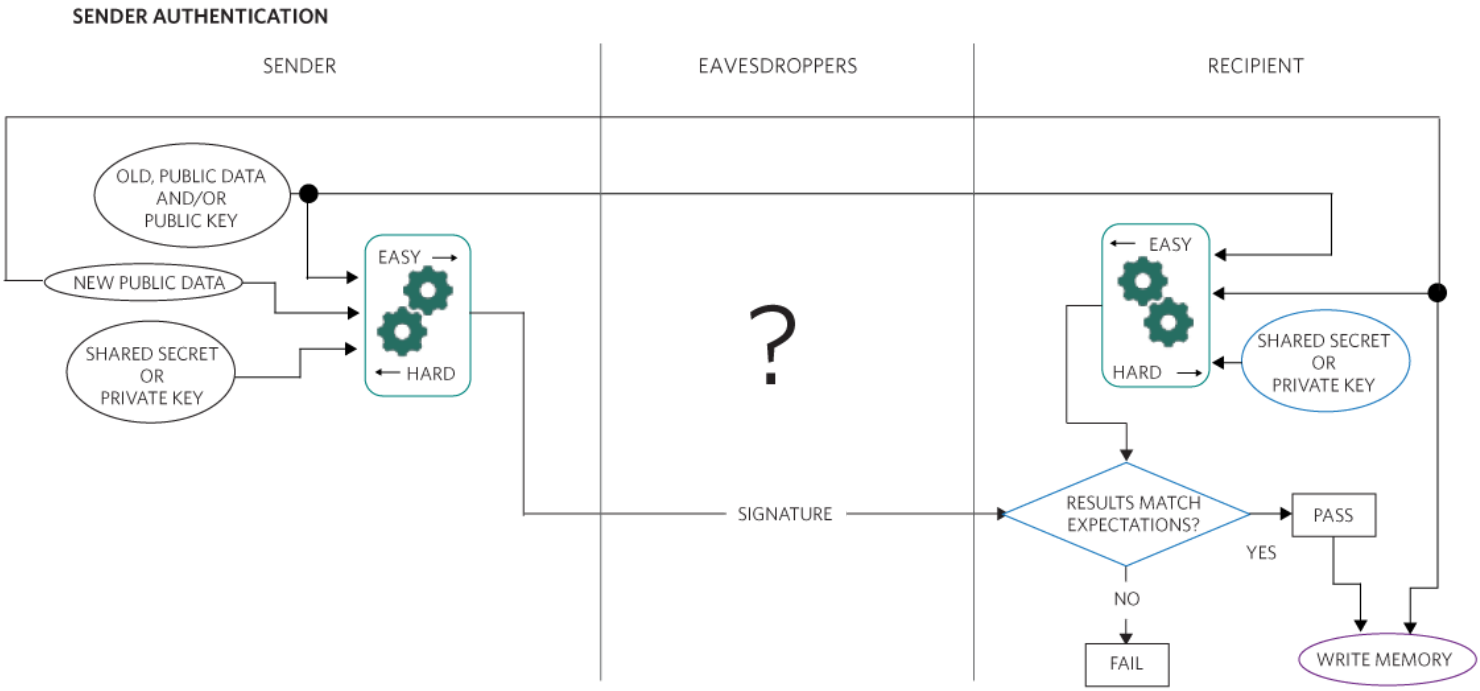

Figure 7. A sender writes new data into a recipient device.

The use of 1-way functions may allow any eavesdroppers to see all data being transmitted, but it prevents them from determining the hidden information that produced the signatures associated with the data. Without this hidden information, eavesdroppers cannot become impersonators.

This two-way authentication model can easily be used to make sure that intellectual property stored in a device will be well-protected from counterfeiters.

TRNG (True Random Number Generator) Output and Typical Use Maxim's ChipDNA™ secure authenticators have a built-in TRNG (**Figure 8**). This is used by the device for internal purposes. But they also have a command that sends out the TRNG output if the user requests it. At this time, the maximum length of the TRNG output length is 64 bytes. This hardware NIST-compliant random number source can be used for cryptographic needs such as "challenge (nonce)" generation by a host processor.

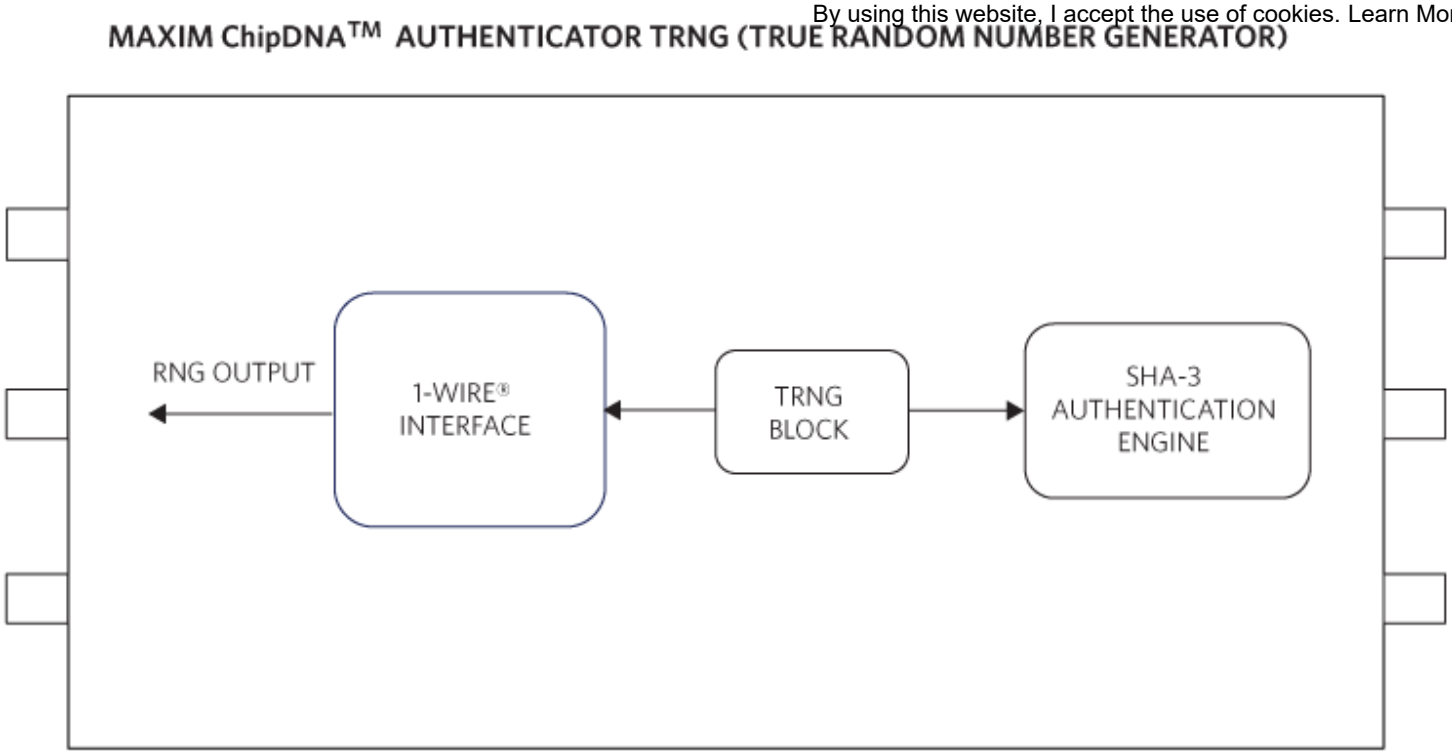

Figure 8. ChipDNA secure authenticator includes a built-in true random number generator.

There are three different specifications related to TRNGs

- NIST SP 800-90A
- IST SP 800-90B
- NIST SP 800-90C

For more details please visit the NIST website. Learn more about specific symmetric and asymmetric key-based hardware authenticators from Maxim Integrated that can be used to accomplish all the concepts discussed in this chapter. Watch for other segments in our series of cryptography application notes to continue deepening your understanding of this important security technique.

Related Parts		
DS2477	DeepCover Secure SHA-3 Coprocessor with ChipDNA PUF Protection	Free Sample
DS28E16	1-Wire Secure SHA-3 Authenticator	Free Sample
DS28C16	I²C Low-Voltage SHA-3 Authenticator	Free Sample
DS28E36	DeepCover Secure Authenticator	Free Sample

Related Parts		
DS28C36	DeepCover Secure Authenticator	Free Sample
DS28E39	DeepCover Secure ECDSA Bidirectional Authenticator with ChipDNA PUF Protection	Free Sample
DS28C39	DeepCover Secure ECDSA Bidirectional Authenticator with ChipDNA PUF Protection	Free Sample
DS28E50	DeepCover Secure SHA-3 Authenticator with ChipDNA PUF Protection	Free Sample
DS28C50	DeepCover I^2C Secure SHA-3 Authenticator with ChipDNA PUF Protection	Free Sample
DS28E83	DeepCover Radiation Resistant 1-Wire Secure Authenticator	Free Sample
DS28E84	DeepCover Radiation Resistant, High-Capacity 1-Wire Secure Authenticator	Free Sample
MAX66240	DeepCover Secure Authenticator with ISO 15693, SHA-256, and 4Kb User EEPROM	Free Sample
MAX66242	DeepCover Secure Authenticator with ISO 15693, I²C, SHA-256, and 4Kb User EEPROM	Free Sample
MAXQ1061	DeepCover Cryptographic Controller for Embedded Devices	Free Sample
MAXQ1062	DeepCover Cryptographic Controller for Embedded Devices	Free Sample

Next Steps	
EE-Mail	Subscribe to EE-Mail and receive automatic notice of new documents in your areas of interest.

APP 7274: 08 Jul, 2020
TUTORIALS 7274, AN7274, AN 7274, APP7274, Appnote7274, Appnote 7274

Customer Testimonials

Careers

Contact Us

Customer Support

Technical Support

Ordering FAQ

Worldwide Franchised Distributors

Investor Relations

Corporate Responsibility