Empirical Evaluation of Memory-Erasure Protocols

Reynaldo Gil-Pons[1][a], Sjouke Mauw[1][b] and Rolando Trujillo-Rasua[2][c]
[1]*University of Luxembourg, 6 Avenue de la Fonte, Belval, Luxembourg*
[2]*Universitat Rovira i Virgili, 26 Avinguda dels Països Catalans, Tarragona, Spain*

Keywords: Security Protocol, Memory Erasure, Malware, Internet-of-Things, Empirical Evaluation.

Abstract: Software-based memory-erasure protocols are two-party communication protocols where a verifier instructs a computational device to erase its memory and send a proof of erasure. They aim at guaranteeing that low-cost IoT devices are free of malware by putting them back into a safe state without requiring secure hardware or physical manipulation of the device. Several software-based memory-erasure protocols have been introduced and theoretically analysed. Yet, many of them have not been tested for their feasibility, performance and security on real devices, which hinders their industry adoption. This article reports on the first empirical analysis of software-based memory-erasure protocols with respect to their security, erasure guarantees, and performance. The experimental setup consists of 3 modern IoT devices with different computational capabilities, 7 protocols, 6 hash-function implementations, and various performance and security criteria. Our results indicate that existing software-based memory-erasure protocols are feasible, although slow devices may take several seconds to erase their memory and generate a proof of erasure. We found that no protocol dominates across all empirical settings, defined by the computational power and memory size of the device, the network speed, and the required level of security. Interestingly, network speed and hidden constants within the protocol specification played a more prominent role in the performance of these protocols than anticipated based on the related literature. We provide an evaluation framework that, given a desired level of security, determines which protocols offer the best trade-off between performance and erasure guarantees.

1 INTRODUCTION

The low-cost characteristics of some classes of IoT (Internet of Things) devices and the ecosystem in which they operate make them particularly vulnerable to malware infection and an attractive target for cyber-attackers. Defenders, in a context of limited computational resources, cannot rely on health-monitoring software installed on the IoT device to fight malware. Instead, defenders ought to resort to security services provided by an external agent that, by interacting with the IoT device and observing its behaviour, can conclude whether the device is malware-free.

There exist two fundamental security services to ensure that computationally-constrained devices are malware-free: memory attestation and memory erasure. The former starts from an expectation on the contents of a device's memory and the assumption that its contents are malware-free, followed by a mechanism to verify the integrity of the device's memory. That is, memory attestation checks whether a device's memory is in a *known* safe state. Memory erasure, instead, replaces the contents of a device's memory with random data, effectively removing all information, including malware, from the device. Both memory attestation and memory erasure have their use-cases. Our focus in this article is on memory erasure, which allows defenders to remove malware without making assumptions on the contents of the device's memory. This service is particularly useful for updating the memory of a device with new software securely, allowing the device to be redeployed elsewhere.

Memory-erasure mechanisms that depend on having physical access to the IoT device or using dedicated secure hardware on the IoT device, such as a Trusted Platform Module, have the potential to offer high security assurances. However, the first requirement makes memory erasure unscalable and time-consuming, while the second requirement makes memory erasure expensive or infeasible in low-cost

[a] https://orcid.org/0000-0003-1804-3319
[b] https://orcid.org/0000-0002-2818-4433
[c] https://orcid.org/0000-0002-8714-4626

Gil-Pons, R., Mauw, S. and Trujillo-Rasua, R.
Empirical Evaluation of Memory-Erasure Protocols.
DOI: 10.5220/0013554800003979
In *Proceedings of the 22nd International Conference on Security and Cryptography (SECRYPT 2025)*, pages 209-220
ISBN: 978-989-758-760-3; ISSN: 2184-7711

devices. Mechanisms requiring none of those features, which are, hence, suitable for low-cost IoT devices, are called *software-based memory-erasure* protocols. In theory, these protocols have the advantage of being able to operate on low-cost, even legacy, devices. In practice, they have not been thoroughly evaluated for their feasibility on off-the-shelf IoT devices.

Before introducing software-based memory erasure, it is important to make a distinction between *permanent memory erasure* (a.k.a. data destruction) and the type of memory erasure we are referring to. Permanent memory erasure requires the erasure process to be irreversible against advanced forensics techniques (Reardon et al., 2013), including those that rely on physical access to the device. Irreversibility, however, is not necessarily required for malware removal, hence the literature on memory erasure (Perito and Tsudik, 2010; Karvelas and Kiayias, 2014; Karame and Li, 2015) where our work fits in does not aim at irreversibility.

Software-based memory erasure is a two-party communication protocol executed between a verifier, acting as a powerful computational device, and a prover, acting as a devise with limited computational resources. The role of the verifier is to instruct the prover to fill its memory with random data and verify the erasure proof generated by the prover. Even though several software-based memory-erasure protocols have been proposed, they have never been compared within the same experimental setting. Moreover, those that have been implemented and tested on real-world devices (Perito and Tsudik, 2010; Karame and Li, 2015; Ammar et al., 2018), have not made their source-code publicly available for further scrutiny and analysis. Hence, it is still an open question how existing software-based memory-erasure protocols perform on low-cost IoT devices and how they compare in terms of computational and communication complexity, erasure guarantees, and security. This article provides an answer, which we argue is a necessary step towards the adoption of software-based memory-erasure protocols by the IoT industry.

A brief discussion on related work. The special characteristics of constrained IoT devices have pushed practitioners to find the most performant and efficient algorithms for each task. Ultimately, testing the behaviour of an algorithm requires deploying them on real devices and conducting performance evaluations. This has been done recently for lightweight hash functions (Rao and Prema, 2019), cryptographic algorithms (Silva et al., 2024), and data protection mechanisms (Lachner and Dustdar, 2019). However, as pointed out by recent surveys on the topic (Banks et al., 2021; Kuang et al., 2022), such level of scrutiny has not yet been achieved for memory erasure and memory attestation. That is not to say that these protocols have never been compared with each other. For example, Aman et al. (Aman et al., 2020) compare their memory-attestation protocol against three alternatives from the literature. In the case of memory erasure, Karame and Li (Karame and Li, 2015) and Perito and Tsudik (Perito and Tsudik, 2010) evaluate the performance of their protocols on off-the-shelf IoT devices, but do not directly compare them, nor implement other proposals. A common issue of these examples is the lack of available open-source implementations, making it hard to reproduce their evaluation and to comprehensively compare existing proposals within a common empirical setting.

Contributions. This article provides the first comparison and evaluation of software-based memory-erasure protocols by directly observing and analysing their performance in a real-world experimental setting. We claim this to be a necessary step towards their adoption by industry and their deployment in the real-world. Crucially, we aim at answering the following questions about existing software-based memory-erasure protocols:

1. Can they be implemented in low-cost IoT devices? If so, what is their memory footprint and execution time?
2. How much is their execution time affected by the computational power of the device, the size of the memory to erase, the implementation of the underlying hash function, and the speed of the communication channel?
3. Is there a dominant protocol in terms of performance and security, i.e. a protocol that performs better than the others in all settings? If not, what are the best trade-offs?

Our experimental setting consists of three off-the-shelf IoT devices from the microcontroller unit family MCU, namely F5529, FR5994 and CC2652. The first one does not have cryptographic accelerator support, the second one comes with AES built-in, and the third one supports AES and SHA256. Aiming at a more comprehensive evaluation, we provided those three devices with software-based implementations of various prominent hash functions.

We implemented seven software-based memory-erasure protocols and evaluated them in terms of performance, security and erasure guarantees. In particular, we measured the overall time of the protocol execution in relation to the speed of the communication channel, the device's computational characteristics and the choice of the hash function imple-

mentation. We placed the obtained performance values alongside other relevant protocol features, such as their security and erasure guarantees, with the goal of determining the protocols offering optimal trade-offs. Specifically, given a desired security level, we identify which protocols strike the best balance between performance and erasure assurances.

Structure of the article. The next section (Section 2) describes existing software-based memory-erasure protocols, highlights their most relevant features, and provides a comparison and analysis based on those features. The remainder of this article is dedicated to extending that analysis with empirical data on the execution time of memory-erasure protocols, with the goal of determining their feasibility and providing a more detailed comparison. Section 3 provides our experimental setup, Section 4 reports on the results obtained after running the experiments, and Section 5 discusses the results. Concluding remarks are given in Section 6.

2 SOFTWARE-BASED MEMORY ERASURE: A SURVEY

The earliest software-based protocol for secure memory erasure was introduced by Perito and Tsudik in 2010 (Perito and Tsudik, 2010). At the start of the protocol, the verifier sends a fresh random value of the same size as the prover's memory. The prover computes the HMAC of this value, using as key the last bits received. The security proof of this protocol relies on the fact that, to compute the HMAC on a random value, the prover needs to first receive the key. The proof of security is informal, though, and does not specify which assumptions the HMAC function must fulfil to make the protocol secure. This protocol, in addition, has the drawback of sending over the network a message as large as the prover's memory. An improvement upon Perito and Tsudik's protocol in terms of efficiency was later proposed by Karame and Li (Karame and Li, 2015).

To reduce the size of the random value sent over the network by the verifier, Dziembowski, Kazana and Wichs (Dziembowski et al., 2011) introduced a software-based memory-erasure protocol that, given a nonce of standard size, asks the prover to store in memory a random labelling function. Ideally, this step requires the use of the entire memory of the prover. The verifier then challenges the prover to quickly give the output of the labelling function on a number of inputs. Dziembowski, Kazana and Wichs provide an elegant security proof based on a reduction to a combinatorial game known as *pebbling*, placing the first stone for the study of memory-erasure protocols whose security depends on the structure of a graph-based labelling function. The drawback of their protocol is that the computational complexity of building the labelling function is quadratic in terms of the memory size of the prover, which might quickly become inefficient as the prover's memory increases. This computational complexity was reduced by Karvelas and Kiayias (Karvelas and Kiayias, 2014), using a labelling function whose underlying graph has quasilinear size in terms of the memory size of the prover. A major limitation of this design is that it can only provably erase $\frac{1}{32}$ of the prover's memory. This means that Karvelas and Kiayias's protocol offers a lower erasure guarantee than the protocols we have reviewed so far. The work in (Karvelas and Kiayias, 2014) also introduces an erasure protocol that is not based on graphs, but on hard-to-invert hash functions. The security of this protocol is proven assuming that the adversary cannot query a hash function during the memory-challenge phase. We are not aware how this could be enforced in practice.

A key security assumption made by all software-based memory-erasure protocols up to 2019 is that prover and verifier should run the protocol in isolation, i.e. without interference from an external attacker. This assumption is cumbersome to enforce successfully for wireless communication, limiting the use of software-based memory erasure to rather specific use-cases. Trujillo-Rasua questioned in (Trujillo-Rasua, 2019) whether such assumption is necessary, offering a symbolic security proof of a memory-erasure protocol that uses distance bounding to thwart collusion between the prover and an external conspirator. This work, however, left open the question of how to bound the probability of success of an adversary (i.e. malware colluding with an external conspirator) passing the memory-erasure test without removing the malware.

In 2024, Bursuc et al. (Bursuc et al., 2024a; Bursuc et al., 2024b) introduced the first memory-erasure protocols with provable security bounds that do not depend on the isolation assumption. Two of the protocols follow the methodology established in (Dziembowski et al., 2011; Karvelas and Kiayias, 2014), consisting of the construction of a graph-based labelling function and an analysis of the protocol's security via a reduction to a graph-pebbling game. The third protocol is an extension of Perito and Tsudik's protocol with a distance-bounding technique. The goal is to guarantee that the prover does not receive external help during the execution of the protocol. This protocol is proven secure unconditionally,

while those based on graph-based labelling functions are proven secure within the random oracle model. Like in (Trujillo-Rasua, 2019), the three protocols introduced in (Bursuc et al., 2024a; Bursuc et al., 2024b) use a distance-bounding mechanism consisting of several round-trip-time measurements, each requiring a round of communication between prover and verifier. The number of round-trip-time measurements thus become a security parameter in these protocols, which we denote r in Table 1.

The SPEED protocol (Ammar et al., 2018) performs memory erasure using a Trusted Software Module (TSM) as a lightweight alternative to hardware security modules. In practice, it might add a significant overhead to any program running on the platform, as the software-based memory protection continuously monitors the platform to ensure memory isolation at all times. Since memory erasure can start from any state of the device, it is more desirable to avoid imposing constraints on programs running on it. SPEED uses distance-bounding to ensure that only a nearby verifier can start the memory-erasure procedure. This does not make the protocol secure in the presence of external attackers, unless the security of the TSM guarantees that the prover device can only communicate with the verifier during the run of SPEED.

We summarize our study of the literature on software-based memory-erasure protocols in Table 1. The table shows to what extent each of the protocols we have discussed satisfies the following relevant features, as claimed in the literature.

- **Proof:** There exists a formal proof of security, which gives security guarantees with mathematical rigour.
- **Prob.:** There exists a formula to bound the probability of success of the attacker given the device characteristics and protocol parameters. Unlike asymptotic analysis, this feature allows the analyst to measure the actual security of the protocol on a given device.
- **No-Isolation:** The protocol does not assume that the device is isolated during the execution of the protocol, meaning that the protocol resists network attacks to some extent.
- **Erasure:** The proportion of the device's memory that can be erased. Together with security guarantees, this is the most important feature of a memory-erasure protocol.
- **Time:** The time complexity of running the protocol.
- **Comm.:** The communication complexity of running the protocol.

From Table 1 one can already draw a preliminary and theoretical comparison of software-based memory-erasure protocols. The series of protocols $\mathbf{PoSE}_{\mathbf{light}}$, $\mathbf{PoSE}_{\mathbf{graph}}$ and $\mathbf{PoSE}_{\mathbf{random}}$ are the only ones that come with a formal security proof, a bound on the probability of success of the attacker, and a mechanism to operate without relying on the isolation assumption. Their erasure guarantee, however, is not asymptotically close to 1 for realistic values of r. Further, it is unclear how the parameter r affects their computational and communicational complexity in practice. The other two protocols that resist network attacks (no-isolation) are **TR** and **SPEED**. The former provides no erasure guarantees, though, while the latter comes with no security proof and has the drawback of relying on a device-specific Trusted Software Platform that negatively impacts the performance of all programs running on the device. The **DFKP** protocol does ensure full memory erasure while, at the same time, it comes with a formal security proof. However, it offers the worst computational complexity amongst all protocols, hinting that it might be a viable option only for devices with low memory size. This computational complexity is improved upon by **KK**, but at the cost of erasing only a small fraction of the device's memory. Lastly, **KL** and **PT** offer optimal computational complexity, but offer neither a formal proof of security nor a bound on the adversary's success probability.

To enable a more detailed and accurate comparison of the protocols above, it is necessary to obtain empirical data on their performance (last two columns of Table 1). That is the goal of the remainder of this article.

3 EXPERIMENTAL SETUP

In this section we describe the procedure we have followed to implement and test the protocols of interest, providing all the necessary details for the reproducibility of the experiments.

3.1 IoT Proving Devices, Verifier Device and Communication Channel

For our experiments we considered 3 IoT microcontrollers produced by Texas Instruments with different characteristics: FR5994, F5529, CC2652. These devices acted as provers in the memory-erasure protocol. All prover implementations were done in Portable C, making them usable on any platform or architecture. To create the binaries, we configured the compiler to optimize for code size. A personal Dell

Table 1: Characteristics of the implemented memory-erasure protocols. The asymptotic bounds for the execution time and communication complexity are given in terms of the memory size (denoted n) and the number of round-trip measurements (denoted r). The erasure guarantees of the three protocols in (Bursuc et al., 2024a; Bursuc et al., 2024b) depend on r, n and a bound p on the desired probability of success of the attacker. As an example, given $r = 71$, $n = 8$KB and $p = 10^{-3}$, $f(r,n,p) = 0.9$.

	Proof	Prob.	No-Isolation	Erasure	Time	Comm.
DFKP (Dziembowski et al., 2011)	✓	✗	✗	1	$\mathcal{O}n^2$	$\mathcal{O}1$
KL (Karame and Li, 2015)	✗	✗	✗	1	$\mathcal{O}n$	$\mathcal{O}n$
KK (Karvelas and Kiayias, 2014)	✓	✗	✗	$\frac{1}{32}$	$\tilde{\mathcal{O}}(n)$	$\mathcal{O}1$
PT (Perito and Tsudik, 2010)	✗	✗	✗	1	$\mathcal{O}n$	$\mathcal{O}n$
PoSE$_{\text{graph}}$ (Bursuc et al., 2024a)	✓	✓	✓	$f(r,n,p)$	$\tilde{\mathcal{O}}(n+r)$	$\mathcal{O}r$
PoSE$_{\text{light}}$ (Bursuc et al., 2024b)	✓	✓	✓	$f(r,n,p)$	$\mathcal{O}n+r$	$\mathcal{O}r$
PoSE$_{\text{random}}$ (Bursuc et al., 2024a)	✓	✓	✓	$f(r,n,p)$	$\mathcal{O}n+r$	$\mathcal{O}n+r$
TR (Trujillo-Rasua, 2019)	✓	✗	✓	✗	$\mathcal{O}n+r$	$\mathcal{O}r$
SPEED (Ammar et al., 2018)	✗	✗	✓	1	$\mathcal{O}n$	$\mathcal{O}1$

laptop acted as verifier, running Python 3 implementations of each protocol. The Bluetooth protocol was used as communication channel between the verifier and the prover.

Table 2 depicts the characteristics of each device, namely memory (code size [1] + data size [2]), maximum clock speed, on-device crypto accelerators, Bluetooth version, architecture, microcontroller unit family (MCU), and IoT class (Bormann et al., 2014). Notice that some devices have hardware accelerators. For these, we also experimented using a hardware-accelerated version of the hash function.

3.2 Hash Function Implementations

Most software-based memory-erasure protocols invoke multiple hash function calls, while at the same time being independent of the particular hash function that was implemented. Hence, given their prominent role, we provide different hash function implementations to compare their performance and memory requirements.

We selected these functions according to popularity, amenability to software-based implementations, and applicability to the IoT domain. All of them calculate a digest of 256 bits. The following hash functions were selected:

- ascon (Dobraunig et al., 2021): a sponge-based hash function selected in the Lightweight Cryptography Standardization Process by NIST[3].
- blake2 (Aumasson et al., 2013): a widely deployed and highly efficient hash function. It was designed to be especially performant in software implementations.
- blake3[4]: a recent improvement on the blake2 hash function which is claimed to be much faster while offering similar security guarantees. This is the fastest (in software implementations) cryptographic hash function we could find.
- sha256 (Eastlake and Hansen, 2006): a well known and widely used hash function based on the Merkle-Damgård construction proposed by NIST. Up to this day, it is still considered secure, although it is prone to length extension attacks (Tsudik, 1992).
- aeshash[5]: an unpublished hash function whose core utilizes AES instructions. It was selected mainly to check how much speed-up could be achieved in a device with only an AES accelerator (FR5994). As far as we are aware, this hash function has not been thoroughly analysed, so it does not offer (yet) strong security guarantees, and it is only used here for the purpose mentioned before.

Table 3 shows the memory footprint (in bytes) of the hash implementations on each device. The lower the memory footprint the better, as these bytes cannot be erased during the execution of the protocol. One surprising result is that the memory footprint of the sha256hw function, which uses the hardware accelerator, actually occupies more space than the software-based implementation sha256. We conjecture this must be the result of the extra code necessary to access the hardware module. In the case of the hash implementation used by **PT**, which originally uses an

[1]Code size refers to the amount of memory occupied by the executable code running on an IoT device. This includes, for example, applications and libraries.

[2]Data size refers to the memory used by the data used during execution. This includes, for example, variables and buffers used by the running code.

[3]https://csrc.nist.gov/News/2023/lightweight-cryptography-nist-selects-ascon

[4]https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf

[5]https://csrc.nist.rip/groups/ST/toolkit/BCM/documents/proposedmodes/aes-hash/aeshash.pdf

Table 2: Microcontroller characteristics.

Device	Memory	Clock	Crypto	BT	Arch	MCU	IoT
F5529	(128 + 10) KB	25MHz	—	2	RISC-16	MSP430	Class 1
FR5994	(256 + 8) KB	16MHz	AES	2	RISC-16	MSP430	Class 1/2
CC2652	(352 + 88) KB	48MHz	AES,SHA2	5.2	RISC-32	Arm Cortex-M	Class 2

HMAC instantiated with sha256, the memory occupied by the hash function is not shown in the table, because the implementation used was tightly coupled with the HMAC routine, making it incomparable to the others.

The memory footprint of each hash function varies according to the device, due to the use of C implementations optimized for each architecture. For example, the blake2 implementation used for the CC2652 is optimized for the 32 bit architecture, which is not suitable for the others. For some hash functions this leads to lower sizes (blake2, blake3 and sha256) in the CC2652 with a 32bit architecture, or lower sizes (the rest) in the F5529 and FR5994 with a 16bit architecture. Taking into account the three devices, sha256 is the hash function with the lowest memory footprint, followed by ascon and aeshash.

For reproducibility purposes, we remark that, to measure the memory footprint of the hash function implementations on each device, we used as baseline a trivial hash function with minimal memory footprint. This allows us to measure the memory footprint of a hash function on a device by comparing it against the implementation of the trivial hash function on the same device. We measure memory in this way to ensure (i) that the compiler code optimization routine does not remove the hash function and (ii) that all hash functions implementations share the rest of the program code.

3.3 Protocol Selection

Out of the 9 protocols described in Section 2, we selected 7 protocols for their implementation and evaluation. SPEED (Ammar et al., 2018) was discarded because it relies on a memory-isolation technique that is neither open-source nor fully specified in the original article. We also discarded the protocol in (Trujillo-Rasua, 2019) because the protocol specification is given symbolically, abstracting away from important implementation details, such as the number of round-trip-time measurements, the size of the nonces, etc. Even though the author in (Trujillo-Rasua, 2019) offers an instantiation of the symbolic protocol specification, it leaves as future work the analysis of its security and erasure guarantees.

We note that the protocols $\mathbf{PoSE}_{\mathbf{light}}$, $\mathbf{PoSE}_{\mathbf{graph}}$ and $\mathbf{PoSE}_{\mathbf{random}}$ (Bursuc et al., 2024a; Bursuc et al., 2024b), depend on an additional security parameter r that establishes the number of round-trip-time measurements performed by the protocol. For our empirical setting, we set $r = 71$, which gives an erasure guarantee of 90% of the device's memory with probability $(1-10^{-3})$. If one wishes to erase $1/32$ of the memory only, like in the **KK** protocol, then it would be sufficient to set $r = 3$, which is efficient. If one wishes to erase 99% of the device's memory, like in **DFKP**, **KL** and **PT**, then the protocols in (Bursuc et al., 2024a; Bursuc et al., 2024b) would need to execute hundreds of round-trip-time measurements, which is inefficient. We thus believe that $r = 71$ strikes a good balance between erasure guarantees and performance. This means that, for the remainder of this article, when we refer to $\mathbf{PoSE}_{\mathbf{graph}}$, $\mathbf{PoSE}_{\mathbf{light}}$ and $\mathbf{PoSE}_{\mathbf{random}}$, we are assuming that they all execute 71 round-trip-time measurements.

In Table 4, we show the memory footprint of our protocol implementations for each combination of protocol and device. To calculate the memory footprint of a protocol on a device, we considered a dummy implementation of each protocol, that, while sending messages of the same size and in the same order, occupies a negligible amount of memory. The rest of the code remained exactly the same. Comparing this dummy implementation with the original one allowed us to compute the actual size of each protocol implementation taking into account compiler optimizations.

Notice that the values in Table 4 vary between devices because their architectures are different. Another reason for the size variation is that we used the best available implementation for each hash function and architecture. In general, the CC2652 implementations take more space than the F5529 or FR5994 ones. This is probably due to the use of byte size variables throughout the implementations, which can be more efficiently represented in the 16 bits architecture than in the 32 bits architecture. Taking into account the three devices, the protocols with smaller memory footprint are **PT** and $\mathbf{PoSE}_{\mathbf{random}}$.

3.4 Memory to Be Erased

To be able to run all protocols in a reasonable time and avoid the need for device-specific engineering, while erasing the same amount of memory with each

Table 3: Memory footprint of the hash function in device CC2652 while attempting to erase 2KB.

	aeshash	ascon	blake2	blake3	sha256	sha256hw
DFKP	1250	5860	6365	21177	1043	1397
KL	1245	5886	6364	21216	1038	1402
KK	1253	5866	6364	21158	1043	1391
PT	—	—	—	—	3730	—
PoSE$_{\text{graph}}$	1251	5868	6367	21182	1044	1401
PoSE$_{\text{light}}$	1251	5867	6366	21195	1044	1402
PoSE$_{\text{random}}$	—	—	—	—	—	—

Table 4: Memory footprint (in bytes) of the protocol implementations on each device.

	DFKP	**KL**	**KK**	**PT**	**PoSE$_{\text{graph}}$**	**PoSE$_{\text{light}}$**	**PoSE$_{\text{random}}$**
CC2652	620	1023	1343	274	1182	1181	312
F5529	777	1343	1868	356	1610	1644	415
FR5994	781	1353	1876	357	1610	1644	416

protocol, we set up our experiments in such a way that a fixed portion of the memory of each device is erased. Our implementation creates an array with a fixed size at compile time, and this array is exactly what is erased while running the protocol. This makes our implementation simple and device-independent. Because the array must fit in the data size of each device, we restrict ourselves to the following memory sizes. For the devices F5529 and FR5994 we erase exactly 2KB, and for CC2652 we erase exactly 2 KB, 4 KB and 8 KB. Increasing the maximum erased size in each device beyond the limits mentioned before, led to the risk of overwriting memory addresses used during execution, therefore making it unfeasible.

We notice that our implementations are limited to performance testing, and will need to be adapted for deployment in a real setting. The reason being that, in practice, memory erasure protocols define the segment of memory to be erased prior compilation, rather than letting the compiler decides. Doing so, however, requires device-dependent implementations, which we considered would add unnecessary complexity to the comparison task.

3.5 Distance

The distance between the device (prover) and the laptop (verifier) was approximately 1 meter. Each run of the protocol was done independently, as the main objective was to compare the protocols against each other in the simplest possible setting.

4 EXPERIMENTAL RESULTS

This section reports on the time each combination of protocol and hash function implementation takes to (i) erase a fixed amount of memory on a device and (ii) to generate a proof of erasure.

4.1 Impact of the Hash Function Implementations

Most protocols under analysis, namely **DFKP**, **KL**, **KK**, **PoSE$_{\text{graph}}$** and **PoSE$_{\text{light}}$**, resort to several hash function calls to fill the device's memory with fresh data. This contrasts with the approach followed by **PT** and **PoSE$_{\text{random}}$** where the device's memory is filled with random data sent by the verifier. Hence, we start measuring the impact of the hash function implementation on the execution time of the former class of protocols. We do so by measuring the execution time of the protocols right until the point where the device's memory has been erased, thereby ignoring the verification of the proof of erasure. We will refer to this (partial) execution time by *erasure time*, to distinguish it from the *verification time* where the verifier checks the erasure proof, and from the *total execution time* which accounts for both: the erasure and verification time.

Tables 5 to 7 display erasure time values for every combination of protocol and hash function, each table focusing on a given device and memory size to be erased. Notice that the protocols **PoSE$_{\text{random}}$** and **PT** do not appear in the tables, as they do not call the hash function to fill the device's memory.

We observe that it is notably faster to erase memory when the device has a higher clock frequency, as can be seen when comparing performance in the CC2652 with respect to the FF5529 and the FR5994 devices. Amongst the hash functions, sha265hw was the fastest in the CC2652 device, which was expected as it uses the hardware accelerator. One surprising result is that ascon, which is meant to be used in

Table 5: Erasure time in seconds on device CC2652 while attempting to erase 8KB.

	aeshash	ascon	blake2	blake3	sha256	sha256hw
DFKP	10.9	31.5	5.1	3.8	7.3	1.7
KL	0.0	0.0	0.0	0.0	0.0	0.0
KK	7.4	20.9	3.6	2.7	5.1	1.2
PoSE$_{\text{graph}}$	14.8	60.0	5.9	4.4	8.7	2.1
PoSE$_{\text{light}}$	4.4	20.0	2.0	1.3	2.6	0.6

Table 6: Erasure time in seconds on device FR5994 while attempting to erase 2KB.

	aeshash	ascon	blake2	blake3	sha256
DFKP	3.1	80.5	20.1	15.6	27.8
KL	0.1	0.3	0.1	0.1	0.2
KK	4.9	130.8	34.5	26.9	48.1
PoSE$_{\text{graph}}$	8.6	218.7	56.7	44.0	78.5
PoSE$_{\text{light}}$	4.6	108.7	28.7	22.3	39.5

Table 7: Erasure time in seconds on device F5529 while attempting to erase 2KB.

	ascon	blake2	blake3	sha256
DFKP	47.5	12.6	9.9	17.4
KL	0.7	0.6	0.6	0.6
KK	76.3	21.2	16.5	30.1
PoSE$_{\text{graph}}$	128.9	34.0	26.3	48.2
PoSE$_{\text{light}}$	64.1	17.5	13.6	24.6

lightweight cryptography, had a consistently worst performance. In every device blake3 was faster than blake2, although the difference is around 25 percent only. Aeshash was much faster than other hash functions in the FR5994 device, as expected by the usage of the AES hardware module present on this device. On the contrary, it performed worse than blake2, blake3 and even sha256 in the CC2652 device. Therefore, we deduce that if the device is fast enough, pure software implementations might be more performant than hybrid implementations such as the one used in aeshash.

To conclude, the impact of the hash function implementation is high in most of the protocols analysed, with the exceptions of **PT** and **PoSE$_{\text{random}}$**. Interestingly, the smaller erasure time on each device is achieved with a different hash function. This suggests that, before deploying a memory-erasure protocol on a given device, it is worth testing it with different hash functions implementations. Lastly, cross-checking Table 3 (memory cost) with Tables 5 to 7 (computational cost), we conclude that (i) aeshash is the fastest and smaller hash function implementation on device FR5994, (ii) sha256 and sha256hw outperform the other hash functions on the device CC2652, and (iii) on the device F5529 there is a clear looser, namely ascon, but no clear winner.

4.2 Total Execution Time

Next, we measure the total execution time of each combination of protocol and hash function implementation. This is admittedly the most important performance variable in our experiments. The values can be found in Tables 8 to 10, each table focusing on a given device and memory size to be erased.

In the CC2652 device, the fastest protocols were **KK** and **DFKP**, despite the latter being the one with worst (asymptotic) time complexity. We believe this to be possible because (i) the constants hidden in the asymptotic notation are very small for this protocol and somewhat larger for the others, and (ii) the size of the erased memory is small. Looking at the protocols **PT** and **PoSE$_{\text{random}}$**, which follow the approach of sending a large random nonce over the network, we observe that they perform poorly in comparison to the others. Interestingly, that is not case if we shift our attention to the devices FR5994 and F5529. In those devices, **PT** and **PoSE$_{\text{random}}$** are amongst the fastest protocols. This difference in result can be explained by the version of the Bluetooth protocol used. The CC2652 device has a Bluetooth module included, able to run version 5.2, which means that in each challenge-response round the message needs to go through the whole Bluetooth stack. On the other hand, for the FR5994 and F5529 devices, a simple HC-05 module was used, which makes the Bluetooth protocol overhead much smaller, as it does not support as many features.

A summary of the results just described is given in Figure 1. In that figure, we display the total execution time of each protocol on each device by choosing the hash function implementation that gives the fastest execution time. From the figure we derive that the choice of the most performant (fastest) protocol for a device depends on the specific conditions in which it will operate. In particular, clock speed, network cost and memory size determine which protocol is more suitable. Cross-checking Table 4 (memory cost) with Figure 1 (computational cost), we conclude that **PT** offers an optimal trade-off between memory footprint and execution time on the devices F5529 and FR5994; other good alternatives are **DFKP** and **PoSE$_{\text{random}}$**. In the CC2652 the winner is **DFKP**.

Table 8: Total execution time in seconds on device CC2652 while attempting to erase 8KB.

	aeshash	ascon	blake2	blake3	none	sha256	sha256hw
DFKP	11.1	31.7	5.3	4.0	—	7.5	1.9
KL	23.4	23.4	23.3	23.3	—	23.4	23.3
KK	7.6	21.1	3.8	2.9	—	5.3	1.3
PT	—	—	—	—	—	23.2	—
PoSE$_{\text{graph}}$	21.4	66.6	12.5	11.0	—	15.3	8.7
PoSE$_{\text{light}}$	11.0	26.6	8.6	8.0	—	9.2	7.2
PoSE$_{\text{random}}$	—	—	—	—	29.7	—	—

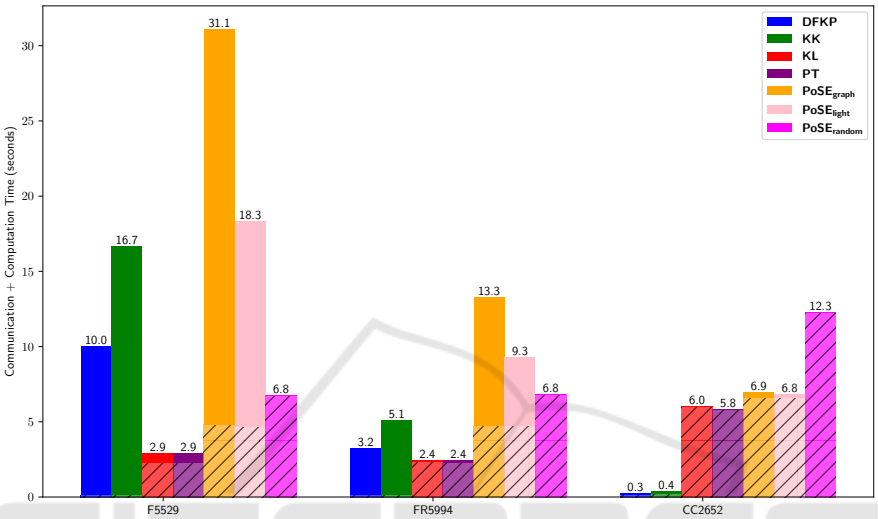

Figure 1: Total execution time in seconds, partitioned by communication time (striped) and computation time, using the best hash function for each device.

Table 9: Total execution time in seconds on device FR5994 while attempting to erase 2KB. For **PoSE$_{\text{random}}$** it was 6.8 seconds.

	aeshash	ascon	blake2	blake3	sha256
DFKP	3.2	80.7	20.2	15.7	27.9
KL	2.4	2.6	2.5	2.4	- 2.5
KK	5.1	131.0	34.6	27.1	48.2
PT	—	—	—	—	2.4
PoSE$_{\text{graph}}$	13.3	223.4	61.4	48.7	83.4
PoSE$_{\text{light}}$	9.3	113.3	33.5	27.2	44.4

Table 10: Total execution time in seconds on device F5529 while attempting to erase 2KB. For **PoSE$_{\text{random}}$** it was 6.8 seconds.

	ascon	blake2	blake3	sha256
DFKP	47.6	12.7	10.0	17.5
KL	3.0	2.9	2.9	2.9
KK	76.4	21.3	16.7	30.2
PT	—	—	—	2.9
PoSE$_{\text{graph}}$	133.5	38.7	31.1	52.9
PoSE$_{\text{light}}$	68.9	22.1	18.3	29.3

4.3 How Much Does the Size of Memory Impact the Execution Time?

A rather surprising result from the empirical data we have presented so far is that the **DFKP** protocol seems to perform very well across all devices, despite having the worst asymptotic computational complexity. As the memory to be erased increases, one should expect a worse performance of **DFKP** in comparison to the other protocols. That is precisely what we test next, i.e. how the execution time changes as we increase the memory size. We perform this experiment on the CC2652 device, which allows us to erase up to 8KB of memory. Figure 2 displays the results.

Observe that the execution time of most protocols seem to increase linearly with the memory size. Exceptions are **PoSE$_{\text{graph}}$** and **PoSE$_{\text{light}}$**, whose execution time is dominated by the communication time rather than by the erasure time. The performance of **DFKP** on the erasure of 8KB of memory starts getting worse than that of **KK**, supporting the hypothesis that its quadratic computational complexity might make it unsuitable for larger memory sizes. We acknowledge, that further experiments are needed to determine the memory threshold where **DFKP** starts behaving worse than the other protocols.

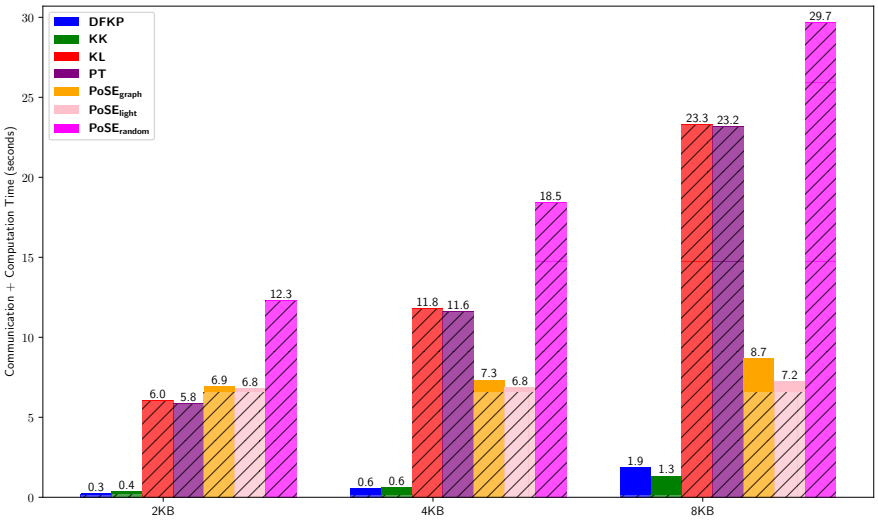

Figure 2: Total execution time in seconds on device CC2652, partitioned by communication time (striped) and computation time, using the best hash function.

5 COMPARATIVE ANALYSIS

Our results have not revealed a clear winner with respect to performance. That said, **PT**, **KL**, **KK** and **DFKP** seem to perform relatively well across all devices. **KK**, however, provides little erasure guarantees (only $1/32$ of the memory is guaranteed to be erased), while **PT** and **KL** come with no formal security proof. Hence, our next step is to provide a holistic comparison of the protocols, one where security and erasure guarantees are considered alongside performance.

To reach our conclusions, we considered the following facts:

- Protocols **KL**, **KK** and **PT** offer a low level of security
- Protocols **PoSE$_{random}$**, **PoSE$_{graph}$**, **PoSE$_{light}$** and **DFKP** offer a high level of security. Note, though, that the **DFKP** protocol is less secure than the others given that it is not resistant against distant attackers.
- Protocols **PoSE$_{random}$**, **PT** or **KL** require sending the full memory of the device through the network.
- **DFKP** has quadratic complexity, hence its performance is worse when the amount of memory is large.

Next, we provide a fine-grained analysis of the results, by projecting them onto specific use-cases. These results are summarized in Table 11. They were obtained by extrapolating the behaviour of the protocols in each device.

- When the network cost is high, communication needs to be minimized, therefore protocols such as **PoSE$_{random}$**, **PT** or **KL** are impractical.
  - When memory is small, protocol **DFKP** is the clear winner (see for example Table 10). Even though it has quadratic complexity, its very low constant factor makes it faster than the rest of the protocols. We are considering that this protocol has high security even though it assumes the isolation assumption. If security against distant attackers is necessary, the winner for this category is the **PoSE$_{light}$** protocol.
  - When memory is large, the most performant protocols are **PoSE$_{light}$** and **KK**, for high security and low security, respectively.
- When the network cost is low, the most performant protocols are **PoSE$_{random}$** and **PT**, for high security and low security, respectively. A surprising result in this setting is that **KL** had a very similar performance to **PT**, which should not have been the case as it was specifically designed to improve its performance. If the device in question has hardware accelerators, then for the high security case it is possible that **DFKP** is faster, as shown for example in Table 8.

It is noteworthy that the clock speed only changed the selection of the winner when the network cost is low and memory is small. In this scenario, the **DFKP** protocol outperforms **PoSE$_{random}$** and **KK** outperforms **PT** (see for example Table 8).

We end our analysis by completing the comparison table given in Section 2 with the empirical data

Table 11: Summary of results. For each combination of Network cost (high, low), Clock speed (high, low), Memory size (large, small) and Security (high, low) the most performant protocol is shown.

				Network cost			
				High		Low	
				Clock speed		Clock speed	
				High	Low	High	Low
Memory size	Large	Security	High	**PoSE$_{\text{light}}$**	**PoSE$_{\text{light}}$**	**PoSE$_{\text{random}}$**	**PoSE$_{\text{random}}$**
			Low	**KK**	**KK**	**PT**	**PT**
	Small	Security	High	**DFKP**	**DFKP**	**DFKP**	**PoSE$_{\text{random}}$**
			Low	**DFKP**	**DFKP**	**KK**	**PT**

Table 12: Summary of results, contrasting performance with security. The last three columns show the total time on each device, when using the highest possible memory to erase (in parentheses) and the hash function leading to the lowest execution time.

	Proof	**Prob.**	**No-Isol.**	**Erasure**		**Total Time**					
						F55292KB		**FR59942KB**		**CC26528KB**	
DFKP (Dziembowski et al., 2011)	✓	✗	✗	1	✓	10.0	✚	3.2	✓	1.9	✓
KL (Karame and Li, 2015)	✗	✗	✗	1	✓	2.9	✓	2.4	✓	23.3	✗
KK (Karvelas and Kiayias, 2014)	✓	✗	✗	$\frac{1}{32}$	✗	16.7	✚	5.1	✓	1.3	✓
PT (Perito and Tsudik, 2010)	✗	✗	✗	1	✓	2.9	✓	2.4	✓	23.2	✗
PoSE$_{\text{graph}}$ (Bursuc et al., 2024a)	✓	✓	✓	0.9	✚	31.1	✗	13.3	✗	8.7	✚
PoSE$_{\text{light}}$ (Bursuc et al., 2024b)	✓	✓	✓	0.9	✚	18.3	✚	9.3	✚	7.2	✚
PoSE$_{\text{random}}$ (Bursuc et al., 2024a)	✓	✓	✓	0.9	✚	6.8	✓	6.8	✚	29.7	✗

obtained during our experiments. This provides another angle to compare the results obtained across devices, this time contrasting performance with the same protocol features shown before in Table 1. The results are displayed in Table 12. For a simpler visual comparison amongst the protocols, we labelled each cell in the table with a symbol that indicates whether the protocol performs well (✓), average (✚) or poorly (✗), relative to the other protocols. For the numerical values in the table, we determined their labels by clustering the values in a column in three clusters. Specifically, we used k-means for clustering. For example, in the last column of the table, which gives the execution time of each protocol on the CC2652 device when erasing 8KB, there are three clusters that minimize the sum of the squared Euclidean distances of each point to its closest centroid, namely ✓ $= \{1.9, 1.3\}$, ✚ $= \{8.7, 7.2\}$ and ✗ $= \{23.3, 23.2, 29.7\}$. The table reinforces the analysis results we have mentioned earlier. An interesting observation is that **PoSE$_{\text{light}}$** is the only protocol that does not perform poorly on any of the features considered.

6 CONCLUSIONS

In this paper we presented the outcome of our experiments with various memory-erasure protocols in an IoT setting. We implemented[6] 7 protocols, each with several variants depending on the hash function used, and tested them on 3 modern IoT devices. Furthermore, we compared the security guarantees provided by each protocol, and contrasted them with their performance in a practical setting.

Our results revealed that current memory-erasure protocols are practical, although erasing the full memory securely could take several minutes for the slower devices. Network speed might be faster than local computation, therefore aiming at minimal communication complexity is not always the best choice. For protocols that use hash functions, the choice of the hash function may dramatically influence the protocol performance and memory footprint. Finally, the most performant protocol might not be the best according to the asymptotic complexity analysis, as for small memory sizes the hidden constants may play a determining role.

[6]https://gitlab.com/uniluxembourg/fstm/dcs/satoss/memory-erasure-experiments

ACKNOWLEDGEMENTS

Reynaldo Gil-Pons was funded by the Luxembourg National Research Fund, Luxembourg, under the grant AFR-PhD-14565947. Rolando Trujillo-Rasua was supported by a Ramón y Cajal grant (RYC2020-028954-I) from the Spanish Ministry of Science and Innovation and the EU, as well as by projects PROVTOPIA (PID2023-150098OB-I00), funded by MICIU/AEI/10.13039/501100011033 and FEDER (EU), and HERMES, funded by INCIBE and the EU's NextGenerationEU/PRTR.

REFERENCES

Aman, M. N., Basheer, M. H., Dash, S., Wong, J. W., Xu, J., Lim, H. W., and Sikdar, B. (2020). HAtt: Hybrid Remote Attestation for the Internet of Things With High Availability. *IEEE Internet of Things Journal*.

Ammar, M., Daniels, W., Crispo, B., and Hughes, D. (2018). Speed: Secure provable erasure for class-1 iot devices. In *Eighth ACM Conference on Data and Application Security and Privacy*.

Aumasson, J.-P., Neves, S., Wilcox-O'Hearn, Z., and Winnerlein, C. (2013). BLAKE2: Simpler, Smaller, Fast as MD5. In *Applied Cryptography and Network Security*.

Banks, A., Kisiel, M., and Korsholm, P. (2021). Remote Attestation: A Literature Review. *ArXiv*.

Bormann, C., Ersue, M., and Keranen, A. (2014). RFC 7228: Terminology for constrained-node networks. *IETF RFC*.

Bursuc, S., Gil-Pons, R., Mauw, S., and Trujillo-Rasua, R. (2024a). Software-based memory erasure with relaxed isolation requirements. In *Proc. 37th IEEE Computer Security Foundations Symposium (CSF'24)*.

Bursuc, S., Gil-Pons, R., Mauw, S., and Trujillo-Rasua, R. (2024b). Software-Based Memory Erasure with relaxed isolation requirements: Extended Version. *arXiv preprint arXiv:2401.06626*.

Dobraunig, C., Eichlseder, M., Mendel, F., and Schläffer, M. (2021). Ascon v1.2: Lightweight Authenticated Encryption and Hashing. *Journal of Cryptology*.

Dziembowski, S., Kazana, T., and Wichs, D. (2011). One-time computable self-erasing functions. In *Theory of Cryptography Conference*.

Eastlake, D. and Hansen, T. (2006). US Secure Hash Algorithms (SHA and HMAC-SHA). Technical Report RFC4634, RFC Editor.

Karame, G. O. and Li, W. (2015). Secure erasure and code update in legacy sensors. In *International Conference on Trust and Trustworthy Computing*. Springer.

Karvelas, N. P. and Kiayias, A. (2014). Efficient proofs of secure erasure. In *International Conference on Security and Cryptography for Networks*, Amalfi, Italy. Springer.

Kuang, B., Fu, A., Susilo, W., Yu, S., and Gao, Y. (2022). A survey of remote attestation in Internet of Things: Attacks, countermeasures, and prospects. *Computers & Security*.

Lachner, C. and Dustdar, S. (2019). A Performance Evaluation of Data Protection Mechanisms for Resource Constrained IoT Devices. In *2019 IEEE International Conference on Fog Computing (ICFC)*.

Perito, D. and Tsudik, G. (2010). Secure code update for embedded devices via proofs of secure erasure. In *European Symposium on Research in Computer Security*. Springer.

Rao, V. and Prema, K. V. (2019). Comparative Study of Lightweight Hashing Functions for Resource Constrained Devices of IoT. In *2019 4th CSITSS*.

Reardon, J., Basin, D., and Capkun, S. (2013). SoK: Secure data deletion. In *2013 IEEE Symposium on Security and Privacy*.

Silva, C., Cunha, V. A., Barraca, J. P., and Aguiar, R. L. (2024). Analysis of the Cryptographic Algorithms in IoT Communications. *Information Systems Frontiers*, (4).

Trujillo-Rasua, R. (2019). Secure memory erasure in the presence of man-in-the-middle attackers. *Journal of Information Security and Applications*.

Tsudik, G. (1992). Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review*.