On-sensor Stream Cipher Encryption for Protecting Smart City Sensor Data directly on Resource-Constrained IoT-Sensors

Jan Seedorf[1], Darshana Rawal[1], Matthias Hamann[1], Sebastian Seid[1], Kai Schneider[1], Vu Danh Anh[1], Maximilian Haag[1], Fabian Milla[1], Yunus Emre Altun[1]
[1] HFT Stuttgart, Schellingstraße 24, 70174 Stuttgart, Germany
jan.seedorf@hft-stuttgart.de, darshana.rawal@hft-stuttgart.de, matthias.hamann@hft-stuttgart.de, 12sese1bif@hft-stuttgart.de, 12scka1bif@hft-stuttgart.de, 22davu1bif@hft-stuttgart.de, maximilian.haag@hft-stuttgart.de, 92mifa1bif@hft-stuttgart.de, 22alyu1bif@hft-stuttgart.de

Keywords: Smart City Sensor Data, Cryptography, Stream Cipher Encryption

Abstract

This paper presents a comparative analysis of the DRACO stream cipher implemented in both C++ and Rust across three microcontroller platforms: ESP32, ESP8266, and Raspberry Pi Pico. DRACO, a lightweight cipher designed for constrained environments, was evaluated in terms of performance, memory efficiency, and initialization behavior. To ensure reliable and repeatable results, a standardized benchmarking framework was developed, including automated testing, key/IV configuration variants, and statistical analysis of execution times. In addition to synthetic benchmarks, the evaluation incorporated representative smart city data sets, simulating real-world sensor inputs to reflect practical encryption workloads. Performance metrics—including keystream generation, encryption, and algorithm initialization—were collected via serial output and processed using Python and visualization tools.

Results show that both implementations perform consistently across varying keystream lengths, with Rust demonstrating faster execution at very short lengths, and C++ slightly outperforming Rust at longer lengths. The study also discusses the implementation challenges and trade-offs between security, compiler support, and platform compatibility. Findings highlight that both languages are suitable for embedded cryptographic applications, with Rust offering stronger language safety guarantees and C++ providing broader microcontroller support. This work further emphasizes the importance of lightweight encryption in smart city sensor networks, where securing real-time, sensitive data is critical to privacy, reliability, and urban infrastructure resilience.

1. Introduction and Motivation

1.1 The Need for Protecting Smart City Data

Encryption is a critical component in safeguarding the vast amounts of data generated by smart city sensors. These sensors are embedded throughout urban environments to monitor everything from traffic flow and air quality to utility usage and public safety conditions. The data they collect plays a vital role in improving city services, enabling real-time decision-making, and enhancing the quality of life for residents. However, much of this data is sensitive in nature—potentially revealing personal movement patterns, household behavior, or critical infrastructure status—which makes it a prime target for cyber threats. (Zhang et al., 2017) (Al-Turjman et al., 2022)

By encrypting this data, cities can ensure that only authorized parties can access or interpret the information, thereby protecting confidentiality and reducing the risk of privacy breaches or espionage. Furthermore, encryption helps maintain data integrity, preventing attackers from altering sensor outputs in ways that could mislead city systems or decision-makers. This is especially important in applications where false data could lead to dangerous outcomes—such as misrouting traffic during emergencies, disrupting public transportation schedules, causing failures in water or power distribution, or delaying first responders.

Thus, encryption not only builds public trust in smart city technologies but also acts as a frontline defense against cyberattacks that could disrupt essential services or endanger public safety. As smart cities continue to evolve and expand, robust encryption practices will remain a cornerstone of secure and reliable urban infrastructure.

1.2 The Advantages of on-sensor Encryption

In principle, sensor data can either a) be encrypted directly on the sensor and then be forwarded over a potentially unsecured communication channel or b) be send over an encrypted communication channel where encryption between communication parties has been established before sending payload data (e.g. by some sort of a communication handshake). On-sensor encryption provides security at the data source, ensuring protection from the moment data is collected.

In contrast to channel encryption like TLS, which secures data only while in transit between two endpoints, on-sensor encryption maintains data confidentiality even if an attacker gains access to the sensor network. By encrypting data directly on the sensor, the data remains protected until it reaches a trusted processing point, reducing potential vulnerabilities at intermediary stages.

Moreover, on-sensor encryption offers an independent security layer that is protocol-agnostic, providing consistent data protection regardless of the network or communication protocols used. This scalability is particularly beneficial in the complex and dynamic networks typical of smart cities, where data may travel through multiple paths and connections.

2. Objectives and Contribution

The overall objective of our work is to study IoT on-sensor encryption with stream ciphers on resource-contrained hardware. In particular, our goals are to i) implement a concrete research prototype of a stream cipher algorithm, ii) evaluate the general performance of this implementation on resource-contrained hardware with various benchmarking experiments, and iii) produce detailed performance results obtained on actual real-world smart city sensor data.

Our contributions in this work towards the aforementioned goals are:

- We realised software-implementations of the recently published Lightweight Stream Cipher DRACO (Hamann et al., 2022) in two different programing languages (C++ and Rust), targeted at microcontrollers.
- Based on these implementations we conducted experiments on different types of resource-constrained Arduino/Raspberry nodes with random as well as real-word smart city sensor data, yielding numerous results.

To the best of our knowledge, our work presents the first performance analysis of the DRACO stream cipher implemented in Rust and C++ on microcontrollers.

3. Stream Ciphers

3.1 Introduction to Stream Ciphers

Stream ciphers are encryption algorithms designed to encrypt data one bit at a time, ideal for applications requiring real-time data processing. This is in contrast to block ciphers, which encrypt data in fixed-size blocks. Stream ciphers are designed to process a continuous stream of data, making them suitable for scenarios where data arrives in varying sizes or needs real-time encryption.

Notable examples for Stream Ciphers include RC4 (now considered insecure due to vulnerabilities), ChaCha20 (Nir and Langley, 2018), Salsa20 (Bernstein, 2008), SNOW (for high-speed encryption in mobile communications), and lightweight ciphers like Grain (Hell et al., 2007) and Trivium (Cannière and Preneel, 2008).

Stream ciphers are ideal for encrypting smart city sensor data due to their ability to efficiently handle real-time data processing with minimal latency and computational overhead. These ciphers encrypt data on-the-fly, making them perfect for the continuous data streams generated by smart city sensors. Unlike block ciphers, stream ciphers do not require fixed-size blocks or data padding, allowing them to seamlessly handle diverse data sizes and formats typical in a smart city environment. Another often underestimated advantage of stream ciphers is that they – unlike block ciphers – do not require a mode of operation (which entails additional computational overhead and may introduce its own security issues). This flexibility enables security across a network of heterogeneous devices, safeguarding the confidentiality and integrity of information vital for applications ranging from traffic management to environmental monitoring.

3.2 Technical Overview of Stream Cipher Operation

At a technical level, a stream cipher starts with a secret key and often an initialization vector (IV) to set up its internal state. Using this input, it generates a keystream, which is a long, pseudorandom sequence of bits. Encryption is typically performed by applying the bitwise XOR operation between the plaintext and the keystream. The same process is used for decryption: the ciphertext is XORed with the same keystream, which reverses the encryption due to the mathematical property of XOR (i.e., $A \oplus B \oplus B = A$).

Internally, the keystream is produced by a generator mechanism such as a linear feedback shift register (LFSR) or nonlinear feedback shift register (NFSR), sometimes combined with complex Boolean functions or internal permutations. The cipher must be designed so that the keystream appears random and cannot be predicted or reproduced without the key and IV.

A key advantage of stream ciphers is their ability to start encrypting data immediately without waiting for a full block of input, making them ideal for applications with low power, limited bandwidth, or real-time requirements. However, reusing the same keystream (i.e., same key and IV combination) is a critical security risk, as it can allow attackers to recover plaintext through simple XOR operations. Therefore, secure key and IV management is essential in any stream cipher application.

3.3 The DRACO Stream Cipher

DRACO (Hamann et al., 2022) is a lightweight encryption algorithm designed for small, low-power devices like RFID tags and embedded sensors. Its main goal is to provide strong data security while using very little hardware, energy, or memory. This makes it ideal for environments where resources are extremely limited.

DRACO is built to be both efficient and secure. It uses a compact internal system to create a stream of encrypted data, based on a secret key and an initialization value (IV). Unlike many traditional ciphers that require large internal memory to stay secure, DRACO achieves strong protection with a much smaller design—helping save cost and power.

One of DRACO's key strengths is that it has been mathematically proven to resist a broad class of generic attacks when used properly. For instance, it achieves 128-bit resistance against generic time-memory-data tradeoff (TMDTO) attacks and distinguishing attacks within the random oracle model—assuming a single-user or single-session setting. This proof makes DRACO notable among stream ciphers, especially those targeting lightweight applications.

DRACO is well suited for IoT applications because it delivers strong security while using very little power, memory, and hardware resources—key requirements for small, battery-powered or passive devices like sensors, RFID tags, and wearables. Its compact design allows it to run efficiently on low-cost microcontrollers, and its energy efficiency helps extend device lifespan. Additionally, DRACO performs well in simple, one-time communication scenarios typical in IoT, offering robust protection without the complexity or overhead of larger encryption systems.

4. Implementation

4.1 Software Implementations of the DRACO Stream Cipher

We implemented the DRACO Stream Cipher in software in two different programming languages: C++ and Rust. DRACO was originally designed for hardware implementations. However, implementing DRACO in software makes sense for generic resource-constrained devices that lack dedicated cryptographic hardware, such as off-the-shelf IoT sensors and embedded systems. Its lightweight design, with a small internal state and simple operations, allows it to run efficiently on low-power processors without draining battery life. Software implementations are also more flexible and easier to update than hardware, making them ideal for evolving or remotely managed systems. Additionally, DRACO's security model is well-suited for single-session or short-lived communications, which are common in IoT applications, offering strong protection with minimal performance impact.

C++ is widely used in low-level IoT programming because it offers the performance and hardware-level control needed for resource-constrained devices, while also supporting modular and maintainable code through object-oriented features. It allows direct interaction with memory and peripherals, making it ideal for real-time operations. Additionally, C++ is compatible with existing C libraries and toolchains commonly used in embedded systems, ensuring broad support across IoT platforms and development environments.

Rust, on the other hand, offers several advantages over C++, particularly in memory safety, concurrency, and developer productivity. Its unique ownership system enforces strict compile-time checks that prevent common bugs like null pointers and data races, all without needing a garbage collector. Rust also promotes safe concurrency, has cleaner syntax, more consistent language features, and excellent tooling through its "cargo" build system. Unlike C++, Rust avoids undefined behavior by default and provides clearer error messages, making it easier to write reliable, maintainable code—especially for security-critical or low-level applications.

We choose to implement DRACO in both C++ as well as in Rust to be able to evaluate and compare their suitability in terms of performance on microcontrollers in a real-world, resource-constrained setting. By developing parallel implementations, we aimed to gain insight into how each language handles low-level cryptographic operations in terms of execution speed and memory usage. In addition, comparing both implementations also allows to assess not only raw performance but also trade-offs in developer experience, code safety, maintainability, and integration effort—informing decisions about language choice for future IoT and embedded security projects.

4.2 Validation Framework

To ensure the correctness and efficiency of our implementations, we designed a comprehensive test framework. This framework covered a wide range of test cases and edge conditions:

- Validation of core bit manipulation functions: The fundamental operations of DRACO were thoroughly tested to ensure all bitwise operations were implemented accurately and efficiently.
- Consistency tests: Identical inputs were required to always produce identical outputs, guaranteeing the deterministic behavior of the implementation.
- Comparison with predefined test vectors: By precisely matching the generated keystreams against the expected values, we were able to reliably validate the implementation against the reference.

Once we successfully reproduced the exact keystreams for all test vectors, it was clear that the implementation had passed all tests. This confirmed that the core computations functioned correctly and that DRACO operated exactly as intended.

4.3 Optimization and Memory Management in Performance Comparison

Another key objective of our work was optimizing memory management and conducting a comparative performance analysis. For this purpose, we compared our C++ implementation with Rust version of DRACO. It became apparent that the C++ version relied on dynamic memory allocations in certain areas, which were not required in the Rust implementation. As a result, the memory management in the original implementation had to be optimized.

To achieve the most efficient implementation possible, we replaced dynamic memory allocations with statically managed arrays and precisely controlled pointers. These measures reduced memory usage and simultaneously improved execution speed. After these optimizations, we achieved a highly performant and stable C++ implementation of DRACO.

5. Testing Environment, Testing Methodology, and Experimental Setup

5.1 Evaluation Environment

We conducted a multitude of experiments on three microcontrollers: ESP32, ESP8266, and Raspberry Pi Pico. The ESP32 and ESP8266 are Wi-Fi-enabled microcontrollers often used in smart cities for applications like smart street lighting, air quality monitoring, and smart parking, while the Raspberry Pi Pico is usually used for local control tasks and sensor data acquisition, often in combination with other devices like the ESP32 for IoT solutions.

Our experiments measure the runtime performance of our two DRACO implementations (one implemented in C++, one implemented in Rust) on these resource-constrained devices. We benchmarked the performance of these implementations on the different microcontrollers for random data of various sizes as well as for more than 700 real-world smart city sensor measurements of various types (e.g. temperature, power units, volume flow rate) from cooling systems (valves, pumps, refrigeration units).

To compare the Rust and C++ versions, it was necessary to record measurement data for later visualization. Specifically, we captured the execution times for keystream generation and initialization in order to conduct a well-founded performance analysis. These measurements enabled a detailed comparison with the Rust implementation. For this purpose, we used CoolTerm (Meier, 2025), a specialized macOS application that allows serial interfaces to be read and their output saved to a text file. Various performance metrics were recorded, including the time required to generate keystreams and the initialization durations. These text files served as raw data for further processing and analysis.

To make the collected measurements usable for future comparisons and visualizations, we developed a Python script that used the Pandas library (Pandas Development Team, 2020) to convert the text files into structured CSV files.

5.2 Testing Methodology

After implementing DRACO in Rust and C++ on three microcontrollers—ESP32, ESP8266, and Raspberry Pi Pico—a standardized testing procedure was developed to enable a comparable analysis of performance across both programming languages.

To ensure consistent data collection, specific test scenarios were defined and executed identically on all platforms. The results of the individual measurements were automatically saved in CSV files to allow for subsequent statistical evaluation.

For data analysis, Python with the Matplotlib library (Hunter, 2007) was used to generate box plots and other visualizations. Additionally, detailed evaluations were carried out in Excel to examine the measurement results in terms of mean values, standard deviation, and other statistical parameters.

This approach ensured a sound basis for comparing the implementations by analyzing not only overall runtime differences but also potential variations and patterns within the measurement data.

5.3 Experimental Setup and Benchmarks Definition

To ensure a statistically sound analysis, each test case used various combinations of random and static keys as well as random and static initialization vectors. This allowed us to examine whether and to what extent these parameters influenced the performance of the implementation.

Each test series was repeated 100 times to minimize the impact of outlier measurements and to create a reliable data set. Tests were conducted with keystream lengths of 1, 8, 128, and 1024 bits to analyze how different output lengths affected the performance of the implementation.

To further improve the validity of the results, the initial test runs were excluded from the analysis, as they were identified as potential outliers that could be distorted by initialization processes or other system-related effects.

The benchmarks were conducted in three categories:

- **Full**: This benchmark measures the time required for complete keystream generation followed by data encryption. In this process, the generated keystream is combined with the original message using an XOR operation and stored. This simulates a typical use case of DRACO for encryption.
- **Keystream_gen**: This test only measures the time taken to generate the keystream, without performing any subsequent encryption. It helps analyze the raw performance of keystream generation and identify speed differences across various microcontrollers and programming languages.
- **Struct_init**: This benchmark measures the initialization time of the DRACO algorithm—that is, the time needed to set up the internal registers and structures before any keystream generation or encryption can occur.

However, the `keystream_gen` benchmark was ultimately used only to validate the results of the full benchmark. A detailed evaluation of the `keystream_gen` data was not performed, as the results were nearly identical to those of the full benchmark. Thus, a separate analysis of keystream_gen would not have provided any additional insights.

6. Results

In the following section, we present and analyze the results of our benchmarking experiments. These results provide insights into the performance of our DRACO implementations across different configurations, platforms, and programming languages. By comparing execution times for initialization and keystream generation, we aim to highlight performance trends, identify bottlenecks, and draw conclusions about the efficiency and suitability of each implementation for resource-constrained environments.

For interpreting the graphs, the following structure should be noted: the labels always follow the format `<Key, IV>`. The number on the X-axis indicates the keystream length in bits. For example, the label `<r, s | 8>` represents a random key with a static IV and a keystream length of 8 bits[1]. The Y-axis always shows the time in microseconds, allowing for a performance comparison between the different configurations. If not specifically noted, the graphs present the results of the "`Full`" benchmark, i.e. the time required for complete keystream generation followed by data encryption. For all our experiments, a very low standard-deviation was observed (i.e. less than 1%, not displayed in the figures).

6.1 Experimental Results on the ESP32 Microcontroller

Figure 1 shows the runtime performance (in µs) of the DRACO stream cipher on an ESP32 microcontroller for various input data sizes and different combinations of random/static key and random/static initialization vector (IV), comparing our two implementations (C++, Rust).

With respect to keystream length, C++ and Rust exhibit similar behavior. A direct comparison of the two languages shows that Rust is approximately twice as fast as C++ when generating a keystream of 1 bit. At 4 bits, the execution times begin to converge, and from 128 bits onward, C++ becomes insignificantly faster than Rust.

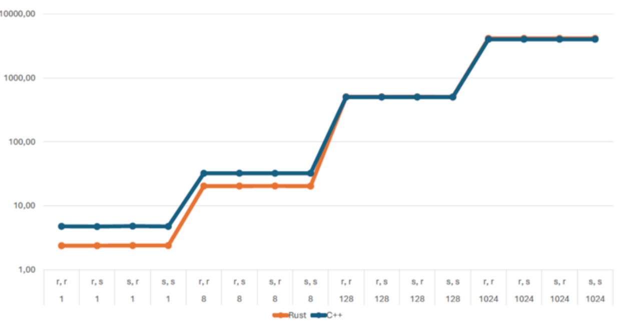

Figure 1: Runtime performance of the DRACO stream cipher on an ESP32 microcontroller for various input data sizes (in bits) and different combinations of random/static key and random/static initialization vector (µs)

[1] In other words, the numbers on the x-axis display the number of input bits to the stream cipher, where the [r/s],[r/s] notation above the input bit size indicates whether the key and/or the IV are random (r) or static (s) for these results.

Figure 2 shows the time need to inizialize the DRACO algorithm (benchmark “`Struct_init`”) on an ESP32 microcontroller. When comparing the four variants of random/static key and random/static IV for initialization on the ESP32, it becomes clear that the execution times remain consistent within the same programming language—regardless of the chosen configuration. The C++ implementation is consistently slightly faster than the Rust version throughout.

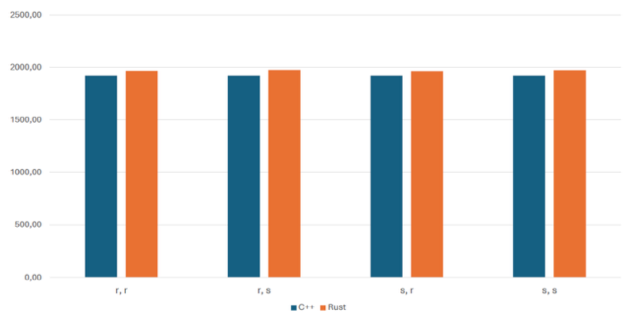

Figure 2: Initialization performance of the DRACO stream cipher on an ESP32 microcontroller for different combinations of random/static key and random/static initialization vector (µs)

6.2 Experimental Results on the ESP8266 Microcontroller

Figure 3 shows the runtime performance (in µs) of the DRACO stream cipher on an ESP8266 microcontroller for various input data sizes and different combinations of random/static key and random/static initialization vector (IV), comparing our two implementations (C++, Rust).

For the execution time of the full benchmark on the ESP8266, we observe that the differences between the two implementations are minimal across the entire range of keystream lengths. However, outliers appear in the Rust implementation at 1-bit keystream length, particularly in the configurations with a random key & static IV and static key & random IV. The exact cause of this behavior could not be clearly determined, but it is likely related to the limited compiler support for the ESP8266 in the Rust ecosystem.

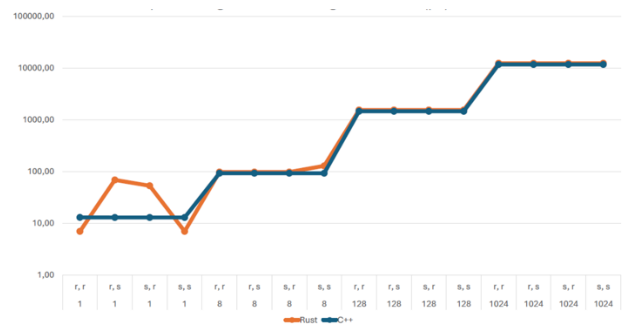

Figure 3: Runtime performance of the DRACO stream cipher on an ESP8266 microcontroller for various input data sizes (in bits) and different combinations of random/static key and random/static initialization vector (µs)

Figure 4 shows the time need to inizialize the DRACO algorithm (benchmark “`Struct_init`”) on an ESP32 microcontroller. Initialization on the ESP8266 also shows that the different combinations of key and IV result in the same execution time within each respective implementation. However, the difference between C++ and Rust is more pronounced in this case. Compared to the ESP32, the overall execution times are significantly higher, reflecting the lower performance capabilities of the ESP8266.

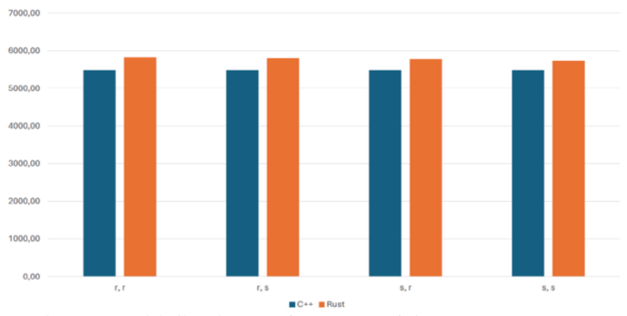

Figure 4: Initialization performance of the DRACO stream cipher on an ESP8266 microcontroller for different combinations of random/static key and random/static initialization vector (µs)

6.3 Experimental Results on the Raspberry Pi Pico Microcontroller

Figure 5 shows the runtime performance (in µs) of the DRACO stream cipher on an Raspberry Pi Pico microcontroller for various input data sizes and different combinations of random/static key and random/static initialization vector (IV), comparing our two implementations (C++, Rust). A similar pattern in execution times can also be observed on the Raspberry Pi Pico. Rust is slightly faster than C++ up to a keystream length of 8 bits. After that, the two implementations begin to converge, and at 1024 bits, C++ ultimately becomes faster.

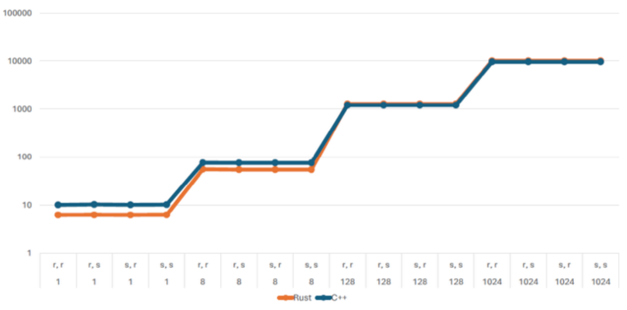

Figure 5: Runtime performance of the DRACO stream cipher on an Raspberry Pi Pico microcontroller for various input data sizes (in bits) and different combinations of random/static key and random/static initialization vector (µs)

6.4 Benchmarks with Smart City Data Sets from the iCity Project

The iCity: Intelligent City initiative by HFT Stuttgart[2], launched in 2017, is a multidisciplinary research partnership aimed at developing holistic solutions for a livable, intelligent, and sustainable urban future. It brings together over 50 researchers and 45 industry, municipal, and SME partners to create

[2] https://www.hft-stuttgart.com/research/projects/i-city

innovative methods, services, and products across domains such as energy management, mobility, data platforms, and urban infrastructure. The project places a strong emphasis on systemic integration—not only technologically, but also socially and ecologically—ensuring citizen well-being remains central to smart city development. Current work focuses on advancing digital urban twins, open urban data platforms, and 5G-supported IoT infrastructure, aiming to bridge academic research and practical deployment while supporting open-source adoption and data-driven decision-making.

In addition to the experiments using randomly generated payload data (see sections 6.1, 6.2, 6.3), we also conducted tests with real sensor data sourced from HFT Stuttgart's iCity project, specifically from building cooling systems. This dataset includes measurements from valves, pumps, and refrigeration units, capturing parameters such as temperature, power consumption, and volume flow rate. By incorporating this operational data into our benchmarking framework, we were able to evaluate the performance and reliability of the DRACO cipher in a realistic context, emulating the encryption demands of smart city infrastructure systems and ensuring its practical suitability for securing critical sensor communications.

The benchmarks in this experiment were executed on the ESP8266 microcontroller, operating at a clock frequency of 80 MHz. For this test, the Rust implementation of the DRACO cipher was used. A keystream of 22,464 bits was generated, corresponding to 702 individual raw sensor measurement data values of 32 bits each, reflecting the size of the real-world sensor dataset. The full benchmark was performed, meaning that both the keystream generation and the encryption of the dataset using a bitwise XOR operation were included in the measurement.

This setup was chosen to simulate a realistic encryption workload based on actual sensor data from smart city cooling systems. In a typical deployment scenario, a sink node—such as a building controller or edge gateway—would collect these individual measurements from distributed cooling system sensors and perform on-device encryption before forwarding the data to a central server, ensuring confidentiality from the moment of acquisition.

A total of 400 benchmark runs were conducted for each dataset—iCity sensor data and random payloads—using the Rust implementation of DRACO on the ESP8266 at 80 MHz. Both benchmarks were run under identical conditions, with the only difference being the type of payload: real sensor data from iCity cooling systems vs. randomly generated data. These runs were evenly distributed across four key/IV configurations: random key & random IV (r, r), random key & static IV (r, s), static key & random IV (s, r), static key & static IV (s, s). In other words, each configuration was tested 100 times, ensuring consistent and comprehensive coverage across different initialization scenarios. This setup allowed us to evaluate whether key and IV variability influenced encryption performance under realistic and synthetic conditions.

Table 1 summarizes the overall execution time statistics based on all 400 benchmark runs for each dataset (i.e. for encrypting iCity sensor data versus random data) using the DRACO cipher, regardless of the specific key/IV configuration (i.e., whether random or static keys and IVs were used). By aggregating results across all four combinations (r,r; r,s; s,r; s,s), the table provides a global performance comparison between the two data types. This approach offers a broad view of how the nature of the payload—real-world sensor data vs. synthetic random data—affects execution time, independent of initialization settings.

Metric	iCity Data	Random Data
Mean (µs)	282,891.4	282,871.5
Median (µs)	282,897.0	282,870.0
Standard Deviation	31.82	25.60
Min (µs)	282,820	282,790
Max (µs)	283,287	282,977

Table 1: Average execution times (µs) and standard deviation for the DRACO full benchmark, comparing iCity sensor data and random payloads on an ESP8266 microcontroller

Both datasets yield nearly identical mean and median execution times—around 282,890 µs for iCity data and 282,870 µs for random data—indicating that the type of data has negligible impact on performance. The standard deviation is slightly (albeit insignificantly) higher for the iCity data, suggesting marginally more variability, possibly due to subtle differences in memory access patterns. Overall, the data confirms that DRACO delivers consistent and stable performance regardless of whether the payload is synthetic or from a real-world smart city source.

Table 2 presents in more detail the average execution times (in microseconds) for the DRACO full benchmark with respect to the four different key/IV configurations, using both iCity sensor data and random payloads on the ESP8266 (80 MHz) with the Rust implementation.

randomKey YesNo	randomIV YesNo	iCity Mean (¬µs)	Random Mean (¬µs)
False	False	282908.82	282870.31
False	True	282887.76	282883.78
True	False	282885.3	282881.53
True	True	282883.72	282850.21

Table 2: Average execution times (µs) for the DRACO full benchmark across four different key/IV configurations, comparing iCity sensor data and random payloads on an ESP8266 microcontroller

It can be observed that across all configurations, the execution time differences between iCity sensor data and random payloads are consistently small. This implies that the type of data being encrypted—whether real-world sensor data (iCity) or randomly generated payloads—has little to no impact on DRACO's performance, affirming its reliability for real-world smart city workloads.

Also, note that across all four key and IV configurations the execution times for the iCity and random datasets are very close, with differences ranging from ~4 to ~39 microseconds. This implies that DRACO's performance is largely unaffected by whether the key or IV is static or randomly generated.

6.5 Summary of Results and Discussion

In summary, our C++ Implementation is slightly faster than our Rust implementation, starting at 128-bit keystream lengths.

Also, C++ has good compiler optimizations, resulting in stable performance.

The Rust implementation, on the other hand, was faster in our experiments for short keystream lengths (1 and 8 bits), except on the ESP8266. However, Rust requires platform-specific code for each microcontroller. Also, there was less programming language support for Rust for microcontrollers, especially for the ESP8266. But Rust offers stronger safety guarantees through the Rust compiler.

Overall, both languages are well-suited for implementing the cipher. While there are some small performance differences, these are generally minor and unlikely to be significant in practice. The biggest distinction is that implementing Rust across different microcontrollers is more complex than C++, but this added complexity may be worthwhile due to the additional safety guarantees Rust provides.

7. Conclusion and Future Work

In this work, we implemented and evaluated the DRACO stream cipher in both C++ and Rust on three widely used microcontroller platforms: ESP32, ESP8266, and Raspberry Pi Pico. Our goal was to analyze the correctness, efficiency, and resource usage of both implementations, particularly in the context of lightweight and resource-constrained IoT environments.

Through a standardized testing framework and a carefully designed benchmarking setup, we measured the performance of keystream generation, encryption, and initialization under various key and IV configurations. Our results demonstrate that both implementations perform reliably and efficiently, with minor variations depending on the target platform and language.

In addition to experiments conducted with randomly generated payload data, we also performed tests using real-world sensor data obtained from HFT Stuttgart's iCity project. This data, collected from deployed urban sensors monitoring cooling systems allowed us to evaluate the performance and behavior of the DRACO cipher with raw measurement data from real sensors. By integrating actual smart city data into our test framework, we ensured that the encryption performance measurements reflect practical usage scenarios, helping to validate the suitability of DRACO for real-time data protection in urban IoT deployments.

Overall, our study highlights the feasibility of DRACO as a lightweight encryption solution for smart cities. The results demonstrate that the technology is mature and efficient enough for immediate, affordable deployment at city scale, enabling secure sensor data transmission in smart urban infrastructures without significant cost or performance trade-offs.

The on-sensor stream-cipher encryption approach demonstrated in this work is well-suited for low- to mid-power sensors and edge devices with processing capabilities comparable to the ESP8266 or ESP32 microcontroller. This includes a wide range of sensors commonly found in smart city environments, such as those used for measuring temperature, humidity, air quality, pressure, and flow within infrastructure systems like HVAC and utility networks. It is also applicable to building automation devices, smart meters, traffic sensors, and environmental monitoring units deployed across urban infrastructure. These devices typically operate on constrained hardware with limited memory and compute resources, yet the results show that efficient, on-device encryption with DRACO is feasible and practical even under these conditions. This confirms the suitability of the approach for scalable, secure data transmission in real-world smart city applications.

Further, our work provides practical insights into the trade-offs between C++ and Rust for cryptographic applications in embedded systems. Our results show that both C++ and Rust are well-suited for implementing the DRACO stream cipher on microcontrollers. Although some minor performance differences were observed, they are generally negligible and unlikely to impact practical applications. The most notable distinction lies in the implementation effort: while Rust introduces greater complexity when targeting different microcontroller platforms, it also offers stronger safety guarantees through its strict compile-time checks. This trade-off may justify the added development effort, particularly in security-critical or safety-sensitive environments.

While this study provides a strong foundation for evaluating the DRACO stream cipher on embedded platforms using both C++ and Rust, several directions remain open for future exploration. One key area is extending the benchmarks to a broader range of microcontrollers and hardware architectures, such as ARM Cortex-M0+ or RISC-V, to further assess performance and portability. Additionally, future work could involve integrating DRACO into real-world applications—particularly within the context of smart cities, where lightweight encryption is crucial for protecting sensitive, distributed data streams.

Smart city infrastructures rely heavily on sensor data for decision-making in areas like traffic management, public safety, environmental monitoring, and utility control. This data is often real-time, continuous, and privacy-sensitive, making efficient and secure encryption essential. Evaluating DRACO's performance in such settings could reveal its strengths and limitations under real-world constraints, including network latency, packet loss, and power consumption.

From a software engineering perspective, further investigation into compiler optimizations and memory usage in both languages could help fine-tune the cipher for deployment in ultra-constrained devices. Exploring hardware-accelerated implementations, or embedding DRACO into existing cryptographic libraries for IoT ecosystems, could also improve integration and adoption. Finally, analyzing long-term reliability, security resilience in multi-session or multi-user environments, and the maintainability of the C++ and Rust codebases will be important for ensuring the cipher's viability in large-scale, mission-critical smart city deployments.

Acknowledgment

This work is an outcome of the project "Datasecurity4iCity", a subproject of "iCity: Intelligent City" (https://www.hft-stuttgart.com/research/projects/i-city). We extend our gratitude for the funding received through the FHImpuls program under the number 13FH9E04IA by the German Federal Ministry of Education and Research (BMBF).

References

Al-Turjman, F., Zahmatkesh, H. and Shahroze, R., 2022. An overview of security and privacy in smart cities' IoT communications. *Trans. on Emerging Telecommunications Technologies*, 33(3). https://doi.org/10.1002/ett.3677

Bernstein, D.J., 2008. The Salsa20 family of stream ciphers. In: Biryukov, A. and Wagner, D. (eds), New Stream Cipher Designs: The eSTREAM Finalists. Springer, Berlin, Heidelberg, pp. 84–97. https://doi.org/10.1007/978-3-540-68351-3_8

Cannière, C. and Preneel, B., 2008. Trivium. In: Biryukov, A. and Wagner, D. (eds), New Stream Cipher Designs: The eSTREAM Finalists. Springer, Berlin, Heidelberg, pp. 244–266. https://doi.org/10.1007/978-3-540-68351-3_18

Hamann, M., Moch, A., Krause, M., & Mikhalev, V. 2022: The DRACO Stream Cipher: A Power-efficient Small-state Stream Cipher with Full Provable Security against TMDTO Attacks. *IACR Transactions on Symmetric Cryptology*, 2022(2), 1-42. https://doi.org/10.46586/tosc.v2022.i2.1-42

Hell, M., Johansson, T. and Meier, W., 2007. Grain: A stream cipher for constrained environments. International Journal of Wireless and Mobile Computing, 2(1), pp.86–93. https://doi.org/10.1504/IJWMC.2007.013798

Hunter, J.D., 2007. Matplotlib: A 2D graphics environment. Computing in Science & Engineering, 9(3), pp.90–95. https://doi.org/10.1109/MCSE.2007.55

Meier, R., CoolTerm Help – Roger Meier's Freeware. Available at: https://freeware.the-meiers.org/CoolTermHelp/ (Accessed: 30 June 2025).

Nir, Y. and Langley, A., 2018. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439. https://doi.org/10.17487/RFC8439

Pandas Development Team, T., 2020. pandas-dev/pandas: Pandas (latest) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.3509134

Zhang, K., Ni, J., Yang, K., Liang, X., Ren, J. and Shen, X.S., 2017. Security and privacy in smart city applications: Challenges and solutions. *IEEE Comm. Magazine*, 55(1), pp.122–129. https://doi.org/10.1109/MCOM.2017.1600267CM