



# INTRINSIC ID

## BK v2.9

This product is subject to EU export restrictions according to Council Regulation (EC) No. 428/2009, dual-use control category 5D002.

## Data Sheet

**Doc version** 1.4

**Status** Approved

**Reference** IID-BK2-9-DS

**For internal use by customer only**

**Confidential**

[www.intrinsic-id.com](http://www.intrinsic-id.com)



This product is subject to EU export restrictions according to Council Regulation (EC) No. 428/2009, dual-use control category 5D002.

This document contains information which is proprietary and confidential to Intrinsic ID B.V. and is intended for internal use only. The document is provided with the express understanding that the recipient will not divulge its content to other parties or otherwise misappropriate the information contained herein. Please destroy this document if you are not the intended recipient. Thank you.

Copyright in this document rests with Intrinsic ID B.V. Reproduction or publication in any medium of this document, in whole or in part, is expressly prohibited without the prior written permission of Intrinsic ID. Intrinsic ID reserves the right to make any changes to this document without prior notice. The contents of this document is provided AS-IS and without any warranties or guarantees as to accuracy or completeness. Receipt or possession of this document conveys no license under any patent or other intellectual property right of Intrinsic ID.

Intrinsic ID, QuiddiKey, QuiddiKey RNG, Apollo, BK, BK-Demo, Zign, Zign RNG, Zign Tag, Citadel, iRNG and other designated brands included herein are trademarks of Intrinsic ID B.V. All other trademarks are the property of their respective owners.



## History Information

Version	Date	Change Description	Modified by:	Reviewed by:
1.0	2020-05-01	First approved version	RM	
1.1	2020-07-27	Added NIST curves P-384 and P-521	BO	JW
1.2	2021-04-16	Latex version	GJS	NB,JG
1.3	2021-04-29	Update wording	GJS	
1.4	2022-04-19	Added new SKUs	NM	



# Table of Contents

History Information .....	3
Table of Contents .....	4
1. Introduction .....	5
1.1. Document Objective .....	5
1.2. Open Source Licenses used .....	5
1.2.1. Micro-ECC .....	5
1.3. Definitions, acronyms and abbreviations .....	6
1.4. References .....	6
1.5. Product Information .....	6
1.6. Product Use .....	8
2. BK Configurations and Function Sets .....	10
2.1. Configurations .....	10
2.2. BK Characteristics .....	11
2.2.1. PUF SRAM and Activation Code sizes .....	11
2.2.2. Length of Key Codes .....	11
2.2.2.1. Symmetric Key Codes .....	12
2.2.2.2. Asymmetric Key Codes .....	12
2.3. Ordering Information .....	13
2.4. Core Function Set .....	13
2.4.1. Product Information Function .....	13
2.4.2. State Management Functions .....	13
2.4.2.1. Device Power-up/Reset and Initializing BK .....	14
2.4.2.2. Enrolling BK with AC Output .....	14
2.4.2.3. Starting BK with Activation Code Input .....	15
2.4.2.4. Stopping BK .....	15
2.5. Randomness and Device-Key Generation Function Set .....	15
2.5.1. Unique Device Key Generation .....	16
2.5.1.1. Generating Device-Unique Symmetric Keys .....	16
2.5.1.2. Generating Device-Unique and Random Elliptic Curve Private Keys ...	16
2.5.2. Random Value Generation Function .....	17
2.6. Symmetric Key Crypto Function Set .....	17
2.6.1. Key Wrapping Functions .....	17
2.7. Elliptic Curve Crypto Function Set .....	18
2.7.1. ECC Key Management Functions .....	18
2.7.1.1. Managing ECC Private Keys as Private Key Codes .....	18
2.7.1.2. Managing ECC Public Keys as Public Key Codes .....	19
2.7.2. ECC Signing Functions .....	20
2.7.3. ECC Key Agreement and Encryption Functions .....	20
2.7.3.1. ECDH Key Agreement .....	20
2.7.3.2. ECDH-based Cryptogram Generation and Processing .....	21



2.7.4. CSR and Self-Signed Certificate Generation .....	22
2.7.5. ASN.1/DER ECC Key and Signature Read/Write.....	22
3. BK Module .....	24
3.1. Library.....	24
3.2. Operational States.....	24
3.3. Programming Interface .....	25
3.4. Profiling Information .....	25
4. Integration Guidelines.....	26
4.1. Integration Considerations.....	26
4.2. Reliability Optimizations .....	26
4.3. Power-up Recommendations .....	26



# 1. Introduction

## 1.1. Document Objective

This document explains the use of the BK embedded software (SW) library and is mainly intended for SW developers deploying BK in their application project. Targeted readers are expected to understand the basics of embedded SW development.

Section 1 explains the use of the embedded (SW) library. Subsequent sections present: the available configurations and functional sets of BK (Section 2), the main PUF operations and operational states of the BK module (Section 3), and guidelines for integrating BK into a product (Section 4).

This data sheet is valid for the following product version: **BK 2.9**, i.e. a call to the **bk\_get\_product\_info** function should return:

- product\_id = 0x42 (character 'B' indicating BK)
- major\_version = 0x2
- minor\_version = 0x9

## 1.2. Open Source Licenses used

### 1.2.1. Micro-ECC

This product contains code which is copyright 2014 Kenneth MacKay and licensed under the BSD license:

Copyright (c) 2014, Kenneth MacKay

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY



THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### 1.3. Definitions, acronyms and abbreviations

AC	Activation Code
API	Application Programming Interface
BK	Intrinsic ID's PUF-based embedded software IP
bk_	BK (as prefix in function/variable names)
CSR	Certificate Signing Request
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
ECDH	Elliptic Curve Diffie-Hellman
EEPROM	Electrically Erasable Programmable Read-Only Memory
HW	Hardware
IID_	Intrinsic ID (as prefix in return codes)
IP	Intellectual Property
k	x1,000
M	x1,000,000
MCU	Microcontroller Unit
MMU	Memory Management Unit
N.A.	Not Available
NIST	National Institute of Standards and Technology (US agency)
NVM	Non-Volatile Memory
OTP	One-Time Programmable
PKI	Public Key Infrastructure
PUF	Physical Unclonable Function
RNG	Random Number Generator
SRAM	Static Random Access Memory
SSC	Self Signed Certificate
SW	Software
Vdd	Supply Voltage

### 1.4. References

- [1] FIPS PUB 140-2 "Security Requirement for Cryptographic Modules", <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf>
- [2] Open SSL, <https://www.openssl.org/>

### 1.5. Product Information

The BK key generation and management software solution combines the benefits of SRAM PUF technology with the flexibility of software. It contains all the crypto functionalities to



upgrade the security in any 32-bit MCU/CPU e.g by building a key vault or provisioning the chip with an unclonable identity.

Depending on its configuration, BK comes with the following **functional features**:

- Generation, storage and reconstruction of device-unique keys (128, 192 and 256 bit symmetric keys and NIST P-192/224/256/384/521 elliptic-curve key pairs)
- Generation of random numbers seeded by device noise
- Secure key storage based on device-unique wrapping keys
- Importing of external elliptic curve key pairs as device-protected key code formats
- Exporting of elliptic curve public keys, e.g. for certificate generation
- Elliptic-curve signature generation and verification based on protected key codes
- Elliptic-curve key agreement function based on device-protected key codes
- Elliptic-curve-based secure cryptogram generation and processing based on device-protected key codes, for secure and authenticated messaging
- Compatibility with generic ASN.1/DER formats for elliptic-curve keys and signatures
- Generation of elliptic-curve-based certificate signing requests and self-signed certificates for integration with public-key infrastructures (PKI)

Its main **benefits** are:

- Protects your data with the electronic fingerprint of the chip:
  - no need to store secrets in NVM;
  - the calling application does not have to handle unprotected private keys
  - keys and secrets are not present in the system when powered off
- Cryptographically secure random number generator based on chip power-up noise
- SW-only solution<sup>1</sup> with HW security based on standard SRAM available on target
- Cost efficient; small footprint for the specified key strength
- Easy to use and easily scalable to billions of devices
- Overall security strength up to 256 bits for the root secrets of BK<sup>2</sup>
- Applies countermeasures against side-channel attacks

SRAM PUF responses have been qualified for use with BK over a wide operating range:

- Temperatures ranging from -55°C to +150°C
- Supply voltage variation of  $\pm 20\%$
- Qualified on semiconductor nodes ranging from 350nm to 7nm, including lowpower, highspeed and highdensity processes
- Lifetime >25 years, or lifetime of the microcontroller

<sup>1</sup>BK SW is being developed to meet FIPS 140-2 Appendix B “Recommended Software Development Practices”[1]. Please contact Intrinsic ID sales support for a compliance overview and list of deviations.

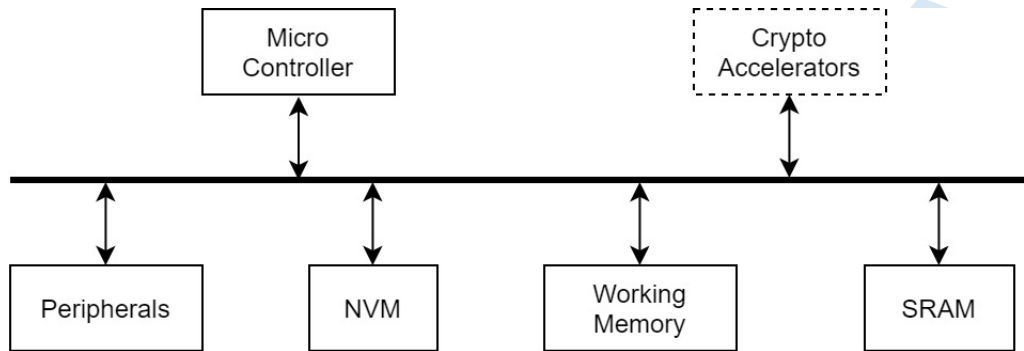
<sup>2</sup>The security strength of individual functions can be lower, depending on their cryptographic properties; e.g., the security strength of the elliptic curve cryptographic functions is limited to 128 bits, when the NIST P-256 curve is used.





## 1.6. Product Use

Figure 1 shows, very basically, the block diagram of a microcontroller unit (MCU). BK is an (embedded) software module which is loaded from memory (e.g. NVM) and executed by the 32-bit microcontroller. Typically, the BK library is integrated into the customer's SW project.



**Figure 1: Basic MCU block diagram**

BK enables secure key extraction from unique physical properties of the underlying hardware (in particular the embedded SRAM block), instead of storing keys in tamper resistant NVM (e.g. secure EEPROM) or even hard-wiring keys into the encryption core. This approach is based on the concept of PUFs; in this case an SRAM PUF is employed. BK generates an *Activation Code* (AC) which, in combination with SRAM start-up behavior, is used to reconstruct device-unique secrets for use by the system, without these secrets ever having to leave the device:

- The key storage functionality (both generation and protection) can be called via the commands of the BK SW API.
- The key generation commands provide access to the device key-generation functionality (both symmetric and elliptic curve). When a device key is no longer needed by the SW it can be removed from memory. When it is needed later it can be reconstructed again.
- The key protection commands provide access to the functionality for protecting the use of keys:
  - A user key protected by BK can only be retrieved on the same device and will be meaningless on other devices.
  - The BK elliptic-curve functionality is only called with device-protected keys. This avoids the calling application having to handle sensitive key material. The key values themselves are only present internally to BK.

The BK security IP is a stateful SW module, which means that its available functionality at any given time is dependent on the state it is in. In addition, compatibility between BK function calls at different points in time also depends on the AC that is used to put BK in its operational state.

BK relies on and/or integrates with the standard building blocks of an MCU system (Figure 1) in the following way:



- Microcontroller:
  - Loads the BK machine code and executes it.
- NVM:
  - Typically stores the BK machine code (e.g. in Flash, or ROM).
  - Typically used to store the generated AC (e.g. in Flash) in between generation and reconstruction.
  - Could be used to persistently store keys in the form of key codes that are protected with device-unique secrets.
  - For the storage of ACs and key codes, no security assumptions about the NVM need to be made, since they are only intelligible to BK running on the same device.
  - Optionally used for the storage of monotonic counters for particular cryptographic operations.
- SRAM/Working Memory:
  - BK requires dedicated access to a block of uninitialized embedded SRAM which is used as SRAM PUF. BK needs both read and write access to this block of SRAM.
  - SRAM (or other embedded memory) is also used as regular working memory in the conventional way.
  - The SRAM block used by BK should be inaccessible by other applications. Typically memory management capabilities such as MMUs or firewalls are leveraged to achieve this. The SRAM startup values are considered to be secret.
- Peripherals:
  - BK has no reliance on any peripherals.
- Crypto Accelerators:
  - BK embeds all cryptographic components it needs internally in SW. In that sense, it has no reliance on the availability of dedicated crypto accelerators on the MCU.
  - If crypto accelerators are available on the MCU, BK could leverage them to improve performance, given that they can be accessed by the BK custom hardware abstraction layer<sup>3</sup>. BK can benefit from access to accelerators for the following cryptographic operations:
    - \* SHA-256 and/or HMAC-SHA-256
    - \* AES
    - \* ECC

<sup>3</sup>Please contact Intrinsic ID for questions related to integrating BK with on-board crypto accelerators.



## 2. BK Configurations and Function Sets

BK v2.9 is available in 3 different configurations, as described in section 2.1. The BK-Plus and BK-Pro configurations are export controlled according to EU Council Regulation (EC) No. 428/2009 under dual-use control category 5D002.

### 2.1. Configurations

Table 1: Standard Configurations of BK

Function Sets		Configurations [Security Strength 128 or 256]		
		BK-Safe	BK-Plus	BK-Pro
Core	Product information (see Section 2.4.1)	Y	Y	Y
	State management (see Section 2.4.2)	Y	Y	Y
Randomness and Device-Key Generation Functions	Unique device key generation (see Section 2.5.1)	Y	Y	Y
	Random value generation (see Section 2.5.2)	Y	Y	Y
Symmetric Key Crypto Functions	Key Wrapping (see Section 2.6.1)		Y	Y
Elliptic Curve Crypto Functions	ECC key management (see Section 2.7.1)			Y
	ECC signing (see Section 2.7.2)			Y
	ECC key agreement and encryption (see Section 2.7.3)			Y
	CSR and self-signed certificate generation (see Section 2.7.4)			Y
	ASN.1/DER read/write of ECC keys and signatures (see Section 2.7.5)			Y

The BK software module is available in off-the-shelf configurations with code size ranging between 7 KB and 20 KB. The standard BK configurations are listed in Table 1. A standard BK configuration is differentiated by the *function sets* that it supports, where a function set is a



collection of software functions that naturally belong together. Table 1 lists the different function sets that can be implemented by BK:

The Core function set, containing functions related to product information and BK state management. The functions of this set are always implemented by BK, regardless of its configuration.

- The Randomness and Device-Key Generation function set, containing functions related to generating device-unique keys (symmetric as well as ECC private keys) and fully random bits.
- The Symmetric Key Crypto function set, containing functions related to protecting and retrieving external application keys.
- The Elliptic Curve Crypto function set, containing functions related to managing device-unique as well as external private/public key pairs, and functions for performing public-key crypto operations based on these managed keys.

Table 1 shows how these function sets are implemented by the three BK standard configurations in an incremental manner:

1. BK-Safe, implementing the Base and the Randomness and Device-Key Generation function sets.
2. BK-Plus, implementing the Base, the Randomness and Device-Key Generation, and the Symmetric Key Crypto function sets.
3. BK-Pro, implementing the Base, the Randomness and Device-Key Generation, the Symmetric Key Crypto, and the Elliptic Curve Crypto function sets.

Each of these three standard configurations is available with *security strengths* of either 128-bit or 256-bit. This value is specified by the `BK_SECURITY_SIZE_BITS` constant and indicates the highest security level offered by any of the BK cryptographic functions. Moreover, this value will also determine the size of some other product parameters for the given configuration, including the required SRAM PUF size and the size of the AC.

## 2.2. BK Characteristics

This section gives an overview of certain BK characteristics, in particular the sizes of external data structures used by BK. More details can be found in the API Reference Manual of the specific BK version you are using.

### 2.2.1. PUF SRAM and Activation Code sizes

The amount of uninitialized SRAM that is needed by BK for extracting PUF data depends on the security strength. The same holds for the length of the Activation Code (AC) that is generated at enrollment (see section 2.4.2.2) and needs to be stored in NVM. An overview of the required sizes is provided in Table 2.

### 2.2.2. Length of Key Codes

Cryptographic keys that are protected with BK, are stored in the form of a so-called Key Code (KC). This is an encrypted and authenticated representation of the key data that can be safely

**Table 2: PUF SRAM size and AC length**

BK Security Strength	PUF SRAM size	AC length
128 bits	688 Bytes	656 Bytes
256 bits	1024 Bytes	968 Bytes

stored outside the security perimeter in which BK resides. A KC can only be successfully decrypted with BK on the specific chip it was generated on.

### 2.2.2.1. Symmetric Key Codes

Key Codes for storing symmetric keys are created using the BK wrap function, see Section 2.6. The length of such a KC is equal to the size of the stored key data plus a header of 44 Bytes. Table 3 provides some examples. Note that the BK wrap function can also be used to create KCs for bigger blocks of data than typical cryptographic keys. It supports input sizes of up to 8192 Bytes in size.

**Table 3: Examples of sizes of symmetric KCs**

Key data size	Corresponding KC size
128 bits	60 Bytes
256 bits	76 Bytes
1024 bits	172 Bytes
4096 bits	556 Bytes
1024 Bytes	1068 Bytes
(max) 8192 Bytes	8236 Bytes

### 2.2.2.2. Asymmetric Key Codes

The elliptic curve cryptography functions in BK (see Section 2.7) work with private and public keys that are stored in Private Key Code and Public Key Code data structures respectively. The size of the asymmetric Key Code structure depends on size of the largest elliptic curve that is supported with the BK version and is computed as follows:

- The KC data structure size for storing an elliptic curve private key corresponds to 4 times the maximum curve size in words plus 48 bytes of header.
- The KC data structure size for storing an elliptic curve public key corresponds to 4 times the maximum size of the elliptic curve point (x,y coordinates) in words plus 48 bytes of header.



Whereas the reserved data structure size depends on the size of the largest supported curve, the actual bytes used inside that data structure depend on the size of the actual key that is stored. Examples of KC data structure sizes as well as the actual bytes of KC data used inside these structures are provided in Table 3.

**Table 4: Examples of KC sizes for asymmetric keys**

BK Security Strength	elliptic curve key stored	KC structure size	KC bytes used
128 bits	P256 private key	80 Bytes	80 Bytes
128 bits	P256 public key	112 Bytes	112 Bytes
128 bits	P192 private key	80 Bytes	72 Bytes
128 bits	P192 public key	112 Bytes	96 Bytes
256 bits	P521 private key	116 Bytes	116 Bytes
256 bits	P521 public key	180 Bytes	180 Bytes
256 bits	P256 private key	116 Bytes	80 Bytes
256 bits	P256 public key	180 Bytes	112 Bytes
256 bits	P192 private key	116 Bytes	72 Bytes
256 bits	P192 public key	180 Bytes	96 Bytes

## 2.3. Ordering Information

Table 5 provides the relation between the ordering number and the BK configuration and options that are delivered.

## 2.4. Core Function Set

The *Core function set* contains a number of functions that are always available in every configuration of BK. These *Core* functions do not provide any cryptographic functionality but are solely intended for inspecting and controlling the BK module.

### 2.4.1. Product Information Function

It is important to verify the version of BK to confirm what is available in the product, and which product documentation applies to it. To do this, BK has a function that can be used to determine the exact version of the SW library (**bk\_get\_product\_info**).

### 2.4.2. State Management Functions

The BK security IP is a stateful software module, meaning that its available functionality is dependent on the state it is in (also see Section 3.2 for a detailed state diagram). In addition,



**Table 5: BK Ordering Information**

SKU	Configuration	Key Wrap	Key Strength (bits)	Max ECC Curve	Cryptogram, CSR, SSC	No Enroll
BKPR-2.9.x-13	BK-Pro	✓	256	P256	✓	✓
BKPR-2.9.x-7	BK-Pro	✓	256	P521	✓	
BKPR-2.9.x-5	BK-Pro	✓	256	P256	✓	
BKPR-2.9.x-4	BK-Pro	✓	128	P256	✓	
BKPR-2.9.x-3	BK-Pro	✓	256	P521		
BKPR-2.9.x-1	BK-Pro	✓	256	P256		
BKPR-2.9.x-0	BK-Pro	✓	128	P256		
BKPL-2.9.x-1	BK-Plus	✓	256			
BKPL-2.9.x-0	BK-Plus	✓	128			
BKPA-2.9.x-1	BK-Safe		256			
BKPA-2.9.x-0	BK-Safe		128			

compatibility between BK function calls at different points in time also depends on the AC that is used to put BK in its operational state. This subsection provides more information on the functional states of BK, and the function calls for managing them.

#### 2.4.2.1. Device Power-up/Reset and Initializing BK

The security of BK is built upon an SRAM PUF, which consists of the startup values of an SRAM range. After a cold reset (device power-up) or a warm reset (software/hardware reset), BK always needs to condition and/or verify the SRAM startup values for consistency and security. For this reason, after a cold or warm reset, the BK module will be in an *Uninitialized* state. Before any other operation, BK must be brought to the *Initialized* state by calling **bk\_init**.

#### 2.4.2.2. Enrolling BK with AC Output

After successful initialization, BK can be Enrolled by calling **bk\_enroll**. Enrollment instantiates a new *cryptographic context*<sup>1</sup> from the secret SRAM startup values, and produces an AC that is output to the calling software. ***The cryptographic functionality offered by the BK API only becomes available after a cryptographic context is instantiated (either through enrolling or starting BK).***

<sup>1</sup>A *cryptographic context* is the BK internal representation of the device-unique cryptographic data that is derived from the SRAM start-up data (i.e. the PUF), and which is used as the root secret of the BK cryptographic functionality.



It is important to note that the combination of SRAM startup values *and* AC defines the instantiated cryptographic context. If, at a later point, the same context is required in order to perform compatible operations, the same AC needs to be provided to the BK module on the same physical device (in a **bk\_start** call, see Section 2.4.2.3). This means that the AC needs to be stored in between usages of BK, and it is the responsibility of the calling software to take care of this. The AC does not contain any confidential information and can therefore be stored publicly in a NVM, on- or off-chip, without additional protection. Because the AC is also device-specific, using it on another device will result in an error. This is because the SRAM startup values of every device is unique and so cannot be used with the AC of another device. This is one of the core security features of BK that protects the system against cloning and counterfeiting.

Over the lifetime of a device, **bk\_enroll** needs to be called at least once to be able to use the BK functions. After that, the same cryptographic context can always be re-instantiated using **bk\_start**. Technically, **bk\_enroll** can be called multiple times, but note that this will result in different ACs and contexts that are separated and hence incompatible with each other. Multiple calls to **bk\_enroll** over a device's lifetime are only meaningful if the above is desired behavior.

#### 2.4.2.3. Starting BK with Activation Code Input

After successful initialization, BK can be *Started* by calling **bk\_start**. Starting re-instantiates a cryptographic context that was previously generated with **bk\_enroll**, from the secret SRAM startup values and the AC created by **bk\_enroll**. After the cryptographic context is re-instantiated, the BK cryptographic functionality becomes available.

Over the lifetime of a device, **bk\_start** needs to be called every time the BK cryptographic functionality is needed, after each device power-up or reset and **bk\_init**, or after each call to **bk\_stop** (see Section 2.4.2.4). It is the responsibility of the calling software to retrieve the correct AC from storage and provide it as an input to **bk\_start**.

#### 2.4.2.4. Stopping BK

BK also provides a **bk\_stop** function that has the opposite effect of **bk\_start** and **bk\_enroll**; i.e. it *un*-instantiates the cryptographic context and removes (zeroizes) all internal secrets related to it from its internal memory. Hence, after **bk\_stop** the BK cryptographic functionality is effectively *Stopped*. To make it available again, **bk\_start** can be called again. After a call to **bk\_stop**, **bk\_enroll** is no longer available until the next reset.

### 2.5. Randomness and Device-Key Generation Function Set

The *Randomness and Device-Key Generation function set* contains a number of functions for generating symmetric device keys and random numbers. Device keys are random and device-unique and can be regenerated because they are derived from the reproducible secret extracted from the SRAM PUF. In contrast, random values are completely unpredictable and





can only be produced once (and can never be reproduced). This function set constitutes the most basic cryptographic functionality of a PUF-based security module.

## 2.5.1. Unique Device Key Generation

### 2.5.1.1. Generating Device-Unique Symmetric Keys

BK can output a range of symmetric device keys by calling the **bk\_get\_key** function. Based on the provided *key\_type* parameter, **bk\_get\_key** will generate keys with different lengths for symmetric cryptographic primitives (supported lengths are 128, 192 and 256 bits, with the latter two only available in configurations with security strength 256). For each key length, up to 256 different and independent device keys can be produced with the same length, by changing the provided *index* parameter.

Each device key is unpredictable and unique per device *and* per instantiated cryptographic context (i.e., per combination of SRAM startup values and AC), and moreover is cryptographically separated from other keys and secrets used by BK. This means that re-enrolling the same device (by calling **bk\_enroll** more than once), or starting BK with different ACs, will result in different device keys that are unrelated and incompatible. On the other hand, when the same cryptographic context is instantiated on the same device, device keys can be reconstructed, i.e. calling **bk\_get\_key** with the same parameters will always return the same key value.

### 2.5.1.2. Generating Device-Unique and Random Elliptic Curve Private Keys

The **bk\_get\_private\_key** function generates elliptic curve private keys. These private keys can come from two different sources:

1. Device-unique keys, generated from the device's secret fingerprint, which are always reconstructible on the same device in the same cryptographic context. This is similar to symmetric device keys as generated by **bk\_get\_key** but with the mathematical properties to be an elliptic curve private key.
2. Random keys, generated randomly from the BK internal random number generator. This is similar to a call to **bk\_generate\_random** but with the mathematical properties of an elliptic curve private key.

The calling application optionally can include external information in the private key derivation by providing it through a *usage\_context* input. For both device-unique and random private keys, a calling application can employ this usage context input to contribute additional context information or entropy to the key generation process. Moreover, for device-unique private keys this usage context allows the calling application to generate multiple different private keys over the same elliptic curve.

It is important to note that all derived private keys are cryptographically separated, e.g. two derived device-unique keys over the same curve with different usage context will be completely different, and two derived device-unique keys with the same usage context over consecutive curves will be completely unrelated.



If **bk\_get\_private\_key** is called twice within the same cryptographic context to generate two private-key codes for *device-unique* private keys, and if all input parameters (curve, usage context) are equal for both calls, the same device-unique private key will be returned. With that in mind, a calling application does not need to store device-unique private keys, since they can always be (re)generated from the same inputs. For *random* private keys this will *not* be the case, as they are randomly derived every time.

### 2.5.2. Random Value Generation Function

BK can generate arrays of random bytes by calling the **bk\_generate\_random** function. Random bytes are generated internally using a cryptographically secure pseudorandom number generator, which is seeded with entropy originating from the random noise in repeated measurements of a device's SRAM PUF startup values. Arbitrary length random arrays can be generated.

This functionality can also be used to generate random symmetric keys (as opposed to device-unique symmetric keys). Note that, in contrast to device-unique keys, random keys generated in this way can never be reconstructed by BK and need to be stored by the calling application if they are needed at a later time.

## 2.6. Symmetric Key Crypto Function Set

The *Symmetric Key Crypto function set* contains a number of functions for protecting (wrapping) and retrieving (unwrapping) external application keys based on a device-unique PUF-derived secret. Plain keys in their *wrapped* form are referred to as *key codes*. This function set is a natural extension to the BK cryptographic functionality that allows an application to work with externally generated keys yet benefit fully from the protection offered by the device-unique PUF secret.

### 2.6.1. Key Wrapping Functions

The BK functions for protecting and retrieving provided application keys are respectively **bk\_wrap** and **bk\_unwrap**. A call to **bk\_wrap** securely wraps a provided application key value into a key code. The key code fully protects the wrapped key value (in terms of confidentiality and integrity), which can hence be handled without any further protection, e.g. it can be stored in NVM, on- or off-chip, without additional protection. A call to **bk\_unwrap** on the same device instantiated with the same cryptographic context will successfully retrieve the originally wrapped key value from the key code. It is the responsibility of the calling software to store and retrieve key codes in between **bk\_wrap** and **bk\_unwrap**.

Since **bk\_wrap** and **bk\_unwrap** internally operate with device-unique secrets, key codes are only intelligible to BK running on the same device *and* in the same cryptographic context, and completely unintelligible and meaningless otherwise. This means that re-enrolling the same device (by calling **bk\_enroll** more than once), or starting BK with different ACs, will result in the inability to retrieve keys from key codes that were produced with a different cryptographic context.



In addition, **bk\_unwrap** can *only* unwrap key codes that were successfully wrapped by a call to **bk\_wrap**. BK can have other cryptographic functions that output key codes (e.g. **bk\_create\_private\_key**, **bk\_compute\_public\_from\_private\_key** and **bk\_import\_public\_key**), but these *cannot* be unwrapped by **bk\_unwrap**. Instead, these special-purpose key codes should be used as inputs to the dedicated public-key cryptographic functions.

## 2.7. Elliptic Curve Crypto Function Set

The *Elliptic Curve Crypto function set* contains a number of functions for combining the strength of PUF-derived device-unique secrets with public-key cryptography, in particular elliptic-curve cryptography (ECC). This function set allows an application to:

- Derive device-unique ECC private keys
- Protect and manage ECC private and/or public keys based on device-unique secrets
- Call basic ECC crypto functions (ECDSA, ECDH) with protected private- and/or public-key codes, such that no sensitive information needs to be passed as a parameter in a function call.

This function set constitutes a more advanced extension of the BK cryptographic functionality.

### 2.7.1. ECC Key Management Functions

BK has a set of functions for dealing with public/private key pairs based on ECC. The basic functionality of these key management functions is the transformation of private and/or public keys from different sources (device keys, random keys or external keys) into dedicated key codes, which can be used for the ECC functions described in Sections 2.7.2 and 2.7.3. These key codes are secure and device-bound representations of the actual key values and associated data. As a result, applications calling the ECC functions do not need to handle or pass sensitive data.

#### 2.7.1.1. Managing ECC Private Keys as Private Key Codes

The **bk\_create\_private\_key** function generates private-key codes containing elliptic curve private keys. These private keys can come from three different sources:

1. Device-unique keys that are always reconstructable on the same device in the same cryptographic context, similar to symmetric device keys as generated by **bk\_get\_key** but with the mathematical properties of an elliptic curve private key.
2. Random keys, generated randomly from the BK internal random number generator similar to a call to **bk\_generate\_random** but with the mathematical properties of an elliptic curve private key.
3. External private keys that are provided as an input in the API call. Private-key codes for various elliptic curves are supported, including NIST/SEC P-192, P-224, P-256, P-384 and P-521.

In addition to the private-key values, private-key codes generated by **bk\_create\_private\_key** also contain additional input information related to the private key:

- The elliptic curve over which the contained private key is defined.



- Purpose flags that indicate the allowed usage of the contained private key.

These two fields will determine the allowed use of the private-key code. A private-key code can only be used in an ECC function if its purpose flag allows it (e.g. a private key that is only purposed for decryption/key agreement cannot be used for signing, and vice versa), and only for operations over the same elliptic curve on which it was defined.

Device-unique private keys and random private keys are derived internally by **bk\_create\_private\_key**, respectively from the device fingerprint (SRAM PUF startup values) or from the device noise entropy, and from the provided curve and purpose inputs. It is important to note that all derived private keys are cryptographically separated, e.g. two derived device keys over the same curve with different purpose flags will be completely different, also two derived device keys with the same purpose flags over consecutive curves will be completely unrelated.

In addition, the calling application optionally can include external information in the private key derivation by providing it through a *usage\_context* input. For both device-unique and random private keys, a calling application can use this usage context input to contribute additional context information or entropy to the key generation process. Moreover, for device-unique private keys this usage context allows the calling application to generate multiple different private keys for which all other parameters (curve, purpose) are equal. For externally provided private keys, the usage context input is not used, since the private key is not derived internally.

If **bk\_create\_private\_key** is called twice within the same cryptographic context to generate two private-key codes for *device-unique* private keys, and if all input parameters (curve, purpose flags, usage context) are equal for both calls, both returned private-key codes will contain the same device private key. With that in mind, a calling application does not need to store device-unique private-key codes, since they can always be regenerated from the same inputs. For *random* private keys this will *not* be the case.

### 2.7.1.2. Managing ECC Public Keys as Public Key Codes

The **bk\_compute\_public\_from\_private\_key** function takes as input a private-key code and generates a corresponding public-key code; i.e., a key code that contains the elliptic curve public-key counterpart to the private key contained in the private-key code. The additional information (curve, purpose) contained in the private-key code will be copied to the public-key code.

The **bk\_derive\_public\_key** function derives the public key value corresponding to a provided private key. This provides the same functionality as **bk\_compute\_public\_from\_private\_key**, but avoids the need to present private and public keys as key codes. This is convenient for applications that do not need to/want to work with BK's key code formats. On the other hand, this function requires private and public keys to be presented on the API in an unprotected format. When using this function, it is hence up to the calling application to ensure the protection of these values.



The **bk\_import\_public\_key** function converts an external elliptic-curve public-key value to a public-key-code format that can be used with the ECC functions described in Sections 2.7.2 and 2.7.3. This function is needed, for example, when a public-key value from an external certificate is needed as an input for the BK ECC functions described in Sections 2.7.2 and 2.7.3.

The **bk\_export\_public\_key** function for the API exports an elliptic curve public key value from its public-key code container, for use by operations external to BK. This function is needed, for example, when a device-unique public key needs to be output for certification by an external party.

## 2.7.2. ECC Signing Functions

BK has a set of functions for performing basic ECDSA signature generation and verification (respectively **bk\_ecdsa\_sign** and **bk\_ecdsa\_verify**). The **bk\_ecdsa\_sign** function takes as input a message string, or a message hash, and computes an ECDSA signature on it based on a provided private-key (code). The **bk\_ecdsa\_verify** function takes as input a message string, or a message hash, a signature value, and a public-key code, and verifies whether the provided signature matches the message under the given public key.

These functions take private and/or public *key codes* as inputs, instead of key values directly. This means that an application calling these functions never needs to pass possibly sensitive key data, only key codes, which are protected both in confidentiality (important for private keys) and integrity (important for public keys). An important consideration is that public- and private-key codes will also contain purpose flags that indicate whether they are allowed to be used for ECDSA-based operations, or not.

## 2.7.3. ECC Key Agreement and Encryption Functions

BK has a set of functions for performing basic ECDH-based key agreement between two parties, and in extension for setting up a secure messaging protocol between a sender and a receiver, based on cryptograms that are protected with an ECDH-derived shared secret.

These functions take private and public *key codes* as inputs, instead of key values directly. This means that an application calling these functions never needs to pass possibly sensitive key data, only key codes that are protected both in confidentiality (important for private keys) and integrity (important for public keys). An important consideration is that public- and private-key codes will also contain purpose flags that indicate whether they are allowed to be used for ECDH-based operations, or not.

### 2.7.3.1. ECDH Key Agreement

BK has a function (**bk\_ecdh\_shared\_secret**) for generating a shared-secret value from a private-key code and a public-key code based on the ECDH algorithm. The returned shared secret can be processed into one or more shared-secret keys by the calling application, using the proper key derivation functions.





### 2.7.3.2. ECDH-based Cryptogram Generation and Processing

BK has a set of functions for enabling a secure one-pass messaging protocol based on hybrid ECC. The conceptual idea is that a sending system, embedding BK (or a compatible implementation), can transform a plaintext message into a secure cryptogram using its own elliptic-curve private key and a receiver's elliptic-curve public key. The corresponding receiving system can use BK (or a compatible implementation) to unpack the message from the cryptogram using its own elliptic-curve private key (corresponding to the public key used by the sender), and the sender's public key (corresponding to the private key used by the sender). When used correctly, the basic properties achieved by this one-pass protocol are:

- The message contained in a cryptogram is **confidential**, i.e., the message is unintelligible except to the sender and the intended receiver.
- The cryptogram is **integrity-protected**, i.e. the receiver can verify that cryptogram is exactly as the sender created it.
- The receiver can **authenticate the sender** of the message in the cryptogram, i.e., the receiver can obtain proof that the message originated from the sender.
- In addition, the cryptogram functions offer the possibility to ensure that cryptograms are **non-replayable**, i.e., the receiver can detect whether a cryptogram has been received before, or whether it is replayed or received out-of-order.

The fact that these cryptogram functions constitute a one-pass secure protocol with these security properties, makes them well suited for use as import/export functions of external secrets (e.g. in a key provisioning scheme) or as a payload protection mechanism (e.g. in a secure update flow).

The **bk\_generate\_cryptogram** function takes as input a plaintext message, the sender's private-key code and the receiver's public-key code, and creates a cryptogram from it. Two additional inputs are:

- A message counter value, needed for keeping track of the order of produced cryptograms. The **bk\_generate\_cryptogram** function updates this counter and returns the new counter as an output. It is up to the calling application to provide reliable storage and update mechanisms for this counter value.
- The cryptogram type to be used. BK provides two different cryptogram types with slightly different efficiency and security properties.
  - Cryptogram type = "BK\_ECC\_CRYPTOGAM\_TYPE\_ECDH\_STATIC". This is the baseline cryptogram type, which achieves the basic security properties listed above. Since it is solely based on static key pairs on both the sending and receiving side, this cryptogram type cannot achieve non-repudiation, nor forward secrecy.
  - Cryptogram type = "BK\_ECC\_CRYPTOGAM\_TYPE\_ECDH\_EPHEMERAL". This is an extension on the baseline type in which the sending side uses an ephemeral key pair. It achieves the basic security properties listed above, and in addition it achieves forward secrecy with respect to loss of the sender's long-term private key. It does not achieve forward secrecy with respect to the receiver's private key, this cannot be accomplished in a one-pass protocol. It also does not



achieve non-repudiation. If non-repudiation is required, the cryptogram functionality can be combined with the ECDSA signing functionality.

The **bk\_process\_cryptogram** function performs the opposite operation of **bk\_generate\_cryptogram**. It takes as input an earlier produced cryptogram, the receiver's private-key code and the sender's public-key code, and returns the contained plaintext message. Additionally:

- The **bk\_process\_cryptogram** function takes as input the last received message counter value. If the message counter of the received cryptogram does not exceed this counter input, it means that the cryptogram has been received out of order (e.g., it has been replayed) and it will not be accepted. If the cryptogram is accepted and successfully processed, the **bk\_process\_cryptogram** updates this counter and returns the new counter as an output. It is up to the calling application to provide reliable storage and update mechanisms for this counter value.
- The **bk\_process\_cryptogram** function outputs the type of a successfully processed cryptogram (see above). Based on this type, the calling application knows which security properties are obtained by the cryptogram.

The **bk\_get\_public\_key\_from\_cryptogram** helper function optionally aids a receiver of a BK cryptogram in obtaining the public key used by the sender, for further verification.

## 2.7.4. CSR and Self-Signed Certificate Generation

BK provides functionality for generating PKCS#10-compliant certificate-signing requests (CSRs) and X.509-compliant self-signed certificates for elliptic-curve public-private key pairs (with **bk\_create\_csr** and **bk\_create\_selfsigned\_certificate**). These data structures can be used for easy integration of the BK functionality in a public-key infrastructure (PKI).

## 2.7.5. ASN.1/DER ECC Key and Signature Read/Write

The native BK ECC functions (as described in Sections 2.7.1, 2.7.2 and 2.7.3) work with a minimal raw binary format for keys and signatures. While efficient, this can sometimes be inconvenient when interfacing with third-party tools that expect or produce different formats (e.g., for importing keys, or for verifying an externally generated signature). To overcome this issue, BK provides functionality for reading (i.e., for importing) and writing (i.e., for exporting) elliptic-curve public and private keys, and ECDSA signatures in their industry-standardized ASN.1/DER binary format <sup>2</sup>:

- for elliptic-curve public keys, the functions **bk\_read\_SubjectPublicKeyInfo** and **bk\_write\_SubjectPublicKeyInfo** respectively read and write a DER-encoded elliptic-curve public key following the ASN.1 syntax as defined in IETF RFC5480 (e.g. as used in an X.509 certificate), and transform it to/from the BK raw binary format (X9.62).
- for elliptic-curve private keys, the functions **bk\_read\_ECPrivateKey** and **bk\_write\_ECPrivateKey** respectively read and write a DER-encoded elliptic curve

<sup>2</sup>These are the formats as produced by the popular OpenSSL tool when using the '-outform der' option (see [2])



private key following the ASN.1 syntax as defined in IETF RFC5915, and transform it to/from the BK raw binary format.

- for ECDSA signatures, the functions **bk\_read\_ECDSA\_Sig\_value** and **bk\_write\_ECDSA\_Sig\_value** respectively read and write a DER-encoded ECDSA signature value following the ASN.1 syntax as defined in IETF RFC3279 (e.g. as used in an X.509 certificate), and transform it to/from the BK raw binary format.





## 3. BK Module

### 3.1. Library

The BK library software IP is delivered as a collection of one C code header file (`iidbroadkey.h`) and an object file containing the compiled binary.

### 3.2. Operational States

The BK library module is stateful, and it will always assume one of five operational states. Among other things, the operational state of BK determines which functions of its API are available. Calling an unavailable function will result in that function call returning immediately with return code `IID_NOT_ALLOWED`, and will have no further effect or output. The diagram below shows the different operational states of BK, and the possible state transitions between them.

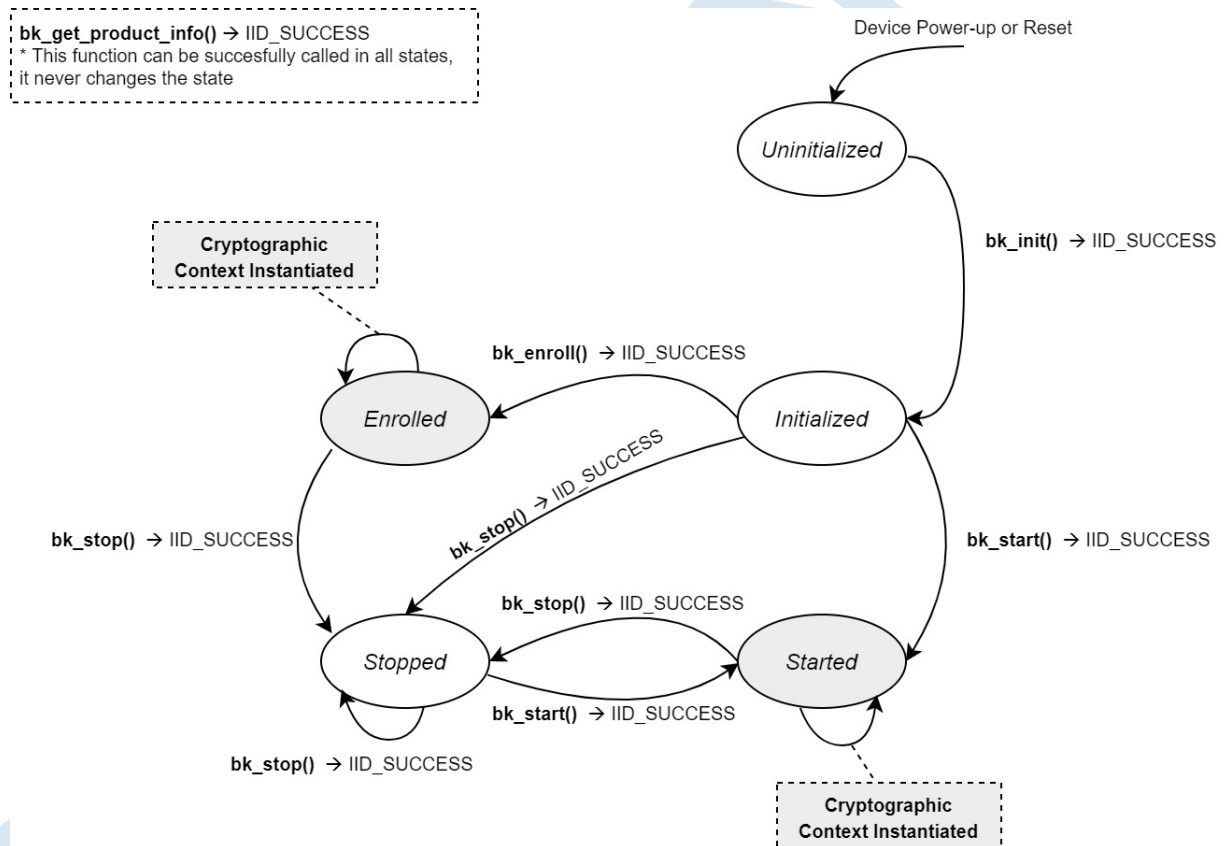


Figure 2: BK State Diagram

The five operational states of BK are:

- **Uninitialized**: the state assumed after a reset or repower of the device on which BK is running. The only allowed function calls in this state are `bk_init` and `bk_get_product_info`.



- *Initialized*: the state assumed after a successful call to **bk\_init**. The only allowed function calls in this state are **bk\_enroll**, **bk\_start**, **bk\_stop** and **bk\_get\_product\_info**.
- *Enrolled*: the state assumed after a successful call to **bk\_enroll**. When in the *Enrolled* state, BK will have an instantiated cryptographic context, and all cryptographic function calls become available. The only unallowed function calls in this state are **bk\_init**, **bk\_enroll** and **bk\_start**.
- *Started*: the state assumed after a successful call to **bk\_start**. When in the *Started* state, BK will also have an instantiated cryptographic context, and all cryptographic function calls become available. The only unallowed function calls in this state are **bk\_init**, **bk\_enroll** and **bk\_start**.
- *Stopped*: the state assumed after a successful call to **bk\_stop**. The only allowed function calls in this state are **bk\_start**, **bk\_stop** and **bk\_get\_product\_info**.

As shown in the diagram, there are also just four functions in the BK API that can invoke an operational state transition when called successfully (return code is IID\_SUCCESS):

- **bk\_init**: brings BK from the *Uninitialized* into the *Initialized* state by pointing it to the SRAM which it can use as the SRAM PUF.
- **bk\_enroll**: brings BK from the *Initialized* into the *Enrolled* state by creating a new cryptographic context, and generating an AC for it.
- **bk\_start**: brings BK from the *Initialized* or *Stopped* state into the *Started* state by reinstantiating a previously created cryptographic context based on a presented AC.
- **bk\_stop**: brings BK from *any* state (except the *Uninitialized* state) into the *Stopped* state. If a cryptographic context was instantiated, **bk\_stop** will unstantiate it.

No other API function call, whether called successfully or not, will ever change the operational state of BK. The only other action which can change the operational state of BK is a *device reset* or *repower*, which will always put BK back into the *Uninitialized* state.

### 3.3. Programming Interface

The programming interface is defined in **iid\_bk.h**.

*The full details of the programming interface can be found in the API User Manual “BK 2.9API User Manual” [IID-BK2-9-API].*

### 3.4. Profiling Information

*Detailed profiling information (performance, memory requirements, code size) of BK in different configurations and for different platforms can be found in the accompanying document “BK 2.8 Profiling Report” [IID-BK2-9-PR].*



## 4. Integration Guidelines

### 4.1. Integration Considerations

The following integration considerations need to be taken into account when using the BK module:

- The PUF SRAM location and size are fixed over the lifetime of the device.
- The PUF SRAM may not be manipulated outside of the control of BK.
- BK does not allocate any memory: it is the responsibility of the calling application to allocate all required buffers with the correct sizes.
- The AC generated during enrollment must be stored by the caller. The availability and integrity of the AC is a requirement for the use of BK. Loss or corruption of an AC inevitably leads to the inability to ever *re-instantiate* the cryptographic context associated with it, potentially resulting in loss of data protected under that context.
- Since the AC does not contain any sensitive information, further protection (confidentiality, authenticity) is not required.
- Re-enrollment of a device will generate a new AC and instantiate a new cryptographic context that is incompatible with earlier contexts instantiated on the same device. As a result, device keys generated in this new context will be different from those generated in earlier contexts, and key codes generated in earlier contexts cannot be unwrapped or used in the new context.

### 4.2. Reliability Optimizations

While a chip is in use (i.e., powered on) the physical parameters of the device will change slightly over the course of years, a process known as *silicon aging*. How soon these changes start to occur depends heavily on the conditions under which the chip is used (temperature, power supply voltage level, etc.). Silicon aging potentially also affects the SRAM PUF startup values over time.

To improve long-term reliability, *anti-aging measures* are taken during **bk\_init**, and during a successful **bk\_start** operation. These measures put the SRAM used by BK in an optimized state such that it can remain powered on for a long time without being degraded by the aging process. While in this optimized state, BK functions in the regular way. It is not recommended to leave BK in the Uninitialized for a prolonged time. Rather it is recommended to call **bk\_init** at the earliest possible convenient time, as the optimal anti-aging measures are only completely taken when a correct AC is provided during **bk\_start**.

### 4.3. Power-up Recommendations

To get good SRAM start-up behavior, it is recommended that the supply voltage power-up curve of the SRAM power supply meets the following guidelines:

- Power-supply voltage must be monotonously increasing (no power dips during voltage ramp-up).
- Power-supply voltage rise time from 0V to 90% of Vdd must be less than 0.5 ms.

- Before power-on, the power supply must have been at 0V for a sufficiently long time in order to guarantee fresh startup values in the SRAM memory. Typically, a power-off time of 100 ms is enough. For extreme low temperatures ( $-40^{\circ}\text{C}$  and below) the required power-off time may need to be extended to around 500 ms.