

Canonical Byte-String Encoding for Finite-Ring Cryptosystems^{*}

Kyrylo Riabov^{1,*} and Serhii Kryvyi^{1,†}

¹ Taras Shevchenko National University of Kyiv, Faculty of Computer Science and Cybernetics, 4d Akademika Glushkova Ave., Kyiv, 03680, Ukraine

Abstract

Ring-mapping cryptosystems need a canonical procedure that converts ordinary byte strings into valid residues before any algebraic encryption step can begin. This paper isolates that software representation layer and presents the base- m -len codec, a canonical map from byte strings of length less than 2^w to lists of residues modulo m . The construction uses two fixed-width base- m headers, for the byte length and the final normalized state, together with a uniform payload transduction inspired by the normalized-state viewpoint of ANS and rANS. The resulting format is deterministic, self-delimiting, and parameterized by the target modulus, which makes it suitable as a transport layer between byte-oriented software interfaces and finite-ring protocols rather than as an entropy coder. Its length-first wire format keeps the abstract specification close to the implementation interface: the decoder can recover the declared output size before processing the remaining payload stream, while the encoder remains linear in the emitted digit stream. For every supported modulus, decoding reconstructs the original bytes exactly; because the byte length is encoded explicitly, decoding also tolerates appended valid suffix digits and stops after the declared payload has been recovered. The paper links the abstract codec to a companion Rust implementation and a Lean 4 formalization with machine-checked proofs. The formal development establishes fixed-width header inversion, normalization-window bounds on the stored payload state, stream-level roundtrip correctness, and validity of every emitted residue. We also analyze asymptotic representation cost, report representative native-code throughput, and discuss practical limits such as exposed message length, implementation-level allocation bounds, and the distinction between deterministic serialization and cryptographic protection.

Keywords

finite-ring cryptography, message representation, base- m encoding, formal verification

1. Introduction

Finite-ring cryptosystems such as the ring-image construction in [1] require messages to be represented as elements of $\mathbb{Z}_m$ before encryption or decryption. Existing examples use hand-built alphabets and small fixed symbol tables, which illustrate the algebra but leave open the general problem of how arbitrary UTF-8 text [2] or binary data should be represented before the ring-level encryption step begins.

This representation problem extends beyond the mentioned papers. Any protocol that operates on plaintext data over $\mathbb{Z}_m$ must first represent that data as residues in $\mathbb{Z}_m$ before applying its core transformations. Canonical byte-level encodings are standard in other settings, including base- N transfer encodings [3], ASN.1 canonical and distinguished encoding rules [4], deterministic CBOR-based interchange [5], and public-key standards with explicit octet-string and integer conversion rules [6]. Those formats solve adjacent interoperability problems, but they do not provide a modulus-parameterized residue stream tailored to protocols that natively operate over $\mathbb{Z}_m$. This aspect is also reflected in the ERC project SYMPZON [7], which studies symmetric primitives over integer rings

2nd Workshop "Intelligent information technologies" UkrProg-IIT'2026 co-located with 16th International Scientific and Practical Programming Conference UkrPROG'2026, May 5-6, 2026, Kyiv, Ukraine

^{*} Corresponding author.

[†] These authors contributed equally.

✉ kyryl.ryabov@gmail.com (K. Riabov); sl.krivoi@gmail.com (S. Kryvyi)

ORCID 0009-0003-4118-8492 (K. Riabov); 0000-0003-4231-0691 (S. Kryvyi)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

for applications such as format-preserving encryption, multi-party computation, fully homomorphic encryption, and zero-knowledge. The present work provides a concrete representation layer that can support such constructions.

The problem addressed in this paper also is an interoperability problem at the boundary between byte-oriented software systems and algebraic cryptographic protocols. Real applications exchange UTF-8 strings, binary protocol frames, serialized objects, files, and network messages as byte strings, whereas finite-ring cryptographic constructions operate on residues modulo a fixed integer. Without a canonical byte-to-residue representation, two implementations of the same protocol may disagree about byte order, message length, padding, invalid symbols, or decoding termination. The proposed base- m -len codec therefore serves as a deterministic transport representation layer for software implementations of finite-ring protocols.

This paper defines a canonical byte-to-residue codec for finite-ring cryptosystems. The preferred base- m -len construction maps each byte string of length less than 2^w to a residue stream in $\mathbb{Z}_m$. Its wire format consists of three components: a fixed-width base- m encoding of the byte length and the final payload state, and a payload stream produced by a uniform base- m transduction. The decoder reconstructs the original bytes exactly and stops after the declared length, so any additional valid suffix digits are ignored.

The present construction is not an entropy coder. Its asymptotic payload cost reflects a change of output alphabet rather than a reduction in information content; larger moduli may reduce the number of transmitted residue symbols, but bit-level compression would require an explicit probabilistic source model and lies outside the scope of this paper. The payload mechanism follows the normalized-state perspective of ANS and rANS, but is used here to obtain a deterministic, self-delimiting residue format rather than a new entropy coder or security primitive [8], [9]. The problem addressed is orthogonal to ring-based lattice cryptography such as NTRU and ML-KEM [10], [11], which rely on different assumptions and target different protocol goals.

The paper makes four contributions.

1. It defines a canonical, modulus-parameterized byte-to-residue codec for finite-ring protocols, centered on the base- m -len wire format.
2. It establishes exact decoding, suffix tolerance, and fixed-width header inversion for supported moduli and byte strings of length less than 2^w .
3. It provides an implementation-oriented representation layer for finite-ring cryptographic software: the codec exposes byte-level Rust entry points, produces residue streams suitable for protocol processing, and is accompanied by a Lean 4 specification that formalizes the intended behavior of the software interface.
4. It evaluates the native implementation by measuring encoding and decoding throughput, comparing it with exact radix-conversion baselines, and reporting the practical trade-off between modulus size, stream length, and runtime performance.

The remainder of the paper formalizes the model and proof scope, specifies the codec, states the main guarantees, and evaluates its complexity and practical performance.

2. Model, implementation scope, and Lean 4 formalization

2.1. Core definitions and boundaries

Throughout the abstract development, we fix a global word width $w \in \mathbb{N}$ and study those moduli m for which the codec parameters are well formed at that width. The evaluation in Section 5 later instantiates this generic theory at $w = 64$.

Definition (Supported modulus).

For the fixed word width w , a *supported modulus* is a modulus m for which the abstract codec parameters are well formed, namely

$$2 \leq m \quad \text{and} \quad 256 \leq \text{decoderLowerBound}(w, m).$$

This definition is formalized in [Params.lean, L51–L55].

Definition (Admissible byte string).

An *admissible byte string* is a finite sequence of bytes of length less than 2^w . We write

$$\text{ByteString}_w = \{\text{bytes} \in \{0,1,\dots,255\}^* \mid |\text{bytes}| < 2^w\}.$$

This definition is formalized in [Spec.lean, L17–L19].

For each supported modulus m , the abstract codec interface is

$\text{Encode}_m: \text{ByteString}_w \rightarrow \text{List}(\text{Fin}(m))$, $\text{Decode}_m: \text{List}(\text{Fin}(m)) \rightarrow \text{Except}(\text{Error}, \text{ByteString}_w)$.

The encoder produces valid base- m digits only, and the decoder either reconstructs an admissible byte string or reports failure. The abstract encoder and decoder are formalized in [Spec.lean, L146–L182], while the fact that every emitted symbol lies in $[0, m)$ is recorded in [Stream.lean, L27–L33].

The core model treats residue digits as elements of $\text{Fin}(m)$ from the outset, so malformed raw naturals outside the residue range do not appear inside the proved interface; see [Params.lean, L32–L34] for the digit type and [Spec.lean, L184–L239] for the boundary validators on plain numeric inputs. The zero-length case is explicit as well: if the length prefix decodes to 0, the decoder returns the empty byte string immediately, as in [Spec.lean, L159–L169]. The corresponding zero-length decoding lemmas appear in [Stream.lean, L135–L172].

2.2. Implementation and proof scope

The public artifacts consist of a Rust library and a Lean 4 formalization with machine-checked proofs. Both are released in the same project repository [12]. On the Rust side, the main byte-oriented entry points are `encode_bytes_base_m_len` and `decode_bytes_base_m_len`, with UTF-8 frontends above them. On the proof side, Lean formalizes the abstract codec, including supported parameters, fixed-width prefixes, payload transduction, and end-to-end stream properties in Lean 4 [13]; the trusted-kernel and elaboration lineage behind the environment is described in [14]. The current development is width-parametric rather than tied to one machine word size, and the repository includes concrete 64-, 128-, and 256-bit instantiations and smoke theorems in [Examples.lean, L58–L70].

The proof scope is intentionally limited to the abstract specification boundary. It does *not* verify the Rust implementation line by line. Parser hardening, allocation limits, low-level I/O, and other engineering details therefore remain implementation concerns rather than machine-checked theorems.

3. The Base- m -len codec

3.1. Fixed-width base- m headers

Definition (Prefix width).

For the fixed word width w and a supported modulus m , let

$$k = \min\{j \in \mathbb{N}_0 : m^j \geq 2^w\}.$$

The fixed-width prefix encoder EncodePrefix_m represents every $x < 2^w$ as exactly k little-endian base- m digits.

The same width k is used for both fixed prefixes in the wire format: the byte length and the final payload state. In the Lean 4 formalization, the definition of this width and its least-width property appear in [Params.lean, L123–L157], with bundled forms for parameter packages in [Params.lean, L215–L271].

3.2. Uniform base- m payload transduction

Let L denote the decoder lower bound associated with the supported modulus. The payload layer maintains a base- m state and applies the uniform byte update

$$x \leftarrow 256x + b.$$

In the Lean 4 formalization, the bundled definitions of the decoder lower bound L and the encoder threshold appear in [Params.lean, L220–L231], and the encoder-side and decoder-side renormalization procedures together with the payload transducer are defined in [Spec.lean, L81–

L144]. Operationally, the encoder scans the input bytes from right to left. Before each byte update, it emits base- m digits until the current state is below the encoder threshold. After the full reverse scan, the emitted payload digits are exposed in FIFO order, so the decoder can read the final wire format from left to right.

This mechanism is close in spirit to ANS and rANS, but the purpose is different. Classical ANS work studies entropy coding relative to a probabilistic source model. Here the byte update is uniform, the output alphabet is fixed by the target modulus m , and the goal is a deterministic, self-delimiting residue representation that can serve as a canonical message layer for finite-ring protocols [8], [9].

3.3. Wire format and algorithms

Definition (Wire format).

For every admissible byte string $bytes$,

$$\text{Encode}_m(bytes) = len \parallel state \parallel payload,$$

where $len = \text{EncodePrefix}_m(|bytes|)$, $state$ is the fixed-width encoding of the final payload state, and $payload$ is the FIFO base- m digit stream produced by the payload transducer.

In the Lean 4 formalization, this concatenated wire format is the definition of the abstract encoder; see [Spec.lean, L146–L153].

Decoding is staged in the same order. The decoder first reads the length prefix. If that prefix is 0, it returns the empty byte string. Otherwise, it reads the state prefix and then consumes exactly as many payload digits as are needed to reconstruct the declared number of bytes. Any valid digits that remain after those bytes are recovered are left unread. In the Lean 4 formalization, the staged decoder is defined in [Spec.lean, L159–L182]; its immediate zero-length return already appears in [Spec.lean, L159–L169], with dedicated zero-length stream lemmas in [Stream.lean, L135–L172]. The padding-tolerant end-to-end theorem that trailing suffix digits are ignored appears in [Stream.lean, L174–L266].

Algorithm 1. Base- m -len encode.

- Set $k \leftarrow \min\{j \in \mathbb{N}_0 : m^j \geq 2^w\}$.
- Encode $|bytes|$ as k little-endian base- m digits.
- Initialize the payload state x at the decoder lower bound L .
- For byte values b of $bytes$ in reverse order, emit base- m digits while x exceeds the encoder threshold, then set $x \leftarrow 256x + b$.
- Encode the final state x as k little-endian base- m digits.
- Reverse the emitted payload digits and return $len \parallel state \parallel payload$.

Algorithm 2. Base- m -len decode.

- Set $k \leftarrow \min\{j \in \mathbb{N}_0 : m^j \geq 2^w\}$.
- Read the first k digits as the declared byte length ℓ .
- If $\ell = 0$, return $\text{ok}([])$.
- Read the next k digits as the initial payload state x .
- For $i = 1$ to ℓ , output $x \bmod 256$, set $x \leftarrow \lfloor x/256 \rfloor$, and while x is below the decoder lower bound consume the next payload digit d and set $x \leftarrow xm + d$.
- Ignore any remaining trailing suffix and return the reconstructed bytes.

In the Lean 4 formalization, Algorithms 1 and 2 correspond directly to the definitions in [Spec.lean, L81–L182].

4. Correctness guarantees

The proof development is split across six Lean files, from parameters and fixed-width prefixes to renormalization, payload inversion, and stream-level correctness.

4.1. Main theorems

Theorem (Stream correctness).

For every supported modulus m and every admissible byte string $bytes \in \text{ByteString}_w$,

$$\text{Decode}_m(\text{Encode}_m(bytes)) = \text{ok}(bytes).$$

Moreover, for every valid suffix $t \in \text{List}(\text{Fin}(m))$,

$$\text{Decode}_m(\text{Encode}_m(bytes) \parallel t) = \text{ok}(bytes).$$

Proof. In the Lean 4 formalization, the suffix-tolerant theorem is [Stream.lean, L174–L254]. Its empty-suffix specialization is [Stream.lean, L256–L266]. The proof uses the header-parsing lemmas in [Stream.lean, L35–L133]. The zero-length lemmas appear in [Stream.lean, L135–L172], and the payload roundtrip theorem in [Payload.lean, L79–L141]. It first recovers the declared length prefix, then the stored state prefix, and finally applies the payload roundtrip theorem to the remaining suffix. The extra-suffix clause follows because decoding stops after reconstructing the declared number of bytes and therefore leaves any remaining valid suffix unread.

Theorem (Header correctness).

For every supported modulus m , let

$$k = \min\{j \in \mathbb{N}_0 : m^j \geq 2^w\}.$$

Then

$$m^k \geq 2^w \quad \text{and} \quad \forall j < k, m^j < 2^w.$$

Moreover, for every $x < 2^w$,

$$\text{DecodePrefix}_m(\text{EncodePrefix}_m(x)) = \text{some}(x).$$

Proof. In the Lean 4 formalization, the least-width property is formalized in [Params.lean, L123–L157], with the bundled statement in [Params.lean, L260–L271]; fixed-width prefix inversion is proved in [Prefix.lean, L65–L89]. Minimality is immediate from the definition of k as the least exponent with $m^k \geq 2^w$. Once that bound is known, every $x < 2^w$ lies in the domain of the fixed-width base- m prefix decoder.

Proposition (Payload-state bounds).

For every supported modulus m and every admissible byte string $bytes$, the payload encoder returns a state in the normalization window

$$L \leq \text{state}(bytes) < Lm.$$

In particular,

$$\text{state}(bytes) < 2^w.$$

Proof. In the Lean 4 formalization, the normalization-window bound and its 2^w corollary are proved in [Payload.lean, L17–L77]. The first part is the inductive invariant of the payload transducer; the second follows from the global estimate $Lm \leq 2^w - 1$.

4.2. Proof idea and traceability

The end-to-end proof has three layers. In the Lean 4 formalization, fixed-width prefix inversion appears in [Prefix.lean, L30–L89], renormalization roundtrip in [Renorm.lean, L219–L294], payload roundtrip in [Payload.lean, L79–L141], and the final stream theorem in [Stream.lean, L174–L266], with the zero-length branch in [Stream.lean, L135–L172]. Fixed-width base- m headers invert because the chosen width covers the bounded w -bit range; renormalization blocks roundtrip the state while preserving untouched suffixes; and a byte-level induction lifts that result to the payload and then to the full stream theorem.

5. Evaluation

The evaluation addresses two questions: what the representation costs in theory, and whether the preferred streaming construction is practical in native code. Throughout this section, we instantiate the general theory at $w = 64$, so the reported header widths and thresholds are the concrete 64-bit values of the generic formulas from the preceding sections.

5.1. Complexity and representation cost

Let $n = |\text{bytes}|$ and let $k = \lceil \log_m 2^{64} \rceil$. The two fixed-width headers contribute exactly $2k$ digits. The payload contributes a number of digits proportional to the byte length, with asymptotic rate

$$\log_m 256 = \frac{8}{\log_2 m}$$

digits per byte. In the Table 1 we summarize the representative points of this trade-off over the range $25 \leq m \leq 257$: larger moduli reduce the payload rate smoothly, while the fixed header width drops only when k crosses an integer threshold. Since each emitted digit is produced once and each consumed digit is read once, encoding and decoding are linear in the stream length. For every fixed supported modulus m , this yields $O(n)$ time and $O(1)$ working state beyond the input and output buffers.

Table 1

Asymptotic payload cost and fixed header size for representative moduli in the $w = 64$ instance.

Modulus m	Payload digits per byte	Header digits $2k$
25	1.723	28
50	1.417	24
257	0.9993	16

5.2. Experiment setup and native performance

The experiments evaluate the Rust implementation of the base- m -len codec as a software component. We measured encoding and decoding throughput in MiB/s, so higher values indicate better performance. Benchmark inputs were deterministic pseudo-random byte arrays of fixed sizes, and each workload was measured separately for encoding and decoding. The native implementation was compared with an exact whole-payload radix-conversion baseline implemented with arbitrary-precision arithmetic. This baseline represents a straightforward implementation strategy for converting a byte string into a residue stream, but it does not use the streaming normalized-state mechanism proposed in this paper. The ANS reference is included only as contextual throughput information; it does not implement the same length-delimited residue wire format and is therefore not a semantic replacement for base- m -len.

The companion Rust code includes Criterion benchmarks for the preferred codec, exact BigUint radix-conversion baselines, UTF-8 smoke tests, and the external ANS reference. The checked benchmark harness covers byte transduction at moduli 25, 50, 257, which are the same representative moduli used in Table 1. The representative measurements below were recorded on an Apple M3 Pro machine running macOS 26.5 with rustc 1.94.0; the benchmark command is `cargo bench -p sym-adv-encoding --bench base_m_len`. The table reports median throughput from Criterion’s stored estimates and the speedup of the native implementation over the BigUint baseline for the same operation, input size, and modulus.

Table 2

Representative throughput measurements in MiB/s for $m = 50$. Here BU denotes the BigUint baseline, and the ratio column is Native/BU speedup. The ANS column provides context only; it does not implement the same length-delimited residue wire format.

Op.	Size	m	Native	BU	Ratio	ANS
Encode	1 KiB	50	152.5	24.2	6.29×	612.4
Decode	1 KiB	50	471.9	129.4	3.65×	180.6
Encode	64 KiB	50	169.4	1.177	143.8×	784.6
Decode	64 KiB	50	488.6	3.347	145.9×	176.0

The implementation tests also exercise the software-interface properties that connect the Rust artifact with the formal specification. For moduli 25, 50, and 257, randomized byte strings of lengths

0, 1, 2, 3, 16, 255, 1024, and 65536 are encoded and decoded. The test asserts that every emitted digit is less than m , that decoding the encoded stream returns the original bytes, and that appending 1–32 random valid suffix digits does not change the decoded result.

5.3. Interpretation

The benchmark results show that the proposed codec is practical as a software representation layer. For the measured workloads, the native implementation reaches hundreds of MiB/s for both encoding and decoding, while the exact `BigUint`-based radix-conversion baseline becomes substantially slower on larger inputs. In the 64 KiB measurements at $m = 50$, the native streaming implementation is more than two orders of magnitude faster than the `BigUint` baseline for both encoding and decoding. This supports the main engineering claim of the paper: canonical byte-to-residue conversion can be implemented as a fast streaming component rather than as a costly whole-message arithmetic conversion. The ANS reference gives useful performance context, but it is not a competing format because it lacks the same explicit length prefix, residue-stream interface, and suffix-tolerant decoding semantics.

The experiment measures the cost of the representation layer that a finite-ring protocol implementation would execute before encryption and after decryption. The measurements therefore answer a practical systems question: whether canonical residue serialization can be performed at native-code speeds while preserving deterministic decoding and formal specification traceability.

6. Security role and implementation limits

This codec should be read as deterministic serialization, not as a security primitive in its own right [15]. In practice, the codec is public and deterministic, so confidentiality, integrity, and authenticity must come from the surrounding cryptographic protocol; the preferred wire format also exposes the exact byte length unless the surrounding protocol adds padding. The Lean development proves the abstract specification boundary rather than the Rust implementation, and decoding allocates according to the embedded length, so implementations should enforce a caller-specified maximum decoded length for untrusted inputs.

Accordingly, the specific focus of this paper is the byte-to-residue representation layer itself: a fixed-width, self-delimiting codec with explicit correctness theorems and machine-checked proof traceability.

7. Conclusion

The base- m -len codec isolates the byte-to-residue representation layer needed by finite-ring protocols. Its wire format consists of a fixed-width length prefix, a fixed-width state prefix, and a uniform base- m payload stream. For every fixed word width w , every supported modulus, and every admissible byte string of length less than 2^w , decoding recovers the original bytes and is tolerant of any valid trailing suffix.

The benchmark suite in the project repository and the Lean 4 formalization [12] together indicate that the construction is both practical and mechanically specified. Because the codec outputs canonical residue streams in $\mathbb{Z}_m$, it can serve as a reusable representation layer for finite-ring cryptography, including current work on symmetric cryptography over integer rings [1], [7].

Declaration on Generative AI

During the preparation of this work, the authors used OpenAI Chat GPT 5.4 tool for grammar and spelling checks, writing-style improvement, and formatting assistance. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

- [1] S. Kryvyi, K. Riabov, Symmetric cryptosystem based on ring images, in: Proceedings of the Workshop on Intelligent Information Technologies (UkrProg-IIT 2025), CEUR Workshop Proceedings 4049 (2025). URL: <https://ceur-ws.org/Vol-4049/paper1.pdf>.
- [2] F. Yergeau, UTF-8, a transformation format of ISO 10646, RFC 3629, RFC Editor, 2003. doi:10.17487/RFC3629.
- [3] S. Josefsson, The Base16, Base32, and Base64 data encodings, RFC 4648, RFC Editor, 2006. doi:10.17487/RFC4648.
- [4] International Telecommunication Union, X.690: Information technology – ASN.1 encoding rules: Specification of basic encoding rules (BER), canonical encoding rules (CER) and distinguished encoding rules (DER), 2021. URL: <https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=14779&lang=en>.
- [5] C. Bormann, P. Hoffman, Concise binary object representation (CBOR), RFC 8949, RFC Editor, 2020. doi:10.17487/RFC8949.
- [6] K. Moriarty, B. Kaliski, J. Jonsson, A. Rusch, PKCS #1: RSA cryptography specifications version 2.2, RFC 8017, RFC Editor, 2016. doi:10.17487/RFC8017.
- [7] European Commission, SYMPZON: Getting SYMmetric CryPtography Out of its Comfort ZONE, Horizon Europe Project Fact Sheet 101160608, 2025. URL: <https://cordis.europa.eu/project/id/101160608>.
- [8] J. Duda, “Asymmetric numeral systems: Entropy coding combining speed of Huffman coding with compression rate of arithmetic coding,” arXiv preprint arXiv:1311.2540, 2014, doi:10.48550/arXiv.1311.2540.
- [9] F. Giesen, rANS notes, The ryg blog, 2014. URL: <https://fgiesen.wordpress.com/2014/02/02/rans-notes/>.
- [10] J. Hoffstein, J. Pipher, and J. H. Silverman, “NTRU: A ring-based public key cryptosystem,” in Algorithmic number theory, vol. 1423, in Lecture notes in computer science, vol. 1423., Springer, 1998, pp. 267–288. doi:10.1007/BFb0054868.
- [11] National Institute of Standards and Technology, Post-quantum cryptography standards: FIPS 203 (ML-KEM), FIPS 204 (ML-DSA), FIPS 205 (SLH-DSA), 2024.
- [12] K. Riabov, Phd-symmetric-cryptography: Lean formalization of the base-n-len codec, GitHub repository, 2026. URL: <https://github.com/KyrylR/phd-symmetric-cryptography/blob/8e903222a7df1c4365592e4b16fa752a924b8977/lean/encoding-dr-1>.
- [13] L. de Moura and S. Ullrich, “The lean 4 theorem prover and programming language,” in Automated deduction – CADE 28, in Lecture notes in computer science, vol. 12699. Springer, 2021, pp. 625–635. doi: 10.1007/978-3-030-79876-5_37.
- [14] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, “The lean theorem prover (system description),” in Automated deduction – CADE-25, in Lecture notes in computer science, vol. 9195. Springer, 2015, pp. 378–388.
- [15] C. E. Shannon, “Communication theory of secrecy systems,” Bell System Technical Journal, vol. 28, no. 4, pp. 656–715, 1949, doi:10.1002/j.1538-7305.1949.tb00928.x.