embeddedworld2018
Exhibition&Conference
... it's a smarter world

Physical Unclonable Functions to the Rescue

A New Way to Establish Trust in Silicon

Geert-Jan Schrijen
Intrinsic ID
Eindhoven, The Netherlands
geert.jan.schrijen@intrinsic-id.com

Cesare Garlati
prpl Foundation
USA
cesare@prplFoundation.org

***Abstract* — As billions of devices connect to the Internet, security and trust become crucial. This paper proposes a new approach to provisioning a root of trust for every device, based on Physical Unclonable Functions (PUFs). PUFs rely on the unique differences of each silicon component introduced by minute and uncontrollable variations in the manufacturing process. These variations are virtually impossible to replicate. As such they provide an effective way to uniquely identify each device and to extract cryptographic keys used for strong device authentication. This paper describes cutting-edge real-world applications of SRAM PUF technology applied to a hardware security subsystem, as a mechanism to secure software on a microcontroller and as a basis for authenticating IoT devices to the cloud.**

Keywords — Security; Internet of Things; Physical Unclonable Function; Authentication

I. INTRODUCTION

The Internet of Things already connects billions of devices and this number is expected to grow into the tens of millions in the coming years [5]. To build a trustworthy Internet of Things, it is essential for these devices to have a secure and reliable method to connect to services in the cloud and to each other. A trustworthy authentication mechanism based on device-unique secret keys is needed such that devices can be uniquely identified and such that the source and authenticity of exchanged data can be verified.

In a world of billions of interconnected devices, trust implies more than sound cryptography and resilient transmission protocols: it extends to the device itself, including its hardware and software. The main electronic components within a device must have a well-protected security boundary where cryptographic algorithms can be executed in a secure manner, protected from physical tampering, network attacks or malicious application code [18]. In addition, the cryptographic keys at the basis of the security subsystem must be securely stored and accessible only by the security subsystem itself. The actual hardware and software of the security subsystem must be trusted and free of known vulnerabilities. This can be achieved by reducing the size of the code to minimize the statistical probability of errors, by properly testing and verifying its functionality, by making it unmodifiable for regular users and applications (e.g. part of secure boot or in ROM) but updateable upon proper authentication (to mitigate eventual vulnerabilities before they are exploited on a large scale). Ideally, an attestation mechanism is integrated with the authentication mechanism to assure code integrity at the moment of connecting to a cloud service [3].

However, we are not there yet. We also need to be able to trust the actual generation and provisioning of the cryptographic keys into the security subsystem. Without trust in the key generation and injection process we cannot assure that keys are sufficiently random and that every device in fact obtains a unique key, which is the basic assumption for secure device identification. In addition, the provisioning must guarantee that private keys are not known outside the device, cannot be extracted or cloned, and that public keys are unmodifiable without proper authentication.

A trustworthy Internet of Things requires a trust continuum from chip manufacturing through code development, device manufacturing, software and key provisioning, all the way to connecting to the actual cloud service. Central to the capability of a device to authenticate to the cloud is its digital identity, which is protected by the security subsystem. Devices that make up the Internet of Things use a broad variety of silicon components. It will therefore be a daunting challenge to roll out a universal security solution that works seamlessly for all possible microchip technologies in a consistent cost-effective way.

The further outline of this paper is as follows. Section II articulates the importance of device root keys as a basis for a digital device identity and authentication. Section III introduces SRAM-based PUF as an innovative, flexible and cost-effective way to bootstrap and secure such root keys in a universal way on the widest possible variety of microchip technologies. Finally, section IV highlights some relevant real-world applications.

II. DEVICE IDENTITY AND AUTHENTICATION

To securely authenticate a device that is connecting to a cloud service or for unmanned machine-to-machine connectivity, every single device must provide a strong cryptographic identity. Such identity typically consists of an asymmetric key pair, composed of a public key and a private key. The private key must be kept secret in the device and ideally should never leave the device security boundary. The public key on the other hand can be output and communicated

to external entities. According to the current PKI model, before the key pair can be used for device authentication a trusted entity needs to assert that the public key in fact belongs to a specific device (e.g. specific brand, model, serial number). This assertion is created in the form of a digital certificate. The trusted entity is typically the OEM that manufactures the device, although many variations in the supply chain setup are possible.

Devices are authenticated by sending their digital certificate, which includes the public key, to the verifying entity, e.g. the cloud service or another device. The verifying party checks the contents of the certificate and verifies by the known public key that it is correctly signed by a party it trusts. The device's public key that is in the certificate can then be used to verify the authenticity of the device by means of established authentication protocols. For example, a challenge-response protocol can be used in which the verifying party generates a random number and sends it to the device. The device generates a response value using its private key to compute a digital signature on the received challenge. The verifying party receives the response and verifies that the signature is correct using the device's public key. Alternative authentication schemes based on asymmetric keys are possible. For example, when the device sets up a secure HTTP connection to the cloud service using the TLS protocol, the client authentication check is done as part of the TLS handshake. This use case is described in section IV.C.

The asymmetric key pair that forms the device identity needs to be securely stored inside the security subsystem. This can be achieved via key wrapping, a process that involves encrypting the private key within the security boundary before storing it in non-volatile memory (NVM). The root key, used to encrypt the other secrets, must be device-unique and securely stored inside the security boundary: see the use case in section IV.A. Besides encrypting additional secrets for permanent storage, the root key can also be used to derive additional private/public key pairs directly via a cryptographic key derivation mechanism. Such keys can be used to authenticate and establish secure channels with multiple devices.

Provisioning root keys into a chip is an essential step in establishing a root of trust anchored in hardware. Traditional key storage methods require the root keys to be injected at an early stage in the production chain. This process implies that secret keys are handed over from device manufacturer to silicon manufacturer, and hence are revealed to different parties in the production chain. This creates undesired liabilities for both parties as the root keys are known outside the device's security boundary. In the IoT this problem is enormously amplified by the sheer number of devices. In this emerging scenario, distribution and potential leakage of root keys becomes the single most important problem [9].

To overcome these limitations, a flexible new key provisioning method is needed that enables secure programming of device root keys at any stage in the production process, allowing a device maker to reduce its dependency on other trusted parties. Physical Unclonable Functions (PUFs) based on SRAM memory are an ideal candidate for providing a universal cost-effective solution to this root key programming and storage problem.

III. PHYSICAL UNCLONABLE FUNCTIONS

Physical Unclonable Functions (PUFs) are known as electronic design components that derive device-unique silicon properties, or silicon fingerprints, from integrated circuits (ICs). The tiny and uncontrollable variations in feature dimensions and doping concentrations lead to a unique threshold voltage for each transistor on a chip. Since even the manufacturer cannot control these exact variations for a specific device, the physical properties are de facto unclonable. These minute variations do not influence the intended operation of the integrated circuit. However, they can be detected with specific on-chip circuitry to form a device-unique silicon fingerprint. The implementation of such measurement circuit is what is called a PUF circuit. There are several alternatives to implementing PUF circuits into an IC. They vary from comparing path delays and frequencies of free running oscillators to measuring startup data from memory components [10]. A particularly promising PUF technology is based on SRAM memory. The SRAM PUF has excellent stability over time, temperature and supply voltage variations and it provides the highest amounts of entropy. Furthermore, it is available as a standard component in almost every IC. The latter aspect has important advantages in terms of deployment, testability and time to market. SRAM PUFs can be used in standard chips by software access to uninitialized SRAM memory at an early stage of the boot process. Hence, it is not required to integrate special PUF circuitry into the hardware of the chip when using SRAM PUF technology.

A. SRAM PUF

SRAM PUFs are based on the power-up values of SRAM cells. Every SRAM cell consists of two cross-coupled inverters. In a typical SRAM cell design, the inverters are designed to be nominally identical. However, due to the minute process variations during manufacturing, the electrical properties of the cross-coupled inverters will be slightly out of balance. In particular, the threshold voltages of the transistors in the inverters will show some random variation. This minor mismatch gives each SRAM cell an inclination to power-up with either a logical 0 or a logical 1 on its output, which is determined by the stronger of the two inverters. Since this variation is random, on average 50% of the SRAM cells have 0 as their preferred startup state and 50% have 1. Note that SRAM memory is normally used by writing data values into the memory and reading back the written values at a later point in time. To use SRAM as a PUF, one simply reads out the memory contents of the SRAM before any data has been written into it.

One can evaluate the behavior of this SRAM PUF based on two main properties for PUFs: reliability and uniqueness. Over the past years thorough analysis of SRAM PUF data has been performed. Startup patterns have been measured under various conditions, from SRAM implemented in several technology nodes (180nm down to 14nm) by several foundries with different processes.

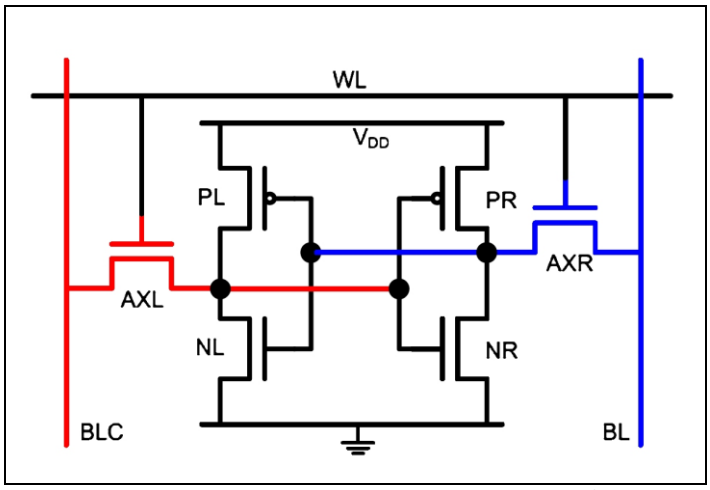

Fig. 1: A 6-transistor SRAM cell; two cross-coupled inverters are formed by left inverter consisting of PMOS transistor PL and NMOS transistor NL and right inverter consisting of PMOS transistor PR and NMOS transistor NR. Left and right access transistors are indicated as AXL and AXR respectively.

Extensive tests performed by leading PUF vendors and universities (e.g. in [10],[17]) have yielded the following results:

- **Reliability**: Most of the bit cells in an SRAM array have a strongly preferred startup value which remains static over time and under varying operational conditions. A minority of cells consist of inverters that are coincidentally well balanced and result in bit cells that will sometimes start up as a 0 and sometimes as a 1. This causes limited "noise" (or, deviation from the initial reference measurement) in consecutive SRAM startup measurements. Tests demonstrate that the noise level of the SRAM PUF under extensive environmental conditions (e.g. temperatures ranging from -55˚C to 125˚C) and over years of lifetime (see also [12]) is sufficiently low to extract cryptographic keys with overwhelming reliability when using appropriate post-processing techniques.
- **Uniqueness**: Extensive testing demonstrates that the startup pattern of an SRAM array is unique for every IC and even for a specific memory (region) within every IC. It is highly unpredictable from chip to chip and hence provides a large amount of entropy. The amount of entropy is sufficiently high to efficiently generate secure and unique cryptographic keys suitable to a broad range of applications.

B. Root Key Storage with PUFs

PUFs can be used to reconstruct a device-unique cryptographic root key on the fly, without storing secret data in non-volatile memory. Since PUF responses are noisy, they cannot be used directly as a cryptographic key. To remove the noise and to extract sufficient entropy, a so-called Fuzzy Extractor is needed. A Fuzzy Extractor or Helperdata Algorithm is a cryptographic primitive that turns PUF response data into a reliable cryptographic root key.

The Fuzzy Extractor (see Fig. 2) has two modes of operation: Enrollment and Key Reconstruction.

In Enrollment mode, which is typically executed once over the lifetime of the chip, the Fuzzy Extractor reads out an SRAM PUF response and computes that so-called Helperdata that is then stored in (non-volatile) memory accessible to the chip [11].

Whenever the cryptographic root key is needed by the chip, the Fuzzy Extractor is used in the Key Reconstruction mode. In this mode a new SRAM PUF response is read out and Helperdata is applied to correct the noise. A hash function is subsequently applied to reconstruct the cryptographic root key. In this way the same key can be reconstructed under varying external conditions such as temperature and supply voltage.

Important: by design the Helperdata does not contain any information on the cryptographic key itself and it can therefore be safely stored in any kind of unprotected Non-Volatile Memory (NVM) on- or off-chip. At rest, when the device is powered down, no secret is ever present in memory making traditional expensive anti-tamper requirements obsolete.

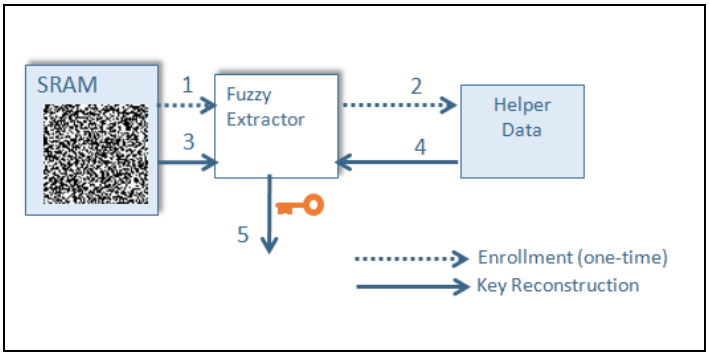

Fig. 2: A Fuzzy Extractor operates in two basic modes: i) In Enrollment mode (steps 1-2) Helperdata is generated based on a measured SRAM PUF response, ii) In the Key Reconstruction mode (steps 3-5) the Helperdata is combined with a fresh SRAM PUF response for reconstructing the device-unique cryptographic root key

C. Fuzzy Extractor implementations

A Fuzzy Extractor is typically implemented inside a chip in one of the following basic forms:

- **Hardware IP**: A hardware IP module that is connected to a dedicated SRAM memory. The Fuzzy Extractor hardware IP block directly controls the SRAM memory interface to read out the PUF values. The cryptographic key can be output via a dedicated interface to a cryptographic accelerator. The security advantages of such an implementation are discussed in the next subsection. Besides security advantages, a Fuzzy Extractor implemented in hardware is typically faster and more power efficient than the equivalent software implementation.
- **Software IP**: A software library that can access a dedicated portion of the overall SRAM memory. It is preferable that the SRAM portion used by the PUF algorithm is not shared with other software. Memory management units, silicon firewalls and trusted execution environments (TEEs) are

likely used if available. The Fuzzy Extractor does not contain any secrets, so it does not need to be encrypted. However, it is important to guarantee the integrity of the software itself. This can be achieved with a secure boot setup or by locking down the software on the chip with alternative mechanisms provided by the chip itself. Advantages of the software variant include flexible deployment options, i.e. retrofitting existing devices in the field and integration with other security components, with minimal or no hardware changes.

D. Security level provided by PUFs

Using the PUF to reconstruct a cryptographic root key has the following security advantages:

- Keys are reconstructed on the fly when needed and are present only temporarily within the security boundary of the chip. This greatly reduces the attack surface and time window for exploiting eventual vulnerabilities.
- When the chip is powered down, no physical traces of the key are present in the chip.
- Guaranteed randomness from the physics of the silicon results in full entropy keys.
- Root keys are generated within the security boundary of the chip rather than being injected from the outside, resulting in a safer and more flexible provisioning process throughout the supply chain.

It is important to observe that the Fuzzy Extractor must be implemented and integrated in a secure manner to minimize the exposure to various attack vectors including software vulnerabilities, side-channel and invasive attacks. Various countermeasures are possible and this is an area where established PUF vendors have developed considerable proprietary IP.

The actual security level achieved depends largely on the integration of the Fuzzy Extractor with the security subsystem. One of the design goals is to make sure that only the Fuzzy Extractor can access the SRAM PUF. In case of a hardware integration this is assured by connecting a dedicated SRAM memory directly to the Fuzzy Extractor and making sure that there are no software interfaces to it. To this end, it is for example preferable to use a Built-In Self Test instead of a scan chain [2]. In case of a software implementation one needs to make sure that access control settings of the chip are set up correctly. For example, this is done by using a memory management unit to reserve access to the SRAM PUF region of the memory dedicated to the Fuzzy Extractor software, by locking down the software image using firmware lock bits, by applying secure boot or by integrating into a TEE. Additionally, the in-circuit debug facilities need to be disabled.

Another design goal is to make sure that the cryptographic key that is output by the Fuzzy Extractor is transported securely to the cryptographic software that requires it. In case of a hardware implementation, this can be arranged by connecting the output bus of the Fuzzy Extractor hardware via a direct internal connection to a cryptographic coprocessor. In case of a software implementation, one needs to make sure that any registers used to store the key are cleared as soon as possible and cannot be accessed by untrusted processes. Similar measures as described in the previous paragraph can be taken to lock down the security boundary of the chip.

E. Known attacks to PUFs

Delay-based PUFs such as Arbiter PUFs and Ring-Oscillator PUFs promise a large space of independent challenge-response pairs that can be used for special authentication schemes [6][7]. In practice, however, it turns out that implementations of such PUFs are broken by modelling attacks, showing that responses are predictable given a limited subset of challenge-response pairs [15][16].

Memory-based PUFs such as the SRAM PUF are not susceptible to such attacks. The attacks that have been demonstrated on SRAM PUFs have been conducted only in non-realistic laboratory setups and do not form a threat to practical implementations. For example, with highly specialized equipment such as laser scanners it seems possible to read out SRAM memory contents by observing photo emissions during repeated read cycles [14]. This method is, however, feasible only in antiquated large technology nodes (e.g. 300 nm) and does not scale down to smaller modern technology nodes. In addition, the documented attacks require a situation where many consecutive SRAM read operations are executed sequentially on the same SRAM address range; a situation that does not occur in a good Fuzzy Extractor implementation. The work presented in [8] uses such a readout method in combination with a Focused Ion Beam to "clone" a PUF response from a first to a second SRAM memory. It should be noted that this is feasible only in obsolete large technology nodes (demonstrated on 600nm technology) and that it is only practical to clone a very limited number of bits with significant effort. In addition, commercial implementations include various proprietary countermeasures that make these kinds of attack simply infeasible. As of today there are no documented successful attacks of commercial-grade SRAM PUF implementations.

IV. Use Cases

This section offers some real-world examples of successful SRAM PUF applications.

A. Secure key vault

The SRAM PUF can be used to provide a cryptographic root key for a hardware security subsystem. The Fuzzy Extractor IP block is integrated with the security system IP. The chip-unique cryptographic root key that is reconstructed from the SRAM PUF feeds directly into the cryptographic module, for example an AES core. Fig. 3 shows a typical security subsystem architecture.

To initialize the system, the PUF must be enrolled: a first readout of the SRAM startup values is used by the Fuzzy Extractor to compute the Helperdata (steps 0 and 1 in Fig. 3). Once the Helperdata is stored in the chip's non-volatile memory (NVM), the enrollment step is completed.

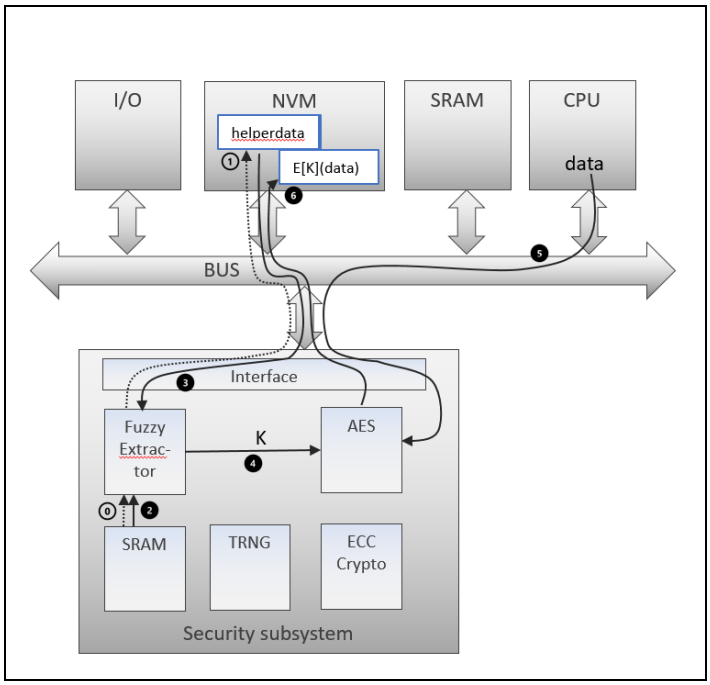

Fig. 3: Secure key vault based on SRAM PUF depicting Enrollment steps 0 and 1 (dotted lines); Key reconstruction steps 2,3,4, and Encryption of data generated on processor in steps 5 and 6.

The enrollment step establishes the device-unique cryptographic root key in the security subsystem. To reconstruct this key for use, the Helperdata is read from NVM and combined with a readout of the SRAM startup values in the Fuzzy Extractor (steps 2 and 3). The reconstructed key is fed into the AES core (step 4). Data that is being processed by the CPU can be securely stored by feeding it to the security subsystem, where it is encrypted using the AES module and stored in NVM (steps 5 and 6). Note that besides just encrypting the data, the AES core can also be used to protect the integrity of the data by computing additional authentication tags or by using an authenticated encryption mode such as AES-GCM.

When the processor requires the secure data, steps 2, 3 and 4 are repeated to reconstruct the cryptographic root key and load it into the AES core. Steps 6 and 5 are then reversed in direction to feed the encrypted data to the AES block, have the AES block decrypt it and feed it back to the CPU.

This mechanism makes it possible to keep secrets in otherwise unprotected non-volatile memory. Note that only encrypted data and non-sensitive Helperdata is ever stored in NVM. No secret is ever stored in permanent memory. The cryptographic root key that is reconstructed from the SRAM PUF is not known anywhere outside the security boundary. Therefore, the data that is securely stored in the chip's NVM can be decrypted only on the same chip on which it has been generated. Transferring them to any other target device is not a concern, even if the Helperdata is copied along with them. The Helperdata can be used only with the specific SRAM fingerprint of the chip that generated it in the first place.

B. Software protection in microcontroller

This section describes a use case where the SRAM PUF is used to protect software IP on a microcontroller. We assume the microcontroller has an internal flash memory where its program code can be stored. Before code is executed it is loaded into an internal SRAM memory. A small part of the SRAM memory is reserved to be used as PUF. This can be achieved by instructing the compiler to exclude a certain part of the SRAM from the memory map, assuring that it will not be "visible" by other software.

We furthermore assume that the microcontroller has some access control mechanisms to:

1. Lock down the software in the flash memory to prevent any modification
2. Disable in-circuit debug facility

Except for a few low-end microcontrollers, these access control mechanisms are quite common.

1) Setup phase

To securely set up the system, we use a provisioning PC in a trusted environment to load the code in the flash memory of the microcontroller. This is depicted in step 1 of Fig. 4. This is software that will be executed at runtime (see next section). The software consists of:

- A boot image containing the Fuzzy Extractor algorithm and the cryptographic cipher algorithms used to decrypt the software image
- A software image encrypted with key S. Initially the software has an empty header. At the end of the setup phase the header will be overwritten with a uniquely encrypted header per device.

After storing the software code in flash memory, the provisioning PC loads a temporary enrollment image in the executable SRAM of the device. This is depicted in step 2. The enrollment image contains the Fuzzy Extractor algorithm, as well as a cryptographic cipher that can be used to encrypt a header for the software image in flash. Furthermore, it contains the software image encryption key S.

When execution of the enrollment image is triggered (step 3), the SRAM PUF is read out (step 4) and Helperdata is created by the Fuzzy Extractor algorithm. The Helperdata is stored in the flash memory (step 5). Based on the Helperdata and the SRAM PUF readout, the cryptographic root key of the device K is reconstructed by the Fuzzy Extractor. Using the cryptographic cipher in the enrollment image, the software image encryption key S is encrypted with the device-unique key K. The resulting value, denoted as E[K](S), is written in the header of the encrypted software image (step 6). The flash memory now contains an encrypted software image, with a header that is specifically encrypted for the device it is stored on.

At the end of the setup phase the enrollment image is removed from SRAM. The provisioning PC triggers the necessary mechanisms in the microcontroller to lock the software images in flash and to disable the debug port.

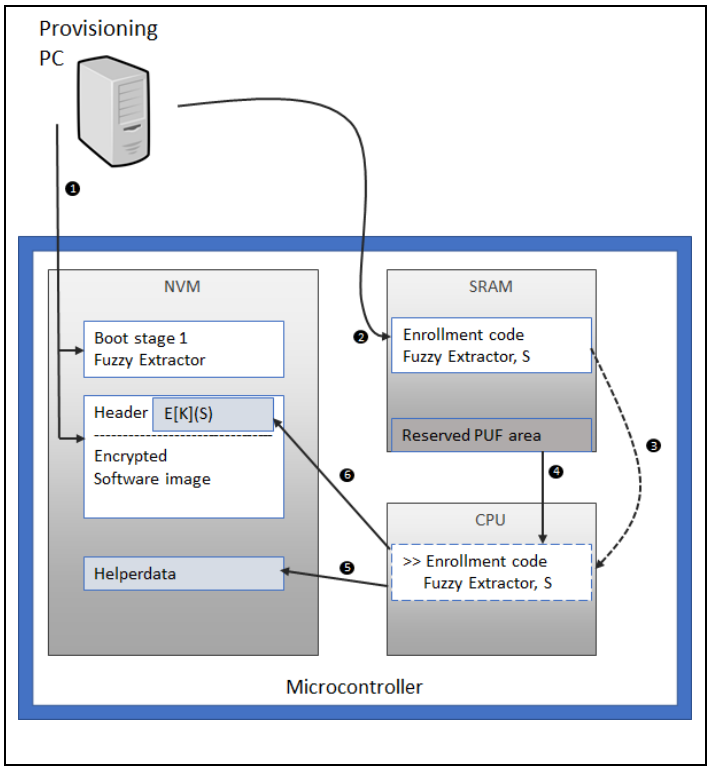

Fig. 4: SRAM PUF-based software protection mechanism, setup phase.

2) Runtime operation

The runtime flow is depicted in Fig. 5. First the microcontroller boot loader copies the first boot image into the SRAM of the microcontroller (step 1) and triggers execution (step 2). The boot stage code reads the SRAM PUF values (step 3) as well as the Helperdata (step 4). The Fuzzy Extractor algorithm in the boot image uses these values to reconstruct the device-unique root key K. The key K is used to decrypt the header of the software image (step 5). Decrypting the software image header results in the software image key S, which is then used to decrypt the software image in flash (step 6) as it is being copied to execution SRAM (step 7). When the full software image is decrypted and available in the SRAM, execution of the image is triggered (step 8).

The PUF plays an essential role in providing the microcontroller with a device-unique cryptographic root key that is used to bind the software image to the specific device. The root key is only temporarily reconstructed in working memory to decrypt the header of the software image. Likewise, the decrypted software image key is only temporarily present in working memory to decrypt the software image. When the device is powered off, the plain software disappears from the execution SRAM memory. Only encrypted values are left in the flash memory.

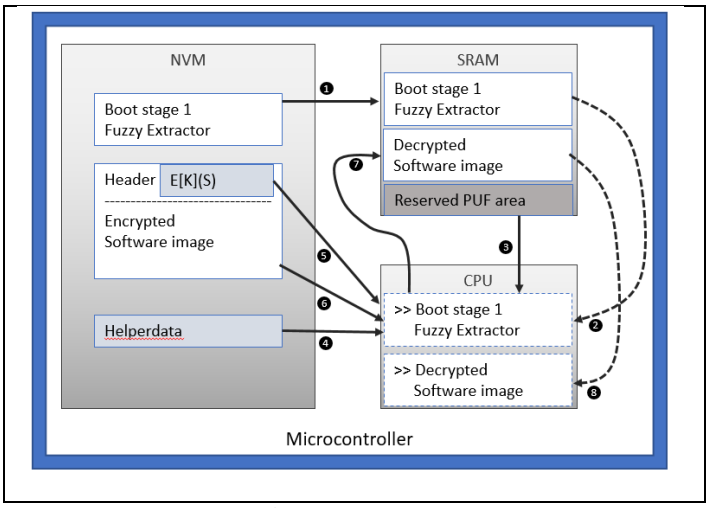

Fig. 5: SRAM PUF-based software protection mechanism, runtime operation.

The software protection method described in this section can be retrofitted to existing devices as it is completely software based. Still, the root of trust originates from the SRAM PUF in hardware. The core component that enables this mechanism is the Fuzzy Extractor that enables key reconstruction from a standard SRAM memory available in the microcontroller.

An open source reference implementation of such a Fuzzy Extractor is available as part of the prpl Security Framework, see [19].

C. Device authentication to the cloud

In this use case scenario, we describe how the SRAM PUF is used as a basis to connect IoT end nodes securely to a cloud service such as Amazon Web Services or Microsoft Azure cloud. We assume that the IoT device employs an off-the-shelf microcontroller as its main processing unit. An OEM (Original Equipment Manufacturer) owns both the devices and the service that is running in the cloud. The situation is depicted in Fig. 6.

1) Installation phase

In the installation phase (step 1) the OEM installs its IoT Service on the cloud platform of choice. The cloud service has its own private/public key pair denoted d_S/Q_S. This key pair is used to authenticate the service toward its clients. Furthermore, the cloud service knows the public key Q_{CA} of a trusted Certificate Authority. This public key is used to verify device identity certificates of the end nodes that connect to the cloud service.

The OEM also provides a software image to the Contract Manufacturer for installation on the IoT devices (step 2). Embedded in this software image is the URL of the cloud service, as well as the public key Q_S of the cloud service. This key is used to authenticate the OEM IoT service toward the device. The software image contains the following submodules:

- **Fuzzy Extractor**: The software library that reads out the uninitialized SRAM values from a reserved part of the

SRAM of the IoT device in order to reconstruct a device-unique cryptographic key K.

- **TLS & crypto library**: A software library that contains cryptographic functionality for securing a network connection using the Transport Layer Security protocol [20].
- **Connectivity library**: A network stack running on the IoT device, which enables the device to connect to Internet services. It will typically set up a TCP/IP stack over a physical network connection such as ethernet or Wi-Fi. Furthermore, it will support a connectivity protocol such as MQTT (Message Queuing Telemetry Transport) to run on top of the TCP/IP stack [21].
- **OEM Application**: The actual application software that provides the device with the intended functionality.

2) Setup phase

Every device will go through a setup phase in the production environment of the Contract Manufacturer, which operates on behalf of the OEM. As part of this enrollment and reconstructed on the fly only when needed. The public device key Q_D is sent via the contract manufacturer PC or Automated Test Equipment to the Certificate Authority service (step 6). The CA generates a device certificate, which includes the device public key Q_D as well as a signature created with the CA private key d_{CA}. Optionally the certificate may include other chip or device IDs. The device certificate, denoted as $\$[d_{CA}](Q_D)$, is stored in non-volatile memory on the device (step 7). After this step the device has an "identity" in the form of a public-key certificate

Note that this phase implements a one-time-trust event where the contract manufacturer assures that the device public key Q_D is valid for the specific device and triggers the generation of a certificate at the CA. The contract manufacturer is trusted for correctly requesting certificates for public keys of the devices. It does not have to be trusted to handle any sensitive private keys.

3) Runtime operation

Once the IoT device is in the field, it can now

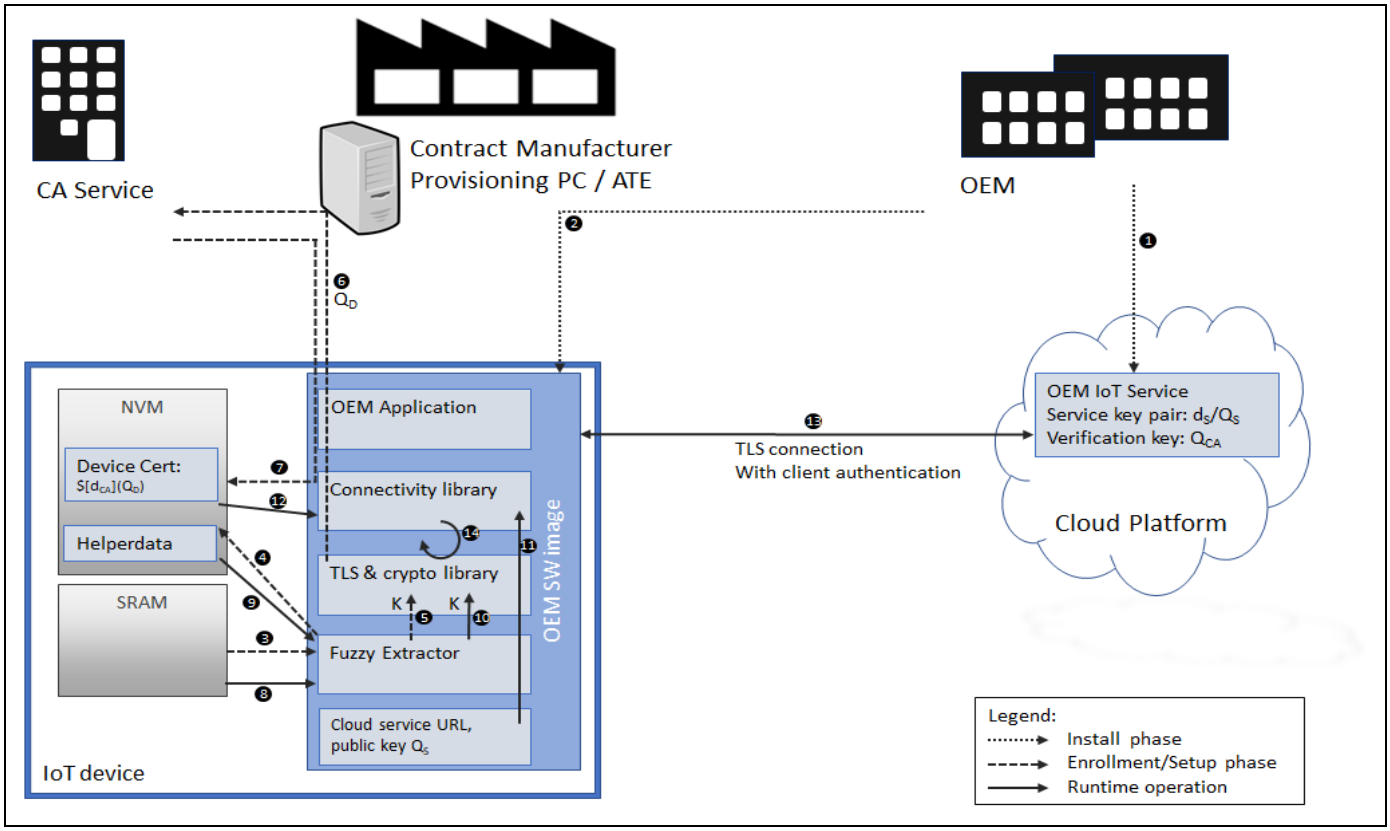

Fig. 6: Cloud authentication mechanism based on SRAM PUF.

step, the Fuzzy Extractor reads out the SRAM PUF values (step 3) and generates Helperdata (step 4), which is stored in non-volatile memory. The device-unique cryptographic key K is output by the Fuzzy Extractor and used with a Key Derivation Function in the TLS crypto library to derive an asymmetric elliptic curve device key pair d_D/Q_D. The private key of this key pair is never stored in any non-volatile memory autonomously set up secure connections to the OEM IoT Service. First, the Fuzzy Extractor is used to reconstruct the device-unique cryptographic key K from a readout of the SRAM PUF (step 8) and the Helperdata (step 9). The cryptographic key K is then used by the crypto library to derive the asymmetric key pair d_D/Q_D (step 10) and prepare for cryptographic support of the secure network connection.

The connectivity library contacts the Internet service via the URL that is fixed in the OEM software image (step 11). A TLS connection is then set up where the server is authenticated toward the device based on the public key Q_S that is stored in the OEM SW image (fetched via step 11). The Device Certificate (obtained via step 12) is used to authenticate the client IoT device toward the OEM IoT cloud service. Setting up the TLS connection (step 13) uses support from the crypto algorithms in the TLS layer (step 14) and on a high level proceeds as follows [20], see also Fig. 7:

a. Client and Server exchange initial messages where the client sends to the server a list of ciphers that it supports. The server compares this list with the ciphers that it supports and selects its preferred cipher that both sides support. In this case we assume that TLS_ECDHE_ECDSA is supported by the client and selected for setting up the secure connection. This cipher combination uses elliptic curve Diffie-Hellman key exchange to set up a shared session key, and the elliptic curve digital signature algorithm for authentication (i.e. message signing).

b. The server determines the elliptic curve parameters, including the elliptic curve base point P. The server randomly generates an ephemeral elliptic curve key pair d_{SR}/Q_{SR}, where $Q_{SR} = d_{SR} \cdot P$ and signs the ephemeral public key Q_{SR} with its private key d_S using the ECDSA signature algorithm. Note that the operator "·" denotes point multiplication over the elliptic curve. The signature value is denoted as $\$[d_S](Q_{SR})$.

c. Then the server sends the signed ephemeral public key $\$[d_S](Q_{SR})$ to the client, together with the elliptic curve parameters.

d. The client uses the server's public key Q_S to verify that Q_{SR} was signed correctly.

e. The client sends its public key certificate to the server. The server uses the CA public key Q_{CA} to verify the certificate and to be assured of the correct device's public key Q_D.

f. The client also randomly generates an ephemeral elliptic curve key pair d_{DR}/Q_{DR}, where $Q_{DR}=d_{DR} \cdot P$. The public ephemeral key Q_{DR} is sent back to the server.

g. The client uses its private key d_D to sign the TLS transcript (messages exchanged in steps a-f) and sends the signature to the server.

h. The server verifies the signature using the previously verified device public key Q_D.

i. The client computes a shared secret as $S = d_{DR} \cdot Q_{SR} = d_{DR} \cdot d_{SR} \cdot P$ over the elliptic curve group.

j. The server computes the same shared secret as $S = d_{SR} \cdot Q_{DR} = d_{SR} \cdot d_{DR} \cdot P$

Now that both client and server side have the same shared key S, symmetric session keys are derived from it to encrypt and authenticate further messages that are exchanged between both sides. Note that authentication of the client IoT device toward the server is done through steps e, g and h. The private device key d_D that is used for this authentication step is derived from the PUF key K. When the IoT device is powered off, no private keys are present. No sensitive data is ever stored in any NVM memory.

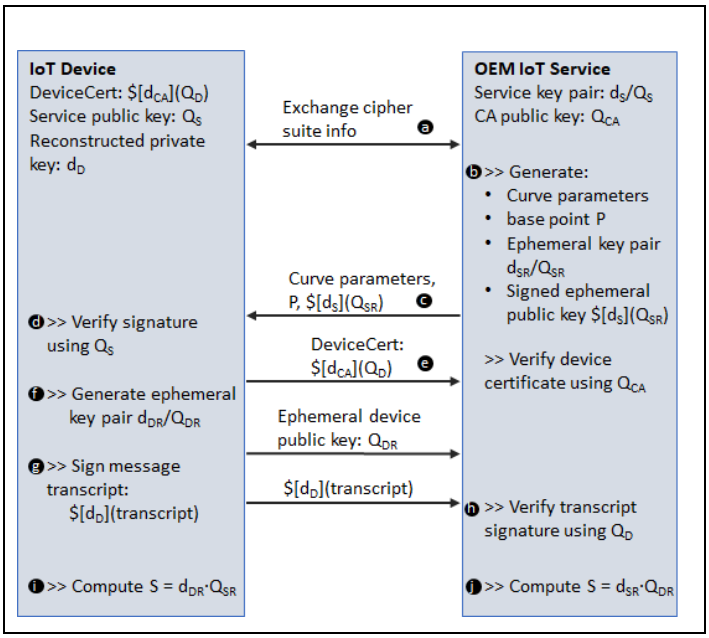

Fig. 7: Simplified overview of TLS key agreement steps based on ECDH protocol.

The SRAM PUF provides the flexibility to instantiate a device-unique key in the device and form the basis of a device identity (through the device certificate). No IDs or keys have to be injected by the silicon manufacturer. The OEM can decide to run the enrollment step at any semi-trusted time and place in the production chain. This has the advantage that the OEM can take device security in its own hands, without having to rely on key injection by the silicon manufacturer and secure handover of installed keys. This reduces key provisioning costs in the production chain considerably.

V. Conclusions

SRAM-based Physical Unclonable Functions form a universal method to securely store cryptographic keys in the chips of IoT devices. SRAM PUF provides hardware-rooted security that is enabled via software. When the device is powered down, no secrets are stored in memory, making cryptographic keys impossible to extract. In addition, SRAM PUF provides a high grade of flexibility all through the device supply chain. Every device can generate its own keys at any wanted point in the production chain. The entropy of these keys is determined by randomness in the physics originating from minute and uncontrollable process variations in the silicon production process. This makes PUF-based implementations much more resilient than traditional key injection options. The flexibility of the SRAM PUF process results in cost reductions as external key management infrastructure is kept to a minimum. SRAM PUF technology works reliably on any device that has silicon SRAM onboard: it will become the option of choice to establish trust in silicon for billions of devices that make the future Internet of Things.

REFERENCES

[1] M. Bhargava, C. Cakir, and K. Mai, "Comparison of bi-stable and delay-based Physical Unclonable Functions from measurements in 65nm bulk CMOS," in Custom Integrated Circuits Conference (CICC), 2012 IEEE, 2012, pp. 1–4.

[2] M. Cortez, G. Roelofs, S. Hamdioui, G. Di Natale, "Testing PUF-Based Secure Key Storage Circuits", DATE conference 2014, https://www.date-conference.com/files/proceedings/2014/pdffiles/07.7_2.pdf .

[3] Trusted Computing Group, Device Identity Composition Engine workgroup, https://trustedcomputinggroup.org/work-groups/dice-architectures/ .

[4] Y. Dodis, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," in Advances in Cryptology - EUROCRYPT 2004, ser. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, vol. 3027, pp. 523–540.

[5] Gartner newsroom, "Gartner Says 6.4 Billion Connected Things Will Be in Use in 2016, Up 30 Percent From 2015", https://www.gartner.com/newsroom/id/3165317 .

[6] B. Gassend, D. Clarke, M. van Dijk, S. Devadas, "Silicon physical random functions" In: ACM Conference on Computer and Communications Security (ACM CCS). pp. 148–160. ACM, New York, NY, USA (2002).

[7] B. Gassend, D. Clarke, M. van Dijk, S. Devadas, "Silicon physical random functions" In: ACM Conference on Computer and Communications Security (ACM CCS). pp. 148–160. ACM, New York, NY, USA (2002).

[8] C. Helfmeier, C. Boit, D. Nedospasov, and J.-P. Seifert, "Cloning physically unclonable functions," in Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on, 2013, pp. 1–6.

[9] Intrinsic ID whitepaper, "Flexible Key Provisioning with SRAM PUF", https://www.intrinsic-id.com/resources/white-papers/white-paper-flexible-key-provisioning-sram-puf/ .

[10] S. Katzenbeisser, U. Kocabaş, V. Rozic, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann, "PUFs: Myth, Fact or Busted? A Security Evaluation of Physically Unclonable Functions (PUFs) Cast in Silicon," in Cryptographic Hardware and Embedded Systems (CHES) 2012, ser. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, vol. 7428, pp. 283–301.

[11] J.-P. Linnartz and P. Tuyls, "New shielding functions to enhance privacy and prevent misuse of biometric templates," in Audio- and Video- Based Biometric Person Authentication, ser. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, vol. 2688, pp. 393–402.

[12] R. Maes, V. van der Leest, "Countering the effects of silicon ageing on SRAM PUFs", HOST 2014.

[13] D. Merli, F. Stumpf, G. Sigl, "Protecting PUF Error Correction by Codeword Masking", Cryptology ePrint Archive, https://eprint.iacr.org/2013/334.pdf .

[14] D. Nedospasov, J.-P. Seifert, C. Helfmeier, and C. Boit, "Invasive PUF analysis," in Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on, 2013, pp. 30–38.

[15] U. Rührmaier, J. Sölter, F. Sehnke, X. Xu, A. Mahmoud, V. Stoyanova, G. Dror, J. Schmidhuber, Wayne Burleson, S. Devadas "PUF Modeling Attacks on Simulated and Silicon Data", IACR Eprint archive 2013, http://sharps.org/wp-content/uploads/RUHRMAIR-IACR.pdf .

[16] U. Rührmaier, J. Sölter, "PUF Modeling Attacks: An Introduction and Overview", DATE 2014, https://pdfs.semanticscholar.org/a023/dd6069b664b0e53dfa5366d3c881a6876583.pdf .

[17] G.-J. Schrijen and V. van der Leest, "Comparative analysis of SRAM memories used as PUF primitives," in Design, Automation Test in Europe Conference Exhibition (DATE) 2012, March 2012, pp. 1319 – 1324.

[18] Synopsys whitepaper, "Securing the Internet of Things – An Architect's Guide to Securing IoT Devices Using Hardware Rooted Processor Security", https://hosteddocs.emediausa.com/arc_security_iot_wp.pdf .

[19] PRPL Foundation, security working group: https://prpl.works/category/prpl-security/ , PRPL PUF-API: https://github.com/prplfoundation/prpl-puf-api/tree/December-2017 , Security Framework application note: https://prpl.works/application-note-july-2016/ .

[20] Wikipedia, "Transport Layer Security", https://en.wikipedia.org/wiki/Transport_Layer_Security#Client-authenticated_TLS_handshake

[21] Wikipedia, "MQTT", https://en.wikipedia.org/wiki/MQTT .