Enhancing Privacy and Security in IoT Environments through Secure Multiparty Computation

RIK VAN DE HATERD, University of Twente, The Netherlands
SUPERVISOR: DR. ING. M.ELHAJJ (MOHAMMED), University of Twente, The Netherlands

Abstract - With the increasing influence of IoT devices in our daily lives, secure data-sharing is becoming ever more important. Sensors and other devices are communicating vast amounts of possibly unencrypted data, which poses a significant privacy concern. To tackle this problem, this research implements two Partially Homomorphic Encryption (PHE) schemes, RSA and the Paillier cryptosystem, to perform Secure Multiparty Computation (SMPC) in the resource-constrained IoT environment. The environment consists of a laptop connected to an Arduino Uno through a serial connection. The RSA-based SMPC protocol has an average completion time of 2007ms. However, due to the inability to use padding, RSA lacks semantic security. Conversely, the Paillier-based protocol is semantically secure but cannot complete the encryption due to dynamic memory issues. Even if resolved, the estimated encryption time exceeds 103.3 minutes. Despite the potential of SMPC in IoT environments for secure data handling, the results from this research suggest that directly implementing PHE schemes on Arduino is not practical based on the observed limitations.

Additional Key Words and Phrases: Secure Multiparty Computation, Internet of Things, Resource-constrained device, Partially Homomorphic Encryption

1 INTRODUCTION

The Internet of Things (IoT) has steadily grown over the last couple of decades[1]. With this growth, IoT devices are increasingly integral in daily life. These interconnected devices share sensitive data, making the confidentiality and integrity of information a substantial concern in the field of IoT security [2]. Furthermore, due to the nature of the IoT, there are additional constraints, such as limited resources, diversity of standards, and network vulnerabilities[3]. In this resource-constrained environment, challenges regarding privacy during data aggregation and transport encryption emerge [4] [5]. Traditional cryptographic algorithms encounter challenges when applied to IoT scenarios due to inherent resource limitations such as power constraints, limited battery capacity, and the need for real-time execution[6]. Thus, this research focuses on addressing the critical privacy and security challenges in resource-constrained IoT environments by applying Secure Multiparty Computation (SMPC) techniques.

1.1 Motivation

The field of SMPC has flourished with the rise of cloud computing and data-sharing in IoT environments. However, the literature on the practical application of SMPC protocols on resource-constrained devices is lacking. Prior research has mostly been focused on developing secure protocols and testing them in virtual environments, rather than on devices like an Arduino. Therefore, this research aims to contribute to the literature by gaining insights into the practical application of SMPC in resource-constrained devices within the IoT environment.

TScIT 40, Feb 2, 2024, Enschede, The Netherlands

1.2 Background

In the context of SMPC, a group of parties aims to collectively compute a function based on their private inputs whilst ensuring only the output is disclosed[7]. The problem was formally introduced as the Millionaires' Problem by Andrew Yao (1982)[8] and describes two millionaires who want to know which one of them is richer, without disclosing their actual wealth. Yao's millionaire problem is a Boolean predicate but was proven to be computationally feasible for any function in 1986, again by Yao[9]. One of the possible building blocks of SMPC is Homomorphic encryption (HE). HE allows for computations to be performed on encrypted data without the need of having to first decrypt it. The three main types of HE are:

- **Partially Homomorphic Encryption (PHE)**: Partially Homomorphic Encryption is the most computationally practical form of HE but is also the most mathematically limited. PHE schemes only support the evaluation of one gate and the two operations of additive homomorphic encryption or multiplicative homomorphic encryption.
- **Somewhat Homomorphic Encryption (SHE)**: Somewhat Homomorphic Encryption can evaluate two types of gates but only for a subset of operations.
- **Fully Homomorphic Encryption (FHE)**: Fully Homomorphic Encryption allows the evaluation of arbitrary circuits made up of multiple gate types of unbounded depth. FHE is the strongest type of HE but is also the most computationally heavy.

Middleware is often deployed for the communication between the hardware and application layer in IoT. IoT middleware generally handles the collection, storage, analysis, processing, and forwarding of results to the data consumers. Since the middleware gets full access to the raw data it becomes a high-value target for attackers [5]. Moreover, the middleware might not be under the control of the owner of the smart environment and thus could be untrustworthy. Finally, when outsourcing data to a third party, sources lose control over how their data is used. An example of this is the aggregation of data from different sensors in an IoT system. The client and a cloud could work together to produce the targeted outcome, but at the same time, private data could be leaked due to the communication of unencrypted data [7]. Another problem in the IoT environment is the resource-constrained nature of IoT devices. IoT devices cannot perform complex computations to encrypt their data that non-IoT devices can [10]. After an extensive literary review, SMPC showed great potential in providing secure data-sharing in the resource-constrained environment of IoT.

1.3 Research questions

This research will investigate the following research questions (RQ).

(1) How can a customized cryptographic protocol based on SMPC be implemented to ensure confidentiality and integrity of data shared among IoT devices, considering the specific constraints and requirements of IoT environments?
(2) What is the practicality, efficiency, and security of the implemented SMPC-based cryptographic protocol in real-world IoT use cases, and how do they compare with existing security solutions in terms of computation time, resource utilization, and power consumption?

1.4 Structure

Section 2 provides an overview of what is presented in the literature regarding the usage of SMPC to enhance privacy and security in IoT applications. Then section 3 documents the hardware and software configurations, implemented PHE schemes, and research metrics. The section concludes with an in-depth review of the testing environment and design choices. Next, section 4 details the performance- and security analysis of the implemented PHE schemes. The performance analysis delves into computation time, power consumption, and memory usage, whilst the security analysis examines the security model, cryptographic key size and semantic security. Additionally, RQ1 and RQ2 are answered at the end of the section. The paper concludes with section 5, which summarizes key findings and outlines potential avenues for future research.

2 RELATED WORK

The field of SMPC has been rapidly evolving over the years and many efficient protocols have been published. With the rise of more efficient protocols, the application of SMPC in IoT is also becoming more relevant. A study by authors in [11] offers two optimized SMPC protocols for the Internet-like setting. Their protocols are based on multiparty garbled circuits as described in the paper of Beaver, Micali and Rogaway[12]. Furthermore, they provide a protocol based on the paper from Ben-Or, Goldwasser and Widgerson[13] that incorporates the free-XOR technique as well as reducing round complexity. With these optimizations they reduced overall runtime from 355 seconds to 25 and the online time from 330 seconds to <0.5 seconds compared to the 1987 paper from Goldreich, Micali and Widgerson[14]. Another paper by authors in [15] looks at optimizing Shamir-Secret Sharing[16] (SSS) to achieve privacy-preserving data aggregation in the IoT environment. They optimized the sharing phase of SSS by lowering the degree of the polynomial. With this optimization, they managed to make their aggregation time 6 and 9 times faster as well as using 7 and 10 times less radio-on time in their testing environments (Flocklab and DCube respectively). Cloud computing has a lot of potential for SMPC in the IoT setting. Authors in [10] discussed the possibility of using the cloud to utilize Homomorphic encryption in the IoT environment. In their architecture, the IoT device encrypts their data and sends it to a cloud. Then a 'data user' can query for the data, which is then computed by the cloud and delivered. They concluded that resource-constrained IoT devices might not be able to afford the costs of Fully Homomorphic encryption. However, they did find that Partial Homomorphic encryption has great potential for resource-constrained IoT devices. The cloud can be used by IoT devices to offload their complex computations. Authors in [17] proposed a protocol based on a modified SSS[16] scheme where the source node can outsource its computation to a set of workers. The paper showed that the proposed protocol met the requirements of full anonymity, confidentiality, verification, and computation synchronization while also shifting most of the computational costs to the workers, ensuring correct results. These papers show that SMPC has great potential to be used in the IoT environment.

3 METHODOLOGY

This section details the methodology employed in the research. It begins with a description of the hardware and software setup, providing the foundation for algorithm implementation. Then, the research metrics are presented, followed by an outline of the chosen PHE schemes. The section concludes with insights into the practical implementation and testing environments used in the research.

3.1 Hardware and software setup

3.1.1 Tools. The hardware configuration is presented in Table 1, providing a detailed account of the devices involved in the experimental setup. Table 2 contains the specific software tools and applications pivotal to this research. Together, these tables serve as references for understanding the components of the research methodology. A schematic overview of the experimental setup can be found in Figure 1.

Table 1. Hardware devices

Device	**Microcontroller / CPU**	**Flash memory**	**SRAM**
Arduino Uno	ATmega328	32K bytes	2K bytes
MacBook Air	1,3 GHz Dual-Core Intel Core i5	121 GB	4 GB
AVHzY USB-Meter C3	-	-	-

3.2 Algorithms

This paper focuses on implementing two PHE encryption schemes to cover both mathematical operations. RSA for its multiplicative property and the Paillier cryptosystem for its additive property.

3.2.1 RSA. RSA [21] is a public-key cryptosystem that relies on the practical difficulty of integer factorization[22], specifically for the product of two large prime numbers. RSA encryption is described by the following equation:

$$E(m) = m^e \mod n \tag{1}$$

In this equation, m represents the plaintext message, e is the public exponent, and n is the product of two large prime numbers, ensuring the security of the encryption scheme. The multiplicative

Table 2. Software environment

IDE	Programming language	Libraries
Arduino IDE 2.2.1	C/C++	microRSA[18], arduino-cryptographic-library[19]
Pycharm CE 2021.3	Python 3.11	OpenSSL + subprocess, serial, python-paillier[20]

homomorphic property of RSA is described by the equation:

$$\begin{aligned} E(m_1) * E(m_2) &= m_1^e m_2^e \mod n \\ &= (m_1 * m_2)^e \mod n \\ &= E(m_1 * m_1) \end{aligned}$$

This property highlights the ability to perform multiplication on the ciphertexts directly without the need for decryption.

3.2.2 Paillier cryptosystem. The Paillier cryptosystem[23] is a probabilistic public-key encryption scheme based on a discrete logarithm trapdoor modulo a large integer that is hard to factor [24]. Paillier encryption is described by the following equation:

$$E(m) = g^m r^n \mod n^2 \tag{2}$$

In this equation, m represents the plaintext message, g is a generator of the multiplicative group modulo n^2, and r is a random value chosen from the set $\{0, ..., n-1\}$. The Paillier cryptosystem exhibits an additive homomorphic property, described by the equation:

$$\begin{aligned} E(m_1) * E(m_2) &= (g^{m_1} r_1^n)(g^{m_2} r_2^n) \mod n^2 \\ &= (g^{m_1+m_2})(r_1 r_2)^n \mod n^2 \\ &= E(m_1 + m_1) \end{aligned}$$

This property enables the computation of the encryption of the sum of corresponding ciphertexts directly.

3.3 Research metrics

This research aims to benchmark the implemented protocols based on two metrics: performance and security To assess the performance of the SMPC protocol, the following performance metrics will be documented:

- **Memory usage:** The memory usage consists of the amount of flash memory used by the program, as well as the amount of dynamic memory used before and during the protocol runtime, measured in bytes.
- **Power consumption:** The power consumption consists of the voltage, wattage, and amperage of the Arduino Uno during protocol runtime, measured in volts, milliwatts, and milliamperes respectively.
- **Computation time:** The computation time consists of the time it takes for the different components of the protocol to complete their runtime, measured in milliseconds.

To assess the security of the SMPC protocol, the following security metrics will be analyzed:

- **Security model:** The security describes the different models that detail adversarial behaviour during the protocol runtime.
- **Cryptographic key size:** Cryptographic key size refers to the number of bits of a key and represents the upper limit of the algorithm's encryption security.
- **Semantic security:** Semantic security denotes the ability of an attacker to guess whether the ciphertext is the result of encrypting message 1 or message 2, i.e. the adversary can gain knowledge based on the encryption alone.

3.4 Implementation

The laptop and Arduino Uno are connected by a USB cable. This connection enables communication through the Serial port, establishing a reliable and straightforward channel for data exchange. The laptop is designated to perform the computationally intensive task of key generation. The key size is set at 1024 bits, which is a tradeoff between security and memory usage. For RSA operations on the laptop, the implementation leverages the Python subprocess library to make use of the computational efficiency of OpenSSL. In contrast, the Paillier encryption on the laptop utilizes the python-paillier [20] library. On the Arduino Uno, a custom data structure 'bignum8' is introduced to handle numbers of up to 512 bytes. For Paillier encryption on the Arduino, the implementation relies on the RNG library to generate cryptographically secure random numbers when computing r (Formula 2). Using modular arithmetic to calculate g^m (Formula 2), g is set to $n + 1$. This transforms g^m into $n * m + 1$, which significantly reduces computational complexity. Additionally, modular exponentiation (ME), outlined in the Appendix (Algorithm 1), is employed to efficiently calculate r^n (Forumla 2). These implementation choices aim to deliver a robust and secure cryptographic system, optimizing performance and ensuring the integrity of communication between the laptop and Arduino Uno.

3.4.1 RSA environments. For RSA encryption two different environments interact with each other, the Python- and Arduino environments. The Python environment generates a 1024-bit RSA key pair in PEM format by utilizing OpenSSL. Then the modulus n (Formula 2) is extracted from the public key and sent to the Arduino to be used in encryption. After the encryption is finished, the script receives the ciphertext from the Arduino and can calculate the final result by multiplying the two cypher texts. Finally, the resulting ciphertext can be decrypted to check the correctness of the computation. The Arduino environment utilizes a modified version of the microRSA library to ensure compatibility with an IoT environment. The Arduino receives the modulus from the laptop and encrypts its plaintext message. The following ciphertext is then sent to the laptop for further calculations.

3.4.2 Paillier environments. The Paillier encryption uses the same environment structure as the RSA encryption. The Python environment generates a 1024-bit Paillier key pair using the python-paillier library. The script then extracts the modulus n and squared modulus n^2 (Formula 2). After extraction, the two values are sent over the serial connection. From here the same steps apply as in the RSA environment. The script waits for the encrypted Arduino message, computes the sum, and decrypts the final result. The Arduino environment uses an extended version of the microRSA[18] library to implement the missing mathematical operations. Once the Arduino receives n and n^2 it can start encrypting its plaintext message. Finally, the Arduino sends the ciphertext back to the laptop.

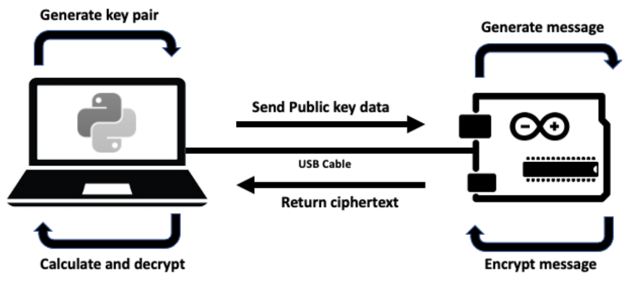

Fig. 1. Schematic overview

4 RESULTS

This section includes an examination of the SMPC protocols, shedding light on the performance and security aspects of the PHE schemes. With a primary focus on RSA and the Paillier cryptosystem, this section delves into the computation time, power consumption, and memory usage analyses. The performance analysis is followed by a security analysis which goes into the security model, cryptographic key size and semantic security. Finally, the section concludes with answering the research questions posed in the introduction.

4.1 Performance analysis

4.1.1 Computation time. This subsection delves into the examination of the computation time of cryptographic operations. To provide a comprehensive view of the minimum, maximum, and average runtime, each reported time in the subsequent tables is based on a sampling of 100 instances.

The runtime analysis presented in Table 3 provides a comprehensive overview of the SMPC protocol using RSA. The execution times for key generation (Keygen), encryption of Python message (EncryptP), decryption (Decrypt) on the laptop, and encryption of the Arduino message (EncryptA) on the Arduino are detailed. Notably, the total runtime of the protocol is heavily influenced by the encryption time on the Arduino, averaging 1807ms, which is approximately 100 times slower than its Python counterpart.

Key generation exhibits notable variability in both minimum and maximum times. However, given that key generation is a less frequent operation in the protocol, the observed variability is unlikely to impact the overall performance of the protocol significantly.

The measured runtimes, encompassing key generation, encryption, and decryption, are crucial metrics for assessing the computational efficiency of the SMPC protocol using RSA with the Arduino Uno.

Table 4 provides a comprehensive breakdown of the computation time for the Paillier-based SMPC protocol on the laptop. Key

Table 3. Computation time RSA

RSA	Min	Average	Max
Keygen	191ms	256ms	578ms
EncryptA	1792ms	1807ms	1834ms
EncryptP	12ms	17ms	32ms
Decrypt	13ms	19ms	29ms
Total	**2008ms**	**2099ms**	**2473ms**

Table 4. Computation time Paillier Python

Pailler Python	Min	Average	Max
Keygen	108ms	356ms	1054ms
Encrypt	24ms	32ms	51ms
Decrypt	7ms	10ms	24ms
Total	**139ms**	**398ms**	**1129ms**

Table 5. Computation time Arduino multiplication

Multiply in bytes	64	128	256
64	25ms	46ms	92ms
128	46ms	99ms	183ms
256	92ms	183ms	367ms

generation (Keygen), encryption of Python messages (Encrypt), and decryption (Decrypt) are the key components analyzed. Key generation, similar to the RSA protocol, displays notable variability in both minimum and maximum times, but, as mentioned earlier, this variability is not expected to significantly impact the overall performance of the protocol due to the infrequency of key generation.

The Paillier encryption faced memory issues preventing a complete run on the Arduino, hindering the acquisition of concrete runtime data. However, an estimate can be derived based on the computational demands of the most intensive task within Paillier encryption: modular exponentiation (Algorithm 1 in the appendix). The algorithm entails 1024 cycles, corresponding to the number of times n needs to be divided by 2 to reach 0. Additionally, within these cycles, the code within the if statement is executed an additional 512 times. This translates to 1536 multiplications of two 256-byte numbers and 1536 modulo operations with a 512-byte number modulo a 256-byte number. Using the information from Table 5 and 6, an estimate is computed. The multiplication step would take $1536*367$ ms, totalling 563,712 seconds or 9.4 minutes. Similarly, the modulation step would take $1536*3668$ ms, totalling 5634,048 seconds or 93.9 minutes. Combined, the total runtime for the function is estimated to be 103.3 minutes. These estimates, though approximations provide insights into the computational demands of the Paillier encryption function on the Arduino.

4.1.2 Power consumption. The power consumption analysis, detailed in Table 7 and 8, provides insights into the energy requirements of the cryptographic algorithms implemented on the Arduino Uno. In terms of amperage, the Arduino exhibited an average

Table 6. Computation time Arduino modulo

Modulo in bytes	**128**	**256**	**512**
128	1ms	922ms	2766ms
256	0ms	2ms	3668ms

Table 7. Ampere usage Arduino

Function	**Min**	**Average**	**Max**
RSA	20,1mA	20,4mA	20,8mA
Multiply	20,2mA	20,3mA	20,5mA
Modulate	20,1mA	20,4mA	21,2mA

Table 8. Wattage Arduino

Function	**Min**	**Average**	**Max**
RSA	102,9mW	103,8mW	105,8mW
Multiply	104,9mW	106,3mW	110,2mW
Modulate	103,7mW	104,2mW	106,2mW

usage of 19.0mA without any algorithms running, operating at a voltage of 5.1 volts. When executing the cryptographic functions (RSA, Multiply, and Modulate), the average amperage only slightly increased to 20.4mA, with the highest peak observed at 21.2mA during the modulation function. These values are well below the Arduino Uno's maximum power draw capacity, which is specified to be at least 200.0mA[25]. It's noteworthy that the recorded amperage values align with the wattages, validating the consistency of the data through Ohm's Law ($W = V * I$). This strong correlation reinforces the reliability of the power consumption measurements across the cryptographic functions. Consequently, the power consumption of these PHE schemes on the Arduino Uno is deemed negligible, emphasizing that power constraints should not pose significant challenges when implementing or discussing these cryptographic algorithms on the Arduino platform.

4.1.3 Memory Usage. In the evaluation of memory usage for the RSA and Paillier implementations on the Arduino, Table 9 provides a comprehensive overview of Flash and dynamic memory consumption before runtime. Both implementations stay well below the maximum thresholds of 32K and 2K respectively, with the RSA utilizing 6076 bytes of Flash and 330 bytes of dynamic memory, while the Paillier implementation uses 10176 bytes of Flash and 493 bytes of dynamic memory. Since the Arduino IDE does not support real-time memory analysis, the subsequent calculations are estimates based on line-by-line code analysis.
To gain insights into the dynamic memory usage during runtime, the memory flow over specific events is depicted in Figures 2 and 3. For the RSA implementation (Figure 2), the memory usage grows during key generation, message generation, and conversion to the custom bignum8 structure. The encryption phase introduces two temporary variables, reaching an estimated highest memory usage of 1226 bytes. The RSA implementation does not encounter memory

Table 9. Memory before runtime in bytes

Memory type	**Flash**	**Dynamic**
RSA	6076	330
Paillier	10176	493

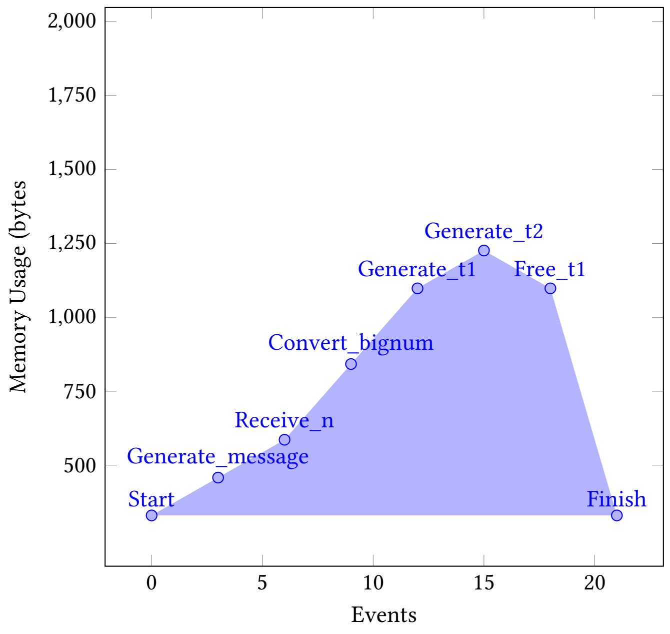

Fig. 2. Estimated memory usage RSA

issues.
The Paillier encryption on the other hand (Figure 3) involves higher memory demands due to the reception of two large variables, n (128 bytes) and n^2 (256 bytes). Additionally, during modular exponentiation, the memory usage variable of r peaks at 512 bytes. The estimated highest memory usage for the Paillier implementation is 1773 bytes, occurring during the modular exponentiation and the final ciphertext calculation. Despite these estimates not reaching or exceeding the Arduino's maximum memory capacity of 2048 bytes, the Paillier implementation experiences program breakdowns and outputs zero values, indicating potential memory-related issues.

4.2 Security analysis

4.2.1 Security model. The security of a protocol is meaningful only when discussed under a specific security model, as the capabilities of the adversary define the security requirements. Three types of security models are often outlined:

- **Semi-honest Adversary Model:** In this model, corrupted parties must execute the protocol correctly. The adversary can obtain information on corrupted parties but will attempt to use this information discreetly. Protocols that achieve this level of security prevent the leakage of information between collaborating parties.

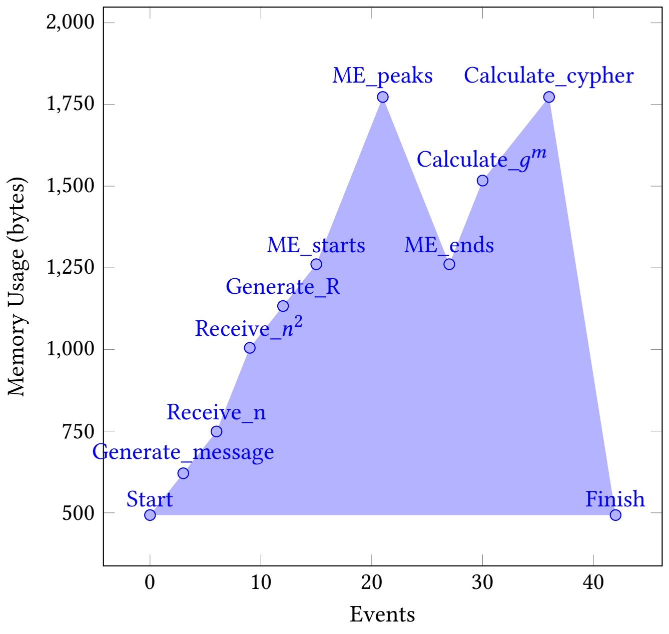

Fig. 3. Estimated memory usage Paillier

- **Malicious Adversary Model:** Corrupted participants in this model may deviate from the protocol's specifications based on the adversary's instructions. A protocol secure against a malicious adversary can guarantee the failure of any adversarial attacks.
- **Covert Adversary Model:** A covert adversary may exhibit malicious behaviour but has a probability of being caught cheating by honest participants.

This paper assumes a Semi-honest Adversary Model as the security model during the execution of the protocol.

4.2.2 Key size. NIST revised its recommendation for RSA key lengths in 2015, now advising a minimum of 2048 bits[26]. This update supersedes the consensus which advocated for a minimum key size of 1024 bits. Consequently, the use of a 1024-bit key in encryption is discouraged for safeguarding sensitive or critical data. However, if the key's lifespan or the protected data spans only days or weeks, the necessity for employing a key resistant to years-long attacks diminishes. The choice of key size and its associated security considerations in the context of the IoT environment would be contingent on specific usage. It's noteworthy that 1024-bit keys represent the upper limit of what an Arduino Uno, without additional memory extensions, can effectively manage.

4.2.3 Semantic security. The intrinsic homomorphic property of RSA makes the algorithm susceptible to semantic insecurity in the absence of proper padding. The lack of semantic security renders the encryption vulnerable to potential attacks such as Chosen Plaintext and Message Replay. The introduction of padding however destroys the homomorphic property, making performing SMPC impossible.

The Paillier cryptosystem does offer semantic security against chosen-plaintext attacks. The successful distinction of the challenge ciphertext boils down to the ability to make decisions about composite residuosity, a task considered computationally intractable under the assumption of decisional composite residuosity. However, despite providing semantic security against chosen-plaintext attacks, the system exhibits malleability due to its homomorphic properties. Thus, it does not achieve the highest level of semantic security, lacking protection against adaptive chosen-ciphertext attacks.

4.3 Answering RQ1

A customized SMPC protocol based on PHE schemes can ensure the confidentiality and integrity of data through the homomorphic properties of the algorithms. PHE schemes are the least computationally complex type of HE and are therefore a logical consideration for the resource-constrained IoT environment. With these homomorphic properties, one can compute results based on ciphertexts rather than on plaintext. This ensures that the context of the numbers is lost and the parties involved do not learn any information about the other participating parties. This protocol can be further extended by implementing secret-sharing techniques like SSS or Oblivious Transfer when distributing keys or ciphertexts.

4.4 Answering RQ2

The practicality of implementing the RSA and Paillier PHE schemes on a resource-constrained device is rather lacking, as efficiency and security perform insufficiently for the IoT environment. For the RSA algorithm, encryption takes on average 1807ms which is too long for the often real-time data needed in the IoT environment. Furthermore, the RSA algorithm cannot make use of padding which makes it semantically insecure. This results in the fact that parties can gain knowledge about other participating parties, which goes against SMPC requirements. The Paillier encryption on the other hand is semantically secure but cannot finish encryption due to running out of dynamic memory. Even if the Paillier algorithm could finish an encryption, this would take at least 103,3 minutes. Again, this is way too long for any practical application in the IoT environment. The security status of the chosen 1024-bit key size is contingent on the usage of the protocol, therefore no concrete conclusion can be made regarding key size other than the fact that the protocols should not be used to store critical data. Power consumption is the only characteristic that both algorithms perform sufficiently. However, this does not weigh up against the lack of computational efficiency and security.

5 CONCLUSIONS

Privacy and security are pivotal in data-sharing within the IoT environment. Despite this, there is a noticeable gap in the literature regarding the practical implementation of SMPC protocols on resource-constrained devices within the IoT. This research details the implementation of two PHE-based SMPC protocols on the resource-constrained Arduino Uno. One protocol is built upon the RSA cryptosystem, and the other is based on the Paillier cryptosystem. The protocols underwent benchmarking and security analysis. For the performance, metrics important to resource-constrained

devices such as memory usage, power consumption, and computation time were tested. Whereas the security analysis delves into the security model, semantic security, and cryptographic key size. Based on the results of this research I conclude that SMPC shows great potential for the resource-constrained IoT environment. However, it is not practical to directly implement Partially Homomorphic Encryption schemes on a resource-constrained device like the Arduino Uno with the goal of SMPC.
I would discourage attempts to optimize RSA and Paillier encryption on the Arduino and instead focus on offloading complex computations and encryptions through the use of cloud computing and secure data offloading. With these techniques, one could utilize even more robust encryption algorithms more efficiently than the Arduino can.

REFERENCES

[1] Adam Thierer and Andrea Castillo. Projecting the growth and economic impact of the internet of things. *George Mason University, Mercatus Center, June*, 15, 2015.
[2] Lo'ai Tawalbeh, Fadi Muheidat, Mais Tawalbeh, and Muhannad Quwaider. Iot privacy and security: Challenges and solutions. *Applied Sciences*, 10(12), 2020.
[3] Asma Haroon, Munam Ali Shah, Yousra Asim, Wajeeha Naeem, Muhammad Kamran, and Qaisar Javaid. Constraints in the iot: the world in 2020 and beyond. *International Journal of Advanced Computer Science and Applications*, 7(11), 2016.
[4] Joseph Migga Kizza. *Internet of Things (IoT): Growth, Challenges, and Security*, pages 517–531. Springer International Publishing, Cham, 2020.
[5] Marcel von Maltitz and Georg Carle. Leveraging secure multiparty computation in the internet of things. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, page 508–510, New York, NY, USA, 2018. Association for Computing Machinery.
[6] Saurabh Singh, Pradip Kumar Sharma, Seo Yeon Moon, and Jong Hyuk Park. Advanced lightweight encryption algorithms for iot devices: survey, challenges and solutions. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–18, 2017.
[7] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu an Tan. Secure multi-party computation: Theory, practice and applications. *Information Sciences*, 476:357–372, 2019.
[8] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982.
[9] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986.
[10] Wang Ren, Xin Tong, Jing Du, Na Wang, Shan Cang Li, Geyong Min, Zhiwei Zhao, and Ali Kashif Bashir. Privacy-preserving using homomorphic encryption in mobile iot systems. *Computer Communications*, 165:105–111, 2021.
[11] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. Cryptology ePrint Archive, Paper 2016/1066, 2016. https://eprint.iacr.org/2016/1066.
[12] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Symposium on the Theory of Computing*, 1990.
[13] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 1–10, New York, NY, USA, 1988. Association for Computing Machinery.
[14] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 218–229, New York, NY, USA, 1987. Association for Computing Machinery.
[15] Himanshu Goyal and Sudipta Saha. Multi-party computation in iot for privacy-preservation. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 1280–1281, 2022.
[16] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, nov 1979.
[17] Oladayo Olufemi Olakanmi and Kehinde Oluwasesan Odeyemi. Trust-aware and incentive-based offloading scheme for secure multi-party computation in internet of things. *Internet of Things*, 19:100527, 2022.
[18] qqqlab. microrsa. https://github.com/qqqlab/microRSA, 2020.
[19] Rhys Weatherley. Arduino cryptography library. https://rweather.github.io/arduinolibs/crypto.html, 2023.
[20] CSIRO's Data61. Python paillier library. https://github.com/data61/python-paillier, 2013.
[21] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, feb 1978.
[22] Kefa Rabah. Review of methods for integer factorization applied to cryptography. *Journal of applied Sciences*, 6(1):458–481, 2006.
[23] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
[24] Dario Catalano, Rosario Gennaro, Nick Howgrave-Graham, and Phong Q. Nguyen. Paillier's cryptosystem revisited. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, CCS '01, page 206–214, New York, NY, USA, 2001. Association for Computing Machinery.
[25] Microchip. megaavr® data sheet. https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061B.pdf, 2019.
[26] Elaine Barker and Quynh Dang. Recommendation for key management part 3: Application-specific key management guidance, 2015-01-22 2015.

6 APPENDIX

```
Data: r: bignum8, n: bignum8, nsq: bignum8
Result: bignum8
res ← bignum8_init(1);
while not bignum8_is_zero(n) do
    if bignum8_is_odd(n) then
        res ← bignum8_multiply_res(res, r);
        bignum8_imodulate(res, nsq);
    end
    bignum8_shift_right(n);
    r ← bignum8_multiply_res(r, r);
    bignum8_imodulate(r, nsq);
end
return res;
```

Algorithm 1: Modular exponentiation function