Enhanced Digital Signature Service Using Split-key Signatures

A Technical Paper prepared for SCTE by

Nicol So, Sr. Staff Systems Engineer, CommScope, SCTE Member
101 Tournament Dr,
Horsham, PA 19044
nicol.so@commscope.com
+1 215 323 1149

Alexander Medvinsky, Engineering Fellow, CommScope, SCTE Member
6450 Sequence Dr.
San Diego, CA 92121
sasha.medvinsky@commscope.com
+1 858 404 2367

Table of Contents

List of Figures

List of Tables

1. Introduction

To prevent unauthorized software from being introduced into a user's computers or embedded devices, there are many existing standards that require executable images to include a digital signature. The device or OS platform validates a code image before it is executed or, in other cases, before it is downloaded and installed on the device. Some well-known standards of signed code images include Android APK [1], Microsoft Authenticode [2] and DOCSIS 3.1 cable modem software image signatures [3]. There are many more standards-based and proprietary code signing formats, some of which are verified in hardware (e.g., in the boot ROM).

A common method of code signing is to use a command-line tool together with a private key file. The process may be initiated by a developer who builds the software or by an automated software build workflow. For example, Android APK can be signed with Android Studio and binaries for Microsoft platforms can be signed with SignTool.exe. A chip vendor that supports boot code signature verification would typically provide a device manufacturer with their own command-line tool for signing boot code and it may be combined with their linker.

However, there are security concerns with this approach. Code signing keys may be stolen from development environments. Some well-known incidents involving compromised code signing private keys include those detailed in [8], [9], and [10]. For this reason, it is good practice to protect code signing keys in hardware security modules (HSMs), which provide strong protection against disclosure of the keys. Multiple companies offer commercial code signing tools and online services with hardware-based protection of private keys.

While protecting the confidentiality of code signing keys addresses one aspect of security, it is also important to ensure that only authorized individuals can exercise the protected signing key to sign code, and can do so only according to policy. A code signing service should support permissions and access control management so that an administrator with appropriate privileges can authorize specific individuals or teams of individuals to use specific signing keys based on their areas of responsibility.

A software development organization that uses a third-party code signing service is, in doing so, trusting the signing service not to sign any software using the organization's signing key without authorization. Note that the signing service has the technical ability to sign any code using its subscribers' signing keys; it is the security controls employed by the service that prevent unauthorized code signing from happening. Code signing services can themselves be targets of, and can even fall prey to, cyberattacks. This concern is validated by recent news of high-profile incidents in which providers of security technologies had themselves become victims of cyberattacks, adversely affecting users of their security products [11], [12], [13].

This paper introduces a code signing service architecture in which a subscriber's code signing key is split into shares controlled separately by the code signing service and the subscriber. If the subscriber's share is compromised, the code signing service will continue to protect the other half of the signing key and will prevent unauthorized code to be signed. At the same time, the code signing service is prevented from signing code in the subscriber's name without the subscriber's participation. If the code signing service becomes compromised in a cyberattack, the attacker will not have possession of the subscriber's share of a code signing key and will still be unable to sign unauthorized software releases.

2. Solution Outline

We will discuss applying split-key digital signature to enhance the security of code signing using an example system. In our example system, a trusted code signing service (TCSS) operates a secure infrastructure and performs code signing for its subscribers. A subscriber in this context may be an organization, but it could also be an individual. A subscriber owns a public-private key pair, which is used to sign and authenticate code (data objects) published by the subscriber. A subscriber has one or more authorized users, who are trusted to exercise the subscriber's private code signing key to produce code signed by the subscriber. A trusted third party (TTP) is a party trusted by a subscriber to generate a code signing key pair for the subscriber. Such trust may be established by technical and non-technical means, such as contractual guarantees, certification, and audits. A TTP should be viewed as a role—trusted key generation may not be the only service it provides.

Figure 1 illustrates the process of generating a split signing key in our example system.

In the description that follows, we use the notation $E(K, m)$ to denote a data object m encrypted using a key K. Cryptographic keys are denoted using symbols of the form K_{XYZ}, where the subscript XYZ is meant to be suggestive of the ownership or the purpose of the key. Where there is no confusion, we do not explicitly state which key of a key pair is used in an operation. For example, if K is a public-private key pair, it is obvious from the context that it is the public key of K that is used in computing $\mathrm{E}(\mathrm{K}, \mathrm{m})$. When discussing the two shares of a split signing key (more specifically the split private key of the signing key), we will use the convention that subscript 1 is associated with an authorized user, whereas subscript 2 is associated with the TCSS. Where it is necessary to distinguish between the public and private keys of a key pair K, we will use K^{PUB} and K^{PRIV} to refer to the public and private keys respectively.

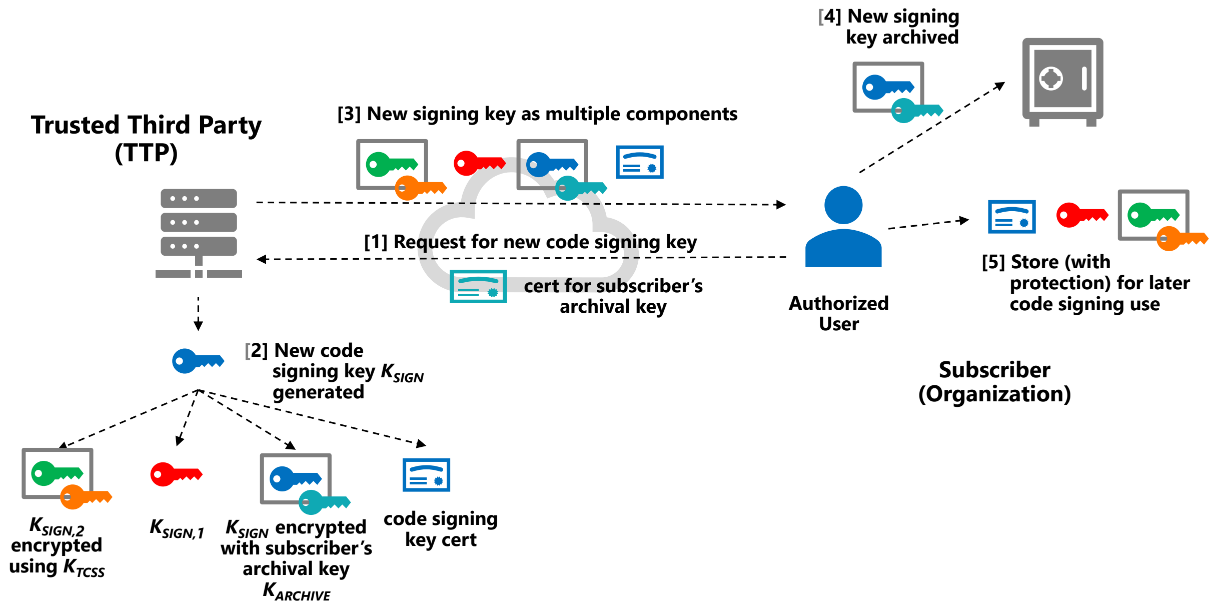
[4] New signing key archived
Trusted Third Party (TTP)
[3] New signing key as multiple components
[1] Request for new code signing key
cert for subscriber's archival key
Authorized User
[5] Store (with protection) for later code signing use
Subscriber (Organization)
[2] New code signing key K_{SIGN} generated
$K_{SIGN,2}$ encrypted using K_{TCSS}
$K_{SIGN,1}$
K_{SIGN} encrypted with subscriber's archival key $K_{ARCHIVE}$
code signing key cert

Figure 1 – Generation of a split signing key

In the key generation process shown in Figure 1, a TTP generates a code signing key K_{SIGN} and then splits it into 2 shares, namely $K_{SIGN,1}$ and $K_{SIGN,2}$. $K_{SIGN,2}$ is encrypted with the (public) key K_{TCSS} of the

TCSS. K_{SIGN} is encrypted with the (public) key $K_{ARCHIVE}$ of the subscriber organization for secure offline archival. The creation of this encrypted archival copy is optional, but some subscribers may find such archiving desirable. The (complete) signing key K_{SIGN} is normally kept offline and not available for code signing, to reduce the security risk of disclosure. $K_{SIGN,1}$, $E(K_{TCSS}, K_{SIGN,2})$ and $E(K_{ARCHIVE}, K_{SIGN})$ are returned to the client used by the authorized user.

For clarity, the keys shown in Figure 1 are color-coded as follows: the complete signing key K_{SIGN} is in blue, the share $K_{SIGN,1}$ is in red, the share $K_{SIGN,2}$ is in green, and K_{TCSS} is in orange.

The TTP also provides the public key K_{SIGN}^{PUB} of the key K_{SIGN}. A code signature that is generated with K_{SIGN} may be validated by anyone using the public key K_{SIGN}^{PUB}. In the figure, $\mathrm{K}_{\mathrm{SIGN}}^{\mathrm{PUB}}$ is provided in the form of a digital certificate, as is common practice, although that is not always necessary. It is assumed that communications between TTP and the client are encrypted and authenticated, for example using HTTPS.

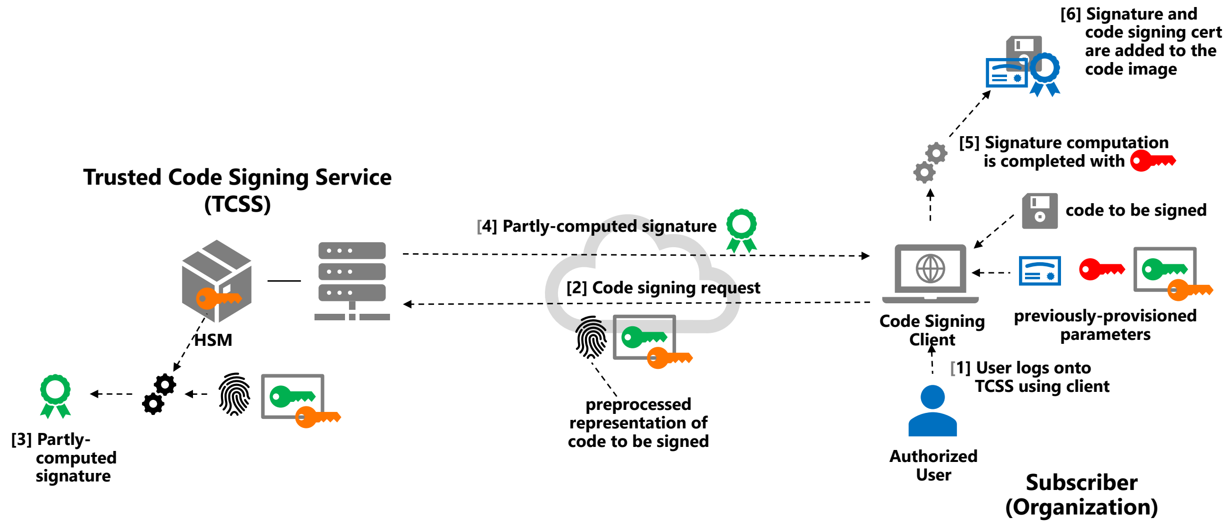
[6] Signature and code signing cert are added to the code image
[5] Signature computation is completed with
Trusted Code Signing Service (TCSS)
code to be signed
[4] Partly-computed signature
[2] Code signing request
Code Signing Client
previously-provisioned parameters
HSM
[1] User logs onto TCSS using client
[3] Partly-computed signature
preprocessed representation of code to be signed
Authorized User
Subscriber (Organization)

Figure 2 – Overview of two-step code signing

Once a client is provisioned with all of the above, the authorized user is able to use an established TCSS account to initiate code signing. Figure 2 illustrates the following code signing sequence:

1) The authorized user logs into the TCSS using a client application and establishes a secure authenticated session with the TCSS. This can be done in many different ways and implementation details are outside the scope of this paper.
2) Using the client, the authorized user submits to the TCSS a code signing request consisting of a preprocessed representation of the to-be-signed software image and $E(K_{TCSS}, K_{SIGN,2})$, which was obtained from the TTP before. The preprocessed representation here may contain a cryptographic digest of the software image, possibly with added padding and randomness.
3) The TCSS unwraps the encrypted $K_{SIGN,2}$ inside an HSM. In other words, $E(K_{TCSS}, K_{SIGN,2})$ is provided to the HSM, which decrypts it with the private key K_{TCSS}^{PRIV} to recover and use $K_{SIGN,2}$ inside the HSM's security perimeter. $K_{SIGN,2}$ is then used to generate a partly-computed code signature.

4) The partly-computed code signature generated using $K_{SIGN,2}$ is returned to the client application.
5) The client application is now able to compute a full code signature using this partly-computed signature and $K_{SIGN,1}$. Details of the underlying mathematics are described in the next section.
6) Finally, the client application combines the generated code signature, the corresponding code verification certificate, and the code image to create the final signed code image.

An advantage of this technique is that a code signing client can use a secure signing service (the TCSS) that uses $K_{SIGN,2}$ only inside an HSM. The TCSS can provide other security advantages, including fine-grained access control to different code signing keys and secure audit logging.

At the same time, the subscriber maintains control of its half of the signing key, $K_{SIGN,1}$, which is required to compute a full code signature. This way, TCSS does not have to be totally trusted – it is not able to sign code in the subscriber's name (i.e. using the K_{SIGN}) without the client's involvement, since the TCSS has access to neither the complete private key K_{SIGN} nor $K_{SIGN,1}$.

The following section provides mathematical details for split-key code signing based on the RSA cryptosystem.

3. Detailed Explanation for RSA-Based Split-Key Signing

3.1. A Quick Review of the Basic RSA Algorithm

RSA-based algorithms are commonly used for public-key encryption and digital signatures. Practical RSA algorithms (see [7], for example) are based on, but not identical to the "textbook" RSA algorithm. Because of security considerations, practical RSA algorithms have additional pre- and post-processing steps. The core of these RSA-based algorithms is the modular exponentiation operation

$$\mathrm{RSA}(\mathrm{m},\mathrm{e},\mathrm{n}) = \mathrm{m}^{\mathrm{e}} \bmod \mathrm{n}, \tag{1}$$

where m is the input to be processed, n is a modulus, and e is an exponent. The same operation is used both in the generation and verification of RSA signatures. In RSA key generation, a public-private key pair is generated together by the key owner, or a party or mechanism trusted by the key owner. An RSA private key (d,n) consists of a modulus $n = p\,q$, which is a product of two large primes p and q, and an exponent d. The factorization of n (i.e., the values of p and q) is kept secret by the key owner. The public key corresponding to (d,n) is (e,n) where e is an exponent such that $(\mathrm{m}^{\mathrm{e}})^{\mathrm{d}} \equiv \mathrm{m} \pmod{\mathrm{n}}$. More relevantly computation-wise, d and e satisfy

$$\mathrm{e} \cdot \mathrm{d} \equiv 1 \left(\mathrm{mod}\ \phi(n)\right), \tag{2}$$

where $\phi(n)$ is Euler's totient function. In RSA, where $n = pq$, $\phi(\mathrm{n}) = (\mathrm{p}-1)(q-1)$. In order for (2) to be satisfied, both e and d must be coprime with $\phi(n)$. The exponents e and d form a matched pair. Note that either e or d can be decided first, with the other derived using equation (2).

In RSA signature schemes, the private key of a key pair is the signing key. The public key is the verification key.

In practical RSA-based signature algorithms, the to-be-signed data object undergoes some preprocessing before the RSA operation is applied. The preprocessing typically includes reducing the data object to a digest using a cryptographic hash function and combining the digest with deterministic and random padding. Similarly, in signature verification, postprocessing is performed after the RSA operation. For the purpose of split-key RSA operations, these pre- and postprocessing steps can be ignored, as they do not involve the RSA keys, private or public.

3.2. Additive Splitting

In additive splitting of an RSA key, the private exponent d used in the signing operation is split into two shares d_1 and d_2, where

$$\mathrm{d}_1 + \mathrm{d}_2 \equiv d \left(\bmod\ \phi(n)\right). \tag{3}$$

With the split, the RSA operation m^d mod n can be computed as $\left(m^{d_1} \bmod n\right)\left(m^{d_2} \bmod n\right) \bmod\ n$. This computation can be carried out in two parts: one by a party with knowledge of d_1 and another by a party with knowledge of d_2.

3.2.1. Key Generation

To make RSA signing a split-key operation, different shares of the private exponent need to be held by different entities, so that compromising one of the entities will not give an attacker the ability to forge signatures. A possible exception to this arrangement is when the private key is archived in heavily protected and controlled offline storage.

There are different ways in which key generation can be accomplished. If a TTP is available, the TTP can generate the signing key in two shares, securely deliver the shares to a TCSS and the subscriber organization's authorized user, and securely erase its copy afterward. Alternatively, the TCSS can generate the signing key and securely deliver one share to the subscriber organization's authorized user. As in the case of key generation by a TTP, the TCSS securely erases the share of the private key for which it is not a custodian.

Yet another possibility for key generation is for the subscriber organization to generate the key pair, split the private key into two shares, upload one share to the TCSS and give custody of the other to an authorized user. All copies of the complete private key are securely erased afterward.

If the subscriber organization desires to keep a complete copy of the private key in archive, that can be achieved by encrypting a copy of the private key during key generation using an archive key belonging to the subscriber organization.

3.2.2. Key Splitting Operation

To split a private exponent d into d_1 and d_2 additively, d_1 may be chosen uniformly at random from $\{2, \ldots, \phi(n) - 2\}$. d_2 can then be determined based on equation (3). Note that d_2 as the additive inverse of d_1 modulo $\phi(n)$ always exists. Note also that because d is odd but $\phi(\mathrm{n}) = (\mathrm{p}-1)(\mathrm{q}-1)$ is even, one of d_1 or d_2 is even, and therefore not a valid exponent for normal RSA operation. However, that is not a problem because (d_1, n) and (d_2, n) don't need to function like normal RSA private keys.

After the private key is split, one share, (d_1, n), is given to the authorized signer. Another share, (d_2, n), is placed under the control of the TCSS. The TCSS can maintain a database of the private key shares entrusted to it by its subscribers. Alternatively, d_2 can be encrypted using a key owned by the TCSS. The encrypted d_2 is given to the authorized signer along with (d_1, n). When the TCSS needs to perform an operation with d_2 in response to a signing request from the authorized signer, the encrypted d_2 can be included in the request.

3.2.3. Signing Operation

To sign a data object m', the authorized signer first performs preprocessing on the data to produce m, a preprocessed representation of m', and sends it to the TCSS.

The TCSS computes a partly-computed signature by performing the RSA operation on the preprocessed representation using its share of the subscriber's signing key

$$s' = m^{d_2} \bmod n, \tag{4}$$

which the TCSS then sends to the authorized signer.

The authorized signer computes the final signature s as

$$s = s' \cdot \left(m^{d_1} \bmod n\right) \bmod n. \tag{5}$$

The authorized signer can verify that the signature is valid by checking that

$$s^e \bmod n = m. \tag{6}$$

Variation: As a variation, the RSA operations in the procedure above can be performed in the opposite order. In that case, the authorized signer computes a partly-computed signature on the preprocessed representation m as

$$s' = m^{d_1} \bmod n. \tag{7}$$

The partly-computed signature s', together with m, is then sent to the TCSS, which computes the final signature s as

$$s = s' \cdot \left(m^{d_2} \bmod n\right) \bmod n. \tag{8}$$

The final signature s is sent to the authorized signer. To catch unexpected computation errors, the final signature can be verified by the TCSS or the authorized signer, or both.

3.3. Multiplicative Splitting

In multiplicative splitting of an RSA key, the private exponent d used in the signing operation is split into two shares d_1 and d_2, where

$$d_1 \cdot d_2 \equiv d \left(\bmod \phi(n)\right). \tag{9}$$

With the split, the RSA operation $m^d \bmod n$ can be computed as $\left(m^{d_1} \bmod n\right)^{d_2} \bmod n$ or $\left(m^{d_2} \bmod n\right)^{d_1} \bmod n$.

3.3.1. Key Splitting Operation

One way to find a pair of d_1 and d_2 that satisfy equation (9) is to choose a random d_1 from among $\{3, \ldots, \phi(n) - 1\}$ that is coprime with $\phi(n)$. In that case, exactly one solution exists for d_2, which can be computed using the extended Euclidean algorithm.

Using this strategy, sometimes it may take more than one try to find a d_1 that is coprime with $\phi(n)$. If the factors p and q are chosen to be safe primes (i.e. $p = 2p' + 1$ and $q = 2q' + 1$, for some primes p' and q'), then choosing d_1 at random is almost guaranteed to succeed on the first try.

3.3.2. Signing Operation

As in the case of additive splitting, to sign a data object m', the authorized signer first preprocesses m' to produce a preprocessed representation m and sends it to the TCSS.

The TCSS computes a partly-computed signature s' by performing the RSA operation on the preprocessed representation using its share of the subscriber's signing key d_2

$$s' = m^{d_2} \bmod n, \tag{10}$$

which the TCSS then sends to the authorized signer.

The authorized signer computes the final signature s as

$$s = (s')^{d_1} \bmod n. \tag{11}$$

The authorized signer can verify that the signature is valid by checking that

$$s^e \bmod n = m. \tag{12}$$

Variation: As a variation, the RSA operations in the procedure above can be performed in the opposite order. In that case, the authorized signer computes a partly-computed signature on the preprocessed representation m as

$$s' = m^{d_1} \bmod n. \tag{13}$$

The partly-computed signature s' is then sent to the TCSS, which computes the final signature s as

$$s = (s')^{d_2} \bmod n. \tag{14}$$

The final signature s is sent to the authorized signer. To catch unexpected computation errors, the final signature can be verified by the authorized signer.

4. Experimental Results

We performed some experiments to compare standard RSA signature performance (with optimization based on the Chinese remainder theorem (CRT)) against the performance of a two-step digital signature approach with a private key that has been split either additively or multiplicatively, as described earlier. The experiments were executed in the following computing environment:

Table 1 – Benchmark Computing Environment

Operating system	Windows 10 64-bit
CPU	Intel® Core™ I7-10610U
RAM	32 GBytes
Cryptographic library	C# BouncyCastle

As expected, the overhead of key splitting is insignificant when compared to the time it takes to generate a random RSA keypair.

On the other hand, there is a noticeable decrease in performance with the two-step digital signatures because CRT-based optimizations are not available. Exploiting the CRT requires knowledge of the prime factors of n (p and q), which would allow the full private exponent to be computed from the public exponent, defeating the purpose of key splitting. Forgoing CRT-based optimizations, we instead used BouncyCastle library functions based on Montgomery's optimization for modular exponentiation.

While this two-step digital signature approach incurs a performance overhead, we consider it acceptable, especially for signing software images, for at least the following reasons:

1. A software image is often signed after a code build, which often involves orders of magnitude more computation than the public-key operations in code signing. This also means code signing is not a very frequently repeated operation for the same software.
2. In code signing, a software image is first reduced to a digest using a cryptographic hash function. The process is likely to be computationally more expensive than the public-key operations involved.
3. Code image signing is normally not expected to be a fast real-time operation. An overhead on the order of 1 second is generally not noticeable.
4. In our proposal, the TCSS's share of a split private key is protected by and used only inside an HSM. HSMs generally have hardware acceleration and very good performance for public-key operations. An increase in execution time of a fast operation by a small multiple is not a big overhead.

4.1. Additive Split Benchmarks

Table 2 – Additive Split Performance Measurements

Operation	Average execution time / ms		
	2048-bit	3072-bit	4096-bit
Key split	0.019	0.028	0.040
Standard CRT-based digital signature	30.479	97.410	290.090
Two-step digital signature[1]	207.100	703.553	2166.583

4.2. Multiplicative Split Benchmarks

Table 3 – Multiplicative Split Performance Measurements

Operation	Average execution time / ms		
	2048-bit	3072-bit	4096-bit
Key split	3.239	6.179	12.678
Number of retries (for the key split)	3.306	3.276	3.561
Standard CRT-based digital signature	30.479	97.410	290.090
Two-step digital signature[1]	206.855	699.414	2159.088

4.3. Benchmark Results Summary

In our experiments, two-step signature computations were very slightly slower with additive key splitting than with multiplicative splitting. This is probably because with additive splitting, an additional modular multiplication is required. The performance difference between the two approaches is insignificant in practical terms.

In our experiments with multiplicative key splitting, we used a simple generate-and-test strategy for selecting d_1. On average it took somewhere between 3 to 4 tries to find a d_1 that is relatively prime to $\phi(n)$. In any case, both multiplicative split and additive split have negligible performance overheads when compared to the computation in RSA keypair generation.

Based on our experimental measurements, we do not see a reason to favor one key splitting approach over the other on performance grounds. Both approaches are viable and are very similar performance-wise. And while two-step digital signatures result in a factor of 7 increase in total execution time over conventional CRT-based RSA signatures, that overhead is generally not significant in the context of software image signing. Note that in the total execution time, roughly half of the computation is normally performed by the TCSS. Between the TCSS and clients used by authorized users, only the TCSS is a centralized resource. Therefore, for the TCSS the performance overhead is only about half of what is suggested by the seven-fold increased execution time.

5. Security Considerations

1) As mentioned in section 4, RSA optimization based on the Chinese remainder theorem cannot be used in this two-step digital signature approach. Doing so would defeat the goal of having two

[1] The measured execution time includes both the computation performed by the TCSS as well as that performed by the authorized user.

parties involved in the code signing process, each having access to only one share of the private key.

2) Key splitting requires a cryptographically strong random number generator to ensure unpredictability and that all valid choices for the combination of private key shares are nearly equally likely. The same random number generator which qualifies for key pair generation can also be utilized for the purpose of key splitting.
3) There are known attacks against RSA based on "small" private exponents, such as the ones in [6] and [14]. We do not believe they are applicable to the key splitting methods we described. We note that when a private exponent is split into two shares, they are not always valid RSA exponents. Even when they are, no corresponding public keys are calculated or published for the shares. Also, in the methods we described, a pair of private key shares d_1 and d_2 is chosen uniformly at random from among all valid choices. Only a negligible fraction of such choices have one or both of d_1 and d_2 is less than, say, half of the bit length of the modulus.

6. Conclusions

We described an architecture for applying split-key digital signature to create a code signing service with enhanced security and explained the advantages it offers. Two approaches of splitting RSA signing keys are presented as examples. We anticipated that signature generation using split keys would incur performance overhead because a commonly employed optimization technique becomes unavailable when split keys are used. We performed experiments using a software implementation to gauge the computational overhead. Measurements from the experiments confirmed our expectation that the overhead is very acceptable in typical code signing usage scenarios.

7. Abbreviations and Definitions

7.1. Abbreviations

CRT	Chinese remainder theorem
HSM	hardware security module
RSA	Rivest-Shamir-Adleman public-key cryptosystem
TCSS	trusted code signing service
TTP	trusted third party

7.2. Definitions

authorized signer	a person authorized by a subscriber (organization) to have possession of a share of a split signing key belonging to the subscriber and to request signatures from a trusted code signing service
partly-computed signature	an intermediate result computed using one share of a split private signing key. It is used later in combination with a second share of the signing key to compute a complete digital signature.
share	one of the outputs from splitting a private key. Knowledge of all the shares of a private key is equivalent to knowledge of the private key. Knowledge of only one share does not make it feasible to generate a valid signature.

subscriber	a party that uses a TCSS for two-step code signing. A subscriber may be an organization but may alternatively be an individual.

8. Bibliography and References

[1] "Application Signing", *Android Open Source Project*, undated. [Online]. Available: https://source.android.com/security/apksigning. [Accessed: November 17, 2021].

[2] Microsoft, "Windows Authenticode Portable Executable Signature Format," March 21, 2008. [Online]. Available: http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/authenticode_pe.docx.

[3] CableLabs, "Data-Over-Cable Service Interface Specifications DOCSIS® 3.1 Security Specification," *CableLabs*, CM-SP-SECv3.1-I04-150910, September 10, 2015. [Online]. Available: https://community.cablelabs.com/wiki/plugins/servlet/cablelabs/alfresco/download?id=00d39889-0af8-4722-b8a2-0063eeaa460a.

[4] B. Lynn, "The Chinese Remainder Theorem", undated. [Online]. Available: https://crypto.stanford.edu/pbc/notes/numbertheory/crt.html.

[5] Ç. K. Koç, "Montgomery Arithmetic", in *Encyclopedia of Cryptography and Security*, 2011 ed., Boston, MA: Springer, 2011.

[6] M. Wiener, "Cryptanalysis of short RSA secret exponents," *IEEE Trans. Inform. Theory*, vol. 36, pp. 553–558, May 1990.

[7] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", *Internet Engineering Task Force*, RFC 8017, November 2016. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8017.txt.

[8] P. Nohe, "Code Signing Compromise Installs Backdoors on Thousands of ASUS Computers," *thesslstore.com*, April 1, 2019. [Online]. Available: https://www.thesslstore.com/blog/code-signing-compromise-installs-backdoors-on-thousands-of-asus-computers/.

[9] T. Anderson, "HashiCorp reveals exposure of private code-signing key after Codecov compromise," *The Register*, April 26, 2021. [Online]. Available: https://www.theregister.com/2021/04/26/hashicorp_reveals_exposure_of_private/.

[10] D. Goodin, "Crooks steal security firm's crypto key, use it to sign malware," *Ars Technica*, February 8, 2013. [Online]. Available: https://arstechnica.com/information-technology/2013/02/cooks-steal-security-firms-crypto-key-use-it-to-sign-malware/.

[11] Center for Internet Security, "The SolarWinds Cyber-Attack: What You Need to Know," *Center for Internet Security*, March 15, 2021. [Online]. Available: https://www.cisecurity.org/solarwinds/.

[12] C. Cimpanu, "Nightmare week for security vendors: Now a Trend Micro bug is being exploited in the wild," *The Record*, April 22, 2021. [Online]. Available: https://therecord.media/nightmare-week-for-security-vendors-now-a-trend-micro-bug-is-being-exploited-in-the-wild/.

[13] T. Seals, "Zoho ManageEngine Password Manager Zero-Day Gets a Fix, Amid Attacks," *Threatpost*, September 9, 2021. [Online]. Available: https://threatpost.com/zoho-password-manager-zero-day-attack/169303/.

[14] D, Boneh and G. Durfee, "Cryptanalysis of RSA with private key *d* less than $N^{0.292}$," *IEEE Trans. Inform. Theory,* vol. 46. pp. 1339–1349, 2000.