Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: 8 January 2026

N. Sullivan
Cryptography Consulting LLC
C. A. Wood
Cloudflare, Inc.

7 July 2025

Guidelines for Writing Cryptography Specifications
draft-irtf-cfrg-cryptography-specification-02

Abstract

This document provides guidelines and best practices for writing technical specifications for cryptography protocols and primitives, targeting the needs of implementers, researchers, and protocol designers. It highlights the importance of technical specifications and discusses strategies for creating high-quality specifications that cater to the needs of each community, including guidance on representing mathematical operations, security definitions, and threat models.

IRTF

This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF. This document may contain material that has not received review from the research community. The IRTF publishes the results of research and development activities. These results might not be suitable for deployment.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 January 2026.

Table of Contents

1. Introduction 3
2. Goals and Requirements 3
3. Guidelines for Cryptographic Specification Presentation . . . 5
3.1. Simplicity 5
3.2. Precision 6
3.3. Consistency 7
3.3.1. Representing Mathematical Operations 8
4. Guidelines for Cryptography Specification Content 13
4.1. Reusability 13
4.1.1. Build on Existing Specifications 13
4.1.2. Modular Design 14
4.1.3. Clear Interfaces and Abstractions 14
4.1.4. Completeness 14
4.1.5. Documentation and Examples 15
4.2. Defining Security Definitions and Threat Models 16
4.2.1. Defining Security Goals 16
4.2.2. Formalizing Security Definitions 16
4.2.3. Describing the Threat Model 17
4.2.4. Addressing Known Vulnerabilities and Attacks 17
4.2.5. Providing Guidance on Secure Implementation and Deployment 17
5. Catering to Target Audiences 18
5.1. Catering to Implementers 18
5.1.1. Test vectors 18
5.2. Catering to Researchers 19
5.3. Catering to Protocol Designers 20
6. General Recommendations 21
6.1. Encourage Open Communication and Feedback 21
6.2. Seek External Expertise 21
6.3. Recognize and Address Conflicting Interests 22
7. Examples of Well-Written Specifications 22
7.1. ChaCha20 and Poly1305 for IETF Protocols (RFC 8439) . . . 22
7.1.1. Introduction and Overview 22
7.1.2. Algorithm Descriptions 22
7.1.3. Test Vectors 23

7.1.4. Security Considerations 23
7.1.5. IANA Considerations and References 23
7.1.6. Problematic Aspects 23
8. Examples of Specifications That Could Be Improved 23
8.1. Test Vectors 24
8.2. Unnecessary Branching 24
8.3. Compatibility and Modularity 24
9. Conclusion 24
10. Security Considerations 25
11. IANA Considerations 25
12. References 25
12.1. Normative References 25
12.2. Informative References 25
Authors' Addresses 26

1. Introduction

High-quality cryptography specifications are critical for the development and deployment of secure cryptographic protocols. This document provides guidelines for specification writers. The guidelines cover mathematical operations, security definitions, and threat models. They help ensure that specifications are of high quality and useful for their intended audience. Adhering to these guidelines helps ensure that specifications are easier to understand, implement, and analyze, leading to high assurance and interoperable systems.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Goals and Requirements

The primary goal of these guidelines is to help guide the authorship of cryptographic specifications so that they are as useful as possible when creating high-assurance cryptographic software.

Specifications that follow these guidelines should be able to be easily understood, implemented, and analyzed by different audiences, including the engineering community, research community, and standardization community. By addressing the unique needs and expectations of each group, these guidelines aim to:

* Minimize ambiguity and misinterpretations, leading to clearer specifications and more accurate implementations.

* Ensure consistent and correct implementations by providing a clear description of both algorithms and their underlying mathematical foundation.

* Facilitate review and analysis by the research community, allowing for the verification of security properties and the identification of potential vulnerabilities.

* Enable interoperability of implementations of these specifications, promoting collaboration and compatibility between various systems and protocols.

Each of these stakeholder groups contributes something different to the overall process of deploying software:

1. Engineering community: Engineers identify technical problems and build solutions using computing tools. They focus on why problems should be addressed, producing requirements that define the problem and solutions that meet those requirements. Their ultimate goal is to implement and ship software or hardware that effectively tackles these challenges.

2. Research community: Researchers explore the design space of different subject areas and evaluate potential solutions. They develop methods for designing tools and performing experiments to validate hypotheses. This work concentrates on how problems should be solved, creating artifacts that help describe solutions. These may include academic, peer-reviewed papers or software that studies or supports the shipping of software.

3. Standardization community: This group develops technical specifications of protocols that others can implement, analyze, and verify. The specifications capture the details of a solution and serve as a foundation for creating interoperable systems. They ensure the correct implementation of cryptographic algorithms and protocols.

By following these guidelines and addressing the distinct needs of each stakeholder group, authors can create well-structured, informative specifications documents that facilitate the development, analysis, and implementation of high assurance cryptographic solutions.

3. Guidelines for Cryptographic Specification Presentation

Technical specifications do not stand on their own. Their value is derived from their usefulness to the various communities that rely on them. A specification can have amazing content but without the appropriate presentation, it may not be as useful as intended. The guidelines in this section are a baseline set of recommendations for authors to consider when writing a cryptographic specification and are applicable beyond just cryptographic standards and are general good practices for specification writers.

3.1. Simplicity

Complexity is one of the main causes of software bugs. The opposite of complexity is simplicity, which is a key aspect of creating effective cryptography specifications. By striving for simplicity in problem statements, technical content, and presentation, authors can make their documents more accessible to a wider audience, including implementers, researchers, and protocol designers. Simplicity reduces the cognitive load required to understand the specification and minimizes the risk of misinterpretation, which can lead to incorrect implementations and security vulnerabilities.

To achieve simplicity, authors should focus on:

Problem Definition Start by presenting a concise and easily comprehensible description of the problem that the specification aims to solve. Avoid unnecessary jargon and strive to make the problem statement accessible to readers with varying levels of expertise in the field.

Component Breakdown When explaining multi-step cryptographic algorithms or concepts, break them down into smaller, more manageable components. This will make it easier for readers to understand the individual parts and their relationships to one another.

Clear Language Write the specification using clear, concise language, and consistent and broadly understood terminology. Avoid overly technical jargon, and define any terms that may be unfamiliar to some readers.

Focused Scope Keep the specification focused on the primary problem or use case, i.e., avoid feature creep. Avoid introducing unrelated or peripheral topics, as this can create confusion and detract from the primary focus.

By focusing on simplicity in document structure and prose in the specification writing process, authors can create documents that are more accessible and easier to understand, ultimately resulting in more reliable and secure implementations of cryptographic algorithms and protocols. Focusing on simplicity in writing does not imply imprecision or brevity. Even long documents can embody simplicity with the right attention to detail and structuring of prose.

3.2. Precision

Precision is essential in cryptographic specifications, as small deviations or ambiguities can lead to severe security vulnerabilities. A precise specification ensures consistent and correct implementations while enabling accurate security analysis.

The following recommendations help achieve precision:

1. Use clear and concise language, avoiding jargon or colloquialisms that may lead to misinterpretation. When introducing technical terms or concepts, provide clear definitions or explanations to ensure that all readers are on the same page.

2. Provide explicit instructions and avoid undefined behavior, ensuring implementers can follow step-by-step instructions with minimal or zero risk of misinterpretation. This helps ensure that all implementations are consistent with the intended design and minimizes the risk of errors or vulnerabilities.

3. Provide test vectors that check for correctness of all behavior in the specification, especially those near edge cases. For example, if a specification involves a branch or condition, then test cases should ideally be written to exercise both paths of the branch. Sometimes this is infeasible, e.g., if probability of a particular branch happening is negligible, though more often than not branches can be adequately covered.

4. Employ formal notation or pseudocode to provide a precise description of algorithms, data structures, and protocols. This ensures that implementers, researchers, and protocol designers can accurately understand the intended behavior and interactions of the components within the specification.

5. Specify data formats and encodings, clearly defining formats, encoding schemes, and serialization methods for all data types used in the specification. This helps ensure that different implementations can interoperate seamlessly and reduces the likelihood of incompatibilities or communication errors.

6. Document assumptions and dependencies, clearly stating any assumptions or dependencies on external components, including other specifications or protocol descriptions. This includes common dependencies like that of a random number generator. This helps implementers and researchers understand the context in which your specification operates and any potential limitations or risks.

Precise specifications minimize ambiguity and reduce the likelihood of implementation errors or inconsistencies.

3.3. Consistency

A specification must be internally consistent. It should also align with the conventions of similar documents.

Consistent use of concepts, vocabulary, language, and presentation reduces ambiguity. This clarity makes the specification easier to understand and implement.

The following recommendations help achieve consistency:

1. Establish a consistent terminology: Develop a clear and consistent set of terms and definitions that will be used throughout the document. Avoid using synonyms or multiple terms for the same concept, as this can lead to confusion. When using acronyms, always provide their full meaning upon first usage and use the acronym consistently afterward.

2. Maintain a uniform style and tone: Write the specification using a consistent style and tone to ensure that readers can easily follow the content. This includes consistent use of grammatical structures, punctuation, and capitalization. If your organization has a style guide, adhere to it when writing the specification.

3. Use a logical structure: Organize your specification in a logical manner, starting with an overview and then progressing through the various components, algorithms, and protocols. Make use of sections, subsections, and other structural elements to break up the content and make it easier to navigate and comprehend. Use forward or backward references to make navigation of the document simpler.

4. Provide consistent formatting: Ensure that all elements within the specification, such as tables, figures, pseudocode and equations, are formatted consistently. This will help readers quickly identify and understand these elements as they progress through the document.

5. Be consistent with conventions and notations: When using mathematical notation, programming languages, or other conventions, apply them consistently throughout the document. This will help prevent confusion and allow readers to focus on the content rather than deciphering different notations.

6. Reference external documents consistently: When referring to external documents or resources, such as other RFCs, standards, or research papers, provide consistent and accurate citations. This will enable readers to locate and review these resources as needed.

7. Keep the broader context in mind: Try to adopt the same terminology and conventions as other related documents the reader may be familiar with, especially for specifications that are developed based on peer-reviewed, published work. Consistency across audiences is important to help lower the bar to successful collaboration and effective communication. If the specification is intended to be part of the RFC series, reuse conventions from other documents in the series.

By focusing on consistency in your cryptography specification, you will make it more accessible and easier to understand for implementers, researchers, and protocol designers. This, in turn, will facilitate the development of correct, secure, and interoperable cryptographic systems based on your specification.

Cryptography specifications are often unique in their use of mathematical objects to define protocols. As such, presenting this content requires special guidance.

3.3.1. Representing Mathematical Operations

Cryptographic protocols rely on mathematical operations. These operations require precise and clear representation in specifications.

Ambiguous or inconsistent mathematical notation leads directly to implementation errors and interoperability failures.

[RFC7748] demonstrates effective mathematical representation through clear introduction of scalar multiplication notation, consistent usage throughout, and concrete examples.

3.3.1.1. Notation Consistency

Consistency in the notation used to represent mathematical operations is essential for avoiding confusion and ensuring that the specification is easy to understand. Specification authors should establish a clear notation system from the beginning and use it consistently throughout the document.

This notation should be introduced with a comprehensive description or a reference to a well-known notation system to ensure that readers can easily follow the mathematical expressions. For example, exponentiation can be represented by superscript or by a carat, but not by both.

3.3.1.2. Use of Standard Mathematical Symbols

Widely recognized mathematical symbols promote clarity and reduce the risk of misinterpretation. However, some symbols have different meanings across contexts or disciplines. The specification should clarify the intended meaning of such symbols. For instance, group operations in multiplicative notation use the * multiplication symbol rather than the x symbol to avoid confusion.

3.3.1.3. Explicitly Defining Custom Operations

Mathematical operations and notation that extend beyond standard conventions require explicit definitions with clear explanations and examples.

Key aspects of defining custom operations: - Provide clear explanations and examples. - Keep new notation minimal to avoid confusion. - Consider including a glossary for multiple non-standard operations.

3.3.1.4. Pseudocode and Algorithmic Descriptions

Mathematical expressions often need to be supplemented with pseudocode or algorithmic descriptions to bridge the gap between theory and implementation. Pseudocode should be written in a style that resembles real programming languages. Comments clarify the logic. Control structures such as loops and conditionals should use consistent notation throughout the document.

3.3.1.5. Visual Representations

Diagrams and other visual aids help convey complex mathematical concepts. These elements must be clear, properly labeled, and consistent with the notation system. Visual representations supplement the text; they do not replace it.

1. Ensure that diagrams remain legible in all output formats, including TXT, HTML, and PDF.

2. For simple state machines or data flows, use ASCII diagrams that display clearly in all output formats.

3. Keep every label, variable name, and symbol in your figures consistent with the notation used in the surrounding text.

3.3.1.6. ASCII-safe Mathematical Notation

Cryptographic specifications MUST use ASCII-only characters in all algorithm descriptions. Symbols that lack direct ASCII representation (for example, , ‖ , , ∞) MAY appear in informative examples or figures, but every such symbol MUST be accompanied by an ASCII equivalent and be defined exactly once in a dedicated Notation section. Each operator or symbol has exactly one meaning; authors MUST NOT overload a glyph (for example, ^) for multiple operations. Following these rules ensures the plain-text RFC renders unambiguously and prevents implementation errors stemming from visual formatting differences across output formats.

Checklist for authors:

* Define a concise notation table covering every non-obvious operator (||, ^, mod, XOR, etc.).

* Prefer XOR or || over Unicode or ‖ in normative text.

* Never reuse ^ for both XOR and exponentiation— spell out one of them instead.

* Verify all formulas in the generated *.txt* file; formatting must not change semantics.

* If Unicode appears in examples, provide the ASCII fallback inline, for example: (*XOR*).

Rendering considerations (HTML/PDF only):

* Authors MAY use <sup> (or Markdown ^) markup so the ASCII ^ exponent indicator renders as a superscript in HTML or PDF outputs. The plain-text RFC MUST still display the caret character.

* It is acceptable for the rendered HTML/PDF to substitute Unicode symbols for clarity — e.g., (U+2295) for XOR or (U+22C5 or U+00B7) for multiplication — provided that the canonical text uses the ASCII equivalents (XOR, *) and the symbol meanings are listed in the Notation table.

* Such styling MUST NOT alter the normative meaning, and the ASCII representation MUST remain authoritative.

3.3.1.6.1. Two-layer rule

Normative layer (canonical text): - Limited to printable ASCII plus SP and LF. - Only the operator glyphs in the table below are permitted.

Rendered layer (HTML/PDF): - Generated automatically by build tools (kramdown-RFC, Sphinx, or similar tooling). - May substitute typographical symbols (for example, *→, XOR→, ^→superscript). - Substitutions are stylistic only; the ASCII source remains authoritative.

Mandatory operator set

Concept	ASCII glyph(s)	Example	Notes
Addition / subtraction	+, -	a + b	
Multiplication	*	x * y	Define early that * is group/field multiplication
Exponentiation	^ or **	g^k, 2**255 - 19	Choose one symbol and use it consistently
XOR	XOR	a XOR b	Avoids clash with ^; all-caps stands out
Concatenation	||	M1 || M2	Define in glossary
Equality / assignment	= / <-	x <- y	<- optional but must be defined

Table 1

3.3.1.6.2. Operator glossary and constant-time annotations

Immediately after the terminology section, include a short table “Mathematical Operators and Symbols” . Each entry MUST provide: 1. ASCII glyph(s) 2. Description of the operation 3. Comment on constant-time versus variable-time expectations.

When pseudocode requires constant-time behavior, mark the line with the CONST tag, for example:

```
z <- CMOV(x, y, e) # CONST: branch-free
```

Style checklist for authors

* If a glyph could be ambiguous (e.g., ^), add an inline reminder the first time it appears: ^ (exponentiation).

* Never overload the same glyph for two different operations within the same specification.

* Prefer italic variables in rendered formats; keep them plain in ASCII.

* Provide at least one worked example that exercises every operator.

* If an uncommon Unicode symbol is truly necessary (e.g., for "perp"), include it only inside <artwork type="html"> with an ASCII fallback in canonical text.

4. Guidelines for Cryptography Specification Content

In addition to cryptographic specification clarity and accessibility through presentation format, the content of a specification also influences the overall value of the specification. The syntax of cryptographic objects introduced and their interfaces, as well as the way in which the object is structured for use in applications, is important for reliable and secure implementations of cryptographic algorithms and protocols. In this section, we discuss factors that relate to the content of the specifications and their impact on overall quality.

4.1. Reusability

Cryptography specifications that rely on bespoke sub-algorithms or lower-level components tend to be brittle and invite implementation issues. To create efficient, interoperable, and widely adopted cryptographic systems, it is preferable to reuse existing components or primitives. Reusability allows developers to build on existing work, reducing the time and effort required to create new implementations while leveraging established security properties and analyses. This section discusses the importance of reusability in cryptography specifications and offers guidance for incorporating reusability principles into the specification development process.

4.1.1. Build on Existing Specifications

When developing a cryptography specification, it is advantageous to build upon existing, well-established specifications, protocols, and primitives where possible.

By doing so, authors can capitalize on the collective expertise of the community, as well as existing security analyses, implementation experiences, and best practices. This approach reduces the potential for introducing new vulnerabilities and inconsistencies while promoting interoperability between different systems.

4.1.2. Modular Design

Emphasizing modularity in the design of cryptography specifications allows for greater flexibility and reusability. By breaking down complex algorithms into smaller, self-contained components or modules, specification writers facilitate the reuse of these components in different contexts or applications. A modular design also simplifies the process of updating or replacing specific components without affecting the overall system, making it easier to incorporate new research findings or technological advancements. An example of a modular design is the prime-order group abstraction. Algorithms that use this abstraction admit a modular design where the group implementation is described in a separate document dedicated to the details of the implementation of the group. This approach simplifies both implementation and security analysis.

4.1.3. Clear Interfaces and Abstractions

To promote misuse resistance and elegant higher-level designs, cryptography specifications should provide clear interfaces and abstractions for the components and primitives they describe.

Well-defined interfaces enable developers to understand and interact with a component without needing to know the details of its internal implementation.

This approach allows for the replacement or modification of components with minimal impact on the overall system and encourages the development of interchangeable components that can be reused across different applications and within protocols.

Cryptographic objects typically have a set of functions associated with them that make up the interface; structuring the functions to fit well-understood and existing abstractions helps make the job of using the object in higher-level algorithms easier and less prone to code duplication.

4.1.4. Completeness

The operations defined in a cryptography specification should be complete, with defined behavior on all inputs. This includes error handling and edge cases which would otherwise not impact the algorithm’s cryptographic properties.

In particular, when deserializing a byte string, the behavior on all byte strings should be defined, including cases which would not be valid outputs of the corresponding serialization function. A complete specification helps avoid implementation variations. These variations can lead to interoperability failures, gaps between formal analysis and real-world practice, or security vulnerabilities.

* Define behavior for all inputs: Ensure that every possible input scenario is accounted for, including edge cases.

* Error handling: Clearly specify how errors should be managed to prevent unexpected behavior.

* Avoid multiple valid behaviors: Consistency is key; avoid leaving multiple implementation options open.

Avoid defining multiple implementation behaviors as valid. Leaving multiple options to implementators leads to compounding complexity: downstream specifications may need to profile the algorithm to pick the preferred option, and validation tools must be configurable to assert either case.

4.1.5. Documentation and Examples

Thorough documentation and illustrative examples play a crucial role in promoting reusability. By providing comprehensive explanations of the specification’s components, interfaces, and intended use cases, specification authors make it easier for developers to understand and implement the specification correctly. Including examples of how components can be combined or applied in various scenarios further clarifies their usage and encourages their reuse in different contexts.

Documentation Tips: - Use clear, concise language - Include illustrative examples - Highlight use cases and scenarios

By focusing on reusability in cryptography specifications, authors can help create secure, efficient, and adaptable cryptographic systems that can be more easily integrated, maintained, and updated, resulting in more robust and widely adopted solutions.

4.2. Defining Security Definitions and Threat Models

Cryptographic protocols are always used within a context of a broader system whose security relies on an understanding capabilities of potential attackers. An incorrect definition or assumption about the threat models to a protocol can make a protocol that is safe in one context unsafe in a different context. Precise definitions help researchers assess the security of the proposed algorithms and protocols, while comprehensible threat models enable implementers and protocol designers to understand the potential risks and limitations of the specification. This section provides guidelines for defining security definitions and threat models in a way that caters to the needs of all target audiences.

4.2.1. Defining Security Goals

Specification authors should explicitly state the security goals that the proposed algorithms or protocols aim to achieve. These goals should be comprehensive, covering all relevant aspects, such as confidentiality, integrity, authentication, non-repudiation, and availability as well as resistance to implementation flaws such as side-channels.

Furthermore, authors should clarify any trade-offs or limitations associated with the security goals, ensuring that the target audiences understand the intended balance between security and other factors, such as performance or ease of implementation.

Common Security Goals: - Confidentiality - Integrity - Authentication - Non-repudiation - Availability - Resistance to side-channels

4.2.2. Formalizing Security Definitions

Formalizing security definitions is essential for researchers to rigorously analyze the algorithms and protocols described in the specification. Specification authors should strive to express security definitions in a formal language, using consistent notation and terminology. Authors should accompany formal definitions with clear explanations and examples to make them more accessible to implementers and protocol designers who may not be familiar with formal methods.

Steps to Formalize Security Definitions: - Choose a formal language - Ensure consistent notation - Provide clear examples

4.2.3. Describing the Threat Model

A well-defined threat model provides an overview of the potential adversaries and the risks they pose to the security of the algorithms or protocols. Specification authors should describe the threat model in detail, including the capabilities, resources, and motivations of adversaries. Additionally, authors should identify any assumptions made about the adversarial model and explicitly state them to help the target audiences understand the intended scope and limitations of the specification's security guarantees. Clear threat models help prevent misuse in inappropriate contexts.

Key Components of a Threat Model: - Adversary capabilities - Resources - Motivations - Assumptions about adversarial models

4.2.4. Addressing Known Vulnerabilities and Attacks

Specification authors should discuss known vulnerabilities and attacks relevant to the proposed algorithms or protocols. This discussion should include an explanation of how the specification addresses or mitigates these issues, as well as any residual risks that remain. This information is valuable for implementers and protocol designers to understand the potential threats and for researchers to assess the robustness of the specification's security claims.

4.2.5. Providing Guidance on Secure Implementation and Deployment

To help ensure that the security definitions and threat models are effectively realized in practice, authors should provide guidance on secure implementation and deployment of the proposed algorithms and protocols. This guidance may include best practices for avoiding common pitfalls, recommendations for cryptographic parameter selection, or considerations for securely integrating the specification into existing systems.

By clearly defining security definitions and threat models in cryptography specifications, authors can facilitate a better understanding of the security properties and limitations of the proposed algorithms and protocols among implementers, researchers, and protocol designers.

Clear security definitions prevent cryptographic algorithms from being used in insecure contexts.

* Following these guidelines and recommendations from [RFC3552] helps create robust security considerations sections

* Complete threat model discussions facilitate better understanding of security properties and limitations

* Proper security definitions enable accurate analysis by target audiences

5. Catering to Target Audiences

When writing a specification, it is important to consider the needs of the three primary audiences: implementers, researchers, and protocol designers. Each group has unique requirements and goals, and the specification should be written in a way that addresses their specific concerns.

5.1. Catering to Implementers

Implementers require a clear, concise, and unambiguous specification to develop production-grade software.

To cater to implementers:

* Provide step-by-step instructions for implementing algorithms or processes, ensuring that all required inputs, outputs, and intermediate steps are defined. Where exceptional cases occur, those should be noted and recommended error-handling steps should be given. Include test vectors to help implementers verify the correctness of their implementations.

* Describe best practices for representing components of the specification in code, addressing exceptional cases and recommended error handling procedures, as well as aspects of the specification that are difficult to implement correctly (e.g., where side-channel attacks might be possible).

* Clearly indicate any optional features, variations, or extensions, specifying their impact on interoperability and security.

5.1.1. Test vectors

Test vectors ideally cover all branches of the specification, with reasonable exceptions, such as branches that occur with negligible probability and as such are computationally infeasible to reproduce. To facilitate writing tests, where possible, all functions should be written with determinism in mind. In particular, this means that functions that produce random outputs, such as a function that produces random elements in a prime-order group, should accept randomness as input and test vectors should specify this randomness as an input to the function. Specifications should minimize internal

calls to PRNGs or similar and emphasize determinism.

Finally, specifications should make the connection between specification and test vectors clear by including explicit reproducibility steps that describe how test vectors were derived for parts of the specification. This might mean pointing to a reference implementation with instructions for how to run it, where the reference implementation is written in a way that is clearly consistent with the specification.

It's possible to include too many test vectors in a specification, which increases document length and decreases readability. Authors should provide test vectors that cover:

* Typical test cases that exercise all logical pathways within an algorithm

* All valid but degenerate cases that result in error or early exit of an algorithm

* Exceptions that can be reached by attacker-controlled inputs

It is NOT necessary to include test vectors for cases that are statistically improbable to be triggered, even by attacker-controlled input, based on the underlying cryptographic assumptions. For example, if an error case is only reachable when an intermediate data point matches the pre-image of a hash value that was randomly generated, finding a test vector to trigger that case would require the ability to compute a hash pre-image, which is deemed unfeasible for sufficiently strong hash functions. Exceptional cases that don't have test vectors should be explicitly noted in the algorithm description.

Lastly, specifications should provide references to machine-readable test vectors (e.g., in JSON format) that persist alongside the specification. This helps avoid possibly error-prone parsing in translating test vectors from a textual specification to test code inputs.

5.2. Catering to Researchers

Researchers need to understand the syntax and functionality of the cryptographic protocol or primitive to ensure its correctness and analyze its security properties. To cater to researchers:

* Clearly define the underlying mathematical concepts and notations used in the specification, ensuring that all symbols, functions, and variables are consistently and accurately represented as explained in the section Representing Mathematical Operations.

* Provide detailed security definitions, goals, and threat models, including the capabilities and limitations of adversaries and their impact on parameter selection. In general, authors should make input requirements that are important for the security of the protocol or construction maximally clear. See: Defining Security Definitions and Threat Models.

* Describe any assumptions made about the underlying primitives or protocols and the justifications for these assumptions. Such assumptions should include references to external documents that describe these underlying primitives or protocols where appropriate, unless there are gaps between how the underlying primitive or protocol is used and how it is described externally.

* Clearly present any security proofs, analysis, or references to existing literature that support the security claims of the specification. If there are gaps between the specification and formal security analysis, these gaps should be noted, along with rationale that justifies the gaps.

5.3. Catering to Protocol Designers

Protocol designers in the standards community use specifications to understand how to safely use the cryptographic protocol or primitive when designing a higher-level protocol that depends on it. To cater to protocol designers:

* Clearly define the interfaces, APIs, or functions exposed by the protocol or primitive, indicating how they should be used and any potential risks associated with their misuse. For example, for each input to the protocol, it should be made clear whether or not these are attacker controlled and, if so, describe what steps must be taken to validate that input.

* Describe any corner cases or situations that may impact security, providing guidance on how to avoid or mitigate potential risks. This includes explicitly stating the probability of an algorithm failing due to invalid operations occurring (such as divide-by-zero) both in the typical case and under attacker-controlled inputs.

* Explain any dependencies or interactions with other protocols, primitives, or system components, highlighting potential compatibility or interoperability issues.

* Provide guidance on configuration, parameter selection, or deployment considerations that may affect the security or performance of the protocol in real-world scenarios. This includes the impact of new discoveries that weaken the security assumptions of a primitive.

By addressing the specific needs of implementers, researchers, and protocol designers, a specification can be more effectively understood, implemented, and analyzed, leading to more secure and interoperable systems.

6. General Recommendations

Developing effective cryptography specifications often requires collaboration between multiple stakeholders in the target audience, including engineers, researchers, and standardization organizations, and engaging in a collaborative process helps ensure that diverse perspectives and expertise are considered, resulting in more robust and widely applicable specifications. This section discusses the importance of collaboration and compromise in specification development and offers recommendations for fostering a collaborative environment.

6.1. Encourage Open Communication and Feedback

Effective collaboration relies on open communication and an ongoing exchange of ideas and feedback. By creating channels for communication, such as mailing lists, pull request threads (as described in [RFC8874]), or regular meetings, authors can facilitate discussions, address concerns, and gather valuable input from various stakeholders. Encouraging an environment where feedback is welcomed and valued helps ensure that the specification benefits from diverse expertise and experiences.

6.2. Seek External Expertise

Involving external experts, such as researchers or engineers from different organizations, can help identify potential issues, uncover new insights, and provide a broader perspective on the specification. Engaging with experts such as those in the IRTF Crypto Review Panel who have different backgrounds or areas of expertise can also help identify potential gaps in the specification or highlight areas where further research or clarification is needed.

6.3. Recognize and Address Conflicting Interests

Collaboration often involves addressing conflicting interests or opinions among stakeholders. It is essential to acknowledge these differences and work towards finding mutually agreeable solutions. This may require making compromises or revisiting previous decisions to ensure that the specification meets the needs of all involved parties. By maintaining a flexible and open-minded approach, authors can:

* Build consensus among diverse stakeholders with varying priorities and technical perspectives.

* Develop a more robust specification that addresses real-world implementation and deployment challenges.

7. Examples of Well-Written Specifications

To provide a better understanding of how to write high-quality cryptography specifications, we will analyze specific sections from a well-written example: ChaCha20 and Poly1305 for IETF Protocols ([RFC8439]).

7.1. ChaCha20 and Poly1305 for IETF Protocols (RFC 8439)

[RFC8439] is a specification that describes the use of the ChaCha20 stream cipher and the Poly1305 message authentication code for IETF protocols. It demonstrates how to write a clear, comprehensive, and precise specification while catering to different audiences.

7.1.1. Introduction and Overview

The introduction in [RFC8439] clearly defines the purpose and motivation for the specification. It provides context on the origins of ChaCha20 and Poly1305, and how they are used together to provide confidentiality and data integrity. By presenting a concise and informative introduction, the specification sets the stage for the detailed technical descriptions that follow.

7.1.2. Algorithm Descriptions

The specification provides detailed and precise descriptions of the ChaCha20 and Poly1305 algorithms, including pseudocode, constants, and mathematical operations. This section caters to implementers, ensuring that they have all the necessary information to create consistent and correct implementations. The mathematical operations are expressed in a clear and unambiguous manner, which helps both implementers and researchers understand the algorithms better.

7.1.3. Test Vectors

[RFC8439] includes test vectors for both ChaCha20 and Poly1305, providing concrete examples of inputs and expected outputs for the algorithms. This section is invaluable for implementers, allowing them to verify that their implementations are correct and compatible with others.

7.1.4. Security Considerations

The specification dedicates an entire section to security considerations, catering to researchers and protocol designers. It discusses potential attacks and their mitigations, recommendations for nonce usage, and the security properties of the algorithms. This section also provides references to academic papers and other resources for further reading, enabling researchers to delve deeper into the security aspects of the specified algorithms.

7.1.5. IANA Considerations and References

[RFC8439] concludes with IANA considerations and a list of references, ensuring that the specification is well-integrated with existing IETF processes and standards. The IANA considerations section is essential for protocol designers who need to register new values or coordinate with existing registries.

7.1.6. Problematic Aspects

A criticism of this document is that it does not cater enough to protocol designers in that it does not explicitly define a decryption algorithm. Researchers familiar with the concept of a stream cipher understand that decryption and encryption are identical in stream cipher constructions, but this may not be clear to implementers.

In summary, [RFC8439] serves as an excellent example of a well-written cryptography specification, providing clear, precise, and comprehensive information for implementers, researchers, and protocol designers alike. By studying and emulating the structure and content of specifications like [RFC8439], authors can create high-quality specifications that cater to the diverse needs of their target audiences.

8. Examples of Specifications That Could Be Improved

[RFC8032] is a specification that describes the Edwards-curve Digital Signature Algorithm (EdDSA). This specification had several errata filed against it for corrections and has had documented criticisms published online.

8.1. Test Vectors

The test vectors included in this document were not comprehensive and did not cover all the cases described in the algorithm, resulting in multiple incompatible implementations. There were also issues with a "greater than" comparison which should have been a "greater than or equal to" which were not explicitly covered by the test vectors.

8.2. Unnecessary Branching

Some parts of EdDSA permit more than one verification path, which can split implementations. For Ed25519, [RFC8032] gives two options: 8_S_B = 8_R + 8_k_A′ (where * denotes scalar multiplication, ′ denotes a derived point, and = denotes equality) or S_B = R + k*A′ (shortcut). The shortcut saves cycles but lets libraries disagree on which signatures are valid. Specs should avoid such optional branches— especially performance-only shortcuts— to keep implementations interoperable.

8.3. Compatibility and Modularity

EdDSA is a variant of the Schnorr signature scheme, but with some small variations that make it incompatible with other related Schnorr signature schemes. This includes a "clamping" operation that makes EdDSA keys and operations incompatible with x25519 ([RFC7748]). Many of the issues in the specification derive from the fact that the specification was written to match an existing implementation rather than define an algorithm. This limited the authors from focusing on compatibility with other related standards and primitives, resulting in numerous issues.

9. Conclusion

Quality matters in cryptographic specification writing. This document provides guidelines for writing effective cryptography specifications, emphasizing the importance of catering to different audiences, such as target audiences, with the end goal of enabling high-assurance cryptographic software. By focusing on simplicity, precision, consistency, reusability, collaboration, and compromise, specification writers can create documents that are easier to understand, implement, and analyze.

We have also discussed the representation of mathematical operations and the importance of clearly defining security definitions and threat models. These elements are critical in ensuring that specifications are not only technically accurate but also convey the necessary information to properly assess the security properties of cryptographic algorithms and protocols.

Finally, we have examined a well-written example, [RFC8439], to demonstrate how these guidelines can be applied in practice, and by highlighting specific sections of this specification, we have shown how authors can create high-quality specifications that cater to the diverse needs of their target audiences.

In conclusion, the process of writing cryptography specifications is both an art and a science. The guidelines presented in this document should serve as a foundation for authors, but it is essential to remain open to feedback and collaboration with the broader community. By doing so, we can continue to develop and refine the specifications that underpin the secure and reliable communication systems of today and the future.

10. Security Considerations

This document discusses best practices for writing and editing cryptography specifications. It does not provide any guidance for the semantic contents of those specifications.

Poor specification practices can lead to serious security vulnerabilities. Ambiguous algorithm descriptions may result in incompatible implementations with different security properties.

11. IANA Considerations

This document has no IANA actions.

12. References

12.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

12.2. Informative References

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/rfc/rfc3552>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/rfc/rfc7748>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/rfc/rfc8439>.

[RFC8874] Thomson, M. and B. Stark, "Working Group GitHub Usage Guidance", RFC 8874, DOI 10.17487/RFC8874, August 2020, <https://www.rfc-editor.org/rfc/rfc8874>.

Authors' Addresses

Nick Sullivan
Cryptography Consulting LLC
San Francisco,
United States of America
Email: nicholas.sullivan+ietf@gmail.com

Christopher A. Wood
Cloudflare, Inc.
101 Townsend St
San Francisco,
United States of America
Email: caw@heapingbits.net