IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2016, No. 2, pp. 52–79. DOI:10.13154/tosc.v2016.i2.52-79

On Ciphers that Continuously Access the Non-Volatile Key

Vasily Mikhalev*, Frederik Armknecht and Christian Müller†

University of Mannheim, Mannheim, Germany
{mikhalev,armknecht,christian.mueller}@uni-mannheim.de

Abstract. Due to the increased use of devices with restricted resources such as limited area size, power or energy, the community has developed various techniques for designing lightweight ciphers. One approach that is increasingly discussed is to use the cipher key that is stored on the device in non-volatile memory not only for the initialization of the registers but during the encryption/decryption process as well. Recent examples are the ciphers Midori (Asiacrypt'15) and Sprout (FSE'15). This may on the one hand help to save resources, but also may allow for a stronger key involvement and hence higher security. However, only little is publicly known so far if and to what extent this approach is indeed practical. Thus, cryptographers without strong engineering background face the problem that they cannot evaluate whether certain designs are reasonable (from a practical point of view) which hinders the development of new designs.
In this work, we investigate this design principle from a practical point of view. After a discussion on reasonable approaches for storing a key in non-volatile memory, motivated by several commercial products we focus on the case that the key is stored in EEPROM. Here, we highlight existing constraints and derive that some designs, based on the impact on their throughput, are better suited for the approach of continuously reading the key from all types of non-volatile memory. Based on these findings, we improve the design of Sprout for proposing a new lightweight stream cipher that (i) has a significantly smaller area size than almost all other stream ciphers and (ii) can be efficiently realized using common non-volatile memory techniques. Hence, we see our work as an important step towards putting such designs on a more solid ground and to initiate further discussions on realistic designs.

Keywords: Lightweight Ciphers · Non-Volatile Memory · Implementation

1 Introduction

Owing to the increased proliferation of resource constrained devices, the design of lightweight encryption schemes is a continuous research topic. One could argue that this topic has become superfluous, given that progress in hardware production often results in designs where resources do not represent a critical factor. In fact, in such cases it is reasonable to use established ciphers like AES [Nat01]. However, various discussions with experts from industry and academia showed that there still remains interest in developing and implementing ciphers that consume as little resources as possible. Similar can be said about the scientific community that regularly develops new lightweight ciphers, e.g., the block ciphers PRESENT [BKL+07] and Midori [BBI+15].

*The author's research was funded by COMET K1, FFG - Austrian Research Promotion Agency and by the Austrian Science Fund (FWF): P12345-B51.
†The author's work was supported by the DFG project "Developing and Applying a Sound Security Framework for Sensor Networks".

Received: 2016-09-01, Accepted: 2016-11-01, Published: 2017-02-03

In this work, we focus on the design of ciphers that consume as little area as possible. Some reasons why such ciphers are of interest are:

1. The less area a cipher consumes, the more area remains for installing other functionalities, e.g., protection mechanisms against physical attackers. In this context, it often becomes easier to protect the cipher when it requires less area.

2. The power consumption of a cipher often relates directly to the area size, giving another incentive to keep it as small as possible.

To this end, the community has developed various techniques for designing lightweight ciphers.[1] One approach that is more and more considered is the idea of using the cipher key that is stored on the device not only for initialization of the registers but also to involve it continuously in the encryption/decryption process. Examples include the block ciphers Midori [BBI+15], KTANTAN [CDK09], PRINTcipher [KLPR10], and LED [GPPR11] as well as the stream ciphers A2U2 [DRL11] and Sprout [AM15]. Moreover, this approach is also used in practice: the cipher and the authentication protocol of the Megamos Crypto transponder [VGE15] also utilize this principle.

To understand the basic idea, recall that a device that operates the cipher usually needs to access a key, which is not given from outside but is stored on the device in some non-volatile memory. We refer to this key as being a *non-volatile key*. Here, the term non-volatile stresses the property that this key cannot be changed by the cipher itself. That is, in most cases the work flow usually looks as follows. After the encryption or decryption process is started, the key is loaded from the location where it is stored, i.e., the non-volatile memory NVM, into some registers, i.e., into volatile memory VM. For block ciphers, this may be a separate register on which the key schedule operates, while for stream ciphers this is usually the internal state. Adopting the terminology from above, one could say that the value in VM now represents a *volatile value* which usually changes during the encryption/decryption process whereas the value stored in NVM, the non-volatile key, remains fixed.

It holds for most designs that after the key has been loaded from NVM into VM, the NVM is usually not involved anymore (unless the key schedule or the initialization process needs to be restarted). However, recently several ciphers were proposed which require access to the non-volatile key, i.e., the key stored in NVM, not only for initialization but also during encryption/decryption.

One of the main motivations for this approach is that using the values stored in the NVM may allow to reduce the VM and hence result in an overall reduction of the area size. The argumentation here is that NVM is needed for storing the key on the device anyway. Hence, if re-using the key stored in NVM allows to reduce the amount of volatile memory, this would provide a direct benefit. In other words, one could say that non-volatile memory that needs to be present anyhow is "traded" for volatile memory so that in total, the area consumption is reduced.

In this context, we want to point out that a similar approach may be considered for block ciphers to reduce the logic. Block ciphers usually deploy a key schedule to derive several round keys from the secret key. Instead of loading the key into VM and to run the key schedule, one may consider to precompute these round keys once and to store them in NVM. Here, one could say that one extends the non-volatile memory to trade it for volatile memory and logic. While this is certainly an interesting approach with its own benefits, it is not completely in scope of our work as we are mainly looking at ciphers which re-use NVM that needs to be present anyway, while storing the round keys would increase the amount of required non-volatile memory. Furthermore, any potential measures applied to NVM to protect it against physical attacks would also require appropriate extension to

[1] In this work, lightweight always refers to consuming as little area as possible in hardware implementation.

remain effective. Finally, this approach makes only sense for block ciphers with a fixed and relatively small number of round keys and is completely impractical for stream ciphers. This is opposed to the idea of re-accessing the same key in NVM which has been considered for block and stream ciphers and hence is more general. Nonetheless, the idea of storing the round keys adds another reason why cipher designs where NVM is continuously accessed can be of interest.

Apart from reducing the demand for resources such as area size, power and energy consumption, the continuous involvement of the key may also result in a higher security level. For example, in [AM15] it is argued that this may increase the resistance of stream ciphers against time-memory-data-tradeoff attacks.

Summing up, the idea of continuously using the key stored in NVM seems to be a promising design approach for several reasons, at least in *theory*. In *practice* however, only little is known if and to what extent this impacts the practicability of the cipher. In fact, most papers treat NVM as "free" memory from which can be read in "zero" time or at least with the same speed as it is possible to access VM. This is certainly not given in practice and raises the question if and to what extent such designs really help to reduce the consumption of resources. The main problem here is that hardly any information is publicly available. That is, cryptographers without a strong engineering background face the problem that they cannot evaluate whether such designs are indeed reasonable and promising or rather represent "cheating". In this work, we revisit the design of ciphers that continuously involve the non-volatile key during the encryption process. More precisely, our contributions are as follows:

Discussion on Non-Volatile Memory: In the first part of this work, we discuss several use cases and explain what types of NVM are practically relevant for the considered scenario. Here, we have to distinguish between the case that the key is set once and forever or that it is rewritable. For the first case, certain types of NVM can be used, e.g., MROM and PROM, where accessing the keybits induces no overhead. That is, in such cases very efficient ciphers are possible but key management is very limited. In the second case, i.e., cases which require the key being rewritable, certain timing limitations for accessing the NVM need to be respected. Motivated by commercially employed schemes like KeeLoq [Mic01], Hitag (S) [NXP14], or Mifare [NXP11] we then put our focus on the case that EEPROM is used for storing the rewritable key and explain practical implications when aiming for low area ciphers using EEPROM.

Reconsideration of Existing Ciphers: Based on our findings, we evaluate existing lightweight ciphers within this scenario with respect to the practical consequences. In case that the NVM needs to be reprogrammable such as EEPROM, some ciphers are better suited for this approach than others, which depends on how the key stored in NVM needs to be accessed.

Proposal of a New Cipher: Our analysis shows that the methods of both the stream cipher Sprout and the block cipher LED for involving the non-volatile key bits are well aligned with the timing limitations of EEPROM compared to other ciphers. This makes them to interesting and flexible designs that work well for both cases: the key is stored in rewritable NVM and non-rewritable NVM. However, it turned out that Sprout suffers from several security weaknesses and hence should not be used. Inspired by this design, we propose a new stream cipher Plantlet, which is well suited for constrained devices, independent of the underlying NVM. The design is based on Sprout but strengthened against the security holes identified for Sprout and also takes into account how the non-volatile key bits can be read efficiently from NVM like EEPROM. The design also includes a novel technique that we term *double-layer LFSR* which uses practically one LFSR for ensuring high period but

avoids the all-zero state without the risk of any internal collisions. This approach is independent of the NVM-based design and hence of its own interest.

Implementation Results: We also discuss the results of our implementation of Plantlet. To allow for comparison, we used the same tools as the authors of Sprout [AM15] and compare the results. The implementation of Plantlet in conjunction with MROM or PROM, i.e., NVM that is non-rewritable, requires 928 GE, whereas when using EEPROM, our best implementation needs an area size of 807 GE only. The main reason for the smaller area size is that we are able to re-use control logic that is part of the EEPROM anyway. We contrast this result to Sprout, which requires 813 GE according to [AM15].[2] This makes Plantlet one of the smallest stream ciphers today while being based on a realistic hardware model.

The structure of this work is as follows. In Section 2, we discuss different technical realizations for NVM and analyze the effort to read from them. In Section 3, we analyze the impact on different ciphers which use the non-volatile key "trick" if the cipher would be implemented with the aforementioned types of NVM. In Section 4, we give the specification of a new cipher named Plantlet, including explanations on the design rationale, and give a security analysis and implementation results in Section 5 and Section 6, respectively. Section 7 concludes this work.

2 On Reading from Non-Volatile Memory

In this section, required technical background pertaining to NVM is presented. First of all, common implementation approaches for NVM are described with a focus on programmability. Then, prominent serial interfaces suitable for accessing NVM are introduced as well as an effort estimation particularly for reading operations. This section also includes a brief summary of the different interfaces' required number of clock cycles for reading from NVM. Readers who are familiar with the technical aspects of NVM and serial interfaces, e.g., EEPROM and I^2C, may safely skip this section and continue with Section 3.

2.1 Approaches

Generally speaking, there are three different categories regarding NVM production approaches.

The first category comprises technologies where the memory content has to be programmed already by the manufacturer and cannot be changed afterwards. In case of lightweight devices this can be realized either by using MROM—where the information is added to so-called wafers—or by realizing lookup tables (using tie-high and tie-low cells) in standard ASIC library. Utilizing tie-high and tie-low cells has the benefit of not incurring any overhead for accessing the memory. The full key is instantly available and its usage comes at no performance overhead, while the area demand of such a logic is also relatively small.

The second category covers techniques where likewise the key can be set only once. The difference however lies in the time of programming, which in this case happens *after* manufacture. This PROM is typically produced with fuses and antifuses that are blown appropriately during the programming process.

The third category differs significantly from the former two in the sense that the key can be freely rewritten, allowing for a higher level of flexibility.

[2] We admit however that we use a trick for EEPROM implementation which is applicable to Sprout as well and discuss this likewise.

In fact, according to our understanding, the authors of PRINTcipher [KLPR10] and A2U2 [DRL11] consider the first category, i.e., to program the key during the IC manufacturing process, while the works on KTANTAN [CDK09] and Sprout [AM15] discuss the second category, i.e., to burn the key in using fuses and antifuses. The authors of Midori [BBI+15] and LED [GPPR11], respectively, do not discuss the concrete types of NVM. With respect to the effort of accessing the bits during the encryption/decryption process, the first two approaches would certainly be the preferred option, as each key bit can be directly accessed without any particular overhead.

However, the price one has to pay is the loss of flexibility. More precisely, the first approach means that the key has to be known already before the IC chip is produced. Moreover, this approach becomes cheap only if a large amount of devices share the same key, since the production of individual wafers, from which chips are produced, is rather expensive. In the works [KLPR10] and [DRL11] utilizing a technology called integrated circuit printing is suggested, claiming that for a printer there will be no costs for changing the circuit while producing each of the new devices, thus enabling individual keys. However, according to [KLPR10] the IC printing technology is not yet fully understood. In any case, if this approach is used it holds that once the key is set it can never be changed, which is true for the second approach as well. That is the process of setting the key cannot be reversed and hence has to be settled before a device leaves the factory which enormously complicates key management.

We conjecture that this is one of the main reasons why the third, yet more expensive approach is often used in practice. That is, the key is stored in some NVM which allows to change the key afterwards. Here, different techniques are imaginable. In case of lightweight devices, EEPROM seems to be the favored choice (for more details, see e.g. [AHM14]). In fact, in many commercial products EEPROM is used for storing the secret key. Examples include MIFARE [NXP11], Hitag [NXP14], KeeLoq [Mic01], CryptoRF [Atm14] and Megamos Crypto transponder [VGE15]. We would like to stress that using EEPROM does not necessarily imply very high costs. For instance, the Hitag μ advanced transporder [NXP15], which offers 1.76 kb of EEPROM memory can be purchased for the price of a few cents per unit from a major retailer. In this work therefore, we focus on a scenario where the key is stored in EEPROM which is continuously accessed by the cipher during the encryption/decryption process and where the goal is to reduce the area size of the cipher without increasing the size of the EEPROM. To this end, we investigate different interfaces for accessing EEPROM in the following and discuss how this impacts the performance of the cipher. As another type of EEPROM-like memory, flash memory is presented in form of a short overview and comparison with a focus on its suitability to lightweight applications in contrast to EEPROM.

2.2 Accessing EEPROM

In principle, two different options exist for using EEPROM in constrained devices: (i) commercially available external EEPROMs which communicate with the device via certain standard interfaces and (ii) NVM cells, e.g. EEPROM, that are directly integrated into an ASIC by design. Although, the first approach may be much cheaper, it is susceptible to side channel attacks. That is, the data lines between the EEPROM module and the device may be easily eavesdropped on. Nevertheless, this approach may be realistic for a scenario, in which physical access to the device is considered infeasible for an attacker, e.g., medical implants. Since our intention is not to promote commercial EEPROMs, but to provide a holistic view on available memory technologies, excluding commercial EEPROMs from the discussion would render it incomplete. Conversely, integrating EEPROM cells into the ASIC is more expensive, but it also offers a higher level of security and more flexibility pertaining to the interfaces. Although most of the commercially available constrained devices use the second approach, we discuss both alternatives for the sake of completeness.

Table 1: Overview of required clock cycles for reading n consecutive memory words using different interfaces and read operations for 8-bit memory words and addresses. SPI and Microwire only offer selective reading. Sequential reading is therefore not *directly* available, indicated by 'n/a'.

Interface	Selective Reading	Sequential Reading
I^2C	$30 + 9n$	$11 + 9n$
SPI	$16 + 8n$	n/a
Microwire	$11 + 8n$	n/a
UNI/O	$50 + 10n$	$30 + 10n$

Standard EEPROMs. In the following, we focus on commercially available EEPROMs and denote these as standard EEPROMs (as opposed to customized EEPROMs that we discuss afterwards). Standard EEPROMs can be accessed using serial or parallel interfaces. EEPROM modules with serial interfaces have a smaller footprint and lower power consumption compared to EEPROM with parallel interfaces of an equivalent density [PSS+08]. As we examine the use of EEPROM in the context of lightweight ciphers, we consider parallel interfaces such as those used e.g. in ARM AMBA protocol out of the scope of this work.

With respect to commercial EEPROMs with serial interfaces, we examined the data sheets of several low-budget EEPROM devices produced by different manufacturers such as NXP Semiconductors, Microchip Technology Inc, Atmel Corporation, On Semiconductor, Renesas Electronics Corporation. According to these, the following serial interfaces are the most commonly used:

1. I^2C [On 14]
2. Serial Peripheral Interface (SPI) [Mic14]
3. Microwire [Atm15]
4. UNI/O [Mic11]

In the following, we discuss for each interface the number of clock cycles required for reading operations. We restrict here to the most relevant aspects and refer to Appendix A (cf. Supplementary Material) for an in-depth discussion.

Although these techniques differ in several details, one can observe that two principal reading methods are possible: accessing the content by providing the address or reading the next word of the EEPROM based on an address counter that is automatically increased. Adopting the terminology of I^2C, we refer to these types as *selective reading* and *sequential reading*, respectively. That is, when we speak about sequential reading we will refer to this mode of operation and not necessarily to I^2C.

Table 1 shows the number of required clock cycles to read n consecutive 8-bit memory words assuming the underlying memory and addressing scheme are as well 8-bit based (or 7 bits as in the Microwire case). One sees clearly that selective reading, i.e., reading from a chosen address, incurs a higher overhead than just reading sequentially the words from the EEPROM. This shows that non-volatile key ciphers that sequentially read the key bits from the EEPROM better match this technology than ciphers that repeatedly have to read from different addresses. In the next section, we analyze how this impacts different non-volatile key ciphers where the key is stored in EEPROM using the discussed interfaces.

EEPROMs Integrated into ASIC. When a designer develops an ASIC, he can integrate EEPROM cells directly so that he has not to rely on any standard interfaces. By doing so, he can achieve that practically almost no limitations on retrieving key bits from an integrated EEPROM apply. In such cases, any cipher design would be equally good with respect to the effort of accessing the key bits.

However, these benefits do not come for free. First of all, developing an ASIC tailored for certain use cases takes more effort than using standard building blocks. Moreover, it seems that certain ASIC EEPROM designs are favorable compared to others. More precisely, in existing work [BL08, CZDL13, NKTZ12, NXDH06, LCK10] that discusses the construction of ASIC EEPROM designs for restricted devices like passive RFID transponders, the access to the EEPROM is organized word-wise where typical memory word sizes are 8, 16, or 32 bits. According to [NKTZ12], the choice of word size is one of the most important factors that directly impact on the area size of the resulting EEPROM. Obviously, this choice also affects the effectiveness of the procedure that involves the key during encryption/decryption.

Although accessing EEPROM on a word-by-word basis is a common technique, designs utilizing bit-by-bit reading may provide a more conservative power consumption [CZDL13, NXDH06]. Such a customized bit-by-bit interface allows to retrieve any desired bit per clock-cycle. Thus, in addition to 8-, 16-, and 32-bit based parallel interfaces, we consider this type of bit-by-bit interface, as we suppose that it implies realistic limitations on the flexibility of reading from customized EEPROM and can be pertinent in practice.

2.3 Flash Memory

After investigating cheap candidate devices for lightweight ciphers, e.g., passive RFID tags, flash memory does not seem to be used commonly. However, since this technology is gaining in popularity and decreasing in cost, it may be used for these kinds of devices in the near future. Therefore, for the sake of completeness, we also shortly discuss flash memory.

Technically, flash memory is another type of EEPROM with higher storage density than 'traditional' EEPROM, sacrificing bit alterability for area [Cyp15], such that minimum erasable data units are organized in *blocks* of several kilobytes [KKN+02], e.g., 64, 128, or 256 kilobytes. Therefore, flash is typically used for applications with high memory demand, i.e., in the range of megabits to several gigabits. Also, different architectures (NOR or NAND), data transfer (serial or parallel), and approaches (external or directly integrated) are prevalent, leading to different characteristics and read protocols.

On the one hand, NOR flash memory offers random read access on a byte level making it a suitable EEPROM alternative [Mic]. On the other hand, NAND flash is page-oriented, whereas several pages, typically 512 [KKN+02], 2048, or 4096 bytes in size, form one block, and it requires a dedicated memory controller. Usually, NAND flash is used for mass storage, e.g., more than 64 megabits [Cyp15, Mac14].

Having compared several data sheets of available low-price external flash memory ICs, it appears that mostly similar serial interfaces are used for reading as in the case of EEPROM [Fre14, Ade16]. We note that external modules suffer from similar weaknesses as EEPROM, e.g., side channel attacks.

Since the structure of the memory cells is similar to EEPROM, integrating NOR flash into ASIC yields a read process organized on either a bit or word basis and therefore, similar limitations as for integrated EEPROM.

All in all, since these two memory technologies use similar read interfaces in cases where flash memory constitutes a potential replacement of EEPROM, we do *not* further consider flash as a substantially different additional technology with respect to the timing limitations of the ciphers and limit our focus to EEPROM.

Table 2: Impact of different standard serial interfaces for EEPROM access on the throughput for several ciphers relative to the original throughput i. e., higher is better.

Cipher	Approach	I^2C	SPI	Microwire	UNI/O
Midori64	wrap	1.4%	1.5%	1.5%	1.2%
	no-wrap	1.2%	1.4%	1.4%	0.9%
LED-128[3]	wrap/no-wrap	81%	82%	82%	81%
KTANTAN32/48/64	std	2.5%	4%	5%	1.6%
	ext	45%	50%	50%	40%
PRINTcipher-48[3]	wrap	17.8%	20%	20%	16%
	no-wrap	13.3%	16.7%	17.6%	10.7%
A2U2	wrap	4.8%	7.1%	8.7%	3%
	no-wrap	4.7%	6.8%	8.4%	3%
Sprout	wrap	88.9%	100%	100%	80%
	no-wrap	66.67%	83.33%	87.91%	53.33%

3 Impact of Different EEPROM Types on Throughput

In this section, we examine for several existing ciphers that continuously access the key stored in non-volatile memory how the use of standard serial EEPROM on the one hand and of customized ASIC EEPROM on the other hand would impact performance. This is motivated by the question on how different designs perform with respect to available standard building blocks. As the concrete throughput depends on many parameters unrelated to the deployed EEPROM, we do not state these values but compare them with the case that the same implementation is used but under the assumption that accessing the key incurs no overhead at all.

The considered ciphers are the block ciphers Midori128 and Midori64 [BBI+15], LED [GPPR11], KTANTAN [CDK09] and PRINTcipher [KLPR10] as well as the stream ciphers A2U2 [DRL11] and Sprout [AM15]. In our analysis we focus on the key selection functions, i.e., the part of the cipher that *continuously* accesses the key stored in non-volatile memory as we expect here the most significant impact. This means for example that we ignore the initialization process which is invoked much less frequently. Therefore, we provide only a description of the key selection function in Appendix B (cf. Supplementary Material) and refer to the respective papers for a full description.

Due to the fact that sequential reading allows for a higher throughput than selective reading, we will opt for the first whenever possible when discussing standard interfaces. With respect to sequential reading, we distinguish between two cases that we call *wrap case* and *no-wrap case*, respectively. In the wrap case, it holds that if the last key bit of the EEPROM is read, the internal address counter automatically wraps around such that it points to the first bit of the key again. In contrast if the no-wrap case holds, the internal address counter does *not* automatically wrap around but has to be reset manually to the start address of the key. Furthermore, we make the assumption that at the beginning of the encryption/decryption process, the address already points to the beginning of the key.

The results of the analysis are summarized in Table 2. It displays the throughput of the respective cipher using the respective standard serial EEPROM in comparison to the same implementation where no effort is assumed for accessing the key which represents 100% throughput. For example, in case of Midori64 the throughput decreases from 100% to 1.4%, i.e., by a factor of almost 72 when switching from an implementation where

[3] Serialized implementation

Table 3: Impact of different ASIC EEPROM realizations on the throughput for several ciphers relative to the original throughput, i. e., higher is better.

Cipher	Custom x1	Custom x8	Custom x16	Custom x32
Midori64	1.5%	12.5%	25%	50%
LED-128	82%	100%	100%	100%
KTANTAN32/48/64				
std.	57%	57%	57%	62.5%
ext.	57%	100%	100%	100%
PRINTcipher-48	20%	72%	84%	89%
A2U2	20%	67%	77.8%	88.9%
Sprout	100%	100%	100%	100%

accessing the key incurs no overhead compared to using I^2C in wrap case. One exception is KTANTAN where the key bits are accessed in random order. Here, the distinction between wrap case and no-wrap case would not make sense. Instead we distinguish between standard and the extended key approach (see also Section 1). Standard approach means that the cipher applies selective reading, i.e., each time the next key bit has to be accessed, the read address needs to be configured before. In the extended approach, the key bits are duplicated and stored in the order they need to be accessed during encryption. This allows to use sequential reading and hence achieves higher throughput, but at the cost of increased area. These two approaches are denoted by std. and ext., respectively.

Furthermore, we investigated for the same ciphers the impact of different EEPROM integrations into ASIC on the throughput. The results are displayed in Table 3. Here, we consider different customized EEPROM types where "custom x1" means that it requires one clock to read one bit, "custom x8" that eight-bit words can be read in one clock cycle, and so on. Analogously to above, we do not state concrete values but state the throughput in comparison to an idealized version where the access to the key creates no overhead at all.

At this point, we would like to stress that our goal is to show the impact on the throughput of lightweight ciphers that continuously access the non-volatile key in the presence of different realizations of NVM. This is by no means intended as an evaluation, neither a judgment, of the ciphers themselves. Whenever possible we did choose the implementations mentioned in the original publication. However, it may well be that for several ciphers, different implementations would result into a lower or higher decrease of the throughput, e.g., the serialized version of Midori would probably be much less affected by the limitations.

Furthermore, since most of the discussed ciphers are available in different variants with different properties, we decided to evaluate area-conservative implementations as provided in the original publications. As our assessments show, the ciphers LED and Sprout keep almost maximum performance regardless of the NVM type due to sequential processing of key bits which make them suitable for various target applications. Also the cipher KTANTAN shows high performance if the extended key approach is used. However, applying such approach would expand the key storage effort for KTANTAN from 80 bits to 508 bits, which may lead to additional costs. This does not hold for the ciphers LED and Sprout.

In a nutshell, LED offers a smaller footprint while Sprout achieves a higher throughput. According to [GPPR12], the 64-bit key version of LED requires 1248 clock-cycles to encrypt 64 bits of data (the block size of LED) and 1872 clock-cycles for key sizes between 65 and

128. In comparison, Sprout requires 320 clock-cycles for the initialization and afterwards encrypts 1 bit per clock-cycle in most cases [AM15]. Therefore, for encrypting 64 bits of data, Sprout requires 384 clock-cycles in total, which would be a factor of 3 faster than LED. Additionally, the more data needs to be encrypted, the higher this factor becomes.

Note that even if speed is not an issue, having a higher throughput may be beneficial as it also provides a better energy-per-bit ratio due to requiring fewer clock-cycles.[4]

To sum up, the approach of constantly accessing the key from rewritable non-volatile memory is practical but implies certain impacts. With respect to area size, there is no big difference if the key has to be read only once or continuously during encryption, since the logic for reading the key (at least once) has to be implemented anyway. Small extra logic may be needed for synchronization with NVM—cipher should not be clocked unless key material is ready.[5] This approach often leads to significantly decreased speed and is thus undesirable for use cases with timing or energy constraints. However, designs with round key functions that require sequential access to the key bits are almost unaffected irrespective of the underlying NVM type.[6]

4 Plantlet Specification

As a consequence of the findings from the previous sections, we discuss in this section a new cipher called Plantlet. The main goal of Plantlet is to design a stream cipher with low area size but high throughput, even with the use of standard NVM.

4.1 Motivation

One might be inclined to question the need for a new stream cipher while there already are many lightweight and efficient ciphers in the field as presented in Section 3, e.g., the block cipher LED. However, in the best case we should have a stream cipher and a block cipher, which are suitable for lightweight scenarios. When certain types of memory are used, the throughput of most of the ciphers, which require continuous access to the key, decreases or needs 'tricks' (e.g. extended key) to weaken the impact. As it turns out, the throughput of both LED and Sprout remains virtually unaffected. As explained above, LED has a lower area size compared to Sprout but a lower throughput as well. However, Sprout, being one of the smallest stream ciphers, suffers from security flaws based on its design. Hence, we consider the question of having a secure lightweight stream cipher that provides a high throughput irrespective of the used NVM as still open that we aim to close. Fixing Sprout seems a promising start as we could build upon existing security analyses.

4.2 Design Goals

Plantlet is designed to meet the following design goals: 80-bit stream cipher with low area requirements, shorter internal state while keeping the security level, which achieves high throughput even if the key is permanently stored in and *continuously* read from re-writable NVM during computation, being thus hardware-friendly and independent of the choice of underlying NVM technology. Essentially, Plantlet is a stronger version of Sprout, hence the name. In particular, it inherits the overall structure from Sprout, adopts the continuous key involvement, but at the same time implements fixes for discovered vulnerabilities, e. g., stronger round key function and avoiding the all-zero state.

[4] Of course, there may be other implementations of LED that provide a higher throughput and/or require less clock-cycles. Nonetheless, we think it is beneficial to have lightweight stream ciphers with similar area size as well-established block ciphers.

[5] For example, see Table 5 "Commercial serial EEPROM modules", where the difference in area stems from the logic required for synchronization.

[6] Secure encrypted memory is out of scope as it is too expensive for ultra-constrained devices.

4.3 Overall Structure

To this end, we follow the design principle suggested in [AM15] to regularly involve the key stored in non-volatile memory in the state update. In fact, the design of Plantlet is based on the Sprout cipher introduced in the same work. The overall structure of Plantlet is depicted in Figure 1. Like Sprout, it is composed of an LFSR, an NLFSR, a

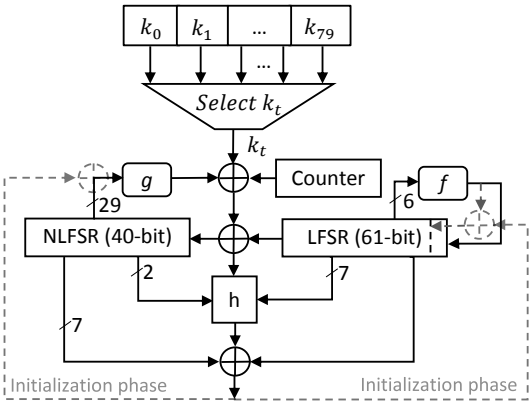

Figure 1: The Structure of Plantlet.

counter, a round key function, and an output function that interact in the same way as in Sprout. That is, the LFSR and NLFSR states represent the internal state which at the beginning are initialized with the key and the IV as explained below. The output of the LFSR is fed into the NLFSR to ensure a minimum period with respect to the internal states. In addition, a key bit is inserted as well to provide resistance against time-memory-tradeoff attacks, which, in turn, allows for a shorter internal state. Since the key needs to be stored in NVM anyway, accessing the key bits directly, i. e., from NVM instead of loading them into registers first, further reduces the area demand. On the one hand, less area is required since fewer registers are needed or existing ones could be used for other purposes than handling key bits. On the other hand, retrieving key bits from NVM is more time-consuming. Overall, this approach is a trade-off between area size and access time. The purpose of the NLFSR is to render certain standard attacks like algebraic attacks infeasible. Moreover, several bits are taken from both FSRs as input to the output function.

However, some modifications are introduced in order to account for attacks which have been discovered against Sprout [MSBD15, LYR15, LN15, EK15, Hao15, Ban15]. More precisely, we enlarge the internal state size by increasing the LFSR's length from 40 bits to 61 bits, we replace the key-selection function by a linear function such that at each update, a key bit is involved, and we introduce a technique that we term "double-layer LFSR" which allows for high period and at the same time avoids the LFSR being initialized with the all-zero case.

4.4 Specification

In this section, we provide the full specification of Plantlet. To this end, we use the following notation:

- t - the clock-cycle number
- $L^t = (l_0^t, l_1^t, \cdots, l_{60}^t)$ - state of the LFSR during the clock-cycle t

- $N^t = (n_0^t, n_1^t, \cdots, n_{39}^t)$ - state of the NLFSR during the clock-cycle t
- $C^t = (c_0^t, c_1^t, \cdots, c_8^t)$ - state of the counter during the clock-cycle t
- $k = (k_0, k_1, \cdots, k_{79})$ - key
- $iv = (iv_0, iv_1, \cdots, iv_{89})$ - initialization vector
- $\tilde{k}^t$ - the round key bit generated during the clock-cycle t
- z^t - the keystream bit generated during the clock-cycle t

As in the case of Sprout, the cipher has an initialization phase and a keystream generation phase. The maximum amount of data that can be securely encrypted under the same key and IV is 2^{30} keystream bits (i. e., 1 Gb). Both phases share the same procedure for producing bits as explained above (and as depicted in Figure 1). However, there is one important difference. While in the initialization an LFSR of length 60 is used, during the keystream generation phase an LFSR of length 61 is deployed. To save area however, both LFSRs use the same register and the same logic. We call this approach a double-layer LFSR. To complete the specification, we explain now in detail the initialization phase, the double-layer LFSR, the key selection function, and the output function.

Initialization phase. In the initialization phase, the NLFSR is loaded with the first 40 bits of the IV, whereas the first 50 LFSR stages are loaded with the remaining IV bits, i. e., $n_i = iv_i$ for $i = 0 \leq i \leq 39$ and $l_{i-40} = iv_i$ for $40 \leq i \leq 89$. The last 11 bits of the LFSR are filled with constant values '0' and '1' such that $l_{50} = \ldots = l_{58} = 1$, $l_{59} = 0, l_{60} = 1$. Then, the cipher is clocked 320 times without producing any keystream bits. Instead, the output is fed back and XORed with the LFSR and NLFSR inputs. In the initialization phase, the first 60 LFSR bits are updated, while the bit l_{60} remains fixed to 1, i.e., it is not involved in the LFSR update. After 320 clock cycles, the initialization phase is complete and the cipher starts to generate keystream bits. Here, the update function of the LFSR is changed such that l_{60} is also updated. We explain this concept next.

Double-Layer LFSR. As mentioned above, the new design utilizes 2 different phase-dependent LFSRs, i. e., a 60-bit LFSR during the initialization phase and a 61-bit LFSR during the keystream generation phase. The novel aspect is that in both phases, the LFSR is instantiated with almost the same hardware. That is the LFSR used in the keystream generation phase reuses significant parts of the LFSR from the initialization phase. More precisely, in the initialization phase the LFSR uses a feedback polynomial P_I, whereas during the keystream generation phase, another polynomial P_K is used. However, both polynomials are almost the same which allows to reuse the same register and almost all of the taps. To ensure maximum period, both polynomials P_I and P_K are primitive and defined as follows:

$$P_I(x) = x^{60} + x^{54} + x^{43} + x^{34} + x^{20} + x^{14} + 1 \tag{1}$$

$$P_K(x) = x^{61} + x^{54} + x^{43} + x^{34} + x^{20} + x^{14} + 1 \tag{2}$$

This means that during the initialization phase, the update function of the LFSR works as follows:

$$\begin{aligned} 0 \leq t \leq 319 : l_{60}^{t+1} &= 1 \\ l_{59}^{t+1} &= l_{54}^t + l_{43}^t + l_{34}^t + l_{20}^t + l_{14}^t + l_0^t + z^t \\ l_i^{t+1} &= l_{i+1}^t, 0 \leq i \leq 58 \end{aligned}$$

Afterwards when the keystream generation phase starts, the LFSR is updated as follows:

$$t \geq 320 : l_{60}^{t+1} = l_{54}^t + l_{43}^t + l_{34}^t + l_{20}^t + l_{14}^t + l_0^t$$
$$l_i^{t+1} = l_{i+1}^t, 0 \leq i \leq 59$$

NLFSR and counter. The NLFSR and the counter are adopted from the original design of Sprout and have respective lengths of 40 and 9 bits. The NLFSR is updated as defined in the following:

$$\begin{aligned} n_{39}^{t+1} &= g(N_t) + \tilde{k}^t + l_0^t + c_4^t \\ &= \tilde{k}^t + l_0^t + c_4^t + n_0^t + n_{13}^t + n_{19}^t + n_{35}^t + n_{39}^t \\ &+ n_2^t n_{25}^t + n_3^t n_5^t + n_7^t n_8^t + n_{14}^t n_{21}^t + n_{16}^t n_{18}^t \\ &+ n_{22}^t n_{24}^t + n_{26}^t n_{32}^t + n_{33}^t n_{36}^t n_{37}^t n_{38}^t \\ &+ n_{10}^t n_{11}^t n_{12}^t + n_{27}^t n_{30}^t n_{31}^t \\ n_i^{t+1} &= n_{i+1}^t, 0 \leq i \leq 39 \end{aligned} \tag{3}$$

That is the next NLFSR bit n_{39}^{t+1} is computed as the XOR of a non-linear combination of several NLFSR bits, the current key bit $\tilde{k}^t$, the output of the LFSR l_0^t, and a counter bit c_4^t. The selection of the key bit is explained below.

The counter is a simple 9 bit register where the value is continuously increased as follows. The first seven bits $(c_0^t \cdots c_6^t)$ of the counter are used to count cyclically from 0 to 79, i. e., it resets to 0 after 79 is reached. The two most significant bits realize a 2-bit counter to determine the number of elapsed clock cycles in the initialization phase, i. e., it is triggered by the resets of the lower 7 bits. Hence, it is reset to 0 after $4 \times 80 = 320$ clock cycles and indicates the end of the initialization phase.

Round key function. The round key function simply cyclically selects the next key bit which is provided as input to the NLFSR. That is it holds

$$\tilde{k}^t = k_{(t \bmod 80)}, t \geq 0 \tag{4}$$

Output function. We use the original output function from Sprout. It has a nonlinear part $h(x) = x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_0x_4x_8$ where $x_0, \cdots, x_8$ correspond to the state variables n_4^t, l_6^t, l_8^t, l_{10}^t, l_{32}^t, l_{17}^t, l_{19}^t, l_{23}^t, n_{38}^t, respectively. The entire output function is determined by

$$z^t = h(x) + l_{30}^t + \sum_{j \in B} n_j^t \tag{5}$$

where $B = \{1, 6, 15, 17, 23, 28, 34\}$.

4.5 Design Rationale

In the remainder of this section, we explain some aspects of the design rationale behind Plantlet, especially on the introduced changes in comparison to Sprout.

Round key function. The task of the round key function is to generate a round key bit that has to be involved in the update function. From a security perspective, one may prefer involved, non-linear functions that compute the round key bit from several key bits. However, more involved functions usually require more logic which increases the area size and often the power consumption as well. Hence, for practical reasons simpler designs should be preferred. Moreover, as elaborated in Section 2, the use of EEPROM for

storing the key advocates solutions where the key bits are cyclically taken from the full key. With this respect, the round key function of Sprout is perfectly suited for reading the key from EEPROM. In Sprout, the round key function also cyclically steps through the key, but is only included into the NLFSR update with a probability of 0.5, i. e., only if the linear combination of several state bits is equal to 1. This has been exploited by several attacks [LN15, Ban15, EK15], for example by looking for longer periods where no key bit is involved. This imbalanced key involvement turned out to be a major weakness in Sprout. To avoid such attacks, Plantlet simply incorporates a key bit during each clock cycle. The modified *linear* key update function utilized by Plantlet balances key involvement such that the key always influences the state, which is also in accordance with the mitigation strategies suggested in [LN15, Ban15, EK15].

Internal state size. There have been several indications that the internal state size of Sprout is too small. The authors of Sprout claimed that the key cannot be easily found even when the entire internal state is known because the round key bits influence the internal state before they propagate to the output function and can be recovered. However, it was shown in [EK15] that if the cipher is clocked backwards, then the key bits can be recovered from the output sequence if the internal state is known. Therefore, having an internal state size equal to the security parameter is a design flaw in this context. In general, all the attacks against Sprout exploit the property of the internal state being too short. Yet, the main reason for increasing the internal state size is to rule out attacks based on the guess and determine technique. Possible ramifications are to increase the size of the LFSR, the NLFSR, or both of them. We chose to increase the size of the LFSR by 21 bits what at the same time allows for a higher period of the output sequences.

The choice of the size was driven by the attack discussed in [Ban15]. Our analysis revealed that increasing the LFSR by 15 bits already makes the attack less efficient than a brute-force attack. We added further 6 bits to the total length to increase the security margin.

Double-layer LFSR. In the initialization phase, similar to Sprout, we do not output any keystream, but feed it back to the NLFSR and to the LFSR inputs. This is done in order to assure both registers depend on all the key and IV bits. Unfortunately, it may lead to the case that the LFSR falls into an all-zero state after the initialization phase. As long as during the keystream generation phase the feedback from the output function is not used, the LFSR would remain in the all-zero state during the entire encryption process which would result in existence of keystream sequences with very short period. In [Ban15], a key recovery attack was suggested based on this weakness. A typical countermeasure is to set one LFSR bit to 1 once the initialization phase is complete. However, this means that there are always two inputs to the cipher that lead to the same initial state, representing another weakness.

To avoid this problem, we use two different LFSRs in different phases, such that the LFSR used in the keystream generation phase is obtained from the LFSR used in the initialization phase by extending it with one additional bit set to 1. With respect to the lightweight hardware implementation, it is preferable that the two update polynomials of these LFSRs share as many terms as possible. To achieve this, we found two primitive polynomials of degrees 60 and 61 which only differ in the maximum-degree term. In hardware, a mechanism to decide how the stages with indexes 60 and 61 are updated depending on the current phase can be easily implemented by using two multiplexers. We call this approach a *double-layer LFSR* and consider it of independent value.

5 Security

In this section, we discuss the security of Plantlet. Since Plantlet is designed for a small area size, the maximum possible period has to be shorter in comparison to other ciphers that deploy a larger internal state. Hence, we assume that distinguishing attacks might be possible but consider them only applicable to scenarios that are less relevant in the context of lightweight devices, e.g., encrypting long data streams. Nonetheless, we will shortly discuss distinguishing attacks at the end of this section. Moreover, as we consider that the key is fixed, variable-key attacks (including related-key attacks) attacks are also out of scope.

Similar to Sprout, our goal for Plantlet is to achieve 80-bit security against key-recovery attacks in the single-key but chosen IV scenario. As explained in Section 4, Plantlet is in fact a modification of Sprout where the incorporated changes primarily target security gaps identified for Sprout. In this section, we first recall attacks that were found against Sprout [AM15], and explain what countermeasures we introduced in Plantlet to render these attacks inapplicable and argue why.

Merging and sieving technique. The first key recovery attack against Sprout was published in [LN15]. The basic idea of this attack is to cleverly enumerate possible states of the two registers used in Sprout. Then, given the observed keystream a sieving technique is applied which allows to discard states that cannot produce the observed bits. It was shown that during the process of sieving, it is feasible to not only reduce the number of possible states, but also to recover some of the key bits. The attack allows for recovering the whole key with a time effort equivalent to roughly 2^{69} of Sprout encryptions. We note that the sieving is viable mainly due to the nonlinear influence of the key on the update function, in other words, due to the fact that the key bits do not affect the internal state in every clock cycle. Therefore, with the new round key function of Plantlet, which takes one key bit every clock cycle and uses it in the state update process, this approach is not directly applicable anymore. Moreover, the complexity of this attack against Sprout is around 2^{69} encryptions and it directly depends on the sizes of the shift registers. Consequently, even if a sieving attack with similar efficiency would be possible, due to the increase of the internal state size by 21 bits, the total effort would exceed the effort of a brute force attack.

Guess-and-determine attacks. In a guess-and-determine attack, the attacker guesses part of the internal state and aims to recover the remaining parts with an overall effort lower than brute force. In [MSBD15] it is shown that if the attacker partially guesses the internal state of Sprout, it is possible to create a system of nonlinear equations which can be solved by a SAT solver in reasonable time to recover the key and remaining state bits. The paper does not provide the complexity of solving such systems, however, some experimental results are provided. For example, when 54 out of 80 bits of the internal state are known, the SAT solver finds 6.6 key candidates on average within approximately 77 seconds. In addition, this paper presents a fault attack on Sprout.

An improved guess-and-determine attack has been described in [Ban15]. It likewise demonstrates the possibility to recover the key from partial knowledge of the cipher state, yet in a more efficient way compared to the attack explained in [MSBD15]. According to the experiments conducted by the author, it holds that if 50 bits of the internal state are guessed, the remaining bits and the key can be found in around 31 seconds on an average PC. In [EK15] it was estimated that the time complexity of this attack equals roughly 2^{70} Sprout encryptions and hence would constitute an attack. Observe however that no theoretical analysis of the effort has been given.[7] Consequently, to estimate the

[7] In fact, the author of [Ban15] referred to his result as an observation rather than an attack.

susceptibility of Plantlet to this attack, we conducted several lines of experiments on our own, which were performed using the Cryptominisat 2.9.10 [Cry] solver in combination with SAGE 6.9 [Sag] and the same source code as the author of [Ban15]. In the first line of experiment, we repeated the experiments reported in [Ban15] to establish a base line for comparison. In our case, when 50 bits of the internal state were guessed for different key/internal state values, the solver needed 30 to 430 seconds for finding the key and the remaining state bits. In order to understand how the new round key function of Plantlet influences the complexity of this attack, we performed a second line of experiments on Sprout but replaced the round key selection mechanism with the one used in Plantlet. In this case, the unknown bits could be determined in 1300 seconds on average, which is significantly longer compared to the unmodified round key function as defined in Sprout.

Finally, we conducted a third line of experiments and performed the attack against Plantlet directly. For different numbers of guessed bits, we measured the time to set up the equations and to compute the solutions. The results are given in Table 4.

Table 4: Time required (in seconds) for applying the attack from [Ban15] against Plantlet.

Guessed bits	Set up time			Solve time			Total time		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
95	0.057	0.096	0.075	0.549	0.949	0.764	0.61	1.04	0.83
90	0.058	0.097	0.078	0.877	1.559	1.193	0.935	1.652	1.271
85	0.089	0.150	0.112	6.400	23.36	12.95	6.49	23.45	13.06
80	0.087	0.095	0.091	33.73	1416.99	669.45	33.81	1417.08	664.05
75			0.095			8992.36			8992.45

For all cases except for the case where 75 bits are guessed, we conducted five experiments for different randomly selected key and state values. For the case of 75 guessed bits, we only carried out one experiment as already this case turned out to be quite time consuming[8]. Observe that one can perform within one second around 2^{10} Plantlet encryptions on the same PC with a non-optimized Sage implementation of the cipher. Hence, the results indicate that each of the considered variants of the attack is much more expensive than exhaustive key search.

Another guess-and-determine attack is reported in [LYR15]. The first step of the attack is to transform the NLFSR used in Sprout to an equivalent NLFSR with a simplified output function, but a more involved update function. More precisely, the goal of this transformation is to decrease the number of variables used in the output function and to introduce several update functions into the NLFSR (Galois configuration). Then, for several rounds at each clock cycle, the attacker guesses all but one bit of the internal state which go into the output function. The last one, which was not guessed can be computed directly from the output function because the keystream bit is known. Then, this process is continued. At some clock cycles, it is possible to compute the bits using not only the output function, but also the feedback functions. As soon as the state is known, the key can be found very efficiently. This attack projects an average number of guesses of around $2^{70.87}$. However, we expect that the increase of the internal state size incurs an effort beyond 2^{80} which is thus worse than exhaustive key search.

Tradeoff-attacks. The authors of [EK15] demonstrated the first practical key recovery attack against Sprout. It allows to find the key in 2^{33} encryption time if the attacker is given data of 2^{40} bits of the keystream output with a memory requirement of 770 terabytes in total. The offline phase requires solving of approximately 2^{42} systems of linear equations

[8] In our experiment, the attack required 8992 seconds which is about 2.5 hours.

with 20 unknowns each. The attack idea is to look for sequences where the key bits are not involved in the update function and to apply the common birthday paradox trade-off attack to these sequences.

A further trade-off attack on Sprout is given in [ZG15]. The time of the attack is 2^{79-x-y} with memory complexity of $[c \cdot (2x + 2y - 58) \cdot 2^{71-x-y}]$ and data complexity of 2^{9+x+y}, where x and y are the numbers of forward and backwards clockings of the cipher under the assumption that there is no involvement of the key bits in the state update.

Both trade-off attacks [EK15, ZG15] against Sprout require the existence of keystream sequences of certain lengths which are generated without any influence of the key. However, due to the fact that in Plantlet one key bit is involved in the state update at every clock cycle and hence propagates to the keystream, we do not see a possibility for these attacks to still be effective.

All-zero LFSR state. In [Ban15] it is shown that for every key approximately 2^{30} IVs exist for which the LFSR gets into the all-zero state during the keystream generation phase. This allows to find Key-IV pairs which result in the sequences which have a period equal to 80 and also to mount a key recovery attack with the complexity equivalent to $2^{66.7}$ Sprout encryptions with negligible memory requirements.

Such a situation cannot occur in Plantlet due to the use of the double-layer LFSR. That is, one of the LFSR bits of the longer LFSR is definitely set to "1" before the keystream generation phase begins. Moreover, the LFSR update polynomial is primitive, hence excluding the all-zero state.

Distinguishing attacks. As explained at the beginning, we are aware that similar to Sprout distinguishing attacks are probable due to the relatively short period (although Plantlet should provide higher periods because of the use of a longer LFSR). From our point of view, it is a general problem that decreasing the area size makes it more and more difficult to ensure high periods. Moreover, one may argue that common scenarios deploying resource-constrained devices do not require the encryption of very long data streams. Nonetheless, we take a short look at existing distinguishing attacks against Sprout and discuss if and how they would apply to Plantlet as well.

In fact the first attack on Sprout was a related-key distinguishing attack presented in [Hao15]. However, we consider such attacks less relevant in the case of hardware ciphers as changing the key takes significant effort or is even impossible. Observe that attacks that use related IVs in the sense that initialization involves the same key but different, yet related public values do not apply here either as the internal update procedure depends on the (fixed) key.

However, Plantlet may also be subject to the distinguishing attack explained in [Ban15] against Sprout. The attack looks for two IVs that result in $(80 \cdot P)$-bit shifted keystreams, where P is a positive integer. This attack is possible due to the simplicity of the round key function. It would be possible to make the design resistant against this attack by either choosing a more complicated key-selection function or by further increasing internal state size. The first countermeasure would by all means result in a lower throughput because of the timing issues inherent to reading from serial EEPROM, the second would result in higher area size. On the other hand, the memory complexity of this distinguishing attack against Plantlet is at least 2^{58} which is about $32,768$ terabytes. As long as we are aiming for ultra lightweight devices, we think that it is not reasonable to introduce countermeasures against this attack at the cost of area and performance characteristics.

6 Implementation Results

Next, we explain and discuss our implementation of Plantlet. The implementation results for Plantlet are provided in Table 5. We do not consider any costs for *storing* the key in the EEPROM memory because we assume that it has to be provided by the device anyway, independent of whether it needs to load the key only once for initialization or requires constant access to the EEPROM. However, we do consider the costs incurred by the control logic which is required to *synchronize* the cipher with commercial EEPROM modules. That is, for the clock cycles when the key bits are not output by the EEPROM, we switch off the clocking of the cipher.

Table 5: Implementation results for Plantlet considering that the key is continuously read from different types of NVM.

Hard-wired[9]		Integrated EEPROM				
Area (in GE)	Throughput (in kbit/s)	Area (in GE)				Throughput (in kbit/s)
		x1	x8	x16	x32	
928	100	807	852	897	929	100

Commercial serial EEPROM modules

Interface	wrap case		no-wrap case	
	Area (in GE)	Throughput (in kbit/s)	Area (in GE)	Throughput (in kbit/s)
I^2C	822	88.9	885	66.7
SPI	807	100	860	83.3
Microwire	807	100	860	87.9
UNI/O	831	80	898	53.3

For the implementation, we used the same tools as the authors of Sprout [AM15]. More precisely, we used Cadence RTL Compiler[10] for synthesis and simulation, and the technology library UMCL18G212T3 (UMC 0.18μm process). We considered also the same clock frequency of 100 kHz which is the most common rate for lightweight devices and supported by most of the existing commercial EEPROMs.

In such a scenario and due to the fact that Plantlet simply reads out the key bits sequentially, there is no need for using multiplexers for selecting the round key bits as this is directly provided by the discussed serial interfaces. Thus, we consider it as reasonable and fair approach to re-use the existing mechanisms when designing a cipher. Depending on the chosen interface, our implementations of Plantlet require a minimum of 807 GE (e.g. for SPI in the wrap case) and a maximum of 898 GE (e.g. for UNI/O in the no-wrap case).

We also investigated cases in which the key is stored in customized EEPROM. The most common architectures convey the possibility to read either one bit from a random memory location or an n-bit word with a given address per memory access. Here, typical word sizes are 8, 16, and 32 bits. Our smallest implementation with respect to these different word sizes requires mere 807 GE. As another alternative, we examined using a lower frequency for accessing the EEPROM compared to the clock-frequency of the cipher itself for which then buffer registers are used. The resulting area sizes are also shown in Table 5. From

[9] By hard-wired we mean all the techniques which do not allow for changing the key, i.e. MROM or PROM

[10] See http://www.cadence.com/products/ld/rtl_compiler/pages/default.aspx

our point of view, this approach might be practically relevant as it could potentially allow for reducing the power consumption of reading from EEPROM considerably.

Due to the fact that Sprout also accesses the key bits cyclically, the same tricks could be used here as well. To validate this, we implemented a corresponding variant of Sprout, i.e., where the additional logic is removed and instead the EEPROM logic is used. As expected, this resulted in a further decrease of the area size of the cipher: from 813 GE to 692 GE. While this implementation of Sprout is significantly smaller than Plantlet, it does *not* contain any additional security measures of any kind, i. e., its security remains unaffected and, of course, it still suffers from the weaknesses that were discussed in Section 5.

We contrast this scenario to the case when the key is stored in PROM (i.e. using fuses and antifuses) or MROM, which incurs additional logic for retrieving the current key bit as required by the respective round. According to [AM15], the Sprout implementation assuming the key being burnt into the device needs 813 GE. The corresponding implementation for Plantlet results in 928 GE. For comparison, Table 6 provides information on several lightweight stream and block ciphers and their respective area requirements, however, since in the previously published papers the authors were not considering the approach of constantly reading from rewritable non-volatile memory, the figures are given for *fixed* keys where applicable.

Table 6: Overview of different lightweight ciphers regarding key and block size, throughput at 100 kHz, the logic process (denoted Tech.), and the required area size. The figures are given for the case, when the key is stored in non-rewritable memory (e.g. MROM or PROM). Note that for A2U2, no logic process information is given and the area size is an estimate as reported in [DRL11]. As there were no throughput figures given for Midori in [BBI+15], we computed them based on the number of rounds and the block size as described for the original implementation.

Cipher	Key size	Block size	Throughput [kbit/s]	Tech. [μm]	Area [GE]
Stream ciphers					
A2U2 [DRL11]	56	1	100	–	300
Sprout [AM15]	80	1	100	0.18	810
Grain 80 [AM15]	80	1	100	0.18	1162
Trivium [GB08]	80	1	100	0.13	2580
Plantlet	80	1	100	0.18	928
Block ciphers					
KTANTAN [CDK09]	80	32	12.5	0.13	462
		48	18.8		588
		64	25.1		688
LED [GPPR12]	64	64	5.1	0.18	688
	128		3.4		700
Midori [BBI+15]	128	64	400	0.09	1,542
		128	640		2,522
PRESENT-80 [BKL+07]	80	64	200	0.18	1,570
PRINTcipher [KLPR10]	80	48	100	0.18	503
	160	96			967

7 Conclusion

In this paper, we revisited the approach of realizing lightweight ciphers that constantly access the non-volatile key during the encryption/decryption process. In use cases where it is sufficient to set the key only once, the assumptions made in existing works are absolutely fine and the schemes should work as predicted. The situation is different however if the use case requires that the key can be rewritten at later points in time. In such cases, different techniques are imaginable but commercial products are mostly using here the established technology EEPROM. In such cases, designs that sequentially access the key bits are better suited for achieving a high throughput than those that require selective access. Based on this result, we revisited existing ciphers for this particular use case. Moreover, we presented and discussed a new cipher, Plantlet. Our new design achieves high throughput and very small area size regardless of the underlying NVM technique ranging from highly customized to commercial standard solutions, thus rendering it universal for all common types of NVM in the domain of ultra-constrained devices. We see this work as a significant contribution towards very small but still practical lightweight ciphers and in particular towards bridging the gap between cryptographic models and practical realizations.

In general, we consider the idea of constantly involving the non-volatile key as an interesting and promising design technique. Further investigating the approach of storing precomputed round keys (in case of block ciphers) in NVM and hence to save logic may also be of interest (although it would require in practice to protect more memory against physical attacks). Another possibly interesting line of research could be the design of more involved round key functions which respect sequential reading, e.g., by buffering some memory words in registers, or in general to investigate the trade-off between involving non-volatile and volatile memory.

References

[Ade16] Adesto Techonologies. *AT25SF041 4-Mbit, 2.5V Minimum SPI Serial Flash Memory with Dual-I/O and Quad-I/O Support*, 2016. http://www.adestotech.com/wp-content/uploads/DS-AT25SF041_044.pdf.

[AHM14] Frederik Armknecht, Matthias Hamann, and Vasily Mikhalev. Lightweight authentication protocols on ultra-constrained RFIDs - myths and facts. In Nitesh Saxena and Ahmad-Reza Sadeghi, editors, *Radio Frequency Identification: Security and Privacy Issues - 10th International Workshop, RFIDSec 2014, Oxford, UK, July 21-23, 2014, Revised Selected Papers*, volume 8651 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2014.

[AM15] Frederik Armknecht and Vasily Mikhalev. On lightweight stream ciphers with shorter internal states. In Gregor Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, volume 9054 of *Lecture Notes in Computer Science*, pages 451–470. Springer, 2015.

[Atm14] Atmel Corporation. *CryptoRF EEPROM Memory 13.56MHz, 4 Kilobits, Summary Datasheet*, 2014. http://www.atmel.com/Images/Atmel-8672S-CryptoRF-AT88RF04C-Datasheet-Summary.pdf.

[Atm15] Atmel Corporation. *3-wire Serial EEPROM 1K (128 x 8 or 64 x 16), Data sheet*, 2015. http://www.atmel.com/Images/Atmel-5193-SEEPROM-AT93C46D-Datasheet.pdf.

[Ban15] Subhadeep Banik. Some results on Sprout. In Alex Biryukov and Vipul Goyal, editors, *Progress in Cryptology - INDOCRYPT 2015 - 16th International Conference on Cryptology in India, Bangalore, India, December 6-9, 2015, Proceedings*, volume 9462 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2015.

[BBI+15] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In *Advances in Cryptology–ASIACRYPT 2015*, pages 411–436. Springer, 2015‘.

[BKL+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[BL08] Raymond E Barnett and Jin Liu. An EEPROM programming controller for passive UHF RFID transponders with gated clock regulation loop and current surge control. *Solid-State Circuits, IEEE Journal of*, 43(8):1808–1815, 2008.

[CDK09] Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - A family of small and efficient hardware-oriented block ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2009.

[Cry] Cryptominisat-2.9.10. http://www.msoos.org/cryptominisat2/.

[Cyp15] Cypress Semiconductor Corporation. Parallel NOR Flash Memory: An Overview, 2015. http://www.cypress.com/file/202491/download.

[CZDL13] Zhaoxian Cheng, Xiaoxing Zhang, Yujie Dai, and Yingjie Lu. Design techniques of low-power embedded EEPROM for passive RFID tag. *Analog Integrated Circuits and Signal Processing*, 74(3):585–589, 2013.

[DRL11] Mathieu David, Damith C Ranasinghe, and Torben Larsen. A2U2: a stream cipher for printed electronics RFID tags. In *RFID (RFID), 2011 IEEE International Conference on*, pages 176–183. IEEE, 2011.

[EK15] Muhammed F Esgin and Orhun Kara. Practical cryptanalysis of full Sprout with TMD tradeoff attacks. In *International Conference on Selected Areas in Cryptography*, pages 67–85. Springer, 2015.

[Fre14] Fremont Micro Devices. *FT25H04/02 DATASHEET*, 2014. http://www.fremontmicrousa.com/pdf/FT25H04_Rev1p0.pdf.

[GB08] Tim Good and Mohammed Benaissa. Hardware performance of eStream phase-iii stream cipher candidates. In *Proc. of Workshop on the State of the Art of Stream Ciphers (SACS'08)*, 2008.

[GPPR11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The LED block cipher. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 326–341. Springer, 2011.

[GPPR12] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The LED Block Cipher. Cryptology ePrint Archive, Report 2012/600, 2012. http://eprint.iacr.org/2012/600.

[Hao15] Yonglin Hao. A related-key chosen-IV distinguishing attack on full Sprout stream cipher. *IACR Cryptology ePrint Archive*, 2015:231, 2015.

[KKN+02] Jesung Kim, Jong Min Kim, Sam H Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[KLPR10] Lars R. Knudsen, Gregor Leander, Axel Poschmann, and Matthew J. B. Robshaw. PRINTcipher: A Block Cipher for IC-Printing. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 16–32. Springer, 2010.

[LCK10] Kyoung-Su Lee, Jung-Hoon Chun, and Kee-Won Kwon. A low power CMOS compatible embedded EEPROM for passive RFID tag. *Microelectronics Journal*, 41(10):662–668, 2010.

[LN15] Virginie Lallemand and María Naya-Plasencia. Cryptanalysis of full Sprout. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 663–682. Springer, 2015.

[LYR15] Gefei Li, Yuval Yarom, and Damith Chinthana Ranasinghe. Exploiting transformations of the Galois configuration to improve guess-and-determine attacks on NFSRs. *IACR Cryptology ePrint Archive*, 2015:1045, 2015.

[Mac14] Macronix International Co.. Ltd. Introduction to NAND in Embedded Systems, 2014. http://www.macronix.com/Lists/ApplicationNote/Attachments/1162/AN0269V2_Introduction%20to%20NAND%20in%20Embedded%20Systems-0220.pdf.

[Mic] Micron Technology, Inc. NOR | NAND Flash Guide. https://www.micron.com/~/media/documents/products/product-flyer/flyer_nor_nand_flash_guide.pdf.

[Mic01] Microchip. *KeeLoq Code Hopping Encoder, Product data sheet*, 2001. http://ww1.microchip.com/downloads/en/DeviceDoc/21137f.pdf.

[Mic11] Microchip Technology Inc. *1K-16K UNI/O Serial EEPROM Family Data Sheet*, 2011. http://ww1.microchip.com/downloads/en/DeviceDoc/22067J.pdf.

[Mic14] Microchip Technology Inc. *SPI Serial EEPROM Family Data Sheet*, 2014. http://ww1.microchip.com/downloads/en/DeviceDoc/22040A.pdf.

[MSBD15] Subhamoy Maitra, Santanu Sarkar, Anubhab Baksi, and Pramit Dey. Key recovery from state information of Sprout: Application to cryptanalysis and fault attack. *IACR Cryptology ePrint Archive*, 2015:236, 2015.

[Nat01] National Institute of Standards and Technology, U.S. Department of Commerce. FIPS PUB 197, Advanced Encryption Standard (AES), November 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[NKTZ12] Andrey Nuykin, Alexander Kravtsov, Sergey Timoshin, and Igor Zubov. A low cost EEPROM design for passive RFID tags. In *Communications and Electronics (ICCE), 2012 Fourth International Conference on*, pages 443–446. IEEE, 2012.

[NXDH06] Yan Na, Tan Xi, Zhao Dixian, and Min Hao. An Ultra-Low-Power Embedded EEPROM for Passive RFID Tags. *Chinese Journal of Semiconductors*, 27(6), 2006.

[NXP11] NXP Semiconductors. *MIFARE Classic 1K - Mainstream contactless smart card IC for fast and easy solution development, Product short data sheet*, 2011. http://www.nxp.com/documents/short_data_sheet/MF1S50YYX.pdf.

[NXP14] NXP Semiconductors. *HITAG S transponder IC., Product short data sheet*, 2014. http://www.nxp.com/documents/short_data_sheet/HTSICH56_48_SDS.pdf.

[NXP15] NXP Semiconductors. *HITAG μ transponder IC.HTMS1x01; HTMS8x01. Product data sheet*, 2015. http://www.nxp.com/documents/data_sheet/HTMS1X01_8X01.pdf.

[On 14] On Semiconductor. *1-Kb, 2-Kb, 4-Kb, 8-Kb and 16-Kb I2C CMOS Serial EEPROM, Data sheet*, 2014. http://www.onsemi.com/pub_link/Collateral/CAT24C01-D.PDF.

[PSS+08] Jivan Parab, Santosh A Shinde, Vinod G Shelake, Rajanish K Kamat, and Gourish M Naik. *Practical aspects of embedded system design using microcontrollers*. Springer Science & Business Media, 2008.

[Sag] Sage mathematics software. http://www.sagemath.org/.

[VGE15] Roel Verdult, Flavio D Garcia, and Baris Ege. Dismantling megamos crypto: Wirelessly lockpicking a vehicle immobilizer. In *Supplement to the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 703–718, 2015.

[ZG15] Bin Zhang and Xinxin Gong. Another tradeoff attack on sprout-like stream ciphers. *Accepted at AsiaCrypt 2015*, 2015.

A Standard Interfaces Used for Reading from Commercial EEPROM Modules

A.1 I²C Interface

If the I²C interface is used, the memory is usually organized in *memory words* with a size of 8 bits. For accessing bits from the EEPROM, the following types of readings are relevant:

Random (Selective) Reading: This type of reading allows to "jump" to a specific EEPROM address for extracting the values stored there. This allows for the highest level of flexibility, but also incurs additional costs. To perform this type of read operation, first the word address must be reset (which usually requires 19 clock cycles). Then, the control byte has to be sent to the EEPROM which in turn has to acknowledge its receipt. This takes 10 clock cycles. Only if this is accomplished, the EEPROM starts to send the data of the requested memory word (8 clock cycles) and waits for the acknowledgment bit (1 clock cycle) and the `STOP` signal (1 clock cycle). Therefore, reading a word of 8-bits from a freely chosen address requires 39 clock cycles in total.

Current Address Reading: The current address reading uses an internal address counter that is increased after each reading. To initiate the read operation, a control byte preceded by a `START` signal is sent to the EEPROM and gets acknowledged (10 clock cycles). Then, the current memory word is sent (8 clock cycles) and the address counter is increased. The read operation is terminated by a `NoACK` signal followed by a `STOP` signal (additional 2 clock cycles). In total, a current address read, i. e., reading 8 bits from the current memory address, requires 20 clock cycles.

Sequential Reading: Similar to current address reading, sequential read allows to read a memory word from the current address and increases the address afterwards. The main difference is that it keeps on doing so until instructed otherwise. That is sequential read outputs memory words from EEPROM one by one starting at the current address and is preceded by a random or a current address read. After each sent memory word, the EEPROM expects an acknowledgement bit. The read operation continues by increasing the address counter and sending the next memory word until the device receives a `NoACK` followed by a `STOP`. Therefore, reading one memory word sequentially requires an additional clock cycle for the acknowledgement, i. e., 9 clock cycles. Considering the condition that a preceding current address or random read must occur, reading n memory words requires $10 + 9n + 1$ or $29 + 9n + 1$ clock cycles, respectively.

A.2 SPI

Usually, the memory is organized in 8-bit words. To read from EEPROM using SPI, the host sends a `READ` instruction (8 bits) followed by a memory address, which may consist of 8, 16, or even 24 bits, to the EEPROM device. After receiving the last address bit, the EEPROM responds by shifting out data on the serial output data pin. Sequentially stored data can be read out for as long as the clock signal is provided. After each provided memory word, the internal address pointer is automatically incremented to point to the next address. If the highest memory address is reached, the address counter "rolls over" to the lowest, and the read cycle can be continued indefinitely. Thus, using an address width of a, reading n memory words requires $8 + a + 8n$ clock cycles, e. g., $16 + 8n$ clock cycles are required in the $a = 8$ case.

A.3 Microwire

This interface is a predecessor of SPI and provides only a subset thereof. The EEPROM is organized in either 8-bit or 16-bit memory words. A `READ` instruction consists of a start bit, a 2-bit op-code, and 7 or 6 address bits for the requested 8-bit or 16-bit memory word, respectively. After receiving and decoding, the data is transferred from the memory to an output shift register. At first, a dummy 0 bit is sent, followed by the memory word, while each next bit is indicated by the rising edge of the clock signal. The device automatically increments the internal address register and sends subsequent memory words until indicated to stop. However, the dummy 0 bit is only sent once, subsequent memory words are output as a continuous stream of data. If the highest memory address is reached, the address counter rolls over and points to the lowest memory address. Reading n words from memory results in $3 + a + 1 + 8n, a \in \{6, 7\}$ clock cycles, e. g., $11 + 8n$ for memory organized in 8 bits.

A.4 UNI/O

Memory accessible via this interface type is usually organized in 8-bit memory words. Before a read operation can be performed, a `Start Header` and a memory address of 8 bits each are transmitted, which are followed by an acknowledgement phase in which both the host and the EEPROM device send 1 bit each. Altogether, this setup phase consumes 20 clock cycles. Then, the actual read command comprising 8 bits is sent and acknowledged in the same way. Immediately, the requested memory address is transmitted. A memory address consists of two 8-bit parts, whereas an acknowledgement phase takes place after each part is sent, thus requiring 30 clock cycles for the command and memory address. Besides this selective read operation, a current read command can be used which relies on an internal address pointer that is automatically increased after each read operation. Using this command, the memory address is omitted and only 10 clock cycles for the command and subsequent acknowledge bits are required. Nonetheless, both read operations allow for continuously reading consecutive memory words. The internal address counter rolls over to the lowest memory address after the highest was reached. Each memory word is followed by 2 acknowledge bits, i. e., reading n consecutive bytes from memory requires $50 + 10n$ and $30 + 10n$ for selective and current read operations, respectively.

B Key Selection Functions of Considered Ciphers

B.1 Midori

Midori [BBI+15] is a block cipher that targets low energy applications and provides a block size of 64 or 128 bits, i. e., Midori64 and Midori128, with a key size of 128 bits for both variants. It does not employ any dedicated key schedule function due to energy consumption concerns. Similar to AES, it consists of a state of 64 bits (or 128 bit for Midori128) which is transformed by 15 (or 19) rounds of substitution, permutation, and (round) key addition layers. Additionally, key whitening resulting in two extra key additions is performed before the first and after the last round. Note that any round key is obtained by XORing the supplied key with pre-defined round constants, in the 64-bit case however, the supplied 128-bit key is reduced to 64 bits by XORing its most significant and least significant half beforehand.

Thus, for encryption of one 64-bit (or 128-bit) block of data, 1088 (or 2688) key bits are required. In other words, 136 (or 336) 8-bit memory words need to be retrieved, which results in equally many EEPROM read accesses in the wrap case. In the no-wrap case, at least 9 (or 19) read operations would be *selective*, while all remaining ones could be performed sequentially. Considering round key precomputation, this approach is almost

equivalent to the wrap case except for the obvious non-negligible increase in memory demand.[11]

B.2 LED

LED [GPPR11] is a block cipher targeting a low hardware footprint with a block size of 64 bits, two *primary*[12] key sizes of 64 or 128 bits, respectively, and no dedicated key schedule. Similar to AES, the cipher consists of several rounds of substitution, permutation, and key addition layers but are applied to a state of 64-bit only. By design, the actual number of rounds depends on the key size. A peculiarity is that 4 rounds constitute one *step* while the key addition is only performed once *before* each step plus one additional key addition after the last step. For 64-bit keys, 32 rounds or 8 steps are performed, which amounts to a total of 9 key additions versus 48 rounds or 12 steps and 13 key additions in the 128-bit case.

For processing one 64-bit block of data, hence, 576 versus 832 key bits or 72 versus 104 8-bit memory words are required, respectively. Due to the absence of any key schedule, storing the key bits redundantly, i. e., in extended form, is essentially equivalent to the wrap case.[13] In the no-wrap case however, after reading all key bits, the memory address needs to point to the first memory word again; this happens 8 times in the 64-bit key case and 6 times in the 128-bit key case. Consequently, 64 versus 98 read operations may be performed sequentially.

B.3 KTANTAN

The KTANTAN family [CDK09] represents a set of block ciphers providing 3 different block sizes, i. e., $32, 48, 64$. Independent of the chosen block size, the family uses an 80-bit key and utilizes a key schedule that relies on the current internal state to choose 5 bits from this key. By design, the key is treated as 5 consecutive 16-bit words and the selected bits have the same relative position inside each 16-bit word. Finally, only 2 of these 5 bits are used, whereas the selection is also based on the current internal state. Occasionally, it may happen with a probability of $1/4$ that the same key bit is picked twice. We consider this to be the best case with respect to the throughput and hence assume in favor of the cipher that this always happens. As the selection of the key should not follow any easily predictable pattern, we furthermore assume that each time a key bit is needed, the device has to initiate a selective reading. For sure, one may increase the throughput by for example buffering the recently read key word and checking whether the same key word is needed again. We do not consider such improvements in our analysis here for two reasons. First, this would require additional logic and registers. Second, different selection patterns may result into different timing behavior and hence may allow for side-channel attacks. Thus, we leave it as an open question if and how the throughput of the KTANTAN family could be increased without consuming too much additional area and without compromising the security.

According to [CDK09], KTANTAN uses 254 rounds in which the key schedule is involved, that is, in each round one access to the key is necessary. In other words, for processing 1 block of input data, 254 EEPROM read operations and 254 clock cycles are required. Note that there is no need to distinguish the no-wrap case from the wrap case since all memory read operations require a specific address which are only provided by selective read.

[11] To the advantage of the cipher, we assume that the length of memory addresses covers exactly the NVM and that no additional effort incurs when addressing higher memory regions.

[12] In fact, any key size from 64 bits up to 128 is imaginable as stated in [GPPR11] by extending keys of more than 64 bits to 128 bits.

[13] We note that the situation is different for any key sizes other than 64 or 128 bits due to the way such keys are handled during the process as of the updated version of the LED specification [GPPR12].

Of course if one aims to avoid the overhead incurred by selectively accessing the key bits, one could deploy the approach already mentioned in Section 1. More precisely, recall that during each of the 254 rounds of KTANTAN, 2 key bits are required that are taken from the 80-bit key. Hence, instead of storing the 80-bit key in EEPROM and being forced to apply selective reading, one could instead store all *necessary* key bit values in the exact order as required by the cipher. That is, given that encryption involves accessing the key 254 times and to extract two bits each, this would result into storing a total of 508 precomputed key bit values. Obviously, this yields a storage overhead since additional 428 bits would be stored in EEPROM compared to storing the 80-bit key only. Any logic responsible for key bit selection can however be omitted. That is logic is traded for EEPROM area in this case. Assuming such a precomputation approach, KTANTAN requires 508 key bits retrieved from EEPROM, i. e., 63.5 sequential read operations for 8-bit memory words.[14] For the sake of comparison, the throughput for this approach is also given in Table 2.

B.4 PRINTcipher

The PRINTcipher [KLPR10] is a block cipher which is available with different block sizes. PRINTcipher-48 uses a block size of 48 bits with an 80-bit key, whereas PRINTcipher-96 uses a block size of 96 bits with a 160-bit key. The number of encryption rounds per input block is defined by the block size, e. g., PRINTcipher-48 performs 48 rounds, on which we focus in the following.

During encryption, the key is used in two different ways. In each round, the first 48 bits are directly used by XORing them with the current state. This state is then shuffled and combined with a round constant. After this, the state and the remaining 32 key bits are used in the key-dependent permutation layer which takes 3 bits from the state and 2 bits from the remaining 32 key bits to produce the input to an S-box. All these operations can be realized by reading the key bits sequentially from the non-volatile memory in each round.

Obviously, this design would fit best into the wrap case. In the no-wrap case, additional effort needs to be taken to restart reading the key bits from the beginning. Of course, this could be circumvented if the expanded key is stored in EEPROM (similarly as discussed for KTANTAN above). This would result into an increased throughput similar to the wrap case, but at the cost of requiring significantly more EEPROM storage ($3,840$ bits instead of 80).

B.5 A2U2

The stream cipher A2U2 [DRL11] uses a 56-bit key. For each generated output bit, 5 consecutive key bits are used in the process. This implies that sometimes it is sufficient to retrieve one memory word only (if all five bits are within the same word), but sometimes two words are necessary (if some of the bits fall into the current memory word and the remaining ones in the subsequent word). This is independent of the underlying memory word size. As eight bits are a common size for memory words, we assume this value in the following. Thus, producing 8 output bits requires retrieving 40 key bits from memory which translates to 4 1-word and 4 2-word EEPROM read operations. Additionally, only 1 of 8 read cases may be performed as a current address read, in all other cases a previously requested memory address has to be read again which requires a selective read. On average, one has to read 1.5 memory words per output bit where a fraction of $\frac{1}{7}$ can be realized as current address reads.

Table 2 provides an overview of the achievable throughput for different interfaces considering both the wrap case and the no-wrap case with an underlying memory word size

[14] Technically, 64 read operations would be performed.

of 8. Additionally, if provided by the interface, we assume to utilize the current address read which is possible for generating 1 out of 8 output bits. Note that it may happen that the last and first memory word are required to produce an output bit which in the no-wrap case requires 2 selective read operations requesting the 2 memory words separately instead of only 1. However, this occurs 4 times per 56 generated output bits. Hence, there is only a slight difference noticeable between those cases.

B.6 Sprout

The stream cipher Sprout [AM15] uses an 80-bit key. In a nutshell, each of the key bits is accessed in a sequential fashion, i.e., starting with the first bit followed by the subsequent bit and so forth, although not necessarily considered for further processing due to the round key function. Nonetheless, it is reasonable to assume that 1 key bit is required per produced output bit. Considering a memory word size of 8 bits, 10 EEPROM read operations are required for producing 80 output bits.

This is particularly useful for the wrap case, as this implies that the read operation needs only be initiated once, while the *current* key bit is available immediately, e.g., to produce output not limited to 80 bits.

In the no-wrap case on the other hand, after the 80^{th} key bit is used, a selective read operation is needed to continue output production which then requires reading the first key bit again. This affects the achievable throughput and limits it to different values for the considered interfaces as presented in Table 2. Contrary to A2U2, there is an absolute difference noticeable between the wrap case and the no-wrap case.