Article

Embedded LUKS (E-LUKS): A Hardware Solution to IoT Security

German Cano-Quiveu *, Paulino Ruiz-de-clavijo-Vazquez, Manuel J. Bellido, Jorge Juan-Chico, Julian Viejo-Cortes, David Guerrero-Martos and Enrique Ostua-Aranguena

Department of Electronics Technology, E.T.S. Ingeniería Informática, University of Seville, Avda. Reina Mercedes s/n, 41012 Seville, Spain; pruiz@us.es (P.R.-d.-c.-V.); bellido@dte.us.es (M.J.B.); jjchico@dte.us.es (J.J.-C.); julian@us.es (J.V.-C.); guerre@dte.us.es (D.G.-M.); ostua@dte.us.es (E.O.-A.)
* Correspondence: germancq@dte.us.es

Abstract: The Internet of Things (IoT) security is one of the most important issues developers have to face. Data tampering must be prevented in IoT devices and some or all of the confidentiality, integrity, and authenticity of sensible data files must be assured in most practical IoT applications, especially when data are stored in removable devices such as microSD cards, which is very common. Software solutions are usually applied, but their effectiveness is limited due to the reduced resources available in IoT systems. This paper introduces a hardware-based security framework for IoT devices (Embedded LUKS) similar to the Linux Unified Key Setup (LUKS) solution used in Linux systems to encrypt data partitions. Embedded LUKS (E-LUKS) extends the LUKS capabilities by adding integrity and authentication methods, in addition to the confidentiality already provided by LUKS. E-LUKS uses state-of-the-art encryption and hash algorithms such as PRESENT and SPONGENT. Both are recognized as adequate solutions for IoT devices being PRESENT incorporated in the ISO/IEC 29192-2:2019 for lightweight block ciphers. E-LUKS has been implemented in modern XC7Z020 FPGA chips, resulting in a smaller hardware footprint compared to previous LUKS hardware implementations, a footprint of about a 10% of these LUKS implementations, making E-LUKS a great alternative to provide Full Disk Encryption (FDE) alongside authentication to a wide range of IoT devices.

Keywords: LUKS; embedded systems; field programmable gate array; IoT

Citation: Cano-Quiveu, G.; Ruiz-de-clavijo-Vazquez, P.; Bellido, M.J.; Juan-Chico, J.; Viejo-Cortes, J.; Guerrero-Martos, D.; Ostua-Aranguena, E. Embedded LUKS (E-LUKS): A Hardware Solution to IoT Security. *Electronics* **2021**, *10*, 3036. https://doi.org/10.3390/electronics10233036

Academic Editors: Juan Antonio López Ramos, Antonio David Escobar Molero and José Antonio Álvarez Bermejo

Received: 12 November 2021
Accepted: 3 December 2021
Published: 5 December 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The Internet of Things (IoT) industry has grown steadily in the last few years. Many facts indicate that this growth is an upward trend [1], with IoT data traffic expected to reach around 2000 petabytes of information by 2024 [2]. The use of IoT devices has reached many fields such as industrial production [3], health care [4], or quality-of-life-related devices [5]. Its implementations in our homes and personal environments has had a great impact on many daily life processes [6,7]. However, IoT is still a recent technology in which numerous devices with low resources share a large amount of personal and sensible data, making data security one of the major issues of IoT. The security problem in IoT has been addressed by many authors, mainly from the perspective of data communication between nodes [8,9]. However, the security of the data stored internally by these devices should also be addressed.

Currently, many IoT nodes based on complex embedded systems use a Flash memory as their main storage for user applications. Some of these systems can expand their storage by adding external Flash mass storage devices such as an SD card. If the application requires some kind of confidentiality in the stored data, some type of data encryption must be used. Data encryption can be implemented at the operating system (OS) level by using dedicated libraries and factory software in order to encrypt single files or the complete block device that holds the file system. Solutions such as Bitlocker from Microsoft, FileVault from Apple, or VeraCrypt, among others, can be used at the user level by running software

tools in the OS. But these kinds of solution have to face challenges such as key storage, key change, and interoperability with other systems; for example, when the Flash storage device is removable and needs to be accessed from a desktop computer.

In any case, the presence and evolution of IoT security must be studied in order to use the available knowledge. Only in doing so can the way new devices and application areas evolve be decided [10–13].

Implementing file encryption is not an option for many IoT devices. On the contrary, block device encryption adds a new layer between the physical device and the read/write device operations. This is much more resource-friendly and allows IoT applications to access encrypted memory in a transparent way. It can pose a solution even for a very simple IoT device that does not run an OS but a standalone application. In fact, there are already solutions that implement this kind of Full Disk Encryption (FDE), such as the Linux Unified Key System (LUKS) [14].

Another important aspect of IoT security is that IoT nodes must be robust against the use of reverse engineering [15,16] and/or malware such as the Mirai botnet [17]. Thus, it is necessary to ensure the integrity and authenticity of user data in addition to its confidentiality.

In this paper, a lightweight IoT protocol based on LUKS is introduced, which provides, on top of the FDE of LUKS, both the integrity, and authenticity of the user data. This protocol has been called Embedded LUKS (E-LUKS) and it has been especially tailored to allow an efficient hardware implementation, making possible the implementation of a small footprint hardware module that is able to read/write into a final storage device. In this sense, the hardware E-LUKS module implemented will act as a transparent layer between embedded systems and their storage memories.

This paper has been organized as follows. First, the Related Work section describes other hardware solutions that can also provide security of the data. Then, in the Section titled Linux Unified Key Setup, the LUKS specification used as blueprints for the proposed solution, necessary to understand it, is described. Next, the section named Embedded LUKS details the proposed solution and its changes compared to LUKS, finishing with the E-LUKS core hardware implementation. The Results section follows, in which the experiment conducted to verify the proposed solution is represented, displaying the execution time of the system with E-LUKS and without it. Furthermore, this section also proves the advantages of E-LUKS via the comparison, and describes the subsequent analysis of the employ of resources between E-LUKS and the alternatives presented in the Related Work section. Lastly, a brief evaluation of the proposed solution is provided in Conclusions.

2. Related Work

LUKS is a block-device encryption specification that aims to be implemented in software, although there are some hardware implementations on FPGA chips as well [18,19]. In Reference [18], the implementation is focused on an energy-efficient LUKS design that can compete in terms of speed against software solutions on higher-end devices. Alternatively, there is another LUKS implementation, which achieves high performance due to a pipeline implementation, proposed in [19]. These designs are not suited for resource constrained devices, since both use multiple instances of a cryptographic algorithm implementation in order to increase the performance at the expense of additional FPGA resources.

Other hardware-solutions such as Sancus [20] are specifically designed for resource constrained devices. Sancus is an open-source solution that focuses on the integrity and authenticity of the data. Its objective is to assure that each file or program on the memory device is untampered and authenticated without trusting any infrastructural software. For this purpose, the device uses a symmetric key to assure the integrity and authenticity of each file independently by means of a Key Derivation Function (KDF) implemented in the hardware. The KDF, together with the device key and file parameters, such as its contents and location on the memory device, generates a digest for each file.

To isolate each in the device; Sancus uses a variation of the program-counter based memory access control [21]. It allows access to the protected data of the file if and only if the program counter is in its text section. In addition, the text section of a file can only be executed if the program counter jumps to its defined entry point. A call to another file can only be executed if a digest of the file to be accessed has been previously deployed in the memory device.

Based on Sancus, several additional solutions have appeared. One of them is Soteria [22], which adds confidentiality to the previous solution, and a specific software loader module. This module is responsible for the confidentiality of the other modules in the node.

Other solutions are those that implement confidentiality on a System-on-Chip (SoC) FPGA RAM memory. In Reference [23], the authors present a transparent encryption/decryption hardware module that uses block ciphers to provide confidentiality of the data. It is also able to achieve authenticity of the data by implementing a Tamper Evident Counter (TEC) tree with a nonce value as a root stored on-chip. This TEC serves to provide a nonce value, which is used only when the block cipher is the Authenticated Encryption (AE) cipher Ascon. Another proposed solution is Atlas [24], which offers confidentiality alone. The idea of Atlas is to use the entry point of the file or code as the Initialization Vector (IV) employed in the encryption/decryption of that file. Currently, the lightweight SIMON block cipher is used.

The proposed solution in this paper, E-LUKS, provides FDE as the LUKS solutions do in [18,19]. Nevertheless, as opposed to LUKS, E-LUKS has been designed for constrained resources devices and also supports data authentication in addition to confidentiality. While the solutions based on Sancus [20] require modifying the processor itself and its own firmware, the E-LUKS solution is independent of the processor and, in fact, it can be used in systems with no processor at all. Sancus-based solutions allow for the isolation of different programs in the external RAM memory of the device. In contrast, E-LUKS uses external Flash storage, which may be present as the main or complementary data storage of the system, such as a microSD card, which is usually more vulnerable to data tampering. The solutions in [23] and Atlas [24] also focus on securing the RAM memory of the system. Atlas implements instructions for the selected processor but does not support the integrity and authentication of the data. Finally, the work presented in [23] depends only on the bus interfaces of the system. However, to allow integrity and authentication of the data, it requires part of the TEC tree in the memory alongside the data, excluding the root nodes, which may consume a significant amount of memory resources.

3. Linux Unified Key Setup

Linux Unified Key Setup (LUKS) is a specification intended to standardize cryptographic key setup for data storage encryption. LUKS was introduced in 2005, and a new version (LUKS2) [25] was released in 2018. This last version extends the previous one, taking all its basic concepts as blueprints. The structure and operations described below are taken from LUKS1.

LUKS performs full user data encryption using a master key that is itself encrypted and stored in front of the encrypted data. The master key can be stored multiple times using different user passwords, allowing many users to have access to the encrypted data.

An LUKS formatted block device consists of two parts: (1) a small unencrypted part that consists of an LUKS header followed by several slots for each granted user; (2) an encrypted part starting with eight slots, each one containing the encrypted master key for one possible user, followed by the user data. All encryption is performed using symmetric cryptography.

To retrieve the encrypted data, a user must unlock one of the eight slots and decrypt the corresponding encrypted master key. A formatted LUKS device must have at least one active slot with the corresponding encrypted master key.

3.1. Cryptography

The type of cryptography used is a key factor in any secure specification. Modern cryptography is mainly supported by ciphering algorithms and cryptographic hash functions. The LUKS specification allows for the use of a variety of block ciphers and hash functions. Possible block ciphers that can be used with LUKS are shown in Table 1, together with supported modes, key lengths, and block lengths. Supported hash functions are listed in Table 2. In addition, LUKS uses Password-Based Key Derivation Function 2 (PBKDF2) as the Key Derivation Function (KDF), following the recommendations of RFC8018 [26].

Table 1. Linux Unified Key Setup (LUKS) block cipher algorithms.

Cipher	Key Lenght (bits)	Block Lenght (bits)	Modes
AES [27]	128–256	128	ecb, cbc-plain, cbc-essiv: *hash*, xts-plain64
Twofish [28]	128–256	128	ecb, cbc-plain, cbc-essiv: *hash*, xts-plain64
Serpent [29]	128–256	128	ecb, cbc-plain, cbc-essiv: *hash*, xts-plain64
cast5 [30]	128–256	128	ecb, cbc-plain, cbc-essiv: *hash*, xts-plain64
cast6 [31]	40–128	64	ecb, cbc-plain, cbc-essiv: *hash*, xts-plain64

Table 2. LUKS hash functions.

Hash	Output Lenght (bits)
sha1 [32]	160
sha256 [33]	256
sha512 [33]	512
ripemd160 [34]	160

3.2. Linux Unified Key Setup Internal Layout

As previously mentioned, the layout of an LUKS block device is divided into two parts, one is unencrypted and the other one is encrypted, as depicted in Figure 1. The unencrypted part is called the LUKS partition header (*LUKS phdr*). It starts at the beginning of the block device and holds two types of blocks: the *header* and various Key Slots (KS_x). The encrypted part contains two types of blocks as well: Key Materials (KM_x) blocks and *User Data* blocks. The *header* identifies the block device as an LUKS partition and stores information about the master key, together with information about the cryptographic algorithms selected in the LUKS block device. Table 3 summarizes the fields of the *phdr*. The Key Slots (KS_x) have information about the user key and the parameters to recover the master key. The 8 KS_x allow up to eight users to access the master key, each one with a different personal user password. Each of the KS_x blocks ($x = 1, 2, \ldots, 8$) points to a KM_x block, which is the encrypted master key that can only be recovered using the corresponding personal user password. Table 4 shows all fields of a KS_x. Following the KM_x blocks are the *User Data*, which are encrypted with the master key.

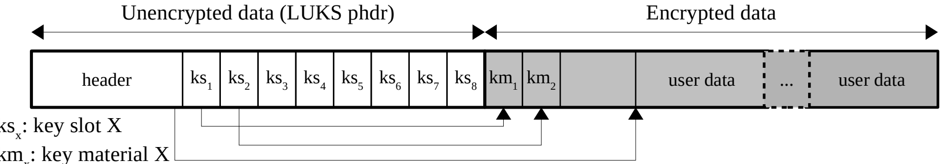

Figure 1. LUKS layout.

Table 3. LUKS partition header (LUKS phdr) fields.

Offset (Bytes)	Field Name	Lenght (Bytes)	Description
0	magic	6	indicates an LUKS partition
6	version	2	indicates LUKS version
8	cipher-name	32	String with the cipher used
40	cipher-mode	32	String indicating the cipher mode
72	hash-spec	32	String with the hash used
104	payload-offset	4	block (512-bytes sector) where the encrypted data begins
108	key-bytes	4	lenght of the master key
112	mk-digest	20	master key digest from Key Derivation Function (KDF)
132	mk-digest-salt	32	salt parameter for master key in KDF
164	mk-digest-iter	4	count parameter for master key in KDF
168	uuid	40	UUID of the partition
208	KS_1	48	Key Slot 1
...	....	...	...
544	KS_8	48	Key Slot 8

Table 4. LUKS Key Slot (KS_x) fields.

Offset (Bytes)	Field Name	Lenght (Bytes)	Description
0	active	4	indicates if the KM_x is enabled
4	iterations	4	count parameter for the KDF
8	salt	32	salt parameter for the KDF
40	key-material-offset	4	block (512-bytes sector) where the KM_x begins
44	stripes	4	number of anti-forensic stripes

3.3. *Operations*

LUKS devices are managed using four types of operation: *initialisation*, *add new password*, *master key recover*, and *password revocation*. By using these functions, LUKS is able to store and retrieve the necessary master key to perform any operation on disk. These functions are summarized below.

3.3.1. *Initialisation*

The *initialisation* operation consists of formatting the block device according to the LUKS layout. This operation requires the following parameters: the master key, the salt for the KDF (*mk-digest-salt*), and a number of iterations (*mk-digest-iter*). These parameters are auto-generated by the software that performs the LUKS formatting. In the process, a new *LUKS phdr* is written in the block device, followed by KS_x. It is also mandatory to specify which cryptographic algorithms (block cipher and hash function) are selected to be used on the block device. To complete the *LUKS phdr*, the master key is used as a parameter, along with the salt and the number of iterations, to generate a digest (*mk-digest*). Lastly, the LUKS format must store the encrypted master key in at least one KM_x. Because the master key is encrypted by a user key, a new user password needs to be added in the way described below.

3.3.2. *Add New Password*

This operation consists of adding a new password for a user, using one of the eight KS_x available. When the block device is being formatted, the unencrypted master key is available from the *initialisation* operation. On the contrary, for an already LUKS-formatted device, the master key is retrieved by the *Master Key Recovery* operation. The *Add New Password* operation begins with the user providing a new user password. After that, it generates random values from the salt and iteration count for the KS_x and stores them. Once the values are generated, the KDF takes the new password along with the salt and the iteration count. The master key is then processed by an anti-forensic splitter, generating a new derived key. This derived key is encrypted using the KDF output as the key for the cipher. Then, the encrypted password is stored on the device as KM_x for later use.

3.3.3. *Master Key Recovery*

The third operation is to recover the master key from a KM_x. This operation consists of decrypting the encrypted master key for two purposes: access to the encrypted user data or to change a user password by using the *Add New Password* operation. The *Master Key Recovery* operation requires the user password associated with the KS_x. It uses the KS_x salt and iteration count values, along with the user password, as parameters for the KDF. Next, the KM_x is recovered from the storage device and is decrypted using the KDF output as the cipher key. Then, the decrypted result is processed by an anti-forensic merge, generating a new candidate key. This candidate key, the *mk-digest-salt* and the *mk-digest-iter* are passed as parameters to the KDF. Finally, the KDF output is compared with the *mk-digest* from the *header*, and if both values match, the candidate key is returned.

3.3.4. *Password Revocation*

The last operation is the password revocation for a selected key slot. This operation consists of deleting the selected KM_x and setting the *activate* field of the KS_x to inactive.

4. Embedded LUKS

Embedded LUKS (E-LUKS) is a new proposal to bring LUKS to devices with limited resources, such as IoT devices. It can be applied in scenarios where the security of the local data is an issue, such as devices in public locations, where an attacker has easy access to the device. E-LUKS performs FDE of the block device. To perform this task, it requires a master key, which is used to create the user keys, as well as in LUKS. The difference being that E-LUKS allows integrity and authentication, on top of confidentiality, of the user data.

Another major aspect to consider is the cryptographic algorithm used for small devices with limited resources. In the last few years, a growing trend of low footprints cryptographic algorithms has emerged for these types of devices. Although these algorithms are designed mainly to optimize the use of internal device resources, at the same time, they also decrease the security level. Nevertheless, the security provided by these algorithms is enough and presents more advantages than disadvantages.

4.1. E-LUKS Internal Layout

The E-LUKS layout is heavily based on LUKS. In Figure 2, a diagram of the layout is shown. However, in E-LUKS, the Key Material (KM_x) is integrated into the Key Slot (KS_x). In addition, some parts present differences in their number of fields, as can be seen in Tables 5 and 6 with respect to the *E-LUKS header* and KS_x. Another characteristic of E-LUKS is that the *E-LUKS header* and the KS_x must be located in the first 512 bytes of the memory. This limitation is due to the fact that embedded devices usually have an external flash memory, such as a microSD card, as their main storage. These flash memories are typically divided into blocks of 512 bytes and, to facilitate the HDL design, it has been decided that the first block of the memory contains all the E-LUKS related data, while the rest of the memory is reserved for the user encrypted data. The fields below are taken by selecting the PRESENT cipher with an 80-bit key and, as a hash function, the same as that

used in the KDF and the HMAC, the SPONGENT-88. With these requirements, the *E-LUKS header* takes 52 bytes, and each KS_x takes 40 bytes. Therefore, the maximum number of KS_x is up to 11.

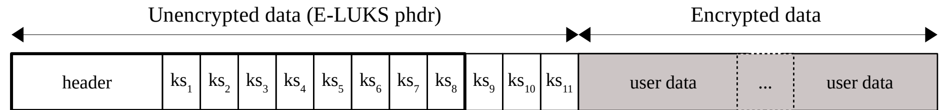

Figure 2. Embedded LUKS (E-LUKS) layout.

Table 5. E-LUKS header fields.

Offset (Bytes)	Field Name	Lenght (Bytes)	Description
0	magic	6	indicates an E-LUKS partition
6	mk-digest	11	master key digest from KDF
17	mk-digest-salt	8	salt parameter for master key in KDF
25	mk-digest-iter	4	count parameter for master key in KDF
29	mk-hmac	11	output from the HMAC of the encrpyted user data
40	mk-IV	8	Initialization Vector (IV) parameter for the block cipher
48	user-data-blocks	4	blocks (512-bytes sector) of user data

Table 6. E-LUKS Key Slot (KS_x) fields.

Offset (Bytes)	Field Name	Lenght (Bytes)	Description
0	activate	4	indicates if the KS_x is enabled
4	iterations	4	count parameter for the KDF
8	salt	8	salt parameter for the KDF
16	pwd-encrypted	16	key material KM_x, encrypted master key
40	IV	8	IV parameter for the block cipher used to generate pwd-encrypted

4.2. Cryptographic Algorithms

The cryptography used in ELUKS inherits from LUKS the way to store the master key. In this sense, ELUKS uses a block cipher, a HASH function, and a KDF. The main feature of E-LUKS is the fact that it uses cryptographic algorithms designed for resource constrained devices. To simplify the design and standardize the internal layout, it has been decided that there will be only one choice for each of the algorithms (block cipher, hash function, HMAC, and KDF). These algorithms are introduced in the following sections.

4.2.1. PRESENT

PRESENT is a block cipher optimized for low resource usage that also has low power consumption. In recent years, multiple implementations of this cipher have appeared with different hardware architectures to boost certain aspects [35]. However, even though there are multiple alternatives for PRESENT, this work has implemented it as was originally

presented in [36]. PRESENT is a substitution–permutation network (SP-network) of 31 rounds of encryption–decryption, all of them with the same structure, which uses a block length of 64 bits and allows keys of 80 bits and 128 bits. Its pseudocode is shown in Algorithm 1.

Algorithm 1 Pseudo-code of the PRESENT encrypt operation.

1: $STATE \leftarrow 0$
2: $ROUNDKEYS = [K_1, K_2, \ldots, K_{32}]$
3: $ROUNDKEYS \leftarrow generateRoundKeys()$
4: **for** $i \leftarrow 1$ to 31 **do**
5: $\quad STATE \leftarrow addRoundKey(STATE, K_i)$
6: $\quad STATE \leftarrow sBoxLayer(STATE)$
7: $\quad STATE \leftarrow pLayer(STATE, K_i)$
8: $STATE \leftarrow addRoundKey(STATE, K_{32})$
9: **return** $STATE$

Each of the functions of the algorithm is described as follows:

- *addRoundKey*: Given the *round key* $K_i = k_{63}, k_{62}, \ldots, k_0$ for $1 \leq i \leq 32$ and the current $STATE = b_{63}, b_{62}, \ldots, b_0$, the function returns a new state $RESULT = r_{63}, r_{62}, \ldots, r_0$, calculated as:

$$r_j = b_j \oplus k_j. \quad \text{for} \quad 0 \leq j \leq 63. \tag{1}$$

- *sBoxLayer*: Is a function with an input of 64 bits that returns an output of 64 bits. Internally, the input data are divided into 16 groups of 4 bits, such that $INPUT = b_{63}, b_{62}, \ldots, b_0 = w_{15}, w_{14}, \ldots, w_0$. Each group ($w_i$) is then processed by *S-boxes*, an S-box S transforms 4-bit input into 4-bit output such that $S : \mathbb{F}_2^4 \rightarrow \mathbb{F}_2^4$. Table 7 shows all possible *S-boxes* results. Finally, the *sBoxLayer* and the result are calculated as follows:

$$\left.\begin{array}{r} w_i = b_{4*i+3} \parallel b_{4*i+2} \parallel b_{4*i+1} \parallel b_{4*i+0} \\ RESULT = S[w_{16}], \ldots, S[w_0] \end{array}\right\} \quad \text{for} \quad 0 \leq i \leq 15.$$

Table 7. PRESENT *S-box*.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

- *pLayer*: The pLayer operation performs a permutation operation defined by:

$$P(i) = \begin{cases} i * 16 \mod 63, & i \in 0, \ldots, 62 \\ 63, & i = 63. \end{cases}$$

- *generateRoundKeys*: Let us assume a length key of 80 bits such as $K = k_{79}, k_{78}, \ldots, k_0$. It is necessary to generate a different round key K_i of 64 bits for each of the rounds. The steps are the following:

$$1.\ [k_{79}, k_{78}, \ldots, k_1, k_0] = [k_{18}, k_{17}, \ldots, k_{20}, k_{19}] = K << 61 \tag{2}$$
$$2.\ [k_{79}, k_{78}, k_{77}, k_{76}] = S[k_{79}, k_{78}, k_{77}, k_{76}] \tag{3}$$
$$3.\ [k_{19}, k_{18}, k_{17}, k_{16}, k_{15}] = [k_{19}, k_{18}, k_{17}, k_{16}, k_{15}] \oplus i \tag{4}$$
$$4.\ K_i = k_{79}, k_{78}, \ldots, k_{16} = MSB_{64}(K). \tag{5}$$

Block ciphers can operate in different modes, the default mode being the ECB mode (Electronic Code Book mode). However, this mode does not provide security as the other modes do. Therefore, for the PRESENT cipher in this work, it has been decided to select the

CTR mode (counter mode). In this mode, the cipher behaves as a stream cipher, thus being used to generate a keystream s. In the equation below, the CTR mode is described where m_i is the i-th 64-bit unencrypted block and c_i is the i-th encrypted block. In this mode, x_1 begins at a random value IV (Initialization Vector) and behaves as a counter. Therefore, this mode allows each encrypted block to be different, regardless of its data.

$$\left.\begin{aligned} c_i &= m_i \oplus MSB_s(ENC_k(x_i)) \\ x_{i+1} &= INC(x_i) \end{aligned}\right\} \quad \text{for} \quad 1 \leq i \leq n.$$

4.2.2. SPONGENT

SPONGENT [37] is a hash function based on the sponge architecture and the use of the permutation operation introduced in PRESENT.

The sponge architecture is an iterative design composed of stages. It takes an arbitrary length of input data and generates an output whose length is related to the number of stages in the architecture. The parameters and operations are the following:

- r: bits length of the ratio.
- c: bits length of the capacity.
- R: number of rounds that take place in each stage.
- n: bits length for the output of the sponge architecture.
- b: bits length for the internal state, which is $b = r + c$.
- π_b: represents the function for the permutation operation. This function $\pi_b(x) = y$ such that $x, y \in 0, 1^b$. In addition, this function is the main operation in each stage of the sponge architecture.

Once the parameters are defined, it is necessary to describe the three phases of the sponge architecture.

- *Initialization phase*: This phase pads the input M with a '1' followed by many '0' until achieving $len(M) \mod r = 0$. Then, the input M is divided into blocks of r-bits such that $M = m_1, m_2, \ldots m_{\frac{len(M)}{r}}$.
- *Absorbing phase*: This phase is composed of stages. In each stage, the input block m_i is added to the current state, as follows $STATE = STATE \oplus m_i$. Once the input is added, a new state is generated by the permutation operation such that $STATE = \pi_b(STATE)$.
- *Squeezing phase*: This phase, as well as the absorbing phase, is composed of stages. Each stage generates r-bits of the output h, from the state $h_i = STATE[b-1 : b-1-r] = MSB_r(STATE)$. Then, a new state is created $STATE = \pi_b(STATE)$. The stages go on until n-bits of the output h have been generated.

SPONGENT offers five different variants to reach different levels of security. Variants, as well as the value of the parameters, are shown in Table 8.

Table 8. Security levels of SPONGENT.

Variant	*n*	*b*	*c*	*r*	*R*	*Preimage*	*2nd Preimage*	*Collision*
SPONGENT-88	88	88	80	8	45	80	40	40
SPONGENT-128	128	136	128	8	70	128	64	64
SPONGENT-160	160	176	160	16	90	144	80	80
SPONGENT-224	224	240	224	16	120	208	112	112
SPONGENT-256	256	272	256	16	140	240	128	128

PRESENT Permutation π_b

The permutation operation π_b is the main part of the design. The behaviour of the function can be defined as follows:

for $(i = 1; i \leq R; i = i + 1)$
$STATE = lCounter_b(i)[0 : \log_2(R) - 1] \oplus STATE[b - 1 : b - 1 - \log_2(R)]$
$STATE = lCounter_b(i)[\log_2(R) - 1 : 0] \oplus STATE[\log_2(R) - 1 : 0]$
$STATE = sBoxLayer_b(STATE)$
$STATE = pLayer_b(STATE)$
end for.

The $sBoxLayer_b$ and $pLayer_b$ functions are based on their homology described in the PRESENT section. The main difference is that these functions are modified to accept either input or output of b-bit. In the case of the $sBoxLayer_b$, more S-boxes have been used in parallel. Thus, $pLayer_b$ is defined as:

$$pLayer_b(i) = \begin{cases} i * (b/4) \mod b - 1, & i \in 0, \dots, b - 2 \\ b - 1, & i = b - 1. \end{cases}$$

Regarding the $lCounter_b$, it is an LFSR of $\log_2(R)$-bits. This register is activated in each iteration $i = 1, 2, \dots, R$. It uses different irreducible polynomials as coefficients and different initial values for each variant of SPONGENT, as shown in Table 9.

Table 9. $lCounter_b$ values.

Variant	Polynomial	Initial Value
SPONGENT-88	$x^6 + x^5 + 1$	$0x05$
SPONGENT-128	$x^7 + x^6 + 1$	$0x7A$
SPONGENT-160	$x^7 + x^6 + 1$	$0x45$
SPONGENT-224	$x^7 + x^6 + 1$	$0x01$
SPONGENT-256	$x^8 + x^4 + x^3 + x^2 + 1$	$0x9E$

HMAC

The hash function is used in E-LUKS as part of the HMAC construction. An HMAC allows both: integrity and authentication. The HMAC used was presented in [38] with the following definition:

$$HMAC_K(X) = Y.$$

First, the key K is XORed with a repetitive pattern of bits called $ipad$. Then, m_0 is calculated as the hash of the input of the HMAC with a suffix of the XORed key.

$$ipad = 0x33, 0x33, \dots, 0x33$$

$$m_0 = h[(K \oplus ipad)||X].$$

After that, it proceeds to calculate the output of the HMAC. To this end, another repetitive pattern of bits called $opad$ is XORed with the key and is used as a suffix with the previous hash output m_0. These data are then passed to the hash function to get the output:

$$opad = 0x5C, 0x5C, \dots, 0x5C$$

$$Y = h[(K \oplus opad)||m_0].$$

4.2.3. KDF

The KDF used in E-LUKS is based on the PBKDF2, the KDF used in LUKS. Therefore, this function will be explained.

$PBKDF2(P, S, c, dkLen)$ contains the following parameters:

- P: the master key.

- S: *salt* value.
- c: represents the number of iterations.
- $dkLen$: length of the derivated key.

First, $PBKDF2$ checks that the desired derived key is in the range in which $hLen$ represents the output length of the hash function.

$$dkLen > (2^{32} - 1) * hLen \rightarrow \textbf{ERROR}.$$

Then, it calculates the number of blocks of length $hLen$ used in the derivated key. This value is stored and, for the last block, in the case of a non-integer result dividing by $hLen$, the number of bytes is also stored.

$$l = \left\lceil \frac{dkLen}{hLen} \right\rceil \tag{6}$$

$$r = dkLen - [(l-1) * hLen]. \tag{7}$$

For each of the blocks of length $hLen$, T_i, the F function will be applied, which takes P, S, c and the index block as a parameter.

$$T_1 = F(P, S, c, 1) \tag{8}$$

$$T_2 = F(P, S, c, 2) \tag{9}$$

$$\ldots \tag{10}$$

$$T_l = F(P, S, c, l). \tag{11}$$

The function F is the XOR of the first c iterations of the HMAC function H.

$$F(P, S, c, i) = U_1 \oplus U_2 \oplus \cdots \oplus U_c \tag{12}$$

$$U_1 = HMAC_P(S||i) \tag{13}$$

$$U_2 = HMAC_P(U_1) \tag{14}$$

$$\ldots \tag{15}$$

$$U_c = HMAC_P(U_{c-1}). \tag{16}$$

Lastly, all the calculated blocks are concatenated in order to generate the derivated key DK.

$$DK = T_1||T_2||\ldots||T_l[r-1:0].$$

However, this construction could be improved as shown in [39]. This has been the alternative chosen for E-LUKS.

In this construction, instead of an HMAC, a hash function was used for the F function. In addition, the parameter c is passed as a parameter for the hash function.

$$y = F(P, S, c) = H^c(P||S||c).$$

Here, H^c represents the hash function executed c times as follows:

$$y_c = H^c(P||S||c)$$
$$y_1 = H(P||S||c)$$
$$y_2 = H(y_1)$$
$$\ldots$$
$$y_c = H(y_{c-1}).$$

4.3. Operations

Similarly to the previous subsection, E-LUKS has the same operations as LUKS, adding a new one that allows the integrity and authentication of the user data.

4.3.1. Initialisation

The main difference in this operation compared to LUKS is the inclusion of integrity and authentication of the user data. Therefore, to allow integrity and authentication, this stage requires the calculation of the HMAC of all the encrypted user data. Below is a brief description of the whole operation.

The *initialisation* operation requires the master key, *mk-digest-iter*, *mk-digest-salt* and *mk-IV*. It can be divided into three parts; the first one populates the simple fields of the *E-LUKS header* as: the random value for the salt of the KDF, the constant value of the magic field, the count value, and the initial value. The second part generates the KDF of the master key with the salt and count parameters. Lastly, the third part generates the HMAC digest of all the encrypted user data.

4.3.2. *Add New Password*

The most significant change in this operation with respect to LUKS is the exclusion of the anti-forensic functions. Therefore, there is no need to calculate the split key.

The *Add New Password* operation requires the master key, the count iteration value for the KS_x, and the user key. This operation begins by setting the selected KS_x to activate. Doing so, random values start to generate for the salt and Initial Value fields. Once all the values have been generated for the KS_x, the user key is processed by the KDF given the user key digest. Then, the master key will be divided into chunks the size of the block cipher size. Each chunk will be encrypted with the user key digest and is concatenated to create the KM_x.

4.3.3. *Master Key Recovery*

It is similar to the *Add New Password operation*; the main difference is the exclusion of the anti-forensic functions.

The *Master Key Recovery* operation requires the user key. First of all, it performs a search for each KS_x. If the activate field is set, then it will create the user key digest with the KDF, using the parameters of the KS_x. Next, it divides the encrypted password into chunks the size of the block cipher; each chunk is decrypted using the user key digest, and each part is concatenated into the master key candidate. This master key candidate is then processed by the KDF with the parameters from the *E-LUKS header* and is compared with the *mk-digest*. If both values are equal, then the master key candidate is returned.

4.3.4. *HMAC Verification*

The *HMAC verification* is a new operation for E-LUKS, which requires the master key. Once the HMAC is initialised with the master key, the user data are then divided into chunks and are fed to the HMAC. When all the data have been fed, the HMAC generates a digest, which is compared with the *mk-hmac* stored in the *E-LUKS header*.

This operation allows for the integrity and authentication of the user data. Therefore, it brings out the possibility of creating a secure boot for the device in which E-LUKS is running.

4.3.5. *Password Revocation*

The *Password Revocation* is the last operation, an exact duplicate of its counterpart in LUKS.

4.4. Comparison between LUKS and E-LUKS

This subsection shows the difference between LUKS and E-LUKS, as shown in Table 10.

Table 10. Properties of LUKS and E-LUKS.

		LUKS	**E-LUKS**
Cryptography	block ciphers	aes, twofish, serpert, cast5, cast6	PRESENT
	block ciphers mode	ecb, cbc-plain, cbc-essiv:*hash*, xts-plain64	ctr
	hash functions	sha1, sha256, sha512, ripemd160	SPONGENT-88
	KDF	Password-Based Key Derivation Function 2 (PBKDF2)	KDF [39]
	HMAC	-	based on SPONGENT-88
Layout	header size (bytes)	208	52
	KS_x size (bytes)	48	48
	phdr size (bytes)	596	512
Operations	Initialisation	yes	yes
	Add New Password	yes	yes
	Master Key Recovery	yes	yes
	HMAC verification	no	yes
	Password Revocation	yes	yes

Before starting to describe the hardware implementation of E-LUKS, Table 11 shows the key differences between LUKS and E-LUKS. LUKS allows more security against brute-force attacks, but lacks integrity and authentication of the data. Therefore, while LUKS cannot provide a mechanism of secure boot, it is possible for E-LUKS.

Table 11. LUKS and E-LUKS comparison.

	LUKS	**E-LUKS**
confidentiality	yes	yes
integrity	no	yes
authentication	no	yes
max. cipher key lenght (bits)	256	80
max. cipher block lenght (bits)	128	64
max. digest lenght (bits)	512	88

4.5. Hardware Implementation

E-LUKS has been implemented in SystemVerilog, the schematic of which is shown in Figure 3. The E-LUKS core has four main parts, three of them related to the aforementioned cryptographic algorithms described earlier: the PRESENT cipher core, the HMAC based on SPONGENT, and the KDF. The last part is the control module, a finite state machine, which implements the Master Key Recovery operation and the *HMAC verification* operation.

In addition, the control module has an SPI interface to communicate with the formatted E-LUKS memory.

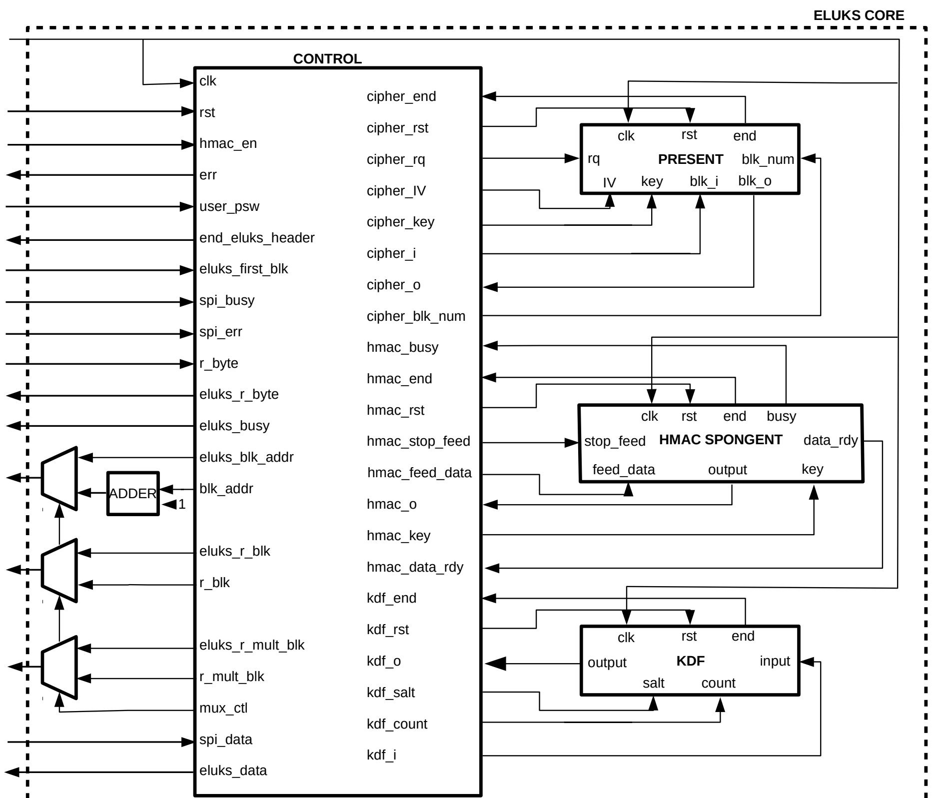

Figure 3. E-LUKS core schematic.

The flow chart that represents the *Master Key Recovery* operation performed by the control module is shown in Figure 4.

The operation begins with the reset signal. After that, the control module resets all the internal counters and registers and proceeds to read from the memory the first block of the E-LUKS partition. Then, it stores all the fields from the *E-LUKS header*. Although some of the fields are not used in this operation, they will be used later by other operations (*HMAC verification* or reading the encrypted user data from the E-LUKS partition). After that, the *magic* field is compared with the `E-LUKS_ID`, which identifies it as an E-LUKS partition. However, in another case, it reaches the *error state* from which the error signal is raised, and finishing the operation. Following the operation, the control module proceeds to read each key slot KS_x in order, checking for each one if the *activate* field is set. If all KS_x are deactivated, the control module goes to the *error state*. If the KS_x is activated, it continues to decrypt the *pwd-encrypted* field. First, it must calculate the key to decrypt, done by calculating the KDF of the *user_psw*, provided as an input of the E-LUKS core. In addition, the KDF takes the *salt* and *iterations* fields related to the KS_x. Once with the key, it is able to decrypt the *pwd-encryted* getting a master key candidate. To verify this candidate, it generates the KDF of the candidate with the fields *mk-digest-salt* and *mk-digest-iter*. Finally, the calculated digest is compared against the field *mk-digest*. If both values are equal, then the candidate is the master key and is returned. Otherwise, the control module goes to the *error state*.

Regarding the *HMAC verification* operation, this is only performed if the *hmac-enable* signal is activated. A flow chart of this operation is shown in Figure 5. The *HMAC verification* operation takes place after a successful *Master Key Recovery* operation. Therefore, it is assumed that the required fields from the *E-LUKS phdr* have been precisely stored. The operation begins by initializing the HMAC with the master key. Then, it reads the encrypted user data that begins in the second block of the E-LUKS partition. It reads

all data byte by byte, each of which feeds the HMAC. When all user data is processed, the control module calculates the HMAC output. Finally, the digest generated by the HMAC is compared to the *mk-hmac* field. If both values are equal, then the user data are proven to be authenticated and unmodified. Otherwise, the control module goes to the *error state* described earlier in the *Master Key Recovery* operation.

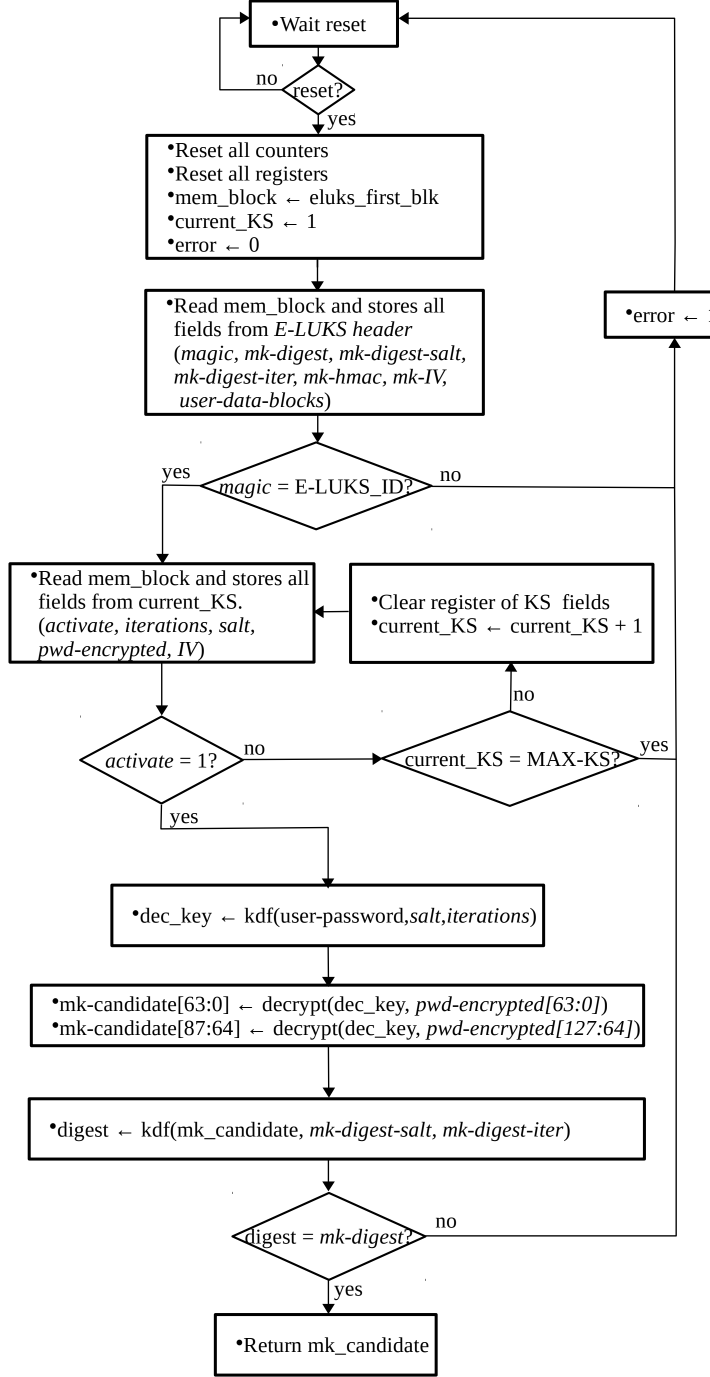

Figure 4. Flow diagram of the Master Key Recovery operation.

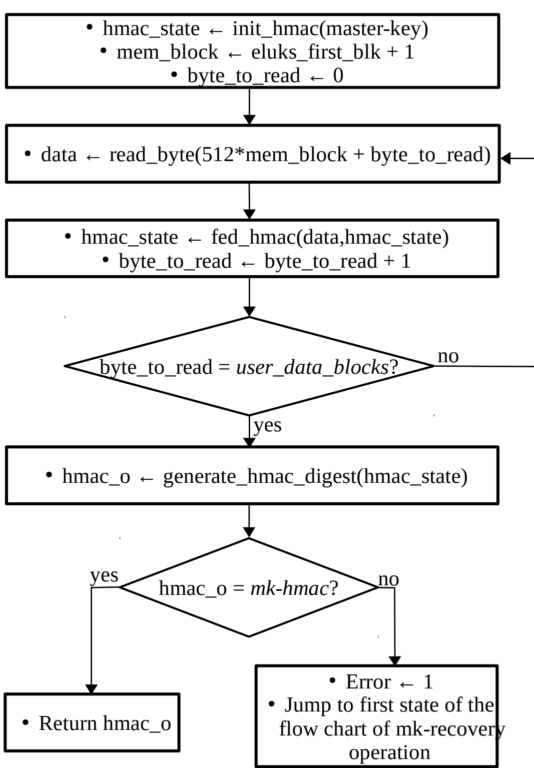

Figure 5. Flow diagram of the HMAC verification operation.

5. Results

In order to validate the proposed solution, an experiment has been conducted that provides a functional verification of E-LUKS and execution times with and without it. These resource results are then compared to the previous solution shown in Related Work.

The experiment has been designed using the SystemVerilog Hardware Description Language (HDL) and is implemented on a Nexys4DDR development board from Digilent Inc. that has an XC7A100T FPGA chip from Xilinx Inc. [40]. The selected development environment is Vivado 2020.1 from Xilinx Inc. [41]. This board features a microSD card slot.

A microSD card is the system's storage device. It is divided into two parts: the first part contains the E-LUKS partition, which stores a target file as the encrypted user data, and the second part contains the unencrypted target file.

The main goal of the experiment is to obtain the execution times when reading the target file in different ways: (1) From inside an E-LUKS partition. It can be done either without the HMAC verification as with LUKS or with the HMAC verification to provide authentication and integrity of the data; (2) From the unencrypted memory area.

To perform the experiment, three different and independent tasks have been developed:

1. Functional verification: The first task begins by reading the unencrypted target file and storing it in a block RAM in the FPGA. Once the unencrypted target file is read, the E-LUKS partition is accessed in order to get the encrypted target file. The E-LUKS can be accessed with HMAC verification or not, depending on the *hmac-enable* signal attached to any switch from the board. For its part, the encrypted file is read and compared with the previously stored values. If all data are equal, then the functional verification is correct.
2. Reading the unencrypted target file: This task reads the unencrypted target file from the microSD card. The duration of this process is measured with an internal counter that gives a precise execution time for this task. The results are displayed in a 7-segments display on the board. This execution time excludes the initialisation time for the microSD card, which takes around 250 ms.

3. Reading the target file from the E-LUKS partition: This last task retrieves the target file from the E-LUKS partition both with or without HMAC verification, depending on the value of *hmac-enable*. It starts by reading the *phdr* from the microSD card to get the master key. Once the master key is retrieved, it is stored in a volatile memory accessible only to the E-LUKS core. Thus, it is only necessary to get the master key once. As in the previous task, an internal counter is used to calculate the execution time and display it on a 7-segments display skipping the microSD card initialisation time.

All the source code is available on the authors' github [42]. The files are ready to be used with the Fusesoc tool [43], which allows the reuse of previous designs and facilitates the creation of the bitfile. This repository has three folders:

1. hdl: Contains the SystemVerilog implementation of the E-LUKS core. In addition, it also holds the core in a format that can be used with the Fusesoc tool.
2. examples: Contains the files needed to replicate the proposed experiment. The files needed to create the bitfile are in the *fusesoc* folder. Whereas the file *create_partition.py* in the *python* folder is a script that generates the layout needed in this experiment for a microSD card.
3. cores: Contains a list of Fusesoc cores used in this experiment taken from previous designs.

In the first task, the file from the unencrypted partition is read and compared with the file read from the E-LUKS encrypted partition, both with HMAC and without HMAC verification. If the comparison is successful, E-LUKS is proven to be functionally valid.

The second and third tasks are intended to obtain the execution times of the reading operations. Different sizes have been used for the target file: 4 KB, 8 KB, and 16 KB. In addition, the count value stored in the *E-LUKS header* and the KS_x, is also variable. The count value increases the security by stretching the original password length (88 bits) with $\log_2$ *count-value* [39]. Therefore, the total password length obtained for the different count values is 93-bit for 32, 94-bit for 64, and 95-bit for 128.

The execution time results from the experiment are shown in Table 12. These results, as expected, indicate that the execution time increases when the target file size or the count value increases. It is also observed that the HMAC verification is the worst case scenario for execution time, but the best case scenario from the security perspective. Therefore, in order to provide more security to the system, the payoff is the execution time. In addition, it is noticed that the percentage of time needed for the E-LUKS, compared with the unencrypted file, is decreased with larger files (+418.8% for 4 KB, +310.3% for 8 KB, and +257.1% for 16 KB).

Regarding the resource taken from the FPGA. E-LUKS has been designed to occupy the least amount of resources. To get a better perspective, the resources' results have been compared with previous solutions. To this end, the E-LUKS core has been synthesized in different FPGAs, XC7Z020 [44] and XC6VLX240T [45]. The results can be divided in two groups: (1) results for solutions that are not specifically for resource constrained devices, and (2) results for solutions that are designed for these devices.

In the first group are the LUKS hardware implementations in [18,19], already presented in the Related Work Section. Its results, shown in Table 13, are an order of magnitude above E-LUKS. It is remarkable that E-LUKS takes less than 90% of the resources when compared to these LUKS hardware implementations. In Figure 6, it is observed that the best LUKS results are those of [18], which takes around 50% of Slice LUTs and 30% of Slice Registers from the FPGA. However, E-LUKS only takes about 5% of Slice LUTs and 3% of Slice Registers. Therefore, E-LUKS proves to be better suited to performing FDE in resource constrained devices which seek to have a small footprint over time execution. In this regard, the LUKS hardware solutions offer better performance due to their pipeline implementation.

The second group is about hardware solutions specifically for resource constrained devices. It contains Sancus [20], Soteria [22], Atlas [24], and [23]. The results are shown

in Table 14 and Figures 7 and 8. All the solutions, except [23] when using authentication with the AE cipher ASCON, utilize less than 10% of any kind of resources for the FPGA. Therefore, these solutions are an adequate fit for resource constrained devices. When E-LUKS is compared to [23], it is observed that E-LUKS presents a slight improvement. Specially, when both designs are used to provide confidentiality, integrity, and authentication, whereas [23] takes up to four times more Slice LUTs. Regarding Atlas, it presents better results in Slice LUTs, 0.7% against 1.8%. However, it has worse results in number of Slices, 2.5% against 6.5%, and Slice Registers, 0.9% against 1.8%. Atlas provides confidentiality, although it lacks integrity and authenticity. Lastly, there are Sancus and Soteria. Both solutions have almost identical results, being the best suited, although it only shows when 1 SM is on the device. The overhead for an additional SM is 0.13% of Slice LUTs, and 0.013% of Slice Registers. E-LUKS takes about two times more Slice Registers and three times more Slice LUTs. However, these values represent less than 2% of both resources. With 10 SM, the Slice LUTs of E-LUKS and both solutions offer similar results. Therefore, to fit a few programs, Sancus and Soteria are the best solutions. However, to accommodate a large number of files, FDE is a better fit. Hence, to accommodate a large number of files, enough memory is required. In these kinds of devices, the storage for large files is usually a flash removable device, such as a microSD card. These memories are the target of E-LUKS, instead of the RAM memory. This leads to the advantage of being independent of any processor. It only needs to implement a bridge to the system bus, as well as [23].

Table 12. Time results for the XC7A100T FPGA.

File Size	Count Value	Unencrypted (Time ms)	E-LUKS Encrypt (Time ms)	E-LUKS Encrypt + HMAC (Time ms)
4 KB	32	1.81 (+0.0%)	4.16 (+128.9%)	6.8 (+275.7%)
	64	1.81 (+0.0%)	5.15 (+184.5%)	7.40 (+308.8%)
	128	1.81 (+0.0%)	7.15 (+295.0%)	9.39 (+418.8%)
8 KB	32	3.41 (+0.0%)	6.75 (+97.9%)	10.77 (+215.8%)
	64	3.41 (+0.0%)	7.75 (+127.3%)	11.76 (+244.9%)
	128	3.41 (+0.0%)	9.74 (+185.6%)	13.99 (+310.3%)
16 KB	32	6.58 (+0.0%)	11.89 (+80.7%)	19.90 (+202.4%)
	64	6.58 (+0.0%)	12.89 (+95.9%)	21.13 (+221.1%)
	128	6.58 (+0.0%)	14.88 (+126.1%)	23.50 (+257.1%)

Table 13. Resources comparison with LUKS solutions for the XC7Z020 FPGA.

Core	FPGA	Slice LUTs	Slice Registers	BRAMs
E-LUKS	XC7Z020	2672	2732	1
LUKS [18]	XC7Z020	28068	29866	36
LUKS [19]	XC7Z020	41656	66447	22

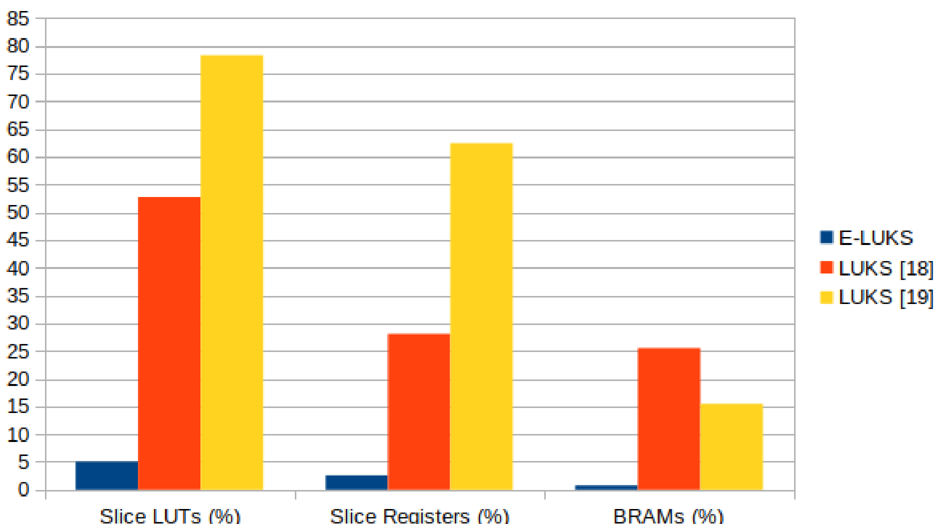

Figure 6. Percentage resources of E-LUKS and LUKS solutions for the XC7Z020 FPGA.

Table 14. Resources comparison with resource-constraint solutions for the XC6VLX240T FPGA, and XC7Z020 FPGA. The - values are not provided in the original papers.

Core	FPGA	Slice	Slice LUTs	Slice Registers	BRAMs
E-LUKS	XC7Z020	946	2672	2732	1
[23] without Authentication	XC7Z020	-	4735	2447	4.5
[23] with Authentication	XC7Z020	-	10214	4682	4.5
E-LUKS	XC6VLX240T	936	2724	2691	1
Sancus [20] with 1 *SM*	XC6VLX240T	-	751	1166	-
Soteria [22] with 1 *SM*	XC6VLX240T	-	792	1374	-
Atlas [24]	XC6VLX240T	2451	1025	5412	-

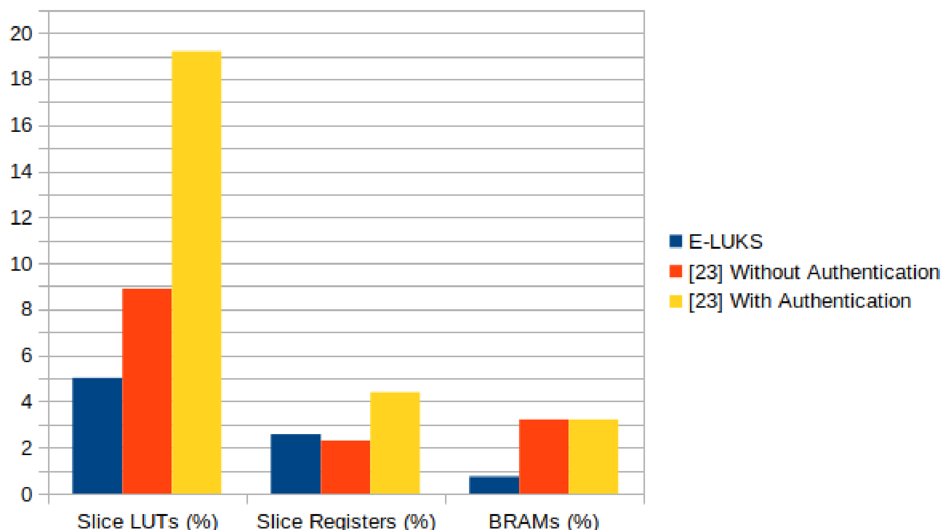

Figure 7. Percentage resources of E-LUKS and [23] solution for the XC7Z020 FPGA.

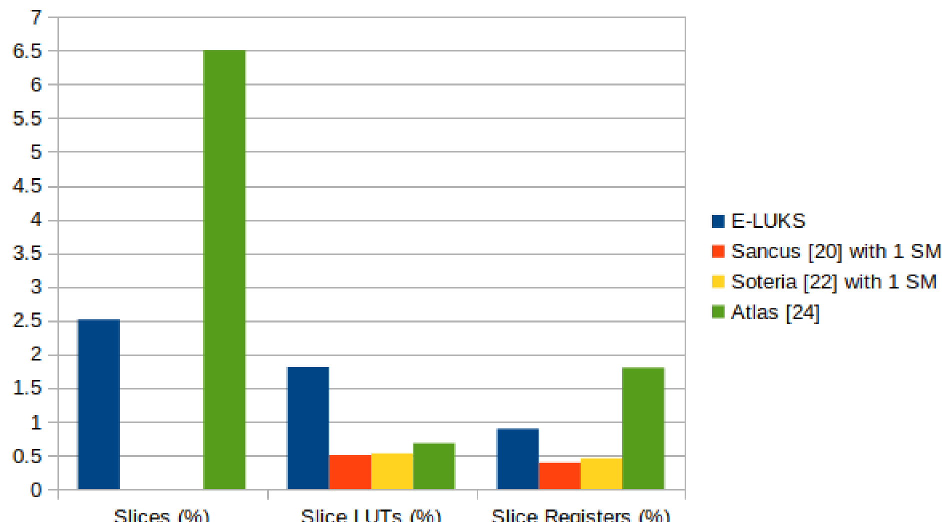

Figure 8. Percentage resources of E-LUKS and *Sancus*, *Soteria*, and *Atlas* solutions for the XC6VLX240T FPGA.

6. Conclusions

In this work, a hardware-solution able to perform FDE on resource constrained devices is presented. In addition, E-LUKS can authenticate and verify the integrity of the user data. The results presented show that E-LUKS takes few resources from the FPGA; less than 5%. However, there are other hardware-solutions, which provide security of the user data. There are the LUKS solutions, which, due to its pipeline implementation and the use of cryptographic algorithms, occupies ten times more resources, in the best of the scenarios, than E-LUKS. The Sancus and Soteria solutions have a smaller footprint than E-LUKS when working with few files. However, to handle a large amount of data, FDE could work better. These solutions need to implement specific instructions to the processor. E-LUKS, on the other hand, is processor agnostic, depending only on the bus system in order to create a bridge module. Atlas solution and E-LUKS have similar resource results. However, while Atlas only provides confidentiality, the work presented in [23], provides confidentiality, authentication, and integrity. Nonetheless, when providing, all three of them have a cost in terms of resources that is worse than E-LUKS, especially in Slice LUTs where it is almost three times greater. Additionally, in this scenario, Reference [23] needs part of the memory to store the TEC tree, which allows the authentication and integrity of the data. Hence, we believe that E-LUKS is an excellent choice for providing security to IoT devices. In those cases, the resources of the devices are the priority.

Our future work with E-LUKS involves Secure Boot, which we believe could be a perfect fit for this solution. Due to its agnostic processor characteristic and lack of software, it could be an excellent Secure Boot to IoT devices that take few resources and boot an embedded Linux image.

Author Contributions: Conceptualization, G.C.-Q., P.R.-d.-c.-V. and M.J.B.; methodology, G.C.-Q., P.R.-d.-c.-V. and M.J.B.; software, G.C.-Q. and P.R.-d.-c.-V.; validation, G.C.-Q., P.R.-d.-c.-V., M.J.B., D.G.-M., J.V.-C., J.J.-C. and E.O.-A.; formal analysis, G.C.-Q. and P.R.-d.-c.-V.; investigation, G.C.-Q., M.J.B. and P.R.-d.-c.-V.; resources, J.V.-C., J.J.-C., D.G.-M. and E.O.-A.; data curation, G.C.-Q. and P.R.-d.-c.-V.; writing—original draft preparation, G.C.-Q., P.R.-d.-c.-V. and M.J.B.; writing—review and editing, G.C.-Q., J.V.-C., J.J.-C., M.J.B., P.R.-d.-c.-V., E.O.-A. and D.G.-M.; visualization, G.C.-Q., P.R.-d.-c.-V. and M.J.B.; supervision, P.R.-d.-c.-V. and M.J.B.; project administration, J.J.-C. and P.R.-d.-c.-V.; and funding acquisition, J.J.-C. and P.R.-d.-c.-V. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partially supported by the Ministerio de Industria y Competitividad of Spain under project TIN2017-89951-P (BootTimeIoT) and by the European Regional Development Fund (ERDF).The author G.C.-Q. is economically supported by the VI-PPITUS.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Global Smart Shade Devices Market 2020–2024 | Emergence of IOT and AI-Based Smart Shade Devices to Boost Market Growth | Technavio | Business Wire. Available online: https://www.businesswire.com/news/home/20200131005322/en/Global-Smart-Shade-Devices-Market-2020-2024-Emergence (accessed on 14 May 2021).
2. Roaming Data Traffic Generated by Consumer & IoT Devices Forecast to Reach 2000 Petabytes in 2024: Kaleido Intelligence | Business Wire. Available online: https://www.businesswire.com/news/home/20200203005036/en/Roaming-Data-Traffic-Generated-Consumer-IoT-Devices (accessed on 14 May 2021).
3. Condry, M.W.; Nelson, C.B. Using Smart Edge IoT Devices for Safer, Rapid Response with Industry IoT Control Operations. *Proc. IEEE* **2016**, *104*, 938–946. [CrossRef]
4. IoT Medical Devices Market Growth Holds Strong; Key Players studied Medtronic, GE Healthcare, Philips Healthcare (Philips), Siemens—MarketWatch. Available online: https://www.marketwatch.com/press-release/iot-medical-devices-market-growth-holds-strong-key-players-studied-medtronic-ge-healthcare-philips-healthcare-philips-siemens-2020-01-30 (accessed on 20 May 2021).
5. Ferrer-Cid, P.; Barcelo-Ordinas, J.M.; Garcia-Vidal, J.; Ripoll, A.; Viana, M. Multi-sensor data fusion calibration in IoT air pollution platforms. *IEEE Internet Things J.* **2020**, *7*, 3124–3132. [CrossRef]
6. Gutierrez-Madronal, L.; La Blunda, L.; Wagner, M.F.; Medina-Bulo, I. Test Event Generation for a Fall-Detection IoT System. *IEEE Internet Things J.* **2019**, *6*, 6642–6651. [CrossRef]
7. Hamdan, O.; Shanableh, H.; Zaki, I.; Al-Ali, A.R.; Shanableh, T. IoT-Based Interactive Dual Mode Smart Home Automation. In Proceedings of the 2019 IEEE International Conference on Consumer Electronics, Las Vegas, NV, USA, 11–13 January 2019; pp. 1–2. [CrossRef]
8. Wazid, M.; Das, A.K.; Bhat K, V.; Vasilakos, A.V. LAM-CIoT: Lightweight authentication mechanism in cloud-based IoT environment. *J. Netw. Comput. Appl.* **2020**, *150*, 102496. [CrossRef]
9. Bodei, C.; Chessa, S.; Galletta, L. Measuring security in IoT communications. *Theor. Comput. Sci.* **2019**, *764*, 100–124. [CrossRef]
10. Meneghello, F.; Calore, M.; Zucchetto, D.; Polese, M.; Zanella, A. IoT: Internet of Threats? A Survey of Practical Security Vulnerabilities in Real IoT Devices. *IEEE Internet Things J.* **2019**, *6*, 8182–8201. [CrossRef]
11. binti Mohamad Noor, M.; Hassan, W.H. Current research on Internet of Things (IoT) security: A survey. *Comput. Netw.* **2019**, *148*, 283–294. [CrossRef]
12. Neshenko, N.; Bou-Harb, E.; Crichigno, J.; Kaddoum, G.; Ghani, N. Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations. *IEEE Commun. Surv. Tutor.* **2019**, *21*, 2702–2733. [CrossRef]
13. Frustaci, M.; Pace, P.; Aloi, G.; Fortino, G. Evaluating critical security issues of the IoT world: Present and future challenges. *IEEE Internet Things J.* **2018**, *5*, 2483–2495. [CrossRef]
14. Fruhwirth, C.; Broz, M. LUKS1 On-Disk Format Specification. 2018. Available online: https://gitlab.com/cryptsetup/cryptsetup/-/wikis/LUKS-standard/on-disk-format.pdf (accessed on 10 March 2021).
15. Shwartz, O.; Mathov, Y.; Bohadana, M.; Elovici, Y.; Oren, Y. Reverse Engineering IoT Devices: Effective Techniques and Methods. *IEEE Internet Things J.* **2018**, *5*, 4965–4976. [CrossRef]
16. Ling, Z.; Luo, J.; Xu, Y.; Gao, C.; Wu, K.; Fu, X. Security Vulnerabilities of Internet of Things: A Case Study of the Smart Plug System. *IEEE Internet Things J.* **2017**, *4*, 1899–1909. [CrossRef]
17. Manos, A.; Tim, A.; Michael, B.; Matt, B. Understanding the mirai botnet. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 1093–1110.
18. Li, X.; Cao, C.; Li, P.; Shen, S.; Chen, Y.; Li, L. Energy-efficient hardware implementation of LUKS PBKDF2 with AES on FPGA. In Proceedings of the 2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, 23–26 August 2016; pp. 402–409. [CrossRef]
19. Li, X.; Wu, K.; Zhang, Q.; Lin, S.; Chen, Y.; Wong, S.Y. A High Throughput and Pipelined Implementation of the LUKS on FPGA. *J. Circuits Syst. Comput.* **2020**, *29*, 2050075. [CrossRef]
20. Noorman, J.; Agten, P.; Daniels, W.; Strackx, R.; Van Herrewege, A.; Huygens, C.; Preneel, B.; Verbauwhede, I.; Piessens, F. Sancus: Low-Cost Trustworthy Extensible Networked Devices with a Zero-Software Trusted Computing Base. In Proceedings of the 22nd USENIX Conference on Security, Washington, DC, USA, 14–16 August 2013; pp. 479–494.
21. Strackx, R.; Piessens, F.; Preneel, B. Efficient Isolation of Trusted Subsystems in Embedded Systems. In Proceedings of the Security and Privacy in Communication Networks, Singapore, 7–9 September 2010; Volume 50, pp. 344–361. [CrossRef]
22. Götzfried, J.; Müller, T.; De Clercq, R.; Maene, P.; Freiling, F.; Verbauwhede, I. Soteria: Offline software protection within low-cost embedded devices. *ACM Int. Conf. Proc.Ser.* **2015**, *7*, 241–250. [CrossRef]

23. Werner, M.; Unterluggauer, T.; Schilling, R.; Schaffenrath, D.; Mangard, S. Transparent memory encryption and authentication. In Proceedings of the 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, 4–8 September 2017; pp. 1–6. [CrossRef]
24. Maene, P.; Götzfried, J.; Müller, T.; de Clercq, R.; Freiling, F.; Verbauwhede, I. Atlas: Application Confidentiality in Compromised Embedded Systems. *IEEE Trans. Dependable Secur. Comput.* **2019**, *16*, 415–423. [CrossRef]
25. Brož, M. *LUKS2 On-Disk Format Specification Version 1.0.0 Document History*; LUKS: Hong Kong, China, 2018; pp. 1–16.
26. rfc8018. *PKCS #5: Password-Based Cryptography Standard v2.1*; RSA Laboratories: Hebron, CT, USA, 2017.
27. Dworkin, M.; Barker, E.; Nechvatal, J.; Foti, J.; Bassham, L.; Roback, E.; Dray, J. *Advanced Encryption Standard (AES)*; NIST: Gaithersburg, MD, USA, 2001. [CrossRef]
28. Schneier, B.; Kelsey, J.; Whiting, D.; Wagner, D.; Hall, C. Twofish: A 128-Bit Block Cipher. *NIST AES Propos.* **1998**, *15*, 1–27. [CrossRef]
29. Anderson, R.; Biham, E.; Knudsen, L. Serpent: A proposal for the advanced encryption standard. *NIST AES Propos.* **1998**, *174*, 1–23.
30. Adams, D.C. *The CAST-128 Encryption Algorithm*; RFC 2144; Network Working Group. 1997. Available online: https://www.rfc-editor.org/rfc/rfc2144.txt (accessed on 15 April 2021).
31. Adams, D.C.; Gilchrist, J. *The CAST-256 Encryption Algorithm*; RFC 2612; Network Working Group. 1999. Available online: https://www.rfc-editor.org/rfc/rfc2612.txt (accessed on 15 April 2021).
32. Eastlake, D., 3rd; Jones, P. *US Secure Hash Algorithm 1 (SHA1)*; RFC 3174; Network Working Group. 2001. Available online: https://www.rfc-editor.org/rfc/rfc3174.txt (accessed on 15 April 2021).
33. Dang, Q. *Secure Hash Standard (SHS)*; NIST: Gaithersburg, MD, USA, 2012. [CrossRef]
34. Dobbertin, H.; Bosselaers, A.; Preneel, B. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption*; Gollmann, D., Ed.; Springer: Berlin/Heidelberg, Germany, 1996; pp. 71–82.
35. Pandey, J.G.; Goel, T.; Karmakar, A. Hardware architectures for PRESENT block cipher and their FPGA implementations. *IET Circuits Devices Syst.* **2019**, *13*, 958–969. [CrossRef]
36. Bogdanov, A.; Knudsen, L.R.; Leander, G.; Paar, C.; Poschmann, A.; Robshaw, M.J.B.; Seurin, Y.; Vikkelsoe, C. PRESENT: An Ultra-Lightweight Block Cipher. In Proceedings of the Cryptographic Hardware and Embedded Systems—CHES 2007, Vienna, Austria, 10–13 September 2007; Paillier, P., Verbauwhede, I., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 450–466.
37. Bogdanov, A.; Knežević, M.; Leander, G.; Toz, D.; Varıcı, K.; Verbauwhede, I. SPONGENT: A Lightweight Hash Function. In Proceedings of the Cryptographic Hardware and Embedded Systems—CHES 2011, Nara, Japan, 28 September–1 October 2011; Preneel, B., Takagi, T., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 312–325.
38. Bellare, M.; Canetti, R.; Krawczyk, H. Keying Hash Functions for Message Authentication. In Proceedings of the Advances in Cryptology — CRYPTO '96, Santa Barbara, CA, USA, 18–22 August 1996; Koblitz, N., Ed.; Springer: Berlin/Heidelberg, Germany, 1996; pp. 1–15.
39. Yao, F.F.; Yin, Y.L. Design and Analysis of Password-Based Key Derivation Functions. In Proceedings of the Topics in Cryptology—CT-RSA 2005, San Francisco, CA, USA, 14–18 February 2005; Menezes, A., Ed.; Springer: Berlin/Heidelberg, Germany, 2005; pp. 245–261.
40. Xilinx. *7 Series FPGAs Data Sheet: Overview (DS180)*; Xilinx Inc.: San Jose, CA, USA, 2010.
41. Xilinx Inc. *Vivado Design Suite User Guide: UG973*; Xilinx Inc.: San Jose, CA, USA, 2020.
42. GitHub—Germancq/ELUKS. Available online: https://github.com/germancq/ELUKS (accessed on 25 October 2021).
43. GitHub—olofk/Fusesoc: Package Manager and Build Abstraction Tool for FPGA/ASIC Development. Available online: https://github.com/olofk/fusesoc (accessed on 8 October 2021).
44. Xilinx Inc. *Zynq-7000 AP SoC Technical Reference Manual*; Xilinx Inc.: San Jose, CA, USA, 2021.
45. Xilinx Inc. *Virtex-6 Family Overview Summary of Virtex-6 FPGA Features*; Xilinx Inc.: San Jose, CA, USA, 2015.