Multi-Party Computation for IoT Environments: Evaluating Information Checking Protocol-Based Verifiable Secret Sharing Under Resource Constraints

Jaishnoor Kaur, Levent Ertaul
CSU East Bay, Hayward, CA, USA.
jkaur122@horizon.csueastbay.edu, levent.ertaul@csueastbay.edu

***Abstract*—As IoT devices continue to proliferate across both consumer and critical infrastructure, securing their communications is paramount. While standardized Lightweight Cryptography (LWC) provides security under resource constraints, it does not facilitate privacy preservation—an essential requirement given the sensitive nature of IoT-collected data. This paper explores the integration of privacy-enhancing cryptographic primitives in constrained devices by evaluating the Information Checking Protocol (ICP), a key component in Rabin's implementation of verifiable secret sharing (VSS), on the Arduino Uno. Using execution time and energy consumption as key metrics, we analyze the viability of implementing ICP under severe computational and memory constraints. Our findings provide empirical insights into the limits of secure protocol deployment on constrained IoT devices, indicating that although certain cryptographic operations are possible at lower bit lengths, resource limitations render full protocol implementation impractical without significant optimization or upgraded hardware.**

Keywords: Security, IoT Security, Privacy Preserving Computations, Multi-party Computation, Verifiable Secret Sharing, Information Checking Protocol.

I. Introduction

The Internet of Things (IoT) has witnessed immense growth, and while estimates vary wildly [1][2][3][4], it is undeniable that they have been deployed across a multitude of sectors, encompassing everyday consumer devices like wearables for healthcare and home automation systems, as well as critical infrastructure in industrial manufacturing and autonomous vehicle [5]. Their pervasive connectivity, coupled with the amount and nature of data collected by these devices, which may include sensitive information like health indicators, location, preferences, habits etc. [6], makes them a treasure-trove of information and thus, prime targets for cyber attacks [7]. In the same vein, they are also critical to secure.

However, these devices are often small, low-powered, and resource constrained [7]. This is where Lightweight Cryptography (LWC), which is aimed at providing security in resource-constrained applications, becomes paramount [8]. The National Institute of Standards and Technology has published standards for LWC [8]; however, while these algorithms provide security in constrained environments, they do not provide privacy-preservation, which is crucial given that these devices have permeated almost every sphere of our lives. Privacy preservation, thus, must instead be achieved by combining LWC with other techniques [9][10]. One of these privacy enhancing techniques is FHE (full homomorphic encryption)[10]. However, as Goyal and Saha[11] write, this often means that most of the current privacy-preserving methodologies, which are "either depend on computation-intensive Homomorphic Encryption based operations or communication-intensive collaborative mechanisms" [11] are not suitable for these systems. It is in these circumstances that secure multi-party computation (MPC) and secure secret sharing come into the spotlight.

Secure multi-party computation (MPC), which first emerged in the 1980s, has become "one of the most active research areas in both theoretical and applied cryptography" [12]. MPC enables a group to jointly perform a computation without disclosing any participant's secret shares [12]. Essentially, each party holds a secret share of the input, and computations are performed directly on these shares. However, if an adversary corrupts some participants and causes them to send incorrect values, it could compromise the entire computation. This is where the importance of Verifiable secret sharing (VSS) becomes apparent [13]. VSS helps mitigate this risk by ensuring that all shared values are verifiable, preventing dishonest participants from injecting false data into the protocol[13].

In Section II of this paper, we introduce Verifiable Secret Sharing, the Information Checking Protocol (ICP) as well as our goals for this study. Section III outlines the steps and participants in the Information Checking Protocol while Section IV goes into specific details of each component of ICP tested. Next, we introduce our experimental environment in Section V. In Section VI, we share our experimental results and analyze the performance

and limitations of Arduino Uno for ICP as explained in Sections II and IV. Finally, in Section 7, we summarize our findings and present our conclusions.

II. Verifiable Secret Sharing (VSS)

Verifiable Secret Sharing (VSS) is a cryptographic protocol that ensures shared secrets remain consistent and reconstructable, even in the presence of a malicious dealer [14].Unlike standard secret sharing, which assumes an honest dealer, VSS incorporates auxiliary verification mechanisms to prevent adversarial manipulation. This property makes VSS fundamental for secure multi-party computation (MPC), where computations are performed on secret shares [13][15],as it allows for security against active adversaries.

Rabin and Ben-Or [16]made key contributions to VSS, and proposed an implementation of VSS under the assumption that participants can securely communicate and broadcast messages. Their work demonstrated that any multi-party protocol or game with incomplete information could be securely executed if the majority of players remain honest. Notably, they introduced the Information Checking Protocol (ICP), a cryptographic tool enabling authentication without relying on computational intractability assumptions.

In this paper, we investigate the feasibility of implementing Identity Commitment Protocols (ICP) within a Verifiable Secret Sharing (VSS) framework on the Arduino Uno—one of the smallest and most resource-limited microcontrollers available. Due to its stringent computational and memory constraints, using the Uno serves as somewhat of an extreme benchmark for assessing practical deployment in lightweight IoT environments. For each critical component of the protocol, defined in section IV, execution time was measured using the millis() function, which leverages the Arduino's hardware timer interrupt mechanism [17]. These timing metrics were then used to estimate energy consumption, offering insights into the real-world cost of cryptographic security on low-power devices. In addition to performance profiling, the study highlights the limitations imposed by the Uno's limited memory, which affected the feasibility of certain cryptographic operations. Together, our results assess whether ICP-based VSS can be effectively adapted to such constrained environments, or whether further optimization is necessary to enable secure communication on ultra-low-power hardware.

III. Information Checking Protocol (ICP)

Rabin's VSS protocol utilizes two sub-protocols [16]:

1) **ICP**: Information Checking Protocol
2) **WSS**: Weak Secret Sharing

In this paper, the focus is on the Information Checking Protocol (ICP). The Information Checking protocol consists of three actors: the dealer (D), the intermediary (I), and the recipient (R).

To achieve an error probability of at most $1/2^k$, set $k = 128$. This is also the value of k used within the ICP protocol.

Additionally, the ICP protocol operates under the assumption that a sufficiently large prime number has been chosen for all computations, such that $p > 2^k$. The generation of such prime numbers was also tested using the Arduino Uno.

The Information Checking Protocol and the steps performed by each actor (D, I, and R) in it are outlined below:

Dealer (D)

1) Chooses $2k$ values $b \neq 0$ and $2k$ values y, both in Z_p.
2) Calculates $c = s + by$ for each of the $2k$ b and y.
3) Transmits $2k$ pairs (s, y) to I.
4) Transmits $2k$ pairs (b, c) to R.

Intermediary (I)

1) Receives $2k$ pairs (s, y) from D.
2) Transmits $2k$ pairs (s, y) to R.
3) Selects k distinct random indices from the range $[1, 2k]$.
4) Requests R to reveal the check vectors (b, c) corresponding to the selected indices.
5) For each revealed check vector, verifies whether $s + by == c$. If all k vectors satisfy this condition, I confirms that R will accept secret s. Otherwise, I determines that R will reject s.

Recipient (R)

1) Receives the $2k$ pairs $(b_1, c_1), \ldots, (b_{2k}, c_{2k})$ from D.
2) Receives the $2k$ pairs $(s, y_1), \ldots, (s, y_{2k})$ from I.
3) For the unrevealed check vectors, verifies whether $s + by == c$. If any fail this check, R rejects s; otherwise, R accepts s.

IV. Arduino Feasibility and Performance Testing Methodology

The ICP protocol was tested on the Arduino Uno, in addition to testing prime number generation. For the ICP, recall that it consists of three actors: the dealer (D), the intermediary (I), and the recipient (R). The Arduino was used to represent each of these actors in turn, and its performance was investigated for the actions performed in each role. The specifics are outlined below.

For each role, the following were tested:

Dealer (D)

1) Choosing $2k$ values of b and y as per the constraints described in Section 2.
2) Calculating c for each pair (b, y).
3) Transmitting the check vector pairs (s, y) and (b, c).

Intermediary (I)

1) Transmit $2k$ pairs (s, y) to R: Unlike the transmission of (s, y) and (b, c) tested in the previous phase, here only 256 (s, y) pairs needed to be transmitted.
2) Selecting k indices as described in Section 2: Choosing indices follows the same process regardless of the

size of the prime, secret, or other data. Therefore, a unified test was performed irrespective of bit size, but with multiple runs.

3) Checking if $s + by == c$ for the k indices selected: Unlike the generation of c tested in the previous phase—where each c was generated and sent without storing any previous values— here, there was a need to store the values of b, y, and c.

Recipient (R)

1) Checking if $s + by == c$ for remaining k indices (the ones *not* selected): Checking 128 times takes the same amount of time as for I, as the same device, architecture, code, and the same number and size of data were used. This step requires verifying c only for those pairs that have not been revealed. We traverse the range $[0, 256]$ linearly, skipping indices that were selected for checking. Since the indices are already sorted (because of the way they were stored upon selection), overhead from searching gets reduced.

For each of these components, execution time was recorded using the ***millis*** function, which is based on the Arduino's hardware timer interrupt [17]. This information was then used to calculate energy consumption for each of the tested operations.

V. Implementation environment

The Arduino Uno R3 board was used for this study. It is one of the smallest and least powerful boards available. Its technical specifications are detailed in the Table 1 [18].

Table 1. Arduino Uno Specs

Feature	Spec
CPU (Microcontroller)	ATmega328P
SRAM	2KB
Flash Memory	32KB
Serial I/O	USB-B
Connection	Serial

It can be seen that this Arduino board has extremely limited resources, with only 2 kilobytes of SRAM and 32 kilobytes of flash (non-volatile) memory. Additionally, we note from Table 1 that the board is severely restricted in terms of communication media as it does not support wireless connections, rather it only has Serial communication available. The features of this communication method have been briefly explained in VI-B3 "Transmission of Check Vector Pairs".

VI. Performance Analysis

This section presents the results of the tests conducted on the Arduino Uno R3. The code for all these tests was written in Arduino, an Arduino-specific variant of C++.

A. Prime Number Generation

To generate n-bit prime numbers on the Uno, the BigNumber library was used for large integer support (128-bits and above).

The process involved:

1) Generating a random BigNumber of the approximately desired length.
2) Finding the next prime number greater than or equal to the generated number
3) Using the Miller-Rabin test to verify primality.

For each bit size, 240 test runs were done.

Table 2. Time Taken (in ms) to Generate Prime

No. of Bits	Mean	Median	Std Dev
32	3050.41	2343	1561.70
64	23644.06	19105	13066.01
128	263865.44	191749	174809.89
152	623644.17	436752	435591.10

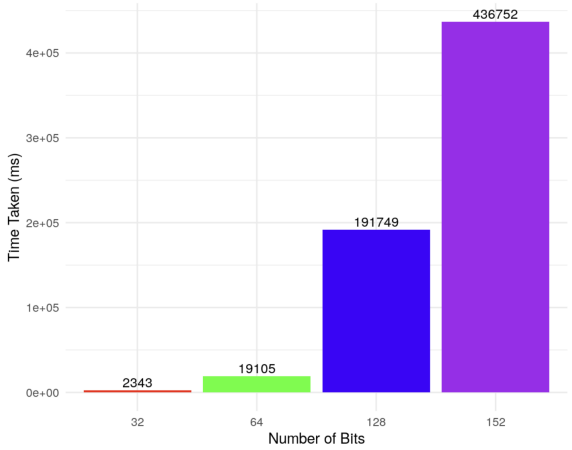

Fig. 1. Time Taken (in ms) to Generate Prime

It was observed (ref. Table 2) that the prime generation time varied significantly depending on the initial seed value, which is reflected in the high standard deviation. Given this variation, the median time was calculated, as it is a more robust statistic and is not affected by extreme outliers. To maintain consistency, we have continued displaying the median in all subsequent measurements, providing a more reliable overview of resource consumption during ICP execution.

From Fig. 1, it is evident that as the bit size increases, the time to generate a prime grows rapidly— almost by an order of magnitude (10^1) per step. For larger bit sizes, prime generation could take several minutes, with the median time for generating a 152-bit prime reaching approximately 7 minutes. This indicates that frequent generation of primes of such large sizes may be impractical on the Arduino Uno.

Additionally, the Arduino Uno R3 was unable to consistently generate primes larger than 152 bits. For primes greater than 152 bits, the processing time occasionally exceeded 2,000,000 ms, at which point the calculation was manually aborted.

Recall that the ICP and VSS protocols require a prime number $p > 2^k$ (i.e., $> 2^{128}$). A prime of this size requires

at least 129 bits for correct representation. This limitation significantly restricts the ability of the Arduino Uno to generate and efficiently handle sufficiently large primes for use in these protocols.

For the remainder of this paper, we include computations with numbers of sizes >152-bits by using predefined prime-numbers (externally generated and hard coded) of the required sizes. This is also a practice that may be used in real-world situations.

B. Arduino acting as Dealer D

1) Generation of 2k b and y for a Given Prime: 256 ($= 2k$) values of b and y were generated for given primes of sizes 64, 128, 256, and 512 bits, within the constraints outlined in the description of ICP in Section III.

Table 3. Time Taken (in ms) to Generate 256 b and 256 y

Bit Size of Prime	Mean	Median	Std Dev
64	3635.86	3673	89.54
128	9143.74	9136	45.95
256	24810.25	24884	183.77

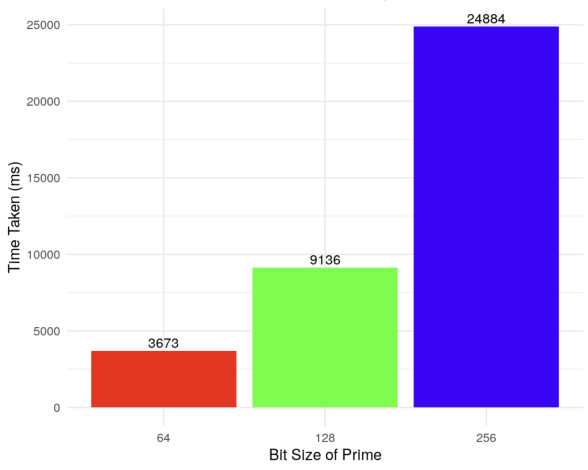

Fig. 2. Time Taken (in ms) to Generate 256 b and 256 y

As seen in Table 3, the generation of the 256 values for b and y was not overly time-consuming. Even though the time consumption increased with increasing bit sizes (Fig. 2), overall the times remain relatively modest across all tested bit sizes.

However, it is important to note that these values were not stored. Instead, they were generated and immediately overwritten. This approach suffices at this stage, but for later stages of the ICP protocol, storage becomes necessary. Specifically, at least one b and one y need to be stored at any given time, and any memory limitations will be explored during those phases.

At this stage, it was also observed that for 512-bit primes, memory constraints on the Arduino Uno prevented the computations required to generate 256 b and y as the Uno was unable to handle all the transient data generated during these calculations. This limitation hindered the Uno's ability to efficiently handle 512-bit primes (including externally generated ones) for use in the VSS protocol. But for primes of sizes like 256 bits, which are >152 bits and <512 bits, it is possible to generate the prime externally and then provide it to the Arduino. The Arduino can then take over and handle the generation and calculation of the check vectors.

2) Calculating c: For a given secret s (of size approximately $=$ sizeof$(b \cdot y)$), c was computed as $c = s + by$ for each of the 256 pairs of b and y.

Table 4. Time Taken (in ms) to Calculate c for 256 b, y

Bit Size of Prime	Mean	Median	Std Dev
64	413.26	413	11.93
128	1102.28	1099.5	18.33
256	3549.31	3548	9.17

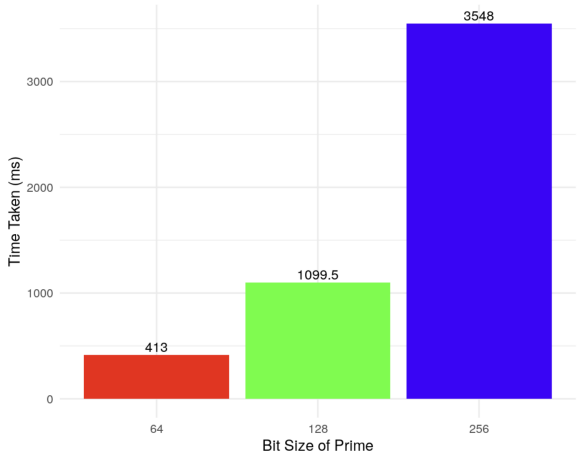

Fig. 3. Time Taken (in ms) to Calculate c for 256 b, y

The calculation of c was very fast across all tested bit sizes, as shown by the mean, median, and standard deviation values in Table 4.

For the 64-bit prime, the time taken to compute c averaged around 413 ms, with a small standard deviation of 11.93 ms, indicating consistent performance. Similarly, for 128-bit primes, the mean time was 1102.28 ms, and the performance remained fairly consistent with a standard deviation of 18.33 ms. For the 256-bit primes, the time taken grew to an average of 3549.31 ms, but the standard deviation remained relatively low at 9.17 ms, again indicating stable performance despite the increase in bit size.

Figure 3 shows the increase in calculation time as the size of the prime increases. The time-taken seems to increase exponentially, possibly because of the large size of the values involved in the calculations as well as the often greater number of values to be checked. Despite this increase, the computation time for calculating c remains relatively low even for the largest tested prime size (256 bits).

However, it is important to note that c must be stored alongside b and y for subsequent phases of the ICP protocol. As the prime size increases, so too does the number of values to be stored, placing additional demands on the Arduino's memory capacity. This factor must be considered when scaling up the protocol to handle larger primes.

3) Transmission of Check Vector Pairs (s, y) and (b, c): The 256 check vector pairs (s, y) and (b, c) were transmitted via serial communication. Serial ports are used to physically connect asynchronous devices to a computer, enabling the transfer of data in a sequential format, where bits are transmitted one after the other in a series [19]. As mentioned in 1, the Arduino board tested here facilitates serial communication using the UART protocol [20], a simple, low-cost and easy to implement serial communication protocol that can be used to send data between an Arduino board and other devices [20].

Table 5. Time Taken (in ms) to Transmit 256 Pairs of (s, y) and (b, c)

Bit Size of Prime	Mean	Median	Std Dev
64	27537.88	27571.5	122.10
128	52298.85	52291	152.34
256	101355.9091	101540	534.4533

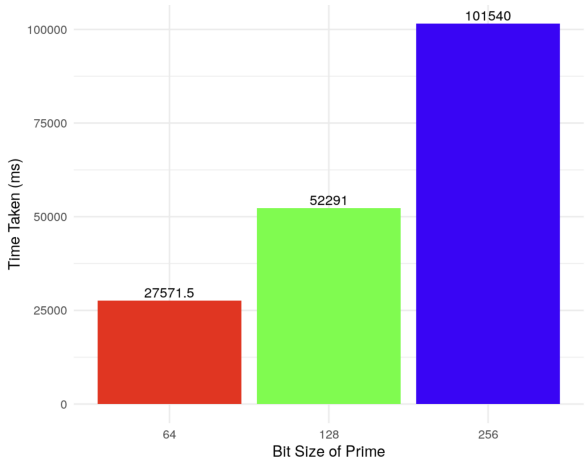

Fig. 4. Time Taken (in ms) to Transmit 256 Pairs of (s, y) and (b, c)

It is observed in Table 5 that the transmission time is largely dependent on the size of the data being sent. For 64-bit primes, transmitting all 256 check vector pairs took approximately 27.5 seconds, whereas for 128-bit and 256-bit primes, the time increased to 52.3 seconds and 101.3 seconds, respectively. The near-linear growth in transmission time seen in Fig. 4 aligns with the increasing bit size of the data being communicated.

Given that the secret s remains constant across all 256 check vectors in a transmission sequence, a potential optimization could be sending s once, followed by the 256 values of y, instead of redundantly transmitting 256 pairs of (s, y). This would effectively cut the transmission size by nearly (for the (s, y) pairs, reducing both time and resource consumption.

C. Arduino acting as Intermediary I

1) Transmission of Pairs (s, y): The 256 check vector pairs (s, y) were transmitted via Serial communication from the intermediary I. The transmission time is primarily dependent on the size of the transmitted data.

Table 6. Time Taken (in ms) to Transmit 256 Pairs of (s, y)

Bit Size of Prime	Mean	Median	Std Dev
64	12380.67857	12380	5.25
128	22181.32143	22182	6.2083
256	38362.10714	38347	37.0833

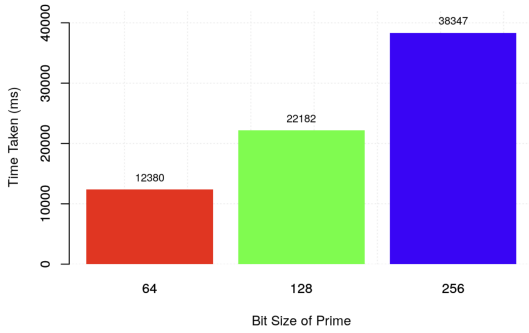

Fig. 5. Time Taken (in ms) to Transmit 256 Pairs of (s, y)

The transmission time increases steadily with the bit size of the prime, as seen in Table 6 and Fig. 5. For 64-bit primes, the process takes 12.4 seconds, whereas for 128-bit and 256-bit primes, the transmission time nearly doubles and triples, respectively. The low standard deviation indicates that the transmission times remain stable across multiple runs, suggesting minimal external interference or buffering issues.

As previously noted, since the secret s remains constant for all check vectors, a more efficient approach would be to transmit s once and then send only the 256 values of y.

2) Selecting k Random Unique Indices: The process used for selecting k distinct random indices was as follows in Fig. 6:

Table 7. Time Taken (in ms) to Select k Random Distinct Indices

Bit Size of Prime	Mean	Median	Std Dev
ANY	24	24	0

As shown in Table 7, the time taken for selecting k random indices was consistently **24ms**, with no variation observed across multiple runs. This indicates that the

```
FOR j FROM 0 TO 127 DO
    SET seed TO j + 1000
    INITIALIZE random number generator
    with seed
    SET temp TO random number between
    0 and 255

    SET exists TO false
    FOR k FROM 0 TO n - 1 DO
        IF ind[k] == temp THEN
            SET exists TO true
            DECREMENT j
            BREAK
        END IF
    END FOR

    // Find correct position and shift
    //elements
    SET i TO n - 1
    WHILE i >= 0 AND ind[i] > temp DO
        SET ind[i + 1] TO ind[i]
        // Shift elements to the right
        DECREMENT i
    END WHILE
    SET ind[i + 1] TO temp
    // Insert the value in the correct
    //position
    INCREMENT n
    // Increase the number of elements
END FOR
```

Fig. 6. Pseudocode for selecting k distinct random indices

operation is both stable and computationally efficient, imposing minimal overhead in the overall protocol.

3) Checking c for k b and y in I: For a given secret s (of size approximately sizeof$(b \cdot y)$), c was computed as $c = s + by$ for each of the 128 pairs of b and y.

Initially, the thought was to store these values in an array. However, since BigNumber does not support arrays directly, the common practice is to store them as strings and create an array of those. The first attempt was made by using the 'String' type, but this approach consumed too much dynamic memory, even for b and y values of 64 bits.

```
Sketch uses 24658 bytes (76%) of program storage space. Maximum is 32256 bytes.
Global variables use 13874 bytes (677%) of dynamic memory, leaving -11826 bytes
Not enough memory; see https://support.arduino.cc/hc/en-us/articles/36001382517
data section exceeds available space in board

Compilation error: data section exceeds available space in board
```

Fig. 7. Memory Exhausted When Using Strings

As shown in Fig. 7, using 'String' resulted in 13,874 bytes of dynamic memory usage, while the Arduino only has 2,048 bytes available.

To overcome this, each BigNumber was stored as a character array instead, as character arrays have much less overhead than 'String' objects. Additionally, the 'PROGMEM' technique [21], which stores these character arrays in program memory rather than dynamic memory, was applied. This is advantageous because the Arduino has much more program memory available (32,256 bytes) compared to SRAM (2,048 bytes).

With this adjustment, it became possible to handle 64-bit numbers effectively. However, when testing with 128-bit numbers, a memory limitation again was encountered again. Despite using the second method, storing 128 b, y, and c values (with b and y of 128 bits each) along with the program code required 34,804 bytes of memory. This exceeds the available program memory of 32,256 bytes (Fig. 8).

Note also, that in both the methods tried previously, s was stored only once as a space saving measure, since it is the same for all values of b, y and c.

```
Sketch uses 34804 bytes (107%) of program storage space. Maximum is 32256 bytes.
Global variables use 644 bytes (31%) of dynamic memory, leaving 1404 bytes for local var
Sketch too big; see https://support.arduino.cc/hc/en-us/articles/360013825179 for tips o
text section exceeds available space in board

Compilation error: text section exceeds available space in board
```

Fig. 8. Program Memory Exhausted with Method 2 for 128-bit Numbers

The results of the timing measurements for checking c are shown in Table 8 below. We observe that the checking was quite fast, taking only ∼270ms or ∼3s.

Table 8. Time Taken (in ms) to Check c for 128 b, y

Bit Size of Prime	Mean	Median	Std Dev
64	270.7931034	269	2.6531
128	-	-	-
256	-	-	-

Note: Optimizations have not been explored yet. One option is the use of F() macro. In general, for IoT applications, a more powerful board with internet capabilities (e.g., ATmega 2560 or Raspberry Pi) would be more suitable, as these boards have more memory and processing power [22][23].

Overall, checking c was relatively quick, with little time taken for 64-bit primes. However, as the size of the primes increases, the memory limitations become more apparent, and handling larger bit sizes may require more capable hardware.

D. Arduino acts as recipient R

1) Checking c for k b and y in R: The same setup and code were used for this as for the intermediary I. The main difference is that R stores 256 b, y, and c values, as compared to the 128 b and c values stored by I. This increased storage requirement implies that calculations that were previously not possible for I are also not feasible for R.

For R, only the c belonging to pairs (b, c) that have not been selected by I for verification need to be checked. This was achieved using the condition:

```
if (ind[j] == i) { j++; continue; }
```

This approach ensures that only the relevant pairs are checked.

Table 9. Time Taken (in ms) to Check c for 128 b, y

Bit Size of Prime	Mean	Median	Std Dev
64	278	278	0
128	-	-	-
256	-	-	-

It was observed that the time required for the checking was minimal (Table 9). The timing is almost identical to the previous section (Table 8), with a slight increase caused by the additional step of checking the indices of the pairs. This extra checking of indices was necessary to ensure that only the relevant pairs of (b, c) that had not been sent to I were processed.

The minimal increase in time is in line with expectations, as the added step of checking indices introduces a small overhead, but the overall computational complexity remains low for 64-bit primes. Larger primes were not tested due to the memory limitations previously discussed in Section VI-C3.

E. Energy Consumption

The energy consumption for each component was calculated using the formula:

$$E = V \cdot I \cdot n \cdot \tau$$

where:
- V is the voltage (5V for the Arduino Uno R3),
- I is the current (20mA for the Arduino Uno R3) [24],
- n is the number of operations, and
- τ is the time taken for each operation.

Since time taken for each component has already been measured, energy can be simplified to:

$$E = V \cdot I \cdot t$$

where t is the time taken for the operations.

The energy consumption was calculated using the median execution times for each component and bit size, with the values shown in Table 10:

Unsurprisingly, the energy consumption across different components of the system is influenced primarily by the size of the data being processed and transmitted as larger bit sizes require more time to process and transmit data, which directly translates to higher energy consumption.

Table 10 also shows that the generation of primes and the transmission of data are the two most expensive operations. In fact, prime generation also shows one of the most drastic increases in energy consumption with increasing bit-size, from 1.91J for 64-bit primes to 19.17 for 128-bit primes. Transmission, be it from D (seen in Fig. 9) or I is also an energy-intensive operation. In this study, Serial communication was, but in practice some form of wireless communication is quite likely to be used[25], and that is generally more expensive than the Serial communication method used here [26].

Table 10. Energy Consumption for Various Components at Different Bit Sizes

Component D	Bit Size	Energy (J)
b and y Gen.	64	0.04
	128	0.11
	256	0.35
Calc. c	64	0.37
	128	0.91
	256	2.49
Transmission	64	2.76
	128	5.23
	256	10.15
Component I		
Transmission from I to R	64	1.24
	128	2.22
	256	3.84
Selecting k indices	Any	0.0024
Checking 128 c in I	64	0.027
Component R		
Checking 128 c in R	64	0.028
General		
Prime Gen.	32	0.23
	64	1.91
	128	19.17

Another relatively expensive operation is the calculation of c when Arduino is acting as dealer D. It can be seen in Table 10 that as the bit-size of operands increases, energy consumption increases quite a lot. Other operations, such as generating b and y, selecting k random indices etc., do not require much energy in comparison to the two most expensive actions (all <0.5J).

Overall, with larger bit-sizes being used for computaion and transmission, the combined energy consumption for each role (D, I, or R) grows. Considering that prime number generation can be outsourced or performed infrequently, in general, transmission emerges as the most expensive operation in terms of energy consumption, followed by operations that involve larger bit sizes like calculation of c. Optimizing calculations and memory handling could lead to substantial power savings, particularly when working with large data sets. Furthermore, using more powerful microcontrollers (e.g., ATmega2560 [23] or Raspberry Pi[22]) could allow for handling larger bit sizes more efficiently, improving execution time and reducing energy consumption per operation.

Putting this in practical terms of battery life, consider a 6V battery. (Arduino ideally requires a 7-12V supply if powered via DC, but it can function with 6V as well.) For a lightweight 6V battery, capacity commonly ranges from 1000-2000mAh [27]. For these calculations, consider a battery with a capacity of 1400mAh such as a Duracell

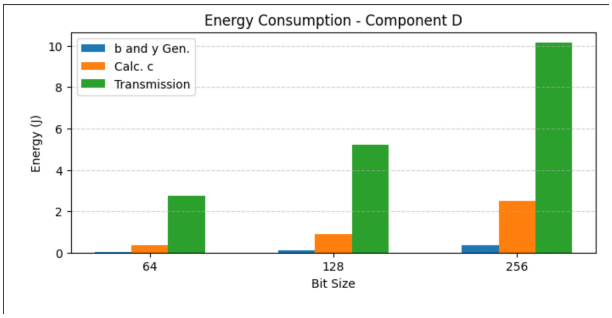

Fig. 9. Energy Usage Comparison of Arduino as D

DL 233 CR-P2. Converting this to Wh using the formula *Capacity(mAh) × Voltage(V) / 1000* gives a battery capacity of 8.4Wh.

Now, consider the Arduino's combined energy consumption for a role and include all operations of that role. For cases where multiple bit-sizes have been tested, only the most expensive for each operation will be considered.

As can be seen from Table 10, the most energy is consumed when the Arduino acts as the dealer D, making it a good candidate case for the upcoming calculations. Thus, considering the case when Arduino acts a D. Here, the main operations are:

1) **Generation of b and y** which consumes $0.35J$ in $25s$. This can be converted to Power consumed using the formula: $Power(P) = Energy(E)/Time(T)$ i.e., $P = 0.35/25 = 0.014W$.
2) **Calculating** c which consumes $2.49J$ in $4s$, giving power usage equal to $0.6225W$.
3) **Transmitting check vector pairs** which consumes $10.15J$ in $101s$, giving a power consumption of $0.1005W$.

Thus, for dealer D, power consumption comes out to be $0.737W$.

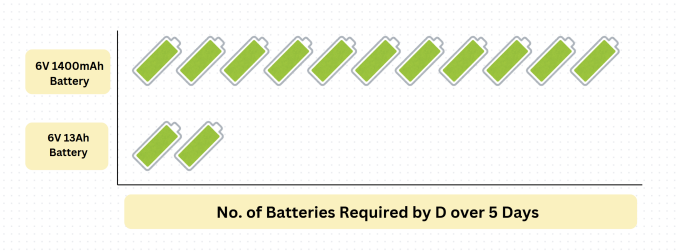

Fig. 10. Battery Consumption of Arduino acting as Dealer D over 5 days

Battery life can now be calculated using the formula: $BatteryLife(hrs) = BatteryCapacity(Wh)/P_{total}(W)$. The result is an approximate battery life of 11.4 hours i.e., when Arduino is acting as Dealer D, a 6V 1400mAh battery will require replacement every 11 hours or so. However, if the use case allows for a larger, more powerful battery such as a 13Ah Duracell LA battery [28], it would require much more infrequent replacement. Specifically, the capacity for such a battery would be $78Wh$, giving a battery life of 105.8 hours for use with Arduino as D. In other words, such a battery would require replacement approximately every 4 days.

In other words, for every 5 days of operating the Arduino Uno as dealer D, a 1400mAh battery will require replacement 10 times and a 13Ah will require replacement once (ref. Fig. 10).

VII. Conclusion

This study evaluated the performance of the Arduino Uno R3 while executing critical components of the Information Checking Protocol (ICP) from Rabin's implementation of the VSS [16]. Through extensive timing and energy measurements across multiple bit sizes (64, 128, and 256 bits), we derived a comprehensive understanding of the limitations and capabilities of the Arduino in the roles of Dealer (D), Intermediary (I), and Recipient (R).

Our findings indicate that while the Arduino performs consistently and reliably for 64-bit operations, scalability becomes a challenge with increasing bit sizes. Notably, the Arduino ran out of memory when attempting to generate large primes or attempting to store multiple 128-bit and 256-bit values and perform operations on them.

From a performance standpoint, execution times for key operations (except prime-number generation) remained stable across repeated trials. However, the transmission of check vectors and generation of primes proved to be the most power-hungry operations. When the Arduino acts as the dealer, a 6V 1400mAh battery can continuously sustain such operations for about 11.4 hours. Larger batteries, such as a 13Ah unit, can extend operational time to 105.8 hours, supporting multi-day deployments. These findings highlight the importance of power-aware protocol design in low-power IoT environments.

In light of these observations, we conclude that while there are a few options, such as aggressively optimizing memory usage, reducing transmission overhead by avoiding redundant data, choosing to use small primes while making concessions on the security of the protocol etc. for implementing ICP in a severely limited device like Arduino Uno, a complete, highly secure implementation requires the use of better, more powerful hardware.

References

[1] Alam T. A reliable communication framework and its use in internet of things (iot). International Journal of Scientific Research in Computer Science Engineering and Information Technology 05 2018;3.
[2] Technologies D. Internet of things and data placement. URL https://infohub.delltechnologies.com/en-us/l/edge-to-core-and-the-internet-of-things-2/internet-of-things-and-data-placement/.
[3] Analytics I. Number of connected iot devices worldwide. URL https://iot-analytics.com/number-connected-iot-devices/.
[4] Statista. Internet of things (iot). URL https://www.statista.com/topics/2637/internet-of-things/#topicOverview.
[5] SoluLab. Iot: The future of innovation technology. URL https://www.solulab.com/iot-the-future-of-innovation-technology/.

[6] TrustCloud. Data privacy in the age of iot: securing connected devices in 2024. URL https://community.trustcloud.ai/docs/grc-launchpad/grc-101/governance/data-privacy-in-the-age-of-iot-securing-connected-devices-in-2024/.

[7] Simmons-Simmons. Technotes – top 10 security and privacy issues within the iot. URL https://www.simmons-simmons.com/en/publications/ckmx5qvys13hb0910ags2nx7o/technotes-top-10-security-and-privacy-issues-within-the-iot.

[8] Meltem Sönmez Turan Kerry A. McKay DCJKJK. Ascon-based lightweight cryptography standards for constrained devices. NIST Special Publication 800 NIST SP 800 232 ipd 2023; URL https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-232.ipd.pdf.

[9] Knudsen LR, Peyrin T, Sasaki Y. Can lwc and pec be friends? https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2020/documents/papers/can-lwc-pec-be-friends-lwc2020.pdf, 2020. Paper presented at the NIST Lightweight Cryptography Workshop 2020.

[10] NIST. Fostering standards for privacy-enhancing cryptography (pec). https://csrc.nist.gov/csrc/media/Presentations/2022/fostering-standards-for-pec/images-media/20220519-PET-Summit-Boston--Fostering-Standards-PEC--Slides-rev20220526.pdf, 2022. Presentation at the 2022 Privacy-Enhancing Technologies (PET) Summit, Boston.

[11] Goyal H, Saha S. Multi-party computation in iot for privacy-preservation. In 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS). 2022; 1280–1281.

[12] Evans D, Kolesnikov V, Rosulek M. A Pragmatic Introduction to Secure Multi-Party Computation. NOW Publishers, 2018. Online version: Apr. 15, 2020. [Online]. Available: https://www.cs.virginia.edu/~evans/pragmaticmpc/pragmaticmpc.pdf.

[13] Das S, Xiang Z, Tomescu A, Spiegelman A, Pinkas B, Ren L. Verifiable secret sharing simplified. Cryptology ePrint Archive, Paper 2023/1196, 2023. URL https://eprint.iacr.org/2023/1196.

[14] Krenn S, Lorünser T. Verifiable Secret Sharing. Cham: Springer International Publishing. ISBN 978-3-031-28161-7, 2023; 45–54. URL https://doi.org/10.1007/978-3-031-28161-7_7.

[15] Chor B, Goldwasser S, Micali S, Awerbuch B. Verifiable secret sharing and achieving simultaneity in the presence of faults. In 26th Annual Symposium on Foundations of Computer Science (sfcs 1985). 1985; 383–395.

[16] Rabin T, Ben-Or M. Verifiable secret sharing and multiparty protocols with honest majority. In Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, STOC '89. New York, NY, USA: Association for Computing Machinery. ISBN 0897913078, 1989; 73–85. URL https://doi.org/10.1145/73007.73014.

[17] Electronics P. millis() arduino function: 5+ things to consider, n.d. URL https://www.programmingelectronics.com/millis-arduino/.

[18] Documentation A. Arduino uno rev3 technical specs, n.d. URL https://docs.arduino.cc/hardware/uno-rev3/tech-specs.

[19] IBM. Serial communication. URL https://www.ibm.com/docs/en/aix/7.3?topic=communications-serial-communication.

[20] Siebeneicher H. Universal asynchronous receiver-transmitter (uart). URL https://docs.arduino.cc/learn/communication/uart/.

[21] Documentation A. Arduino progmem, n.d. URL https://www.arduino.cc/reference/tr/language/variables/utilities/progmem/.

[22] Pounder L. Raspberry Pi vs Arduino: Which Board is Best? https://www.tomshardware.com/features/raspberry-pi-vs-arduino, 2023.

[23] Microchip Technology Inc. ATmega2560 - 8-bit AVR Microcontroller. https://www.microchip.com/en-us/product/atmega2560, 2024.

[24] Forum A. Power consumption discussion, 2020. URL https://forum.arduino.cc/t/power-consumption/661308/5.

[25] Sinha S. State of iot 2024: Number of connected iot devices growing 13 URL https://iot-analytics.com/number-connected-iot-devices/#:~:text=Wi%2DFi.

[26] Kasslack R. Balancing wireless innovation with wired reliability, 2024. URL https://www.securityindustry.org/2024/03/19/balancing-wireless-innovation-with-wired-reliability/#:~:text=Wired%20networks%2C%20with%20their%20lower,the%20overall%20energy%20efficiency%20of.

[27] Junction B. 6v batteries. URL https://www.batteryjunction.com/batteries/6v?srsltid=AfmBOopsK3jNAak-9npp-QyuxOCgc_OpxxLAgWeBvfXJ6GuWYSK2gurg.

[28] Uline. Duracell 6v lantern alkaline battery. URL https://www.uline.com/Product/Detail/S-17590/Batteries/Duracell-6V-Lantern-Alkaline-Battery?pricode=WB0943&gadtype=pla&id=S-17590&gad_source=1&gclid=Cj0KCQjw4cS-BhDGARIsABg4_J1m7Ogc36pCQvLGdLbOkZskzePykjEcJmueOF6tOg9wg4LvT5ZpB1EaArMvEALw_wcB.