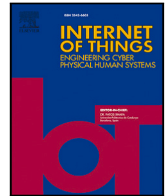
Internet of Things 23 (2023) 100874

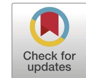
Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot



Research article



IRIS: An embedded secure boot for IoT devices

G. Cano-Quiveu *, P. Ruiz-de-Clavijo-Vazquez, M.J. Bellido, J. Juan-Chico, J. Viejo-Cortes

Departamento de Tecnologia Electronica, Universidad de Sevilla, Av. Reina Mercedes s/n Sevilla, 41012, Andalucia, Spain

ARTICLE INFO

Dataset link: https://github.com/germancq/ELUKS/tree/main/examples/bootloader

Keywords:
Field programmable gate array
Secure boot
Internet of Things (IoT)
Bootloader
Hardware
Embedded device

ABSTRACT

This study proposes a hardware secure boot solution, an instant retrieval information system (IRIS) that is suitable for integrating Internet of Things (IoT) devices. IRIS can boot a Linux kernel image pre-stores in removable media and comprises a data verifier securing the authenticity, integrity, and confidentiality of the boot process. IRIS is fully developed as a hardware module and the results reveal short boot-up times and a small hardware footprint when implemented on field programmable gate array chips. In addition, IRIS is an open-source generic solution that can be adapted to multiple architectures and includes a crypto-core called E-LUKS that can be used outside the boot-loading process to add confidentiality, integrity, and authenticity to data stored on off-chip storage like a flash device.

IRIS shows a reduction in lookup tables footprint when compared to other IoT solutions consuming from 90% to 750% less resources and only slightly greater, around 30%, with solutions that only cover authentication and integrity.

1. Introduction

The significant growth in connected Internet of Things (IoT) devices in recent years increased attacks on these types of devices. Security is a priority in environments where these devices are part of critical systems or deal with private data. Security must be addressed at all levels, from the hardware to the full software stack. The link between hardware and software is the bootloader which is the first piece of software executed on the device. Therefore, to secure a device, the bootloader is a critical part to be secured in the software stack.

There is a wide range of IoT devices that can be broadly classified into two types depending on the underlying hardware: a microcontroller unit or a system on chip (SoC). Both include a bootloader, but the software executed in each case differs because the former usually executes standalone software, while the latter starts an operating system (OS). In the last scenario, the target of the bootloader is to launch the OS kernel; however, it is often impossible to fetch it in a single stage because of its size. The solution adopted in this case is to split the bootloading process into multiple software stages, creating a bootloader chain. The first stage is commonly called *preloader* and is launched after the SoC initializes a minimal hardware set such as peripherals, on-chip memories, debug interfaces, etc [1].

Current SoCs store the full bootloader chain in off-chip flash memory (on-board flash chip or removable media). Hence, it can be tampered with or altered with appropriate tools. In this situation, attacks typically involve altering the flash contents to change parts of the bootloader chain, inducing the malfunctioning device to execute malicious code. To avoid these types of attacks, a procedure called *secure boot* [2] aims at protecting the data and/or code involved in the bootloader chain.

* Corresponding author.
E-mail address: germancq@dte.us.es (G. Cano-Quiveu).

https://doi.org/10.1016/j.iot.2023.100874
Received 22 December 2022; Received in revised form 6 June 2023; Accepted 6 July 2023
Available online 20 July 2023
2542-6605/© 2023 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

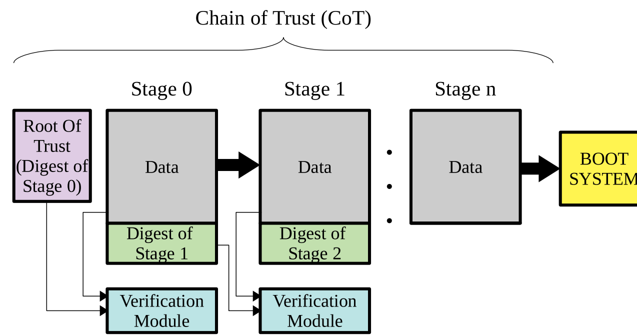


Fig. 1. General overview of the secure boot process.

The secure boot process comprises a chain of stages where the previous stage verifies the next stage before launching it. A general overview is shown in Fig. 1. The component called the *root of trust* (RoT) is considered trusted by default and is usually an immutable preloader stored in an on-chip ROM. Each stage attaches a digest of its data to be used in a verification procedure run before starting.

This verification ensures the stage integrity of the bootloader chain using a *hash-based message authentication code* (HMAC), and is called *chain of trust* (CoT). In addition to integrity, authentication is achieved by encrypting the digest. In addition, secure boot can also support confidentiality by encrypting the data of the stages and preventing them from being read by unauthorized users.

In current IoT scenarios, most devices are small and limited in available resources; hence, complex boot solutions like the *unified extensible firmware*(UEFI) [3] cannot be applied, making secure boot on IoT devices require specific solutions in most cases. This, together with the fact that it is practical to place the RoT in immutable on-chip hardware, makes custom hardware solutions specially suitable for secure boot in IoT devices.

Secure boot implementations such as those proposed in UEFI support only integrity and authentication, but using the *Linux unified key system* (LUKS) [4], it is possible to add confidentiality to the boot process when the Linux kernel is used [5].

This contribution proposes a custom secure boot hardware called an instant retrieval information system (IRIS). It is focused on SoC devices with low footprints, allowing them to boot a Linux OS securely. IRIS guarantees confidentiality, integrity, and authentication when booting a Linux kernel image stored in off-chip memory.

The paper is organized as follows. The next section summarizes the most relevant related work. Section 3 details the IRIS solution. Section 4 shows the results of the implementation on field programmable gate arrays (FPGAs), the execution metrics, and comparisons with related solutions. Finally, the main conclusions are summarized in Section 5.

2. Related work

Currently, multiple secure boot designs have been proposed with features such as authentication, confidentiality, and integrity [6–8].

These three features are valuable against different types of attacks. For example, authenticity and integrity are needed to validate the origin of the data and ensure that the data have not been modified by a third party. These two properties can prevent attacks against firmware modifications such as those presented in [9]. Regarding confidentiality, it is common to customize the boot data or the kernel image to include personal data or private modifications that must be protected. Encrypting the data can avoid reverse engineering attacks on the firmware extracted from the device [10].

In [6], a full-hardware secure boot is proposed to run in a programmable system-on-chip (PSoC) to support authenticity, confidentiality, and integrity. This design is divided into two parts: the *programmable logic region* (PL Region) that has an FPGA and the *processing system region* (PS Region) that has an ARM processor. Secure boot is implemented in the PL region, which grants authentication based on the factory's unique IDs from the *non-volatile memory* (NVM) and the FPGA. Confidentiality is achieved by using a symmetric cipher that encrypts the image, and to achieve integrity, it uses a hash function to process all the data. However, the cryptographic algorithms included, such as AES, are not the optimal solution for mid- and low-range devices, as is reflected in the FPGA resources utilization results, which are very high.

Another solution relevant to this paper is called ITUS; it is presented in [8] and is a complete secure solution with multiple modules. One of them is a module implementing a secure boot CoT where each stage is signed and verified by a hardware core. ITUS is tested over a high-range of FPGA chips because the design has a mid-footprint and is mainly focused on security against quantum computation attacks. Hence, it makes an effort to enhance integrity using algorithms such as the *elliptic curve* digital signature or the *extended Merkle signature* scheme, but these algorithms are not good approaches to resource-constraint devices as typically found in the IoT domain.

Another solution introduced in [7] called CARE mixes hardware and software. The CARE design provides authenticity and integrity, adding a recovery engine that can boot a pre-stored secure image. The hardware module comprises a data verifier securing the authenticity and integrity of an image stored in flash memory. The verification procedure divides the image into chunks with an attached signed digest. Then, the module checks the flash image chunk by chunk. Furthermore, the CARE design does not achieve confidentiality and is not a full-hardware design because only the HMAC module is implemented on hardware.

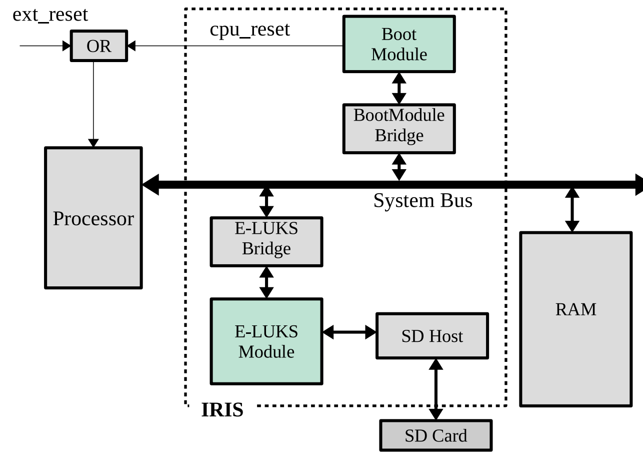


Fig. 2. Schematic of the IRIS subsystem.

3. IRIS secure boot

3.1. IRIS overview

IRIS is a hardware subsystem that attempts to add secure boot functionality to IoT devices. It can be integrated into a device to provide confidentiality, integrity, and authenticity to the boot process. The main features of IRIS are summarized as follows.

- It has a low hardware footprint for resource-constrained devices.
- It is a generic secure boot solution for embedded devices, and it is easy to adapt to different processors and buses.
- The design has an open-source license for easier integration and re-usability in other projects.

When the IRIS subsystem is integrated into an SoC, only a few parts must be adapted to the specific system: mainly the parts connecting to the internal processor bus. Once integrated, the IRIS subsystem will take control of the first steps of the boot process and retrieve a Linux kernel image from a storage device, decrypt and verify it, and load it into RAM such that the processor can run it only if the verification step has succeeded. The encrypted Linux kernel image may be stored in off-chip flash memory or removable media.

IRIS has a modular design with multiple components. Fig. 2 depicts a typical SoC configuration with the IRIS component adapted to the processor's internal bus and an SD card acting as removable storage media.

The main parts of IRIS are the *Boot Module* and the *E-LUKS Module*. Both are reusable. Depending on the system bus the SoC uses, the interface of the *E-LUKS Module* to the bus (the *E-LUKS Bridge*) may have to be adapted as the internal interface to the bus in the *Boot Module*. Rewriting this connection to the system bus in each module (*E-LUKS* and *Boot Module*) allows IRIS to boot a Linux kernel stored in the storage device. The current implementation of IRIS uses an SD card as storage media accessed by an *SD host* reader module.

The *Boot Module* is the main core governing the boot process of the system by taking control of all components connected to the processor's internal bus. The *E-LUKS Module* [11] supports data security and verification. It provides confidentiality, integrity, and authenticity to the data stored in the off-chip storage media during the boot process, as detailed in Section 3.4.

The confidentiality is accomplished by encrypting the Linux kernel image stored in the external memory (SD card). These encrypted data have attached the result of an HMAC function, which is encrypted with the same key used for the data. This ensures data integrity and authenticity, protecting the boot-up process against data tampering, generation, or corruption.

The next subsections will be explain the boot-up procedure executed by IRIS. Then the two main cores of IRIS, *Boot Module* and *E-LUKS*,will be detailed.

3.2. IRIS procedure

The IRIS secure boot procedure starts after an SoC power-on or hard-reset. It begins with the *Boot Module* asserting the processor's reset signal to take control of the internal bus until the secure boot is completed. The boot procedure requires secret parameters to initialize the *E-LUKS Module*. They are preloaded in the *Boot Module*, which is in charge of initializing the *E-LUKS Module*. These parameters are:

- *PSW*: user password to decrypt and verify the E-LUKS boot partition with a maximum size of 80 bits.
- *START*: a 32-bit number with the logical block of flash memory from which the E-LUKS partition starts.

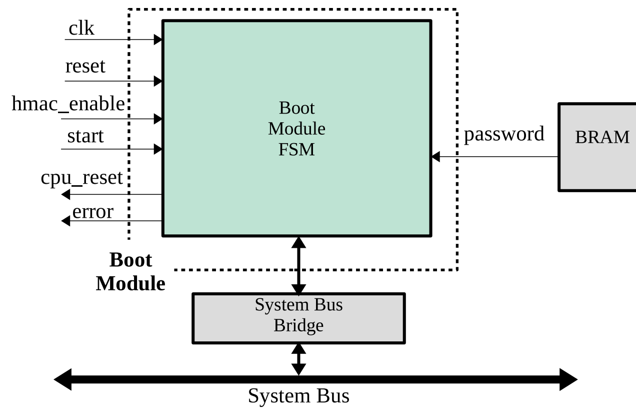


Fig. 3. Schematic of the *Boot Module* subsystem.

- *INIT-BLOCK*: the relative offset within the E-LUKS partition where the Linux kernel image is stored.
- *HMAC-ENABLE*: a flag to activate or deactivate integrity and authenticity checks.

Once the *E-LUKS Module* is configured, the *Boot Module* waits for a response containing the total number of bytes of the E-LUKS partition or an error signal. An error could occur in multiple situations:

- No valid E-LUKS header is present in the partition at offset *START*.
- The HMAC digest verification fails.

When all checks are correct, the *E-LUKS Module* reads a data stream from the SD card, decrypts , and sends it to the *Boot Module*. The *Boot Module* copies the data received into RAM and finally de-asserts the processor's reset signal, leaving internal bus control to the processor, which will execute the first stage of the boot-up process.

3.3. Boot module

A full hardware module, the *Boot Module*, has been implemented to coordinate the boot process. This core has been created expressly for this purpose. This module is independent of the processor and acts as a central piece that activates or deactivates the different participating modules in the boot process.

A schematic of the *Boot Module* is shown in Fig. 3. It has a control signal to reset the processor (*cpu_reset*) and communicates with the system bus to reach the other components. The core of this module is its finite state machine (FSM), which, step by step, determines the activation order of the other modules.

First, the *Boot Module* waits for a start signal. Once the module has been activated, it halts the main processor activating its reset, which remains active until the *Boot Module* finishes its activity. Once the *Boot Module* takes control, it begins to send the required parameters to the *E-LUKS Module*: the password for the encrypted partition, the signal to enable or not the HMAC function of *E-LUKS* and finally in which logical block of the SD card begins the E-LUKS boot partition. When all the data have been transmitted through the system bus, the *Boot Module* requests chunks of data from the boot partition, which the *E-LUKS Module* sends back decrypted. Once the data reaches the *Boot Module*, it stores it in the RAM. This request process continues until all data has been copied to the RAM. At this point, the *Boot Module* deactivates the reset of the processor, finishing the boot process. All this process taken by the FSM is shown in Fig. 4.

The design choice to put the password into a separate BRAM and send it to the *Boot Module* instead of storing it in the *E-LUKS Module* is due to allowing the processor to use the *E-LUKS Module* with another partition on the same flash memory using a different password, as mentioned earlier. It makes the *E-LUKS Module* generic and it may remain available for other modules to use after the booting process.

3.4. E-LUKS

E-LUKS is a full-hardware module that supports the process to secure the data in the IRIS secure boot system. It has mainly been developed for IoT devices and works similarly to LUKS [4]. However, while LUKS is a software driver, E-LUKS is a hardware implementation.

E-LUKS provides confidentiality, integrity, and authenticity to user data stored in a partition. This partition has an unencrypted space containing the E-LUKS header, which includes information on the public cryptographic parameters used. It also has slots for different *users*, each with a unique password derived from the master key used to encrypt and verify the user data. When a user wants to retrieve data, they must provide its user password from which a candidate master key is generated. The digest of this

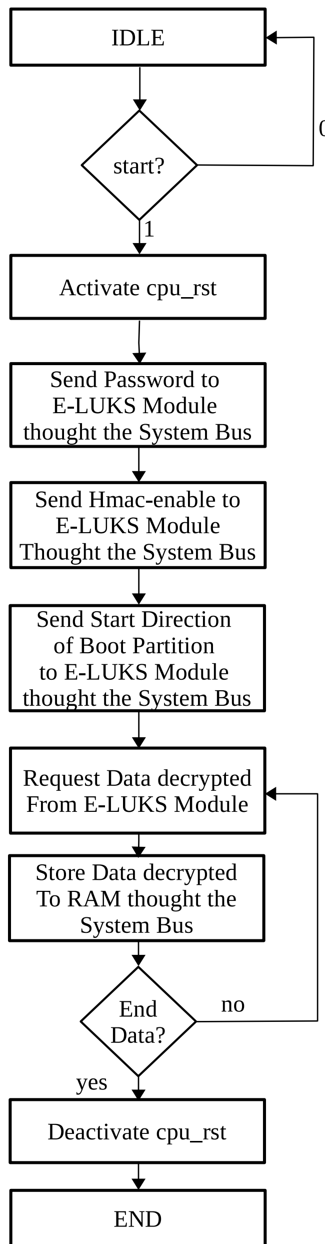


Fig. 4. Diagram of the finite state machine of the *Boot Module* subsystem.

master key is compared to a digest for the original master key stored in the partition. If the candidate is correct, the integrity and authenticity of the user data are checked. If all the data is correct, the user can retrieve the data.

E-LUKS is optimized to be easily integrated as part of an SoC and has some variations with respect to the original LUKS, the most relevant are:

- E-LUKS includes cryptographic algorithms specifically designed for IoT devices with limited resources: PRESENT [12], SPONGENT [13] and a variation of the *key derivation function* (KDF) introduced in [14].
- The KDF function takes three parameters; the parameter *count* is used to hold the number of iterations that must be performed before the KDF generates an output. This *count* parameter can be modified to add more security against brute-force attacks. It is assumed that the security achieved in this way is equivalent to having a key length of $80 + \log_2(count)$.
- E-LUKS adds a new operation to compute the HMAC of the entire encrypted user data. This operation ensures data integrity and authentication , which are essential to achieve a secure boot.

Table 1
IRIS vs. regular bootloader. FPGA resources utilization on the Xilinx Artix7 XC7A100T-1CSG324C.

Core	Slices		Flip flops		LUT's	
	No.	%	No.	%	No.	%
Regular Bootloader	**215**	**1.35**	**428**	**0.34**	**584**	**0.92**
SD host	121	0.76	254	0.20	316	0.50
Boot Module	94	0.59	174	0.14	268	0.42
IRIS	**1345**	**8.48**	**3301**	**2.60**	**3298**	**5.20**
SD host	108	0.68	199	0.16	306	0.48
Boot Module	90	0.57	168	0.13	243	0.38
E-LUKS	1147	7.23	2934	2.31	2749	4.34

- LUKS internal data structures are optimized in E-LUKS by reducing the number of internal fields and matching the logic block size of the physical layer. In the current implementation, the block size is 512 bytes to obtain a better performance on SD cards.

The *E-LUKS Module* is not exclusive to secure boot; it can also be used with software drivers in the OS to protect removable media partitions. In fact, E-LUKS was created to be able to store the boot data in a separate partition for the first booting stage, but it has another E-LUKS partition on the same flash memory with user data that the processor can access to execute a second stage of the boot process if necessary.

4. Results

In this section, two types of results are obtained. First, the utilization of IRIS resources and boot time performance are analyzed. Second, the utilization of IRIS core resources is compared with other solutions introduced in Section 2. The entire code used for this section can be accessed on the author's GitHub [15].

4.1. IRIS resources utilization and boot time performance

To validate the proposed solution, IRIS has been tested in an SoC deployed on the Digilent Nexys4-DDR prototype board, which has a Xilinx Artix7 XC7A100T-1CSG324C FPGA chip [16]. The SoC architecture tested corresponds to that depicted in Fig. 2 in the previous section. The SoC uses OpenRISC [17] as the processor and Wishbone [18] as the system bus. The system includes a removable SD card as boot media, containing an E-LUKS partition with an encrypted embedded Linux image of 6.3 MB.

To evaluate the impact of the secure boot features on the hardware resources needed and the boot time performance, another SoC has been implemented with secure boot functionality removed (a regular hardware boot system). With both the secure and regular boot systems, the test involves loading the Linux image into the RAM and launching it. However, in the regular boot case, the Boot Module fetches the Linux kernel directly from the SD host module and loads it into the RAM. This regular boot system does not include an *E-LUKS Module* or any boot security features.

Table 1 shows the FPGA resources needed by regular and the secure (IRIS) bootloaders. The total number of FPGA slices, flip-flops, and lookup-tables (LUT's) are included, together with the percentage for every resource with respect to the total chip's available resources. Numbers are also specified for the main components of the boot system: SD host, *Boot Module*, and *E-LUKS Module* (IRIS only).

The resources consumed by the SD host and *Boot Modules* are practically the same in both implementations. The small difference is due to small variations in the interfaces of the modules that are not exactly the same when the *E-LUKS Module* is not used.

Most resources in the IRIS system are consumed by *E-LUKS Module* (between 80% and 90% of the total resources consumed) because of the inherent complexity of the cryptographic algorithms implemented within the module. Nevertheless, resource usage is below 10% of the device's availability, making the IRIS solution perfectly viable for the platform used in this SoC.

Table 2 shows the resources taken by the full design. It also shows that the overcost of IRIS compared to *Regular Bootloader* is in the order of 25% more slices (27.14%), as is seen in the last row of the table. This overcost is natural because of the implemented *E-LUKS Module* which comprises the cryptographic algorithms. It can be considered that the manufacturing costs over ASICs could be increased by approximately 25%. Considering that the system is provided with safety during its boot-up and subsequently the *E-LUKS Module* can be used by the processor, the overcost associated with its manufacture can be assumed.

Regarding the impact of the secure boot functionality of IRIS on the boot-up time, Table 3 summarizes the boot-up times of the regular bootloader, the IRIS module with HMAC disabled (no integrity or authenticity), and the full IRIS module. The boot time overhead of the IRIS system compared to a regular bootloader is only 13% when integrity and authenticity are disabled, even if the boot data is still encrypted. The overhead is above 80% when the complete IRIS functionality is used and the boot process is fully secured.

As mentioned in the previous section, the IRIS secure boot supports different security levels by changing the *count* pre-programmed parameter of the KDF in the *E-LUKS Module*. This level of security is noticed in the robustness of bits of the system, which indicates on how much time is required against brute-force attacks. Therefore, the increase in bits is only relevant if the best attack against the system is brute-force attacks, which is the case with PRESENT, one of the algorithms recommended for

Table 2
IRIS vs. regular bootloader. FPGA resources utilization on the Xilinx Artix7 XC7A100T-1CSG324C, Entire SoC implementation.

Core	Slices		Flip flops		LUT's	
	No.	%	No.	%	No.	%
Regular Bootloader SoC	**3282**	**20.71**	**6527**	**5.15**	**9293**	**14.66**
OpenRISC	1411	8.90	2020	1.59	4233	6.68
DDR Controller	1546	9.75	3714	2.93	4041	6.37
Regular Bootloader	215	1.35	428	0.34	584	0.92
IRIS SoC	**4173**	**26.33**	**9378**	**7.40**	**11539**	**18.20**
OpenRISC	1251	7.89	1938	1.53	3666	5.78
DDR Controller	1526	9.63	3714	2.93	4064	6.41
IRIS	1345	8.48	3301	2.60	3298	5.20
Increment of IRIS	**891**	**+27.14**	**2851**	**+43.7**	**2246**	**+24.15**

Table 3
Boot-up time on the Xilinx Artix7 XC7A100T-1CSG324C.

Core	Boot up time (seconds)	Percent (%)
Regular Bootloader	4.51	100
IRIS (HMAC disabled)	5.11	113
IRIS	8.19	181

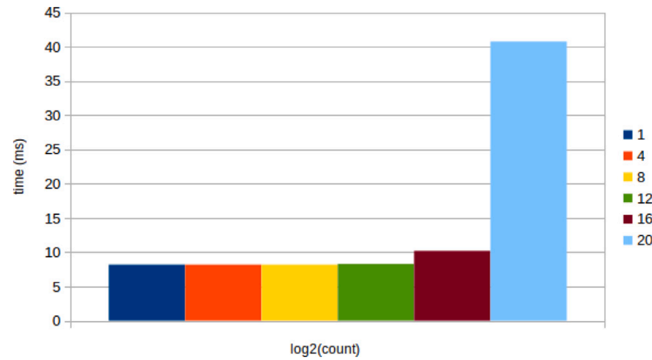


Fig. 5. Execution results of IRIS for boot a Linux Image with different values of the E-LUKS *count* parameter .

Table 4
Robustness of IRIS with different count values against brute-force attacks.

$\log_2$(count value)	Robustness (bits)
1	80
4	84
8	88
12	92
16	96
20	100

IoT applications [19]. Table 4 shows the robustness of IRIS against brute-force attacks with different values of *count*. However, increasing the count value runs against the performance of IRIS, as shown in Fig. 5 which shows the boot-up time for different values *count*. It is appreciable that the boot-up time greatly increases for values of *count* equal to 2^{20} and above. Hence, a great balance of performance and security of the system is a robustness of 96 bits with a *count* value of 2^{16}. Regarding standard values for the *count* in the specifications of LUKS, a minimum of 1000 is suggested for the value *count* [4].

4.2. Comparison with other solutions

It is also interesting to compare the utilization of IRIS resources with the solutions presented in Section 2: CARE [7], ITUS [8], and Streit et al. [6].

Both Streit et al. and ITUS include authenticity and integrity, but Streit et al. also add confidentiality, just as IRIS. In contrast, CARE is a mixed hardware-software solution in which only the HMAC algorithm (used for authenticity and integrity) is implemented on hardware.

Table 5
Resources utilization for IoT secure boot solutions.

Work	Flip flops		LUT's		FPGA family
	No.	%	No.	%	
Streit et al. [6]	5934	16.9	6783	38.5	Zynq
IRIS	3269	9.29	3319	18.86	Zynq
ITUS [8]	6722	1.65	27170	13.33	Kintex-7
IRIS	3299	0.81	3334	1.64	Kintex-7
CrytoCore CARE [7]	1715	1.35	2591	4.08	Artix-7
IRIS	3365	2.65	3390	5.35	Artix-7

Table 5 contains the comparison of resource utilization between the above-mentioned solutions and IRIS on the platforms reported by the authors: Zynq, Kintex-7, and Artix-7 FPGA families for Streit et al., ITUS and CARE respectively. Compared to Streit et al. and ITUS, IRIS requires approximately half the resources and many fewer LUTs than ITUS even if IRIS implements authenticity, which is absent in ITUS. This shows the benefits of using cryptographic algorithms specific to IoT applications.

When comparing IRIS with the CARE CryptoCore, the results show that the IRIS requirements are only slightly larger than those of CARE, which is not a surprise because IRIS is a fully secure bootloader that includes a KDF, a block cipher and an HMAC while the CARE CryptoCore only includes an HMAC.

5. Conclusions

The IRIS system introduced here is a full hardware secure boot solution for IoT devices on FPGA with a low footprint. IRIS provides confidentiality, integrity, and authenticity to the boot process. It remains available at runtime after the boot process and can be used to access secured data on removable media. IRIS is an open-source design and can be adapted to different SoC architectures by rewriting the glue logic for the internal bus.

The results show little impact on the boot-up time, especially if only data confidentiality is required. Comparisons against similar solutions reveal significant improvements in hardware resource requirements even when compared to solutions that do not support all aspects of a secure boot process, making IRIS a valuable option for secure boot in IoT devices.

As future works for this paper, some improvements are currently being developed to issue problems that may appear. One of these improvements is to add more cryptographic algorithms used in IRIS, which are currently being studied, particularly block ciphers mentioned in [19] alongside PRESENT: LEA and CLEFIA. In addition, a recovery mode for the system in case of tampering data, such as implementing a central server to communicate and upload a safe kernel image. However, to create this central server, it is required to implement asymmetric cryptography to ensure secure communication with the server. Moreover, to increase the system's security, it is important to ensure some mechanism to provide each FPGA with its password.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The code is provided in https://github.com/germancq/ELUKS/tree/main/examples/bootloader.

Acknowledgments

This work was partially supported by the Ministerio de Industria *y* Competitividad of Spain under project TIN2017-89951-P (BootTimeIoT) and the European Regional Development Fund (ERDF). The author, G. Cano-Quiveu, is financially supported by the VI-PPITUS.

References

[1] C. Gu, Power on and bootloader, in: Building Embedded Systems: Programmable Hardware, A Press, Berkeley, CA, 2016, pp. 5–25, http://dx.doi.org/10.1007/978-1-4842-1919-5_2.
[2] K. Dietrich, J. Winter, Secure boot revisited, in: 2008 the 9th International Conference for Young Computer Scientists, 2008, pp. 2360–2365, http://dx.doi.org/10.1109/ICYCS.2008.535.
[3] Specifications |Unified extensible firmware interface forum, 2022, URL: https://uefi.org/specifications.
[4] C. Fruhwirth, M. Broz, LUKS1 on-disk format specification, 2018, URL: https://mirrors.edge.kernel.org/pub/linux/utils/cryptsetup/LUKS_docs/on-disk-format.pdf.
[5] J. Hagl, O. Mann, M. Pirker, Securing the linux boot process: From start to finish, in: ICISSP, 2021, pp. 604–610.
[6] F.-J. Streit, F. Fritz, A. Becher, S. Wildermann, S. Werner, M. Schmidt-Korth, M. Pschyklenk, J. Teich, Secure boot from non-volatile memory for programmable soc architectures, in: 2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST, 2020, pp. 102–110, http://dx.doi.org/10.1109/HOST45689.2020.9300126.

[7] A. Dave, N. Banerjee, C. Patel, CARE: Lightweight attack resilient secure boot architecture with onboard recovery for RISC-V based SOC, in: 2021 22nd International Symposium on Quality Electronic Design, ISQED, 2021, pp. 516–521, http://dx.doi.org/10.1109/ISQED51717.2021.9424322.
[8] V.B. Kumar, S. Deb, N. Gupta, S. Bhasin, J. Haj-Yahya, A. Chattopadhyay, A. Mendelson, Towards designing a secure RISC-V system-on-chip: ITUS, J Hardw. Syst. Secur. 4 (4) (2020) 329–342.
[9] Z. Ling, J. Luo, Y. Xu, C. Gao, K. Wu, X. Fu, Security vulnerabilities of internet of things: A case study of the smart plug system, IEEE Internet Things J. 4 (6) (2017) 1899–1909, http://dx.doi.org/10.1109/JIOT.2017.2707465.
[10] O. Shwartz, Y. Mathov, M. Bohadana, Y. Elovici, Y. Oren, Reverse engineering IoT devices: Effective techniques and methods, IEEE Internet Things J. 5 (6) (2018) 4965–4976, http://dx.doi.org/10.1109/JIOT.2018.2875240.
[11] G. Cano-Quiveu, P.R. de Clavijo-Vazquez, M. Bellido-Diaz, J. Juan-Chico, J. Viejo-Cortes, D. Guerrero-Martos, E. Ostua-Aranguena, Embedded LUKS (E-LUKS): a hardware solution to IoT security, Electronics 10 (23) (2021) 1–22, http://dx.doi.org/10.3390/electronics10233036.
[12] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin, C. Vikkelsoe, PRESENT: An ultra-lightweight block cipher, in: P. Paillier, I. Verbauwhede (Eds.), Cryptographic Hardware and Embedded Systems - CHES 2007, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 450–466.
[13] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, I. Verbauwhede, spongent: A lightweight hash function, in: B. Preneel, T. Takagi (Eds.), Cryptographic Hardware and Embedded Systems – CHES 2011, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 312–325.
[14] F.F. Yao, Y.L. Yin, Design and analysis of password-based key derivation functions, in: A. Menezes (Ed.), Topics in Cryptology – CT-RSA 2005, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 245–261.
[15] GitHub - germancq/ELUKS/examples/bootloader, 2022, URL: https://github.com/germancq/ELUKS/tree/main/examples/bootloader.
[16] Xilinx, 7 Series FPGAs data sheet: Overview (DS180), 180, 2010, pp. 1–18, URL: www.xilinx.com.
[17] OpenRISC - OpenRISC, 2022, URL: https://openrisc.io/.
[18] WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores — WISHBONE B3, 2019, URL: https://wishbone-interconnect.readthedocs.io/en/latest/index.html.
[19] ISO/IEC 29192-2:2019 information security — Lightweight cryptography — Part 2: Block ciphers, 2019, URL: https://www.iso.org/standard/78477.html.