RIoT

A Foundation for Trust in the Internet of Things

Paul England, Andrey Marochko, Dennis Mattoon, Rob Spiger, Stefan Thom, and David Wooten

{pengland, andreyma, dennisma, rspiger, stefanth, dwooten}@microsoft.com

Microsoft Corporation

1 Abstract

RIoT (Robust Internet-of-Things) is an architecture for providing foundational trust services to computing devices. The trust services include device identity, sealing, attestation, and data integrity. The term "Robust" is used because the minimal trusted computing base is tiny, and because RIoT capabilities can remotely re-establish trust in devices that have been compromised by malware. The term IoT is used because these services can be provided at low cost on even the tiniest of devices.

This paper describes RIoT's hardware requirements and how the hardware supports a range of foundational trust services.

2 Introduction

Technology is rapidly evolving and enriching the modern world with numerous applications that make computing devices indispensable to our daily lives. Many of these applications are extremely cost constrained and, since traditional approaches to hardware-based security add cost to the device, hardware-based security has often been deemed impractical. Simultaneously, devices are frequently deployed in demanding environments without physical security guarantees. Accelerated time to market, iterative refinements, and continually improving security research have created a need for frequent software updates to products in the field. However, these updates must be administered and verified without human involvement. RIoT addresses these new security realities, and can further augment traditional security techniques in use today.

Cryptographic operations and key management underlie many security scenarios, and the central contribution of RIoT is to provide a secure and manageable foundation for these services. Authentication, integrity verification, and data protection require cryptographic keys to encrypt and decrypt, as well as mechanisms to hash and sign data. Most internet-connected devices also use cryptography to secure communication with other devices and services. More complex scenarios like asset management through device identity, access control, verification of device health, and deployment of manufacturer controlled updates to devices and verifying their installation was completed, are also best accomplished using cryptography.

To support these and other scenarios, the fundamental cryptographic services provided by RIoT include:

Device identity

Devices typically authenticate themselves by proving possession of a cryptographic key. If the key associated with a device is extracted and cloned then the device can be impersonated, so protection of device identity keys is of paramount importance.

Data protection

Devices typically use cryptography to encrypt and integrity protect locally stored data. If the cryptographic keys are only accessible to authorized code, then malware or unauthorized software that may adversely affect platform security will not be able to decrypt or modify the data.

Attestation

Devices often need to be able to report the code they are running as well as their security configuration – a process called *attestation*. In particular, attestation is most often used to prove that a device is running up-to-date and patched code.

Cryptography only contributes to overall system security if the cryptographic keys are not known to adversaries. If keys are managed in software alone, then bugs in software components can result in key compromise, negating their security value. For software-only systems, the primary way to restore trust following key-compromise is to install updated software and provision new keys for the device under conditions of physical security. This is time consuming and expensive for typical PC, server, and mobile devices, and infeasible or impossible when devices are physically inaccessible.

Today's practical solutions to secure remote re-provisioning involve hardware-based security. Software-level attacks may allow adversaries to *use* hardware-protected keys but not *extract* them, so hardware-protected keys are a useful building block for secure re-provisioning of compromised systems. The Trusted Platform Module, or TPM, is an example of a low-cost security module that provides hardware protection for keys, and also allows the device to report (attest to) the software it is running. [1] Hence, a compromised TPM-equipped device can be securely issued new keys, and can also provide attestation reports that prove an update was successful.

TPMs are widely available on contemporary computing platforms, and the emergence of SoC-integrated and processor-mode-isolated firmware TPMs has reduced costs. However, TPMs are still impractical in some circumstances. For example, in a tiny IoT device that would not be able to support a TPM without a substantial increase in cost and power budgets. Another example where TPM use is impractical is securing the TPM itself - TPMs are complex software systems that need to be field-updated. If a TPM is compromised then it is unlikely that it has access to a second TPM to assist in re-provisioning and re-keying, so an alternative approach is needed.

RIoT has been developed to provide this sort of foundational device security for even the smallest of computing devices, although it can be applied to any processor or computer system. The hardware and software requirements for RIoT are extremely modest and should not add to bill-of-materials cost of any device that is sufficiently powerful to run a network stack. In particular, RIoT does not need a dedicated security processor or dedicated processor mode.

RIoT's security foundation is extremely simple and can realistically be free of exploitable bugs (in the simplest case it is little more than an HMAC function). If software components outside of the RIoT core are compromised, then RIoT provides for secure patching and re-provisioning

RIoT is also very flexible: its simple hardware foundation can be used to bootstrap a wide range of software-based security services.

Finally, RIoT takes a somewhat unusual approach to cryptographic key protection. The best-protected cryptographic keys used by the RIoT framework are only available briefly during boot. This method of key management is both the reason for RIoT's low cost, as well as the reason that long-lived RIoT keys are so well safeguarded. However, some operations and protocols must be adapted to work within this limitation.

3 RIoT in a Nutshell

This section provides a brief overview of the principles of RIoT-based systems. The detailed software and hardware architecture is more fully described in the sections that follow.

In a RIoT system, boot progresses in stages. An initial immutable loader program, referred to as L_0 or layer zero (executing, say, in CPU-based ROM), loads and launches a second stage loader L_1. This layer, in turn, loads and launches the next stage loader L_2, and so on until the operating system (OS) and applications are running. This type of progression is illustrated in Figure 1.

RIoT system security is based on a secret value called the Device Secret that is set during manufacture (or later, provided it is set under conditions of physical security). In a typical RIoT implementation, a Device Secret will only ever exist within the device on which it was provisioned. However, in this pedagogical introduction, it will be assumed that the manufacturer maintains a database of Device Secrets.

The Device Secret is accessible to the first stage ROM-based boot loader, L_0, at boot time. RIoT-capable systems provide a hardware mechanism that the ROM-based loader uses to render the Device Secret inaccessible until the next boot cycle.

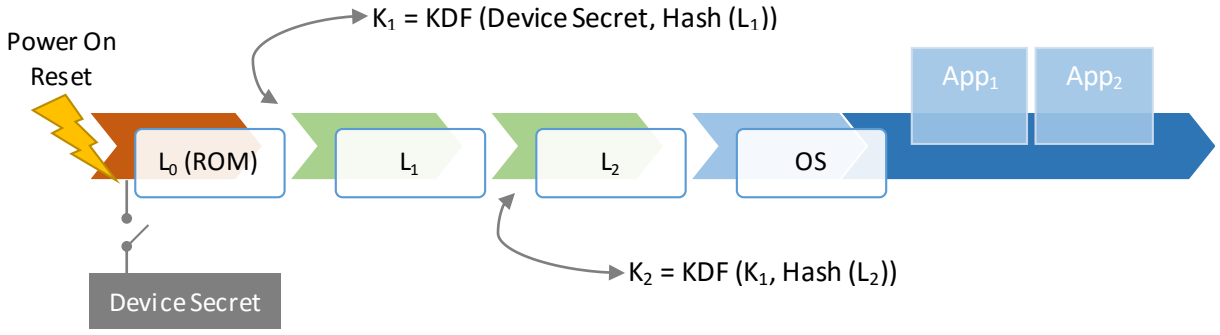

Figure 1: Simple RIoT-style key derivation for layered boot

The ROM-based loader L_0 could naively provide the Device Secret to L_1 and so on up the boot chain. Software could then use the secret value to encrypt confidential data or to authenticate the device to

the manufacturer. The problem with this simple scheme is that if any software with access to the Device Secret is ever exploited and the Device Secret leaks, all device security is lost and there may be no way to securely re-provision the device.

To avoid this weakness, the L_0 code in a RIoT-compatible system never reveals the actual Device Secret. Instead, it provides a *derived* key to the next program in the boot chain. Then, if L_1 is compromised, the *derived* key may be compromised but the Device Secret remains secure. Of course, a compromised L_1 might try to obtain the Device Secret on its own. To thwart this, RIoT-capable systems must provide a hardware-mechanism to ensure that only the L_0 boot loader can ever access the Device Secret.

There are many possible choices for key derivation functions. The simplest construction that has the necessary security properties is a function that depends on the hash of the next program in the boot chain. For example:

$$K_1 = KDF\left(Device\ Secret, Hash\ (L_1)\right) \quad (1)$$

Where, $KDF\ (K,S)$ is a cryptographic one-way key derivation function, e.g., an *HMAC* function. The key produced, K_1, is also referred to as the *Fuse Derived Secret*, or FDS.

This operation is illustrated in Figure 2. The "keyed diode" circuit element represents the one-way function, and the switch illustrates one mechanism for blocking access to the Device Secret during boot.

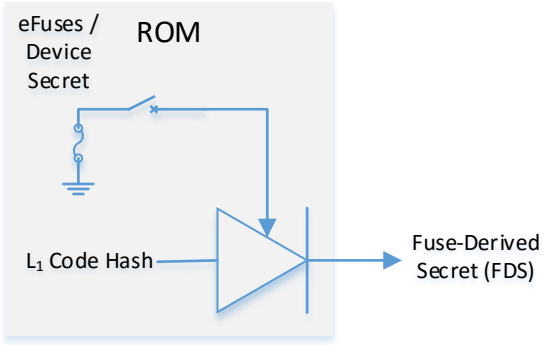

Figure 2: Schematic representation of the behavior of RIoT ROM code

This construction has two interesting characteristics. First, the derived key K_1 depends on both the Device Secret *and* the cryptographic identity of L_1. Since the Device Secret uniquely identifies the device hardware, the program L_1 can use K_1 in a proof-of-possession protocol to prove to the manufacturer both the identity of the platform *and* the cryptographic identity of the code that it is running. In other words, K_1 can be used for both device and software attestation.

Second, consider an authorized update from a vulnerable version $L_{A,1}$ to $L_{B,1}$ as illustrated in Figure 3. Layer $L_{A,1}$ receives the key $K_{A,1}$. If $L_{A,1}$ is compromised, then the manufacturer can push out a software update, $L_{B,1}$, that will receive a different derived key, $K_{B,1}$, that cannot be deduced from the leaked key (the KDF is a one-way function). Now, the updated $L_{B,1}$ can use this new derived key to attest to the manufacturer that the device is running the new and bug-fixed program.

Practically, this means that pushing out a software patch also securely re-keys the device.

A layer's derived keys can also be used to secure private data.[1] L_1 can use K_1 to encrypt and integrity protect data that is stored locally - a security primitive that is referred to as *sealing*. The simple KDF-based key derivation scheme provides excellent protection against malware attempting to obtain decryption keys and decrypt the data. For instance, if an attacker were able to replace L_1 in an attempt to receive or extract K_1, the attacker would obtain a different key and would not be able to decrypt previously stored data. This is because the derived key is based on the identity of the component receiving the key. This is also illustrated in in Figure 3.

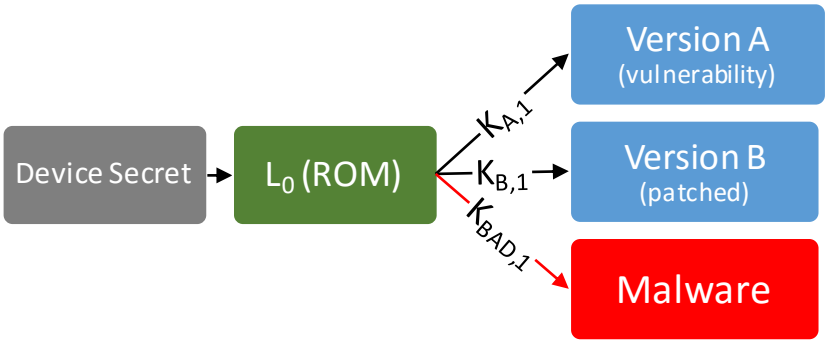

Figure 3: A manufacturer authorized update of a module with a vulnerability and malware

This simple key derivation scheme has one notable side effect. When a manufacturer performs an *authorized* update, the updated program will also obtain a different key ($K_{B,1}$ rather than $K_{A,1}$ in Figure 3). This means that the new component will not be able to decrypt data that was previously stored. There are a few ways to address this issue. The simplest solution involves grouping programs that should be treated equivalently, for example, by means of digital signatures. Another solution, which involves deriving data migration keys, is described in section 6.

The handoff of derived keys from the initial boot loader L_0 to L_1 can be repeated at each stage of boot, where the "layers" may include a sequence of boot loaders, an operating system, or even an application.

For instance, L_1 can use the same KDF-based scheme to provide a second-stage derived key, K_2, that relies on K_1, as well as the identity of the program, L_2, that L_1 will load, measure, and to which control is transferred. This is illustrated in Figure 4.

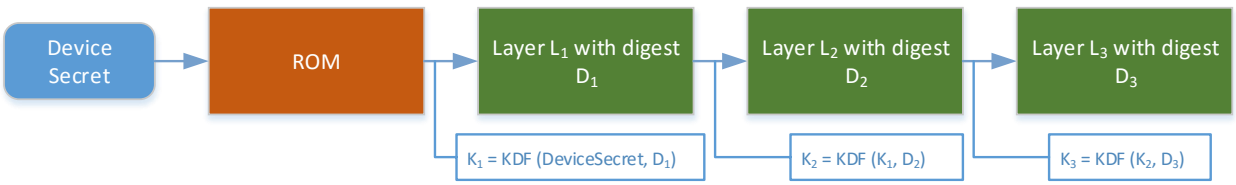

Figure 4: Example RIoT key derivation for multi-stage boot

[1] It is usually advisable to derive separate keys for the purposes of local encryption and device authentication. This is covered more fully in section 6.

The only essential difference between L_0 and the behavior of other layers is that L_0 must use a hardware facility to hide the Device Secret before passing control to L_1. Since subsequent layers will usually obtain derived keys in RAM or registers, "hiding" keys prior to transferring control to subsequent layers is simply a matter of deleting them from memory.

In this layered architecture, the KDF-derived keys at stage *n* of boot depend on both the Device Secret and *all* of the code that is part of the foundational TCB of level *n*. As a result, if the n^{th}-stage derived key is used in an attestation protocol, the manufacturer can determine exactly what code is part of the TCB of level *n*. Additionally, if the n^{th}-stage derived key is used for sealing, then protected data will only be accessible if the TCB does not change.

4 RIoT Hardware and Firmware

This section expands on the basic hardware and firmware requirements of a RIoT system.

Startup of modern processors is complex, with substantial variation in behavior across devices and vendors. RIoT-capable devices must satisfy the following set of minimal requirements.

4.1 Reliable Reset

An essential feature for building secure systems is a reliable processor and system reset. The reset process must clear all processor, device, and volatile memory state that could adversely affect future execution. The device must then restart execution in a well-defined manner. Power-on will typically initiate a processor reset after necessary hardware subsystems have been initialized.

4.2 Device Secret

RIoT-enabled devices require a device-specific secret. This statistically unique secret may be generated externally and installed during manufacture, or may be generated internally during device provisioning (see Section 6).

The Device Secret must be stored in non-volatile write-once memory on the device, e.g. eFuses, or any other suitably protected NV storage subsystem. Its size will depend on the algorithms used as well as the desired security strength of these functions. A 256-bit Device Secret should be considered the minimum at this time.

4.3 Protection of the Device Secret

The Device Secret must be available to early-stage bootstrap code, but the processor must also provide a mechanism to ensure that the Device Secret is inaccessible to later code. There are a number of ways this may be accomplished. For example, the processor or memory subsystem may include a latch controlling access to the Device Secret that is only opened during processor reset, but can be programmatically closed after the ROM loader has read the value.

4.4 Immutable Measurement Code

RIoT-capable processors must be manufactured with immutable code that can read the Device Secret and produce a derived key or keys that depends on the cryptographic identity of the next program in the boot chain.

In principle, a processor could be designed that performs complex functions like certificate verification to establish software identity. In practice it is better to build processors with immutable/ROM code that performs extremely simple cryptographic operations, because simple code is more likely to be bug-free.

In Section 6 it is demonstrated that a single derived value that depends on the Device Secret and the digest of the next program in the boot chain can be used to bootstrap sophisticated key-management schemes. Therefore, in the remainder of this paper it is assumed that the processor provides a Fuse Derived Secret using Equation 1.

Of course, it is essential that the immutable code hide the Device Secret (and delete any copies in RAM, registers, cache, etc.) before passing control to the next program in the boot chain.

5 Fundamental Characteristics of RIoT Devices

In this section some of the fundamental characteristics of the RIoT security architecture are explored.

5.1 Resilience

One of the main responsibilities of the boot measurement code (in ROM) is to protect the Device Secret. On the other hand, adversaries will seek to subvert operation of ROM code to leak the Device Secret to unauthorized entities. If an attacker is not launching a physical attack, then the primary means of attacking computer systems is to provide inputs that were not foreseen by the program designer and cause the software system to malfunction, e.g., through a buffer overrun or other logical error. The preferred implementation of boot measurement code is chosen to be extremely simple; it just computes the hash of a memory region. Because of its simplicity, it is expected that code implementing this behavior can be free from exploitable bugs.

Code that performs more complex functions will naturally have a greater attack surface. Where possible these complex functions should be performed later in the boot sequence where system recovery, in the case of an exploit, can be accomplished with the help of earlier layers.

RIoT security depends on software maintaining the security of cryptographic keys. If software vulnerabilities allow keys to be extracted, then these keys can be used to impersonate the device or claim that the device is running a different software stack. In this situation the recourse is to patch the device so that new derived keys are produced, and then revoke trust in the previous configuration.

A primary consideration of the RIoT system architect is to design and build extremely robust foundational layers that can be used to securely re-provision the more complicated and bug-prone upper-layers.

5.2 Protecting Secrets and Keys

RIoT device resiliency is due, in part, to the approach the RIoT framework takes to securing secrets and keys. A typical approach to maintaining the secrecy of a cryptographic key is to store and operate on the key within an isolation envelope provided by a combination of hardware and software, for example, a separate security processor or dedicated processor mode. However, these keys are then available for use at any time. This is referred to as *spatial* protection.

RIoT is different in that it also relies on *temporal* protection for keys and secrets. RIoT key security is based on receiving keys (or other secrets) from an earlier component in the boot chain, operating on

those keys, and then deleting them from memory before running the next boot program. This means that crypto operations using these received keys can only be performed for a limited time. That is, during boot and within the security boundary in which those keys were available.

These differences are illustrated in Figure 5.

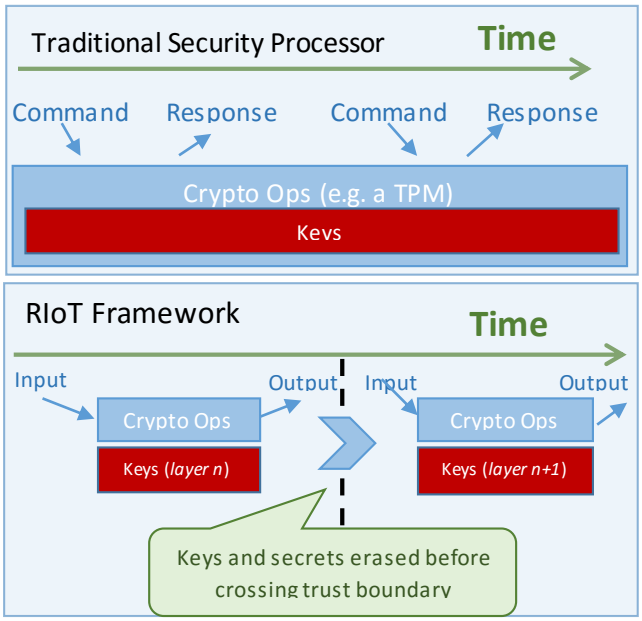

Figure 5: RIoT vs traditional security processors

Note that a program is not required to delete its RIoT-generated keys as long as it is able to prevent access to those keys. An early boot component usually deletes its keys/secrets because it is passing full control of the system to another, usually more complex, program.

A program executing at a higher layer may choose to retain its RIoT keys rather than delete them. This may be because the program is a top-level application that needs to use the key to encrypt data or authenticate itself, or it may be because this layer can adequately protect this key so that even higher layers cannot access the key. For example, a firmware TPM running in ARM TrustZone might retain a RIoT key within Secure World so that the TPM can use it to protect its important keys and state. Upper level software can (directly or indirectly) *use* the RIoT key, but cannot access it directly, just like the Traditional Security Processor in Figure 5. This further illustrates how RIoT technology can be used as the foundation for more traditional security processors.

5.3 RIoT as a General Purpose Security Processor

As illustrated in Figure 5, each layer in a RIoT boot-sequence performs some or all of these operations:

- Receives a key or keys from earlier layers
- Reads inputs from other sources (local storage, local interactive users, the network)
- Measures and/or validates the identity of subsequent layers
- Performs crypto operations on keys, layer-identities, and other inputs
- Deletes layer-private keys from memory and registers
- Transfers control to the next layer

Many cryptographic operations can be adapted for use in the RIoT framework. For example, new keys, including asymmetric keys, can be created through key derivation functions. Symmetric and asymmetric keys can be used to sign and decrypt data. Keys can be used to certify other keys or other aspects of platform state. In addition to secret key operations, RIoT layers can validate certificates and generate new keys.

Much of the power of RIoT arises because system designers can directly implement precise requirements without having to work within the architectural confines of a pre-defined hardware security processor. Additionally, complex functions (with correspondingly greater risk of compromise) can be implemented in upper layers so that security functions in lower layers can recover them if needed.

The fact that some RIoT keys are only available during boot, however, will generally demand adjustments in the protocols and the way that keys are used.

6 RIoT Operations

In Section 3, simple key derivation schemes enabling device identity, attestation, and sealing were illustrated. Some drawbacks of the simple schemes were also noted. For instance, relying on a database of Device Keys for attestation, or the fact that software updates can render previously sealed data inaccessible. In this section a more complete and practical RIoT implementation is described.

For simplicity it is assumed that all RIoT functionality is implemented in a single layer: L_1 or, *the RIoT Core*. This must be the first component measured and started by the processor's boot-loader. The remaining device firmware, implementing the main device functionality, is called the *Application Firmware* and runs at layer 2.

This simplification, the assumption of a single-layer RIoT Core, does introduce a complication, however. That is, a single-layer RIoT Core would not be updateable without loss of device identity. Exactly why will be described in Section 6.2. In reality, more sophisticated RIoT implementations have only a small fraction of non-serviceable code.

It is assumed, however, that the remaining Application Firmware *will* need to be updated to add functionality and fix potential issues. Secure management of the upgrade process is the main subject of this section. The components and data flows discussed in this section are illustrated schematically in Figure 6.

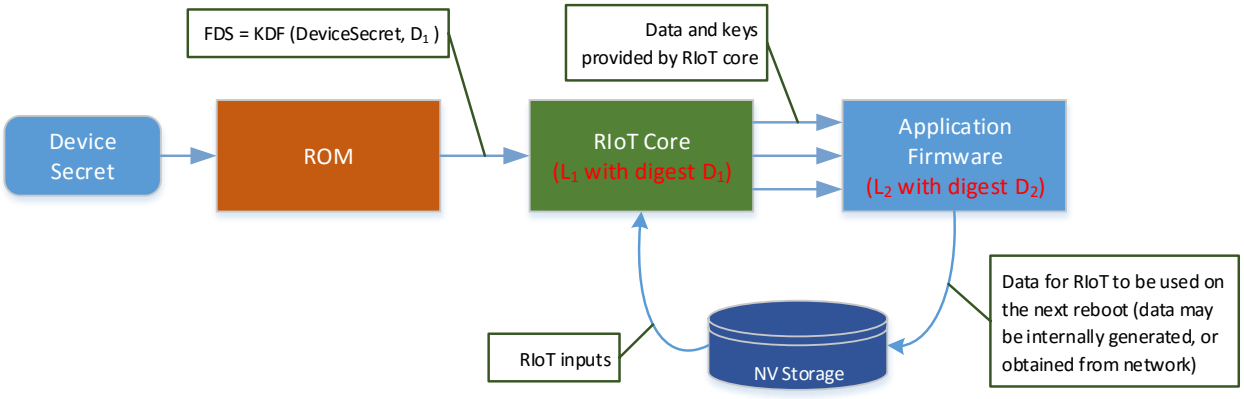

Figure 6: Firmware modules and data flows

6.1 Provisioning

In the RIoT architecture, the simplest provisioning scheme requires the device manufacturer to keep a database of the Device Secrets that were installed on its devices. However, this secret key database represents a single point of attack. Further, even if there is no irreversible loss of device security and identity, this scheme still does not allow the device to keep secrets from the device manufacturer itself.

A favored alternative is to furnish processors with a Device Secret that is *not* known outside the device. Ideally the Device Secret is internally generated early in the life of the processor using a good internal entropy source. If external tools are used to generate and install the secret during device manufacture, then the vendor should not keep a record of Device Secrets.

Since the Device Secret is not known externally, the simple attestation schemes in Section 3 cannot be used to identify the RIoT Core code in layer 1. Instead, the RIoT Core must be installed under conditions of physical security during manufacture.

6.2 Using the Fuse Derived Secret

The Fuse Derived Secret (FDS) is the key provided to the initial boot code by the processor ROM. The FDS will typically be used as the foundation for device identity, attestation, and data protection. Cryptographic best-practice demands that the same key is not used for multiple purposes. This is most easily achieved by deriving additional keys with a cryptographically secure key derivation function.

For example, device identity keys and operations can be based on K_{ID}, where:

$$K_{ID} = KDF\ (\ FDS, \text{"identity"}\ )$$

Other keys can be created similarly. The "purpose" field is included, both explicitly and implicitly, in several of the RIoT primitives described in this section.

The FDS itself is derived from the Device Secret and the software identity of the layer ROM will transfer control to, L_1 (see Figure 6). This is why a single-layer RIoT Core would be non-serviceable. Updating RIoT Core as a single layer would cause the FDS to change. A change in the FDS value would result in completely different keys in each derivation chain, including device identity.

The solution to this is a multi-layered RIoT implementation with a very simple first layer that provides a stable identity and support for update of subsequent layers.

6.3 Device ID

RIoT Core can generate a public key device identity by creating an asymmetric key pair at boot time using a deterministic key-generation function based on K_{ID} (or directly on the Fuse Derived Secret). As long as the first stage RIoT firmware does not change, the device will always generate the same key pair, i.e., the Device ID will be stable for the life of the device. See Figure 7.

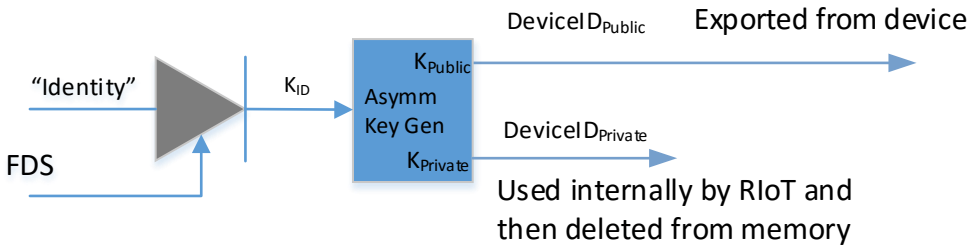

Figure 7: Deriving an asymmetric Device Identity from the Fuse Derived Secret

The Device ID public key can be written to persistent storage or passed to the Application Firmware, and will eventually be securely exported to the manufacturer or to any other party that requires a long-term stable identifier for the device. Manufacturers may also choose to certify the Device ID using a manufacturer PKI or keep a database of Device IDs of the devices that they have manufactured.

Proof-of-knowledge of the Device ID private key can be used as a building-block in a cryptographic protocol to identify the device, as in the following examples.

First, a server generates a challenge nonce that is signed by the RIoT Device ID key on the next reboot.

Second, a server creates a symmetric key that is encrypted by the Device ID public key that can be decrypted by the RIoT Core on the target device on the next reboot (this option is discussed more in the next section).

Third, the RIoT Core generates a new random asymmetric key pair, K_{L2}, for use by the Application Firmware. The RIoT Core can then certify the $K_{L2,Public}$ using the Device Identity key, and pass the new key pair as well as the Riot Core generated certificate that demonstrates that the key was generated securely. However, now $K_{L2,Private}$ can be used at any time rather than just during reboots. See Figure 8.

In practice, the same Device ID key should not be used for more than one purpose without explicit qualification. This can be accomplished by deriving keys specifically for signing and decryption, and by using distinguishing fields in the data structures that are being signed and decrypted.

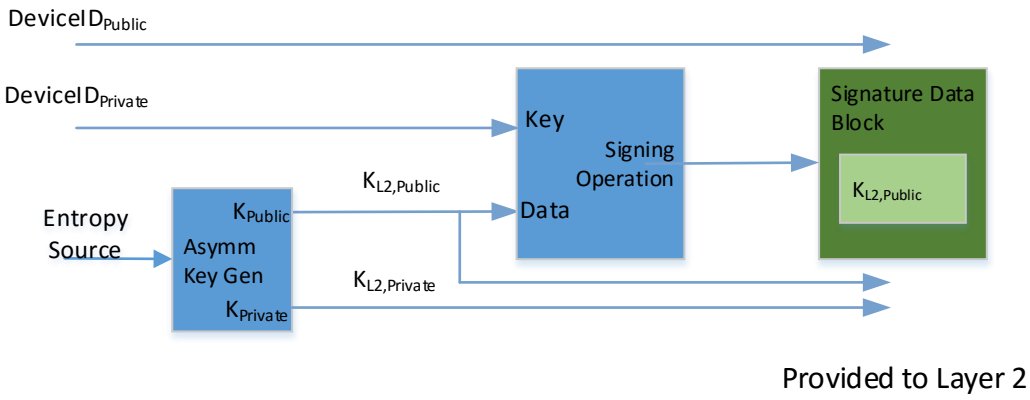

Figure 8: RIoT code creating a key pair for use by layer 2 that is certified by the Device ID

The Device ID private key must always be deleted from RAM and registers before RIoT passes control to the Application Firmware.

6.4 Attestation

The simple attestation scheme in Section 3 requires knowledge of the Device Secret to validate the proof-of-possession of the layer-identity based derived keys. In this section several alternative schemes that use the Device ID as the foundation for attestation are described.

The first scheme uses the Device ID to indirectly attest the identity of the Application Firmware by creating an asymmetric key that is certified (by the Device ID key) to be associated with Application Firmware with a specific digest.

This is a variation on the Application Firmware identity key described in the previous section. However, rather than a simple certificate associating $K_{L2,Public}$ with the Device ID, the certificate now includes the identity of the software that has access to $K_{L2,Private}$. See Figure 9, and the more detailed discussion by Lampson, et. al. [2] .

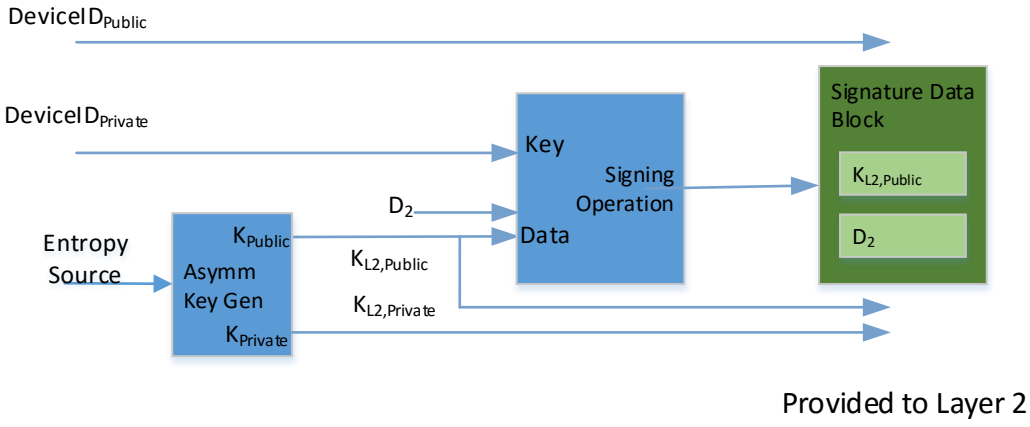

Figure 9: Creation of asymmetric key for layer 2 that is certified by the Device ID as being associated with firmware identity D_2

An alternative scheme, which builds on the symmetric key style attestation described in Section 3, is for the manufacturer to encrypt a symmetric attestation seed value (the *Attestation Seed*) with the public

Device ID. The Attestation Seed can be decrypted at boot time by the RIoT Core, and can then be used for KDF-type key derivation exactly as described in Section 3. I.e.,

$$K_{ATTEST,L2} = KDF\ (\ Attestation\ Seed, D_2)$$

In this simple two-layer firmware example, the Application Firmware can use $K_{ATTEST,L2}$ in a proof of-possession protocol to prove to the manufacturer (or any other entity providing an Attestation Seed value) that a particular device is running the expected firmware. If the system has additional layers, then the key derivation procedure can be repeated at each phase of boot.

Figure 10 illustrates this operation, together with an option to "seal" the attestation seed value (rather than simply encrypting it) so that the seed is only accessible to specified Application Firmware.

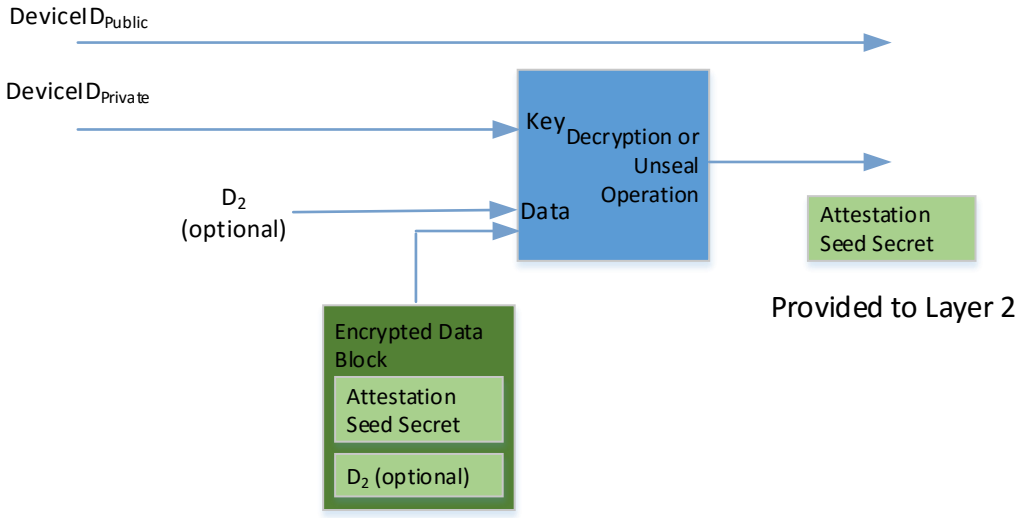

Figure 10: Secure import of an Attestation Seed value from the vendor or other party. RIoT can perform a simple public key decryption operation, or a decryption operation followed by a check that the Application Layer is authorized to receive the encrypted secret.

A third technique is a variation on this scheme, but rather than simply decrypting the Attestation Seed value, the RIoT Core interprets the encrypted data as a tuple consisting of an attestation secret together with the identity of the Application Firmware authorized to receive it (the optional D_2 value in Figure 10). Now, the RIoT Core decrypts the tuple, but only reveals the Attestation Seed value to the layer above if it has the expected hash. In this case the additional key morphing described earlier in this section is not needed. This operation closely resembles how TPMs implement unsealing.

6.5 Sealing and Data Protection

The RIoT Core can a create a variety of keys with different security properties for Application Firmware to use for data encryption and integrity protection. For example:

$$K_{SEALING,ALL} = KDF(\ FDS,"sealing")$$

Creates a key that is unique to the device, but the same for any code that executes at layer 2. This makes updates easy (the data protection key does not change), but if an attacker can extract this key though a vulnerability in Application Firmware, then this key loses value.

The alternative key derivation scheme:

$$K_{SEALING,D} = KDF(FDS, "sealing" | D)$$

Creates a key that is specific to both the device and the exact identity, *D*, of the Application Firmware.[2] This is much more resilient to malware since, if malware boots on the device, it will obtain a different key because it has a different digest than the authorized software. Also, if $K_{SEALING,D}$ leaks through compromise of authorized software, then a device update results in a new key that cannot be inferred from the compromised key (as long as the FDS is not known).

The challenge in the latter case is that users and device vendors will normally want to allow newer firmware versions to access previously stored data. The next section discusses options for enabling controlled access to previously sealed data.

6.6 Sealing and Authorized Updates

Consideration of updates is complicated because the device vendor, owner, and perhaps a user with physical presence, may need to be involved in the update decision. Additionally, there are a number of choices that trade off security with manageability. For example, some vendors may choose to allow devices to be "downgraded" to an earlier, possibly vulnerable, version of Application Firmware and other vendors may not.

This paper refers to the source of updates and accompanying certificates as the vendor. RIoT itself does not impose control over what entities are authorized to perform updates; the same scheme works if updates come directly from the device vendor or if they are created (or re-certified) by the owner of the device.

6.6.1 Using Certificates to Authorize Updates

Existing devices often use public key cryptography to authorize updates. Each firmware version is accompanied by a vendor-provided certificate containing (typically) a firmware version number and the digest of the corresponding firmware. Newer versions are usually downloaded by earlier versions, but the final decision on whether a new version is authorized to run on the device is handled in one of two ways:

1) The current full stack (possibly with bugs) makes a policy decision based on the new version certificate, or
2) The new version is staged in temporary non-volatile storage and the device is rebooted. The (typically much smaller and simpler) loader for the full device firmware makes the final policy decision whether the new version is authorized to run.

Similar strategies can be adopted for RIoT devices. However, in addition to deciding whether to install and run a new version, the RIoT Core will also check whether the new version is authorized to access previously sealed keys and data by validating certificates.

Within this basic framework there are still a number of implementation options. A conceptually simple model is for the RIoT Core to derive a key that is the same for all Application Firmware with certificates signed by the same key, e.g., $K_{PUB,VENDOR}$.

[2] The '|' operator indicates concatenation of some suitable representation of the parameters.

For example:

$$K_{FAMILY} = KDF\ (\ FDS,"family"\ |\ "all"\ |\ K_{PUB,VENDOR})$$

K_{FAMILY} is available to all modules that have been signed by the vendor. This "family" key can be used to encrypt data directly, or can be used by Application Firmware to transfer a subset of data between firmware versions.

A straightforward extension is to create family keys that are only available to a subset of family members. For example:

$$K_{FAMILY,N} = KDF(\ FDS,"\text{family}"\ |n|\ K_{PUB,VENDOR})\quad where\ n = 0,1,2 \dots N$$

Here *N* is a version number extracted from the certificate, and the vendor increments the version number for Application Firmware releases. When the RIoT Core boots version *n* of the firmware, it creates all family keys up to and including *N* and passes them to the Application Firmware. If this software uses $K_{FAMILY,N}$ to encrypt data then no earlier version will be able to decrypt the data, but future versions will. This construction prevents rollback attacks.

6.6.2 Alternatives to RIoT Core Certificate Based Updates

The controlled update scheme in the previous section assumes certificate validation and parsing happens in the RIoT Core firmware. The RIoT framework also allows Application Firmware to make an autonomous decision to migrate all or a subset of its data to a new version, new device, a backup server, etc. It is outside the scope of RIoT to determine how these policy decisions are made by a vendor's firmware, but two RIoT building blocks that can enable Application Firmware-initiated secure update and data migration, both within and between machines, are described next.

Unsealing

$$Unseal(X):$$
$$Decrypt(X) \rightarrow (S, D)$$
$$if(D == D_2)\ return\ S\ else\ return\ error$$

To *Unseal*, Application Firmware running on the platform (or elsewhere) prepares a data structure - (S,D) - containing a secret value and the digest of the Application Firmware authorized to access the secret. An authenticated encryption scheme is then used to encrypt the tuple with the public Device ID of the target device, and the resulting encrypted blob is placed in non-volatile storage on the platform. On the next reboot, the RIoT Core attempts to decrypt the blob and checks that the code that is about to run has the required identity. If all is correct, then RIoT Core provides the secret to the authorized Application Firmware. Practically, Unseal allows programs to migrate any or all of its protected data to a new configuration.

Variations of this scheme also allow the identity of the (local) sealing program to be identified.

Migration

CreateMigrationKey (D_A, D_B):

$$If\ (\ (D == D_A)\ |\ (\ D == D_B))$$

$$K_{MIGRATE,D_A,D_B} = KDF\,(\,FDS,"migrate"|D_A|\,D_B)$$
$$return\ K_{MIGRATE,D_A,D_B}$$

$K_{MIGRATE}$ is a software identity dependent key that is the same for either of a pair of specified versions of Application Firmware. Once the hash of the new version is known (how this is determined is out of scope), version A creates the tuple *(D_A, D_B)* where D_A is the current version identity and D_B is the authorized future version.

The platform is then rebooted. RIoT Core provides $K_{MIGRATE,A,B}$ to version *A* (the source version). Version *A* checks that D_A is its current identity, and D_B is the authorized future state. If these checks succeed, then it uses $K_{MIGRATE}$ to encrypts all keys and state that need to be transferred to the new version. The system is then rebooted into the new version, and if the new software has hash D_B, the previous migration key is re-created and previously encrypted data can be recovered.

6.7 Secure Boot and RIoT

Most IoT devices will implement certificate-based secure boot to increase device resiliency against malware and other threats. RIoT capabilities supplement secure boot systems by adding device identity, attestation, and sealed storage. Secure boot and RIoT functionality can comfortably coexist; in fact, certificate parsing code and other cryptographic primitives can be shared between secure boot and RIoT subsystems.

Both secure boot and RIoT-based systems typically require the correct operation of earlier software versions to download newer versions and initiate updates. Certain classes of vulnerability can interfere with this step, leading to a bricked device. All devices should provide for remediation in the case of such catastrophic failures. Traditionally, remediation of a bricked device is accomplished through physical access, e.g., using an external port to download new firmware. If remote/autonomous recovery is required, then the device must include ROM-based (or very well protected) recovery firmware that executes if normal boot fails. The recovery firmware (and any additional programs downloaded by this firmware) will be treated exactly as any other program booting on a RIoT device. Notably, the recovery firmware will get a Fuse Derived Secret (FDS) that it can use for any purpose. If the recovery firmware restores an earlier backup of the main firmware and state, then all RIoT derived keys will be precisely restored on the next reboot.

Finally, it is noted that RIoT can improve the performance of secure boot systems. For example, systems are typically rebooted many times between updates. A RIoT-equipped system can validate a full code signature on the first boot after an update, and then create a local symmetric certificate, e.g., an HMAC, using a device-specific RIoT key. Subsequent boots can check the validity of the locally created certificate and only fall back to asymmetric certificate validation during the next update, or if the local cached certificate is lost.

7 Conclusions

RIoT provides fundamental device security services at close to zero materials cost. RIoT also allows an exceptionally small and simple foundational trusted computing base for computing devices and allows devices to be updated securely if any other code on the device is compromised. While RIoT is ideally

suited to the smaller and cheaper processors that power the Internet of Things, it is of equal value in all computing devices.

8 References

[1] D. Wooten and D. Grawrock, *Trusted Platform Module Library,* Trusted Computing Group, 2014.

[2] B. Lampson, M. Abadi, M. Burrows and E. Wobber, "Authentication in Distributed Systems: Theory and Practice," in *Proceedings of the 13th ACM Symposium on Operating*, Pacific Grove, CA, 1991.