

“Preview Writeup”: In anticipation of a package submission to the NIST Threshold Call

Title: SplitKey

Subtitle: Two-party signing and decryption with extra features

Version: 0.1 (2026-01-12)¹

Team name: SplitForge

Team members: Peeter Laud, Alisa Pankova, Nikita Snetkov, Jelizaveta Vakarjuk, Petr Muzikant, Aivo Kalu, Burak Can Kus, Semjon/Sona Kravtšenko, Raul-Martin Rebane, Mart Oruaas

Abstract: We describe a number of signing and decryption protocols for two (main) parties, where one of the parties initiates protocol runs and the second one responds while trying to authenticate the first party. The protocols have reasonable security properties even if the first party does not adequately protect its keyshare — we consider the possibility of that party’s encrypted memory leaking to the adversary, while the encryption keys have only low entropy. We discuss what properties a protocol deployed in this setting (which we call “server-assisted” signing / decryption) should satisfy. Large deployments of such protocols do indeed exist; we describe the RSA-based signing protocol (used in the [Smart-ID service](#)) in this write-up. We also describe a protocol for server-assisted ECDSA-based signing, and a public-key encryption scheme with server-assisted decryption. Finally, we present a two-party (with offline correlated randomness generation) protocol for ML-DSA key-generation and signing. This one has not been designed for server-assisted setting, but its adaptation should not be difficult.

Proposed crypto-systems: (I) Split-RSA (Categories N1.3, N4.2); (II) Split-ECDSA (Categories N1.2, N4.1); (III) Split-decryption (Categories S2, N4.1); (IV) Trilithium: (2+1)-party ML-DSA (Categories N1.4, N4.3).

Keywords: Threshold Cryptography; NIST Threshold Call



¹Preliminary version submitted to NIST-MPTC for review

Preview writeup. This document is provided to NIST for online publication, to foster public awareness and support public discussion within the scope of the NIST First Call for Multi-Party Threshold Schemes [NIST-IR8214C]. This “preview writeup” represents a good-faith plan for a subsequent “package submission”. However, until the deadline for package submission, the team may still modify its own composition and the submission plan, including possible changes to the technical scope, and/or the used techniques or achieved results.

Team members: Peeter Laud ^{i1,a1}, Alisa Pankova ^{i2,a1}, Nikita Snetkov ^{i3,a1,a2}, Jelizaveta Vakarjuk ^{i4,a1,a2}, Petr Muzikant ^{i5,a1}, Aivo Kalu ^{i6,a1}, Burak Can Kus ^{i7,a1}, Semjon/Sona Kravtšenko ^{i8,a1}, Raul-Martin Rebane ^{i9,a1}, Mart Oruaas ^{i10,a1}

Open Researcher and Contributor Identifiers (ORCID):

i1 (0000-0002-9030-8142); i2 (0009-0006-8214-6848); i3 (0000-0002-1414-2080); i4 (0000-0001-6398-3663); i5 (0009-0008-7439-9508); i6 (0000-0002-5579-3648); i7 (0009-0002-2363-0913); i8 (0009-0007-8204-9519); i9 (0009-0003-7290-210X); i10 (0009-0005-4944-0570)

Affiliations:

^{a1} Cybernetica AS, Tallinn, Estonia

^{a2} Tallinn University of Technology, Tallinn, Estonia

Main contacts:

- **Team mailing list:** splitforge@lists.cyber.ee
- **Primary technical contact person:** Peeter Laud, peeter.laud@cyber.ee
- **Secondary contact person:** Nikita Snetkov, nikita.snetkov@cyber.ee

Produced by humans. The team hereby confirms that the content in this preview writeup: (i) was produced by the team members, and (ii) was not produced by generative artificial intelligence (AI), with the possible exception of AI-proposed grammar improvements, minor integrated suggestions, or some well-identified and short localized portions of auxiliary content (e.g., some illustration); and (iii) was proofread by the team members.

1. Introduction

We present a number of threshold cryptosystems targeted specifically towards two (main) parties. One of these parties — “Client” — is assumed to be weak in terms of the security measures it is able to use to protect its secrets either during their storage or their use in the protocols. Also, Client is the one who initiates the execution of the main functionality (signing or decrypting) of the cryptosystem, providing the message to sign or the ciphertext to decrypt. The second party — “Server” — participates in the execution with its own private key share, while trying to ascertain that it was indeed Client that initiated that run. We call such protocols “server-assisted”.

Split-RSA. We present Split-RSA: the original two-party RSA signing scheme [BKLO17] that is the basis of Cybernetica’s SplitKey® technology, which is used to provide the Smart-ID (<https://www.smart-id.com>) authentication and signing service in Baltic States. The service, which has been available since 2016, currently has 3.7M active users (out of 6.1M total population). Split-RSA works with two parties, providing security against one malicious party, and creates signatures interoperable with RSA [NIST-FIPS-186-5] (PKCS#1 v1.5 or PSS [MKJR16]).

SplitKey-like properties.. Split-RSA offers more security properties. It considers that Client’s keyshare is stored encrypted with a key that is provided by the (human) user before each signing. As the key (which is typically a PIN) must be remembered by the user, it can only have low entropy. At the same time, Client’s security measures are weak, hence the encrypted private key may leak to an adversary. Split-RSA is designed so that, when an adversary with the encryption of the private key starts brute-forcing for PIN, he can check his guesses only by contacting Server. In this way, Server is made aware of all the guesses of the PIN by some party initiating the signing. There is also a separate *clone detection* mechanism that allows Server to (eventually) detect whether the initiations of signing sessions come from a single device or multiple devices. Both mechanisms are necessary for the Server to adequately keep track of PIN guesses.

Split-ECDSA. ECDSA [NIST-FIPS-186-5], with its increasing popularity compared to RSA signatures, is poised to be the default signature scheme in electronic wallet applications, at least before the transition to post-quantum cryptography. Thus we also present Split-ECDSA, which gives the same SplitKey-like properties to a two-party ECDSA signing protocol. Actually, we have found that these properties can be added to existing protocols in a quite generic manner. The reference material gathered through the Threshold Call may focus on this generic method.

Split-decryption. Can threshold decryption have SplitKey-like properties? We provide a “Split-decryption” protocol that achieves it for an encryption system built upon ElGamal. Unfortunately, the ciphertexts of this protocol do not follow any standard format. This is perhaps less of an issue than for signatures, because the interoperability needs are smaller for encryption. We consider Split-decryption a possible alternative for secure elements in phones, decrypting the credentials that a user wants to present to a relying party. In this setting, we have identified another desirable security property — privacy (unlinkability) against Server. Split-decryption achieves this property by blinding the to-be-decrypted ciphertext before Client sends it to Server. Again, the reference material gathered through the Threshold Call may focus on these security properties.

Trilithium. Finally, we propose Trilithium, a protocol for ML-DSA key generation and signing. It is again meant for two “main” parties — Client and Server. But, being built on top of secure multiparty computation techniques, it requires a third party, the Correlated Randomness Provider (CRP). The protocol is secure against a malicious Client or a malicious Server. It can also be made secure against malicious CRP, thus being an instance of a three-party protocol with security against one malicious party. Currently, Trilithium is without SplitKey-like properties. We believe that the same methods used for all other schemes are broadly applicable here as well, even though algebraic structures are different.

2. Specification

2.1. Split-RSA

During the key generation protocol (similar to [DMS14]), each party generates an RSA key (d_i, n_i, e) , for $i \in \{1, 2\}$ (where n_i is a product of two secret primes and $d_i \equiv e^{-1} \pmod{\phi(n_i)}$) and the composed public key is defined by setting $n = n_1 \cdot n_2$. The Client's private key share is further split into additive shares using user-provided input PIN as $d_1 \leftarrow d'_1 + d''_1 \pmod{\phi(n_1)}$, where $d'_1 \leftarrow \text{genShare}(u, \text{PIN}, n)$ and u is a random bitstring. Further, d''_1 is sent to the Server for storage and u is stored by the Client, deleting d_1, d'_1 . In this way, the scheme ensures protection against offline guessing attacks as an adversary does not have any value to verify the correctness of their guess on the user's PIN without contacting the Server. Additionally, Server generates clone-detection string w that is sent to the Client for storage. For the key generation, but not for signing, we assume authenticated channel between the Client and the Server.

The signing of message m starts with Client reconstructing the share d'_1 of d_1 using user-supplied PIN. Message consisting of the signature share $y \leftarrow m^{d'_1}$ and the clone-detection string w is sent to the Server. Server verifies w and computes Client's signature as $\sigma_1 \leftarrow y \cdot m^{d''_1}$. If σ_1 verifies, then Server considers Client as authentic (and user as having entered the correct PIN). Server computes its signature $\sigma_2 \leftarrow m^{d_2}$ and combines σ_1, σ_2 to σ using Chinese Remainder Theorem. Server sends combined signature σ to Client.

The only other building blocks of the scheme are a padding function P to process the message before signing, and genShare function built from a pseudorandom function.

The security of Split-RSA is shown in the Universal Composability (UC) model, assuming the existential unforgeability (EF-CMA) of RSA signatures with padding function P under chosen-message attacks [SVL24]. The ideal functionality captures the SplitKey-like properties, including the adversary choosing corruptions at run-time.

2.2. Split-ECDSA

Split-ECDSA is a protocol with properties similar to Split-RSA. During the key generation each party generates its keyshare $(x_i, \text{pk}_i \leftarrow x_i \cdot G)$ and the composed public key is defined as

$pk \leftarrow pk_1 + pk_2$. The Client's private key share is further split into additive shares x'_1 and x''_1 similarly to Split-RSA using PIN as input, where `genShare` works by encrypting share x'_1 using user's PIN. After the key generation, Client safely deletes pk_1 .

During signing, the Client recomputes share of the key \hat{x}'_1 using PIN received from the user; if the PIN was correct then $\hat{x}'_1 = x'_1$. Client proceeds with computing public key share $Q'_1 \leftarrow x'_1 \cdot G$. The client proves to the server that it knows the discrete logarithm of Q'_1 (assuming that client initiates the communication, this could be an additional NIZK proof together with the first message from the client). The server tries to verify this proof with regard to pk_1 . Such mechanism of avoiding the brute-forcing of PIN is compatible with a number of two-party ECDSA protocols [DKLS18; CCLST19; Lin21; XAXYC21; KT24].

The security of the protocol is proven in UC-model in presence of adaptive adversary under the assumption that ECDSA signatures are EF-CMA secure. Corresponding ideal functionality [SVL24] captures unforgeability, clone detection and security against offline guessing attacks.

2.3. Split-Decryption

This functionality [LPV25] allows server-assisted decryption of an ElGamal ciphertext, where during the decryption, the ciphertext is blinded from the Server. The protocol design builds upon the threshold ElGamal encryption scheme with a non-interactive zero-knowledge (NIZK) proof that the ciphertext was correctly constructed [SG02]. Additionally, the client proves to the server that it knows the discrete logarithm of the public key share pk_1 , similarly to Split-ECDSA.

Additionally, we want privacy/unlinkability for the Client, hence it has to blind the ciphertext when sending it to the Server for decryption. This, in turn, means that the NIZK proof has to be blinded, too. To achieve this functionality, we have chosen to let the ciphertext include a designated verifier (DV) NIZK proof of correct construction, with Server as the designated verifier. We instantiate the DVNIZK proofs with the Damgård-Fazio-Niccolosi construction [DFN06], because there is a way for the client to blind them. Finally, to reduce the possible confusion that a misformed ciphertext may cause, we have chosen to include in the ciphertext a NIZK proof that the previously mentioned DVNIZK proof is going to convince a honest Server. Therefore, the ciphertext in Split-decryption is contains the regular ElGamal ciphertext $(u \leftarrow g^r, c_2 \leftarrow pk^r)$, accompanied by:

- π – the NIZK proof of knowledge of the exponent r (needed for security against chosen-ciphertext attacks),
- $(\alpha_1, \Gamma_1, \Gamma_2)$ – the DVNIZK proof of knowing r (designated to the Server),
- π_ι ($\iota \in \{1, 2\}$) – the NIZK proofs that $(u, \alpha_1, \Gamma_\iota)$ will be accepted by the verifier (for the Client to be sure that the ciphertext is correct before decryption).

During decryption of (u, c_2) , Client verifies the proofs π, π_ι . Next, Client blinds u by raising it to a random power. It also blinds the DVNIZK proofs and sends them all to the Server. Server verifies proofs and applies its share of the key to the blinded ciphertext and sends the result to the client. Client unblinds the result, and applies its own keyshare to it.

The security of the scheme is proven in UC-model under static corruption under one-more Computational Diffie-Hellman (CDH) assumption [LPV25]. The ideal functionality captures both the SplitKey-like properties and the privacy property.

2.4. Trilithium

Trilithium protocol [DKLS25] allows Client and Server with the support of a Correlated Randomness Provider (CRP) to jointly generate keys and produce a standard ML-DSA signature [NIST-FIPS-204]. As a three-party protocol, Trilithium is secure against one malicious party, even though a malicious CRP can perform *selective disclosure attacks* [KS06]. We consider these attacks harmless for the intended use-cases. On the other hand, Trilithium has simpler communication pattern than a generic three-party protocol: the CRP and one of the parties (e.g. Client) only have to share a random seed, while the CRP sends only a single, input-independent message to the other party.

Trilithium builds on secure multiparty computation (MPC) techniques, based on additive secret sharing [CCD88] with homomorphic authenticators (MACs) [BDOZ11]. It implements both the key generation and the signing protocols on top of these MPC techniques. It handles the non-linear parts of these operations as follows:

Generating random values from a small(er) set is done by getting secret-shared random bits (or small values) from the CRP, randomly flipping them, and linearly combining them (i.e. in standard manner).

Computing the high bits is done by a technique similar to bit extraction, but instead of splitting the value to bits, it is split into digits according to bigger radices. The radices are chosen so, that the result is more-or-less the most significant digit.

Rejection checks, i.e. inequality checks adapt the passively secure two-party protocol for binary rings [AMOST22] to fields, active security against *one* party, and *active privacy* [BLLP14] against the other party. We show that the security model described above is recoverable from such restricted setting.

Trilithium signing protocol makes the commitment value public, and evaluates the hash function in the open. Using a variant of the Module Learning With Rounding assumption, the revealing of commitment values is proven secure [DKLS25], even in rejected runs of the protocol.

3. Open-Source Implementations

Code progress and availability. All Split-RSA, Split-ECDSA, and Split-decryption protocols are in the proof-of-concept stage, which we intend to transform into the reference implementations. The Rust implementation of Trilithium is somewhat more mature, but needs to be improved in terms of readability and code comments.

Code structure. The reference implementations will be written in Go programming language (for Trilithium, the decision to either rewrite it in Go or to keep improving Rust code still needs to be taken, but we are inclined more towards the latter option). We believe that Go is the perfect language for prototyping and sharing proof-of-concept implementation. Indeed, Go has a purposefully simple and minimal syntax, a strict and enforced formatting style, and a build toolchain that enables a very fast single-binary-output compilation. Lastly, Go's out-of-the-box performance is decent enough for showcasing the capabilities of the protocol. We will create the reference implementation's code structure according to our other existing Go prototypes, which we believe to be easy to understand, build and run.

Preliminarily, there are no bundled or external dependencies. If some would become necessary during the implementation, they would be noted in the README.md, and defined in the go.mod file. Building will be handled by Go's toolchain and benchmarking script will be included in the examples folder.

Implementation of the networking model. In our Trilithium prototype, the networking model is implemented by utilizing Transmission Control Protocol (TCP) connection functionalities from the Rust's standard net library. In the upcoming Go implementations, the networking model will be implemented via custom comms package in the reference implementation. The base interface looks like the following code example, and it is used in the protocol-related code.

```
type Comms interface {
    Send(message any)
    Get() any
}
```

Following is the minimal example implementation of such interface, which utilizes Go's channels (i.e. tunnels used for inter-thread communication). Therefore, these ChanComms would be used when running the protocol locally in parallel.

```
type ChanComms struct {
    ToOtherParty chan any
    FromOtherParty chan any
}
func (c ChanComms) Send(message any) {
    c.ToOtherParty <- message
}
func (c ChanComms) Get() any {
    return <-c.FromOtherParty
}
```

Networking is greatly simplified by our protocols only having two (main) parties.

Testing. The protocols can be tested and benchmarked by downloading Go, and by building and running one of the examples in the reference implementation. The code will also contain unit tests, which can be validated by Go's toolchain. The current Trilithium implementation includes a test that validates the key and signature generation against NIST FIPS 204 [NIST-FIPS-204] ML-DSA test vectors.

For convenience, we may include necessary Dockerfiles and Docker-compose configurations in order to increase availability of the testing via containers. Arbitrary networking conditions, including a party's deviation from the prescribed protocol, may be simulated by custom implementation of Comms interface.

Protocols	Client's Key Size	Server's Key Size	Signature Size	KeyGen Time	Signing Time
Split-RSA	3328 bits	9344 bits	6144 bits	17 sec	30 ms*
Split-ECDSA	512 bits	1666 bits	512 bits	9.36 ms*	18.44 ms*

Table 1: Comparison of Split-RSA (3072 modulus) and Split-ECDSA (P-256 curve). Notice: * without network delay

Ciphertext Size	Client decryption message	Server decryption message	Encryption time	Decryption time (Client)	Decryption time (Server)
3758 bytes	1408 bytes	256 bytes	225 ms	117 ms*	39 ms*

Table 2: Split-decryption performance for 32-byte message. Notice: * without network delay

4. Experimental Performance Evaluation

The Split-RSA, Split-ECDSA, and Split-decryption protocols have a simple structure, with a single request sent from Client to Server, followed by the response from Server to Client. The computations of the parties are comparable to the evaluation of algorithms of asymmetric cryptography. Table 1 presents performance of Split-RSA and Split-ECDSA. Table 2 presents performance of Split-decryption.

Trilithium is a more complex protocol. It requires 3 round-trips (between Client and Server) for key generation, and 14 round-trips for signing. It has been optimized for the number of rounds, and the amount of traffic exchanged between Client and Server. One signing attempt requires between 180 KiB and 350 KiB (depending on ML-DSA security level) communication between them (both directions added up). When executing multiple signing attempts in parallel, the mean time to signature is between 1.5 and 3.5 seconds in a network with low latency, and between 4.8 and 7.5 seconds in a network with 100ms latency between Client and Server.

There is no communication between the Client and the CRP; they have a shared random seed. The communication from the CRP to the Server can go up to 100 MiB per signing attempt.

5. Licensing, Patent Claims, and Funding

1. We expect to use BSD or MIT licenses for the code we make public under our public GitHub organization: <https://github.com/ISRI-PQC>. A dependency for the Split-decryption implementation is Paillier encryption, which can be obtained under the same license(s).
2. Split-RSA is covered by [US11251970B2](#), Split-decryption partially by [US20250323791A1](#), Split-ECDSA by [WO2025248081A1](#), Trilithium partially by [GB202500923D0](#).
3. The work leading to the submission has received funding from Estonian Research Council, grants no. PRG1780 and PRG2177.

References

- [AMOST22] Nuttapon Attrapadung, Hiraku Morita, Kazuma Ohara, Jacob C. N. Schuldt, and Kazunari Tozawa. “Memory and Round-Efficient MPC Primitives in the Pre-Processing Model from Unit Vectorization”. In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '22. Nagasaki, Japan: Association for Computing Machinery, 2022, pp. 858–872. DOI: [10.1145/3488932.3517407](https://doi.org/10.1145/3488932.3517407).
- [BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. “Semi-homomorphic Encryption and Multiparty Computation”. In: *Advances in Cryptology – EUROCRYPT 2011*. Ed. by Kenneth G. Paterson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 169–188. DOI: [10.1007/978-3-642-20465-4_11](https://doi.org/10.1007/978-3-642-20465-4_11).
- [BKLO17] Ahto Buldas, Aivo Kalu, Peeter Laud, and Mart Oruaas. “Server-Supported RSA Signatures for Mobile Devices”. In: *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*. Ed. by Simon N. Foley, Dieter Gollmann, and Einar Snekkenes. Vol. 10492. Lecture Notes in Computer Science. Springer, 2017, pp. 315–333. DOI: [10.1007/978-3-319-66402-6_19](https://doi.org/10.1007/978-3-319-66402-6_19). Also at <https://www.academia.edu/66597144>.
- [BLLP14] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. “From Input Private to Universally Composable Secure Multi-party Computation Primitives”. In: *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE Computer Society, 2014, pp. 184–198. DOI: [10.1109/CSF.2014.21](https://doi.org/10.1109/CSF.2014.21). Also at ia.cr/2014/201.
- [CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. “Multiparty Unconditionally Secure Protocols (Extended Abstract)”. In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*. Ed. by Janos Simon. ACM, 1988, pp. 11–19. DOI: [10.1145/62212.62214](https://doi.org/10.1145/62212.62214).
- [CCLST19] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. “Two-Party ECDSA from Hash Proof Systems and Efficient Instantiations”. In: *Advances in Cryptology – CRYPTO 2019*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Cham: Springer International Publishing, 2019, pp. 191–221. DOI: [10.1007/978-3-030-26954-8_7](https://doi.org/10.1007/978-3-030-26954-8_7). Also at ia.cr/2019/503.
- [DFN06] Ivan Damgård, Nelly Fazio, and Antonio Nicolosi. “Non-interactive Zero-Knowledge from Homomorphic Encryption”. In: *Theory of Cryptography*. Ed. by Shai Halevi and Tal Rabin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 41–59. DOI: [10.1007/11681878_3](https://doi.org/10.1007/11681878_3).

- [DKLS18] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. “Secure two-party threshold ECDSA from ECDSA assumptions”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 980–997. DOI: [10.1109/SP.2018.00036](https://doi.org/10.1109/SP.2018.00036). Also at ia.cr/2018/499.
- [DKLS25] Antonín Dufka, Semjon Kravtšenko, Peeter Laud, and Nikita Snetkov. *Trilithium: Efficient and Universally Composable Distributed ML-DSA Signing*. Cryptology ePrint Archive, Paper 2025/675. 2025. URL: <https://eprint.iacr.org/2025/675>.
- [DMS14] Ivan Damgård, Gert Læssøe Mikkelsen, and Tue Skeltved. “On the Security of Distributed Multiprime RSA”. In: *Information Security and Cryptology - ICISC 2014 - 17th International Conference, Seoul, Korea, December 3-5, 2014, Revised Selected Papers*. Ed. by Jooyoung Lee and Jongsung Kim. Vol. 8949. Lecture Notes in Computer Science. Springer, 2014, pp. 18–33. DOI: [10.1007/978-3-319-15943-0_2](https://doi.org/10.1007/978-3-319-15943-0_2).
- [KS06] Mehmet S. Kiraz and Berry Schoenmakers. “A Protocol Issue for the Malicious Case of Yao's Garbled Circuit Construction”. In: *Proceedings of 27th Symposium on Information Theory in the Benelux*. 2006, pp. 283–290. Also at <https://www.academia.edu/5080563>.
- [KT24] Sermin Kocaman and Younes Talibi Alaoui. “Efficient Secure Two Party ECDSA”. In: *Cryptography and Coding*. Ed. by Elizabeth A. Quaglia. Cham: Springer Nature Switzerland, 2024, pp. 161–180. DOI: [10.1007/978-3-031-47818-5_9](https://doi.org/10.1007/978-3-031-47818-5_9). Also at ia.cr/2023/1455.
- [Lin21] Yehuda Lindell. “Fast Secure Two-Party ECDSA Signing”. In: *Journal of Cryptology* 34.4 (September 2021). DOI: [10.1007/s00145-021-09409-9](https://doi.org/10.1007/s00145-021-09409-9). Also at ia.cr/2017/552.
- [LPV25] Peeter Laud, Alisa Pankova, and Jelizaveta Vakarjuk. “Privacy-Preserving Server-Supported Decryption”. In: *38th IEEE Computer Security Foundations Symposium, CSF 2025, Santa Cruz, CA, USA, June 16-20, 2025*. IEEE, 2025, pp. 127–142. DOI: [10.1109/CSF64896.2025.00004](https://doi.org/10.1109/CSF64896.2025.00004). Also at [arXiv:2410.19338](https://arxiv.org/abs/2410.19338).
- [MKJR16] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. November 2016. DOI: [10.17487/RFC8017](https://doi.org/10.17487/RFC8017).
- [SG02] Victor Shoup and Rosario Gennaro. “Securing Threshold Cryptosystems against Chosen Ciphertext Attack”. In: *J. Cryptol.* 15.2 (2002), pp. 75–96. DOI: [10.1007/s00145-001-0020-9](https://doi.org/10.1007/s00145-001-0020-9).
- [SVL24] Nikita Snetkov, Jelizaveta Vakarjuk, and Peeter Laud. *Universally Composable Server-Supported Signatures for Smartphones*. Cryptology ePrint Archive, Paper 2024/1941. 2024. URL: <https://eprint.iacr.org/2024/1941>.
- [XAXYC21] Haiyang Xue, Man Ho Au, Xiang Xie, Tsz Hon Yuen, and Handong Cui. “Efficient Online-Friendly Two-Party ECDSA Signature”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual

Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 558–573. DOI: [10.1145/3460120.3484803](https://doi.org/10.1145/3460120.3484803). Also at ia.cr/2022/318.

[NIST-FIPS-186-5] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. (Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) NIST FIPS 186-5. 2023. DOI: [10.6028/nist.fips.186-5](https://doi.org/10.6028/nist.fips.186-5).

[NIST-FIPS-204] National Institute of Standards and Technology. *Module-lattice-based digital signature standard*. (Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) NIST FIPS 204. 2024. DOI: [10.6028/NIST.FIPS.204](https://doi.org/10.6028/NIST.FIPS.204).

[NIST-IR8214C] Luís T. A. N. Brandão and René Peralta. *NIST First Call for Multi-Party Threshold Schemes*. (National Institute of Standards and Technology) NIST Internal Report (NISTIR) 8214C. 2026. DOI: [10.6028/NIST.IR.8214C](https://doi.org/10.6028/NIST.IR.8214C).