New Key Extraction Attacks on Threshold ECDSA Implementations

Duy Hieu Nguyen
Verichains

Anh Khoa Nguyen
Verichains

Huu Giap Nguyen
Verichains

Thanh Nguyen
VNSecurity & Verichains

Anh Quynh Nguyen
Nanyang Technological University

Abstract

Threshold ECDSA, a Threshold Signature Scheme based protocol for the widely used digital signature algorithm ECDSA, has gained much attention in the distributed ledger industry. The use of this protocol was introduced as a means to increase security in these systems, as many ledgers have a dependency on ECDSA. However, as with any novel cryptographic protocol, the security of Threshold ECDSA has not been thoroughly tested over time. In light of this, a survey was conducted to evaluate the security of various implementations of Threshold ECDSA.

The survey conducted on the security of Threshold ECDSA implementations yielded some unexpected results, revealing the persistence of implementation flaws that can leave systems vulnerable to various attack vectors. Our research has identified three potential attacks that have the potential to recover the private key of ECDSA, depending on the particular implementation flaw. As a result, this paper provides a comprehensive analysis of the security of Threshold ECDSA, starting with a review of the cryptographic primitives utilized in its implementation, followed by a detailed explanation of the attack process and how to mitigate them.

Our research highlights the need for ongoing security evaluations in the distributed ledger technology field, especially when it comes to new technologies like Threshold ECDSA. Through our analysis, we have identified several implementation flaws that could be exploited to trigger attacks on these systems. It is critical to understand the potential risks associated with the use of Threshold ECDSA and to take proactive measures to ensure the security of assets stored in these systems. By sharing our findings and attack instructions, we hope to raise awareness and encourage both the academic and practical communities to prioritize security in their implementations.

1 Introduction

Threshold Signature Scheme (TSS) is a cryptographic scheme allowing multiple parties to jointly generate keys and sign messages. For signing a message, at least t (the threshold) out of n (the number of parties participating in the generation ceremony of the key in use) parties are required. There is no trusted dealer as the TSS private key is never constructed (each party only keeps a private key share).

In the blockchain ecosystem, ECDSA is the most popular signature scheme as it is used by Bitcoin and Ethereum, to name a few. Designing a TSS for ECDSA is not straightforward, since it requires MPC (multi-party computation)-unfriendly operations over shared secrets such as inversion or multiplication. To accomplish this, many cryptographic toolboxes (e.g., homomorphic encryption, zero-knowledge proof, ...) are involved, thus making the scheme complex and adding more attacking surface. Most open-source TSS implementations today follow the research line of Rosario Gennaro and Steven Goldfeder [1–3], which is also the only design of concern throughout this paper.

Section 2 presents an overview of the cryptographic techniques employed in the implementation of Threshold ECDSA. Section 3 details the various attack vectors that can exploit commonly observed implementation flaws in Threshold ECDSA. Specifically, we provide a comprehensive list of the triggering flaws, attack flow, and potential mitigation strategies for each of the three identified attacks. Finally, in Section 5, we summarize our contributions and highlight the significance of our findings for the security of Threshold ECDSA implementations in practical settings.

2 Background

This section provides the necessary details to understand the attacks to be discussed. For complete specification of the Threshold ECDSA protocol, please refer to [1–3].

2.1 Fiat–Shamir heuristic

Fiat-Shamir heuristic is a technique for removing interactivity from interactive, public-coin proof systems.

In these systems, the verifier generates and sends to the prover some random values acting as challenges which can only be solved if the statement being proved is correct (the soundness property). Usually if these random values are known in advance, the prover will be able to forge proofs for incorrect statements.

Fiat-Shamir heuristic replaces random values with outputs from a cryptographic hash function, hashing the verifier context whenever randomness is needed. Public parameters, the statement being proved, previously exchanged messages, ... all contribute to this context. Since randomness is removed, the verifier becomes deterministic and simulatable, thus making the proof system non-interactive.

2.2 `dlnproof`

`dlnproof` is used to verify, in zero-knowledge manner, that a prover knows $\log_g h$ modulo a composite number N.

At the key generation ceremony, each party is required to generate and broadcast a triple $\tilde{N}, h_1, h_2$, which is later used in subsequent signing ceremonies, together with a `dlnproof` proving that the party knows $\log_{h_2} h_1 \mod \tilde{N}$. Some implementations also require an additional `dlnproof` for $\log_{h_1} h_2 \mod \tilde{N}$.

Here is the interactive version of `dlnproof`:

1. Peggy (the prover) commits to a random value $\rho \in \mathbb{Z}_{\phi(N)}$ ($\phi(N)$ known only to Peggy) by sending $\alpha = g^\rho \mod N$ to Victor (the verifier).
2. Victor chooses and sends Peggy a random challenge bit $c \in \{0,1\}$.
3. Peggy computes and sends back $\tau = \rho + c \log_g h$.
4. Victor accepts if and only if $g^\tau = \alpha h^c$ modulo N.

This protocol is sound since, given a successful prover, one can apply the rewind technique to extract the discrete logarithm value similar to proving soundness of the well-known Schnorr protocol. Note that knowing c in advance allows an attacker Eve, who does not have knowledge of $\log_g h$, to trick Victor into accepting by sending him $\alpha = g^\tau h^{-c}$ for an arbitrary τ of Eve's choice in round 1.

The interactive proof above is converted to non-interactive `dlnproof` by applying the Fiat-Shamir heuristic described earlier. The proof is also repeated λ times (usually $\lambda \geq 80$) to reduce the soundness error from $\frac{1}{2}$ (the chance of making a correct guess for c at the beginning of the protocol) to $\frac{1}{2^\lambda}$.

2.3 MtA sub-protocol

MtA stands for multiplicative-to-additive, which is a sub-protocol involving 2 parties Alice and Bob holding secret values a, b respectively. At the end of MtA, Alice obtains α and Bob obtains β such that $ab = \alpha + \beta \mod q$ in which q is the order of the ECDSA group in use. During a TSS signing ceremony, for each pair of 2 different parties P_i, P_j, MtA is run four times (Table 1). The input secret values are:

Iteration	Alice	a	Bob	b
1	P_i	k_i	P_j	γ_j
2	P_j	k_j	P_i	γ_i
3	P_i	k_i	P_j	w_j
4	P_j	k_j	P_i	w_i

Table 1: The 4 iterations of MtA between 2 parties P_i and P_j.

- k_i: P_i's private share of the nonce inverse. If all k_i are leaked, the nonce inverse k could be reconstructed ($k = \sum k_i$) and combined with the message hash m and the signature r, s (m, r, s are public information) to recover the private key d of the TSS group based on the ECDSA formula $s = k(m + rd) \mod q$.
- γ_i: P_i's private share of γ, a temporary secret value used to calculate the shares of $k^{-1} \mod q$. Since $\delta = k\gamma$ is published, the leakage of all γ_i also allows k to be reconstructed ($k = \frac{\delta}{\gamma} = \frac{\delta}{\sum \gamma_i} \mod q$).
- w_i: a value depends on P_i's private key share x_i and which members of the TSS group are currently running the signing ceremony (requiring t/n members). If all w_i are leaked, the TSS private key d can be reconstructed by $d = \sum w_i \mod q$.

As a conclusion, leaking MtA input a or b both lead to TSS private key recovery.

In the MtA protocol, both parties need to exchange encrypted values with each other in order to perform the desired computations. However, since there is a lack of trust between the parties, it is difficult to ensure that the encrypted values being sent are valid. To mitigate this problem, the protocol requires both parties to also send a range proof of the encrypted values.

At first glance, the requirement for a range proof may seem unnecessary, but it has been shown in [4] that the absence of a range proof can lead to a major breakdown in the security of the protocol. The reason for this is that range proofs ensure that the encrypted values are within a specified range, and prevent attackers from tampering with the values or sending invalid data.

2.4 $\tilde{\mathbf{N}}, \mathbf{h_1}, \mathbf{h_2}$

The triple $\tilde{N}, h_1, h_2$ is used in MtA range proofs. Let $\tilde{N_A}, h_{1A}, h_{2A}$ and $\tilde{N_B}, h_{1B}, h_{2B}$ denote Alice and Bob's $\tilde{N}, h_1, h_2$

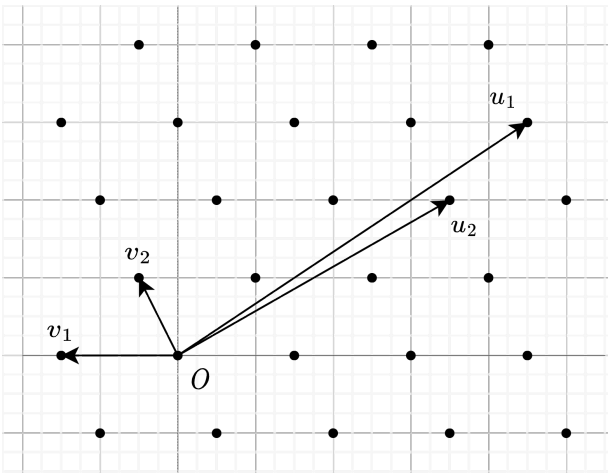

Figure 1: This lattice (represented by black dots) in $\mathbb{R}^2$ can be constructed from either $\{u_1, u_2\}$ or $\{v_1, v_2\}$.

respectively. It turns out that the following values are revealed by the range proofs:

- $z_A = h_{1B}^a h_{2B}^{\rho_A} \bmod \tilde{N}_B$ via Alice range proof for a. ρ_A is a random value in $\mathbb{Z}_{q\tilde{N}_B}$.
- $z_B = h_{1A}^b h_{2A}^{\rho_B} \bmod \tilde{N}_A$ via Bob respondent range proof for b. ρ_B is a random value in $\mathbb{Z}_{q\tilde{N}_A}$.

It can be seen that if Bob is able to eliminate $h_{2B}^{\rho_A}$ and compute discrete logarithm modulo $\tilde{N}_B$, then he could learn Alice's private input a. A similar result can be obtained when the attacker plays on the side of Alice.

In many implementations of TSS protocols, the values of h_1 and h_2 are chosen in such a way that there exists a value x that satisfies the equation $h_1 = h_2^x \mod \tilde{N}$. To ensure the other parties that their chosen values are generated as such, a `dlnproof` of $\log_{h_2} h_1$ is appended when they broadcast these values. This serves as a means of verifying that the values of h_1 and h_2 were indeed generated as specified, and that there are no discrepancies or malicious alterations made by any of the parties.

2.5 Vector enumeration in lattice

A lattice in $\mathbb{R}^n$ is a set of linear combinations with integer coefficients of n linearly independent vectors[*]:

$$\{a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \mid a_i \in \mathbb{Z}, b_i \in \mathbb{R}^n \text{ for } 1 \le i \le n\}$$

These independent vectors are together called a lattice basis. Different bases may generate the same lattice as shown in Figure 1.

[*]Only full-dimensional lattices are considered.

A lattice basis is good if it consists of nearly orthogonal, short vectors (e.g., $\{v_1, v_2\}$ in Figure 1). To convert a bad basis (e.g., $\{u_1, u_2\}$ in Figure 1) to a good one, we need a lattice basis reduction technique such as the well-known LLL algorithm. LLL runs in polynomial time and guarantees some qualities on the output basis. It is widely implemented and has useful applications in many areas, especially cryptanalysis.

In practice, there are often times when we need to search or enumerate through all vectors of a lattice (its basis is given) inside a particular region (e.g., a hypersphere or box) until some certain condition is met. To solve this kind of problem, we may use the following approach:

1. Apply LLL to the given basis, obtain $B = \{b_1, b_2, ..., b_n\}$. The result vector v can now be represented by $\sum_{i=1}^n c_i b_i$.

2. Determine the possible range $[c_{i\text{-min}}, c_{i\text{-max}}]$ for each c_i. This is done by first projecting each b_i and the searching region onto the orthogonal line of the subspace generated by $B \setminus \{b_i\}$ to obtain b_i^* and a line segment. Then, only c_i such that $c_i b_i^*$ lies in that segment is considered valid. Note that the purpose of projection is to make sure that we are working on a subspace that is only affected by b_i. In other words, if c_i is out of range ($c_i b_i^*$ falls out of the segment), then no combination of the remaining vectors $B \setminus \{b_i\}$ is able to fix that error.

3. Looping over all c_i to search for the desired vector v. Specifically, the search space S for v is equivalent to the Cartesian product of the ranges $[c_{i\text{-min}}, c_{i\text{-max}}]$. Hence, its size $|S|$ is equal to $\prod_{i=1}^n (c_{i\text{-max}} - c_{i\text{-min}} + 1)$.

One may figure out that the enumeration approach above is not optimal since it does not make use of early-exit or pruning. For example, when c_1 is determined, the actual range for c_2 may be adaptively smaller than the generic range $[c_{2\text{-min}}, c_{2\text{-max}}]$ but this fact is ignored.

However, the approach allows for seamless integration of the meet-in-the-middle technique (MITM) to significantly improve its time complexity (possibly down to $O(\sqrt{|S|})$) when the required condition on v can be converted into some form that allows separated working on 2 complementary subspaces S_1, S_2 of S. For example, if $v = c_1 b_1 + c_2 b_2$ and the condition on v can be expressed in c_1, c_2 as $f_1(c_1) = f_2(c_2)$, then $S_1 = [c_{1\text{-min}}, c_{1\text{-max}}]$, $S_2 = [c_{2\text{-min}}, c_{2\text{-max}}]$ and by applying MITM, the searching time will be reduced from $O(|S|)$ to $O(|S_1| + |S_2|)$ at the cost of some memory[†].

For some optimization, one may consider applying a more advanced lattice enumeration technique inside each of the subspaces S_1, S_2 whenever possible.

[†]Basically, by the technique, we loop over S_1 and cache all $f_1(c_1)$ to memory as a table (requiring $O(|S_1|)$ time and space), then loop over S_2 until an entry for $f_2(c_2)$ is found (requiring $O(|S_2|)$ time).

Figure 2: Hashing (i_1, i_2) by simply concatenating their byte representations.

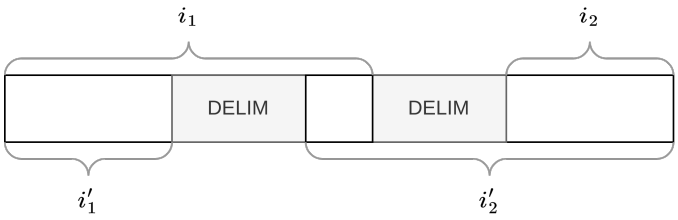

Figure 3: (i_1, i_2) and (i'_1, i'_2) both produce the same hash output.

3 Attacks

The Threshold ECDSA technology is a recent development in the field of cryptography. As with any new technology, there is a risk of implementation flaws that can lead to potential attacks. In sections 3.1 and 3.3, we will discuss some potential attack vectors that can arise in the context of Threshold ECDSA and their triggering conditions.

The focus of our attack will be on the leaking of secret shared values x in $z = h_1^x h_2^\rho \bmod \tilde{N}$ during the MtA rounds. This can be accomplished if a malicious party generates a triple $\tilde{N}, h_1, h_2$, where h_1 and h_2 are chosen in such a way that it is possible to eliminate h_2^ρ from z and compute discrete logarithm to base h_1 modulo $\tilde{N}$.

For eliminating h_2^ρ, we let $h_2 = h_1^e \bmod \tilde{N}$[‡] in which e is a divisor of $\text{ord}(h_1)$. Now, raising z to the power of $f = \frac{\text{ord}(h_1)}{e}$, we have $z^f = (h_1^f)^x \bmod \tilde{N}$. Solving it $(\log_{h_1^f} z^f)$ yields $x \bmod e$. When e is greater than q (the ECDSA group order), x is fully recovered. However, this approach will not work unless we can forge a `dlnproof` for the inexistent $\log_{h_2} h_1 = \frac{1}{e} \bmod \text{ord}(h_1)$. By utilizing some implementation flaws, we are able to do so!

To make discrete logarithm easy to compute, we choose $\tilde{N}$ to have one of the following properties, depending on the exploitation context (e.g., which checks are performed on $\tilde{N}$):

- (*square*) $\tilde{N} = p^2$. Computing a discrete log is similar to how the Paillier cryptosystem decrypts a ciphertext, exploiting the fact that $(1+kp)^x = 1 + kpx \bmod p^2$. Since the unknown x has been moved out of the exponent position, solving for it becomes easy.
- (*smooth*) $\phi(\tilde{N})$ is smooth. It is well-known that the hardness of the discrete logarithm problem depends on the size of the largest prime factor of the base's order. When this order is a product of only small primes, computing discrete logarithm becomes easy.
- (*unbalanced*) $\tilde{N} = \tilde{p}\tilde{q}$ for just small $\tilde{p}$ (e.g., 256-bit $\tilde{p}$ compared to 2048-bit $\tilde{N}$). Computing discrete logarithm over $\mathbb{Z}_{\tilde{p}}$ instead of $\mathbb{Z}_{\tilde{N}}$ drastically reduces the difficulty.

[‡]Currently, we consider working over $\mathbb{Z}^*_{\tilde{N}}$. However, as $\mathbb{Z}^*_{\tilde{N}}$ can be decomposed into a direct product of smaller subgroups, working over any of these subgroups is also fine. In that case, only the projections of h_1, h_2 onto the working subgroup are of concern and it may not hold that $h_2 = h_1^e$ modulo $\tilde{N}$ (it holds over the subgroup only). For example, let $\tilde{N} = \tilde{p}\tilde{q}$, then $\mathbb{Z}^*_{\tilde{N}} = \mathbb{Z}^*_{\tilde{p}} \times \mathbb{Z}^*_{\tilde{q}}$ and one may consider working over $\mathbb{Z}^*_{\tilde{p}}$ instead of $\mathbb{Z}^*_{\tilde{N}}$. In that case, h_1 and $h_2 \bmod \tilde{q}$ are ignored, only $h_2 = h_1^e \bmod \tilde{p}$ holds.

3.1 α-shuffle Attack

3.1.1 The Flaw

In practical context, the `dlnproof` is carried out in a non-interactive manner by using the Fiat-Shamir heuristic, which requires hashing a list of integers including N, g, h, α_1, α_2, ..., α_λ (α_i is α at the i-th iteration of the proof). To achieve this, some implementations just concatenate the byte representations of the integers with some delimiter before feeding to a Cryptographic hash function as shown in Figure 2.

Let `bytes()`, `int()` denote integer-to-bytes and bytes-to-integer conversion functions respectively. Let '|' denote byte concatenation (should not be confused with '|' as 'divides' in the context of integers). Let `rand()` denote the function that returns a random element from an input set. Let $H()$ denote the vulnerable hashing implementation and D denote the byte representation of the delimiter used by H.

Recall that at the beginning of a `dlnproof`, Peggy (the prover) is required to commit to an α_i for each of the λ iterations of the corresponding interactive proof. The idea is, Peggy first commits to a stream of bytes in the format of $a|D|a|D|a|...|D|a$. When H is applied, the challenge bits c_i are determined and Peggy can then flexibly choose each α_i to be equal to $\texttt{int}(a)$ or $\texttt{int}(a|D|a)$, depending on which value causes Victor (the verifier) to output 'accept' for that iteration. Figure 4 demonstrates this idea.

3.1.2 The Attack

Algorithm 1 allows Peggy to forge a valid `dlnproof` for arbitrary g, N of his choice. The algorithm outputs h and a `dlnproof` for $\log_g h$ modulo N consisting of λ pairs of α_i, τ_i.

Note that if the condition at step 4c of Algorithm 1 holds, subsequent modification of α_i will not affect the challenge bits c_i since $H(g, h, N, \alpha_1, \alpha_2, ..., \alpha_\lambda)$ will be unaltered as long as the number of instances of a remains the same (regardless of how they are interpreted). Moreover, it can be verified that $g^{\tau_i} = \alpha_i h^{c_i}$ for all i when the algorithm returns, hence the output `dlnproof` is correct. As the list of α_i $(\alpha, \alpha, ..., \alpha, \beta, \beta, ..., \beta)$ is rearranged based on the challenge bits c_i, we name this technique α-shuffle.

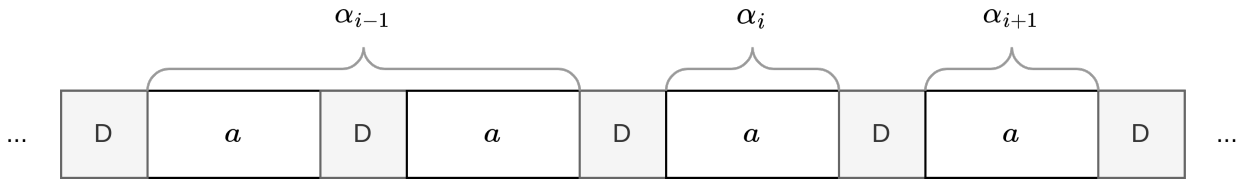

Figure 4: α_i are chosen after the challenge bits c_i are revealed.

Algorithm 1 α-shuffle `dlnproof` forging

Input: g, N.

Output: h, `dlnproof` for $\log_g h \bmod N$.

1. Let $\tau = \texttt{rand}(\mathbb{Z}_{\text{ord}(g)})$. Let $\alpha = g^\tau \bmod N$. Set all $\tau_i = \tau$.
2. Let $a = \texttt{bytes}(\alpha)$. Let $\beta = \texttt{int}(a|D|a)$.
3. Set $h = \frac{\alpha}{\beta} \bmod N$ (so that $\beta = \frac{g^\tau}{h} \bmod N$).
4. For l in $\{0,1,2,...,\lambda\}$:
   (a) Temporarily set $\alpha_i = \begin{cases} \alpha & (1 \le i \le l) \\ \beta & (l+1 \le i \le \lambda) \end{cases}$ (assign α to l first α_i and β to the remaining).
   (b) Let $c_1, c_2, ..., c_\lambda = H(g, h, N, \alpha_1, \alpha_2, ..., \alpha_\lambda)$.
   (c) If $\sum c_i = \lambda - l$ (there are l challenge bits equal to 0), set $\alpha_i = \begin{cases} \alpha & (c_i = 0) \\ \beta & (c_i = 1) \end{cases}$ and return.
5. Go back to step 1.

Now, we know how to forge a `dlnproof`. Ideally, an attacker may execute the above algorithm with $g = h_2 = 1$[§] and a discrete-log-friendly $\tilde{N}$ to obtain h_1 and a malicious `dlnproof` for $\log_{h_2} h_1$ that should be successfully verified by other TSS parties. Additionally, h_1 is likely to have large order (it is just a virtually random element in $\mathbb{Z}_{\tilde{N}}$), allowing the full MtA secret input x to be extracted from $z = h_1^x h_2^\rho \bmod \tilde{N}$. As explained in Section 2.3, this results in full recovery of the TSS private key.

3.1.3 α-shuffle in practice

This part briefly describes how to bypass different checks or policies (P) applied to $\tilde{N}, h_1, h_2$ encountered in practical Threshold ECDSA implementations.

[§]One may consider incrementing α by N ($\alpha = \alpha + N$) instead of picking another random τ in step 1 of Algorithm 1 to avoid an infinity loop in this case.

P1: $\mathbf{h_2}$ must not equal 1.

If h_2 can be wrapped around, then just let $h_2 = \tilde{N} + 1$. Otherwise, let $h_2 = \tilde{N} - 1$ (need z squared to eliminate h_2).

P2: A `dlnproof` for $\log_{\mathbf{h_1}} \mathbf{h_2}$ is also required.

Add an extra condition to ensure that Algorithm 1 will not return unless $\log_h g$ modulo $\tilde{N}$ exists. Since $\tilde{N}$ has already been chosen to be discrete-log-friendly, testing this condition should not be hard. As a result, it is feasible to compute and build an ordinary `dlnproof` for $\log_{h_1} h_2$ after forging one for $\log_{h_2} h_1$.

P3: α_i must be smaller than $\tilde{N}$.

Instead of picking a random τ and computing $\alpha = g^\tau \bmod N$ as in step 1 of Algorithm 1, one can pick a suitable α first, then compute $\tau = \log_g \alpha \bmod N$. Here, 'suitable' means that α is small enough (so that $\beta < N$ for $a = \texttt{bytes}(\alpha)$, $\beta = \texttt{int}(a|D|a)$) and τ can be successfully computed. For example, supposing that $\tilde{N} = \tilde{p}\tilde{q}$, $\tilde{N}$ is 2048-bit, $\tilde{p}$ is 512-bit and $g = h_2 = 1 \bmod \tilde{p}$, we have $1,\ p+1,\ 2p+1, ...$ are some candidates for α since they are small and $\alpha = 1 \bmod p$ is the minimum requirement for the existence of $\log_g \alpha$.

P4: The `dlnproof` hashing context must include an auxiliary input which is only determined after $\tilde{\mathbf{N}}, \mathbf{h_1}, \mathbf{h_2}$ are committed.

The probability that Algorithm 1 succeeds in only one run is:

$$P = 1 - \prod_{l=0}^{\lambda} \left(1 - \frac{{}^{\lambda}C_l}{2^\lambda}\right)$$

For $\lambda = 80$, $P \approx 0.6441$, which is also the probability that Eve, an attacker, successfully applies α-shuffle in this situtation. This is because Eve has to execute Algorithm 1 without a complete definition of H since it depends on an unkown input. To continue the attack, Eve has to break early (after step 3) and broadcast a commitment for a might-be-working triple of $\tilde{N}, h_1, h_2$. Later on when the unknown input is revealed, Eve resumes to step 4 of Algorithm 1 to check if a `dlnproof` can be successfully forged with respect to the committed triple. If

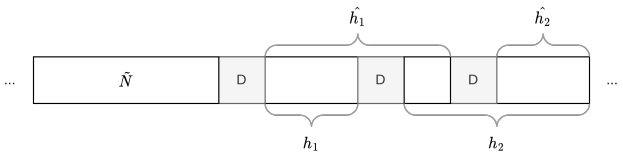

Figure 5: $\tilde{N}, h_1, h_2$ and $\tilde{N}, \hat{h_1}, \hat{h_2}$ having the same commitment.

Eve is unlucky, the key-generation ceremony has to abort and other TSS parties might be able to identify Eve as the culprit.

However, it is likely that the commitment scheme is hash-based and also depends on the vulnerable hash implementation H (module/function reuse is common in software development). If it is true, Eve will have a way to escape in case the `dlnproof` can not be forged. The idea is to reveal a different triple $\tilde{N}, \hat{h_1}, \hat{h_2}$ such that it has the same commitment as $\tilde{N}, h_1, h_2$ and $\log_{\hat{h_2}} \hat{h_1} \bmod \tilde{N}$ exists, allowing a correct `dlnproof` to be built. Figure 5 demonstrates this idea. Note that the commit function is supposed to simply return $H(L)$ in which L is a list constructed from a secret decommitment and to-be-commited values.

Since the input $g = h_2$ can be freely chosen, putting a delimiter inside its byte representation is usually not a problem. However, if $0 \le h_1, h_2, \hat{h_1}, \hat{h_2} < \tilde{N}$ is required, the uncontrolled output $h = h_1$ will also need to be small enough so that $\hat{h_1} < \tilde{N}$ can be satisfied[¶]. This can be achieved by carefully choosing α, β such that $b = \texttt{bytes}(\beta)$, $\alpha = \texttt{int}(b|D|b)$ and $\beta \,|\, \alpha$ beforehand[‖]. As a result, $h = \frac{\alpha}{\beta}$ is small regardless of $\tilde{N}$.

To sum up, whenever Eve fails to complete an attack, she can safely fallback to the usual workflow for not being detected while trying to trigger (or simply wait for) another key generation or re-sharing ceremony to conduct another attack. Eve may keep repeating this process until success. The expected number of failures is:

$$\sum_{\nu=0}^{+\infty} \nu(1-P)^{\nu} P = \frac{1-P}{P}$$

For $\lambda = 80, P = 0.6441$, this value is approximately 0.5526.

3.1.4 Mitigation

To mitigate α-shuffle attack, it is necessary to adopt a non-ambiguous encoding scheme to construct the list-of-integer hashing function H. There should not exist two different lists encoded into the same byte sequence. A rule of thumb is that if you do not have a deterministic way to decode the byte sequence (back to the original list of integers), you are likely doing it wrong. A simple fix is to always include the length of each integer. Other popular encoding schemes like Protocol Buffers (Protobuf [5]), Abstract Syntax Notation One (ASN.1 [6]) or Tag-Length-Value (TLV [7]) are also fine.

It is important to note that changing the way H works requires an update to all existing systems and software in a TSS group. However, the potential security benefits that result from implementing such measures are substantial, and can help to protect against the risks associated with hash collision.

3.2 c-split Attack

3.2.1 The Flaw

The `dlnproof` requires repeatedly generating random challenge bit $c \in \{0,1\}$. However, instead of repeating the interactive proof with binary challenge $c_i \in \{0,1\}$, the implementation decide to use a much larger challenge set (e.g., all possible outputs of SHA-256 or $\mathbb{Z}_{2^{256}}$) in only one run. It turns out that a larger challenge set does not result in a better soundness error ($< \frac{1}{2}$). Let $g \in \mathbb{Z}_N^*$, $h = g^2$ and $2 \,|\, \text{ord(g)}$. Since 2 has no inverse modulo ord(g), $\log_h g = \frac{1}{2}$ does not exist. However, when $2 \,|\, c$ (probability $\frac{1}{2}$), Peggy is able to forge a correct `dlnproof` for it by having $\tau = \rho + \frac{c}{2}$.

3.2.2 The Attack

The attack is straightforward. Let e be a small divisor of $\text{ord}(h_1)$ (recall that $h_2 = h_1^e \bmod \tilde{N}$). When building a `dlnproof` for $\log_{h_2} h_1$, one can just keep brute-forcing ρ until c is divisible by e (probability $\frac{1}{e}$). This will not be hard if e is, say, only 32-bit long. Consequently, $(\alpha, \tau) = (h_2^{\rho} \bmod \tilde{N}, \rho + \frac{c}{e})$ is a valid `dlnproof` for the nonexistent $\log_{h_2} h_1$. Since $e \,|\, c$ is required to forge the proof, we name the technique c-split.

However, it is not quite done yet since only $x \bmod e$ for some secret MtA input x (k_i,γ_i or w_i) can be recovered during a signing ceremony while e cannot be made arbitrarily large or the brute-forcing step above becomes infeasible. To fully recover the TSS private key, some extra work is needed.

Firstly, given a message-signature pair $(m, (r,s))$, one can leverage the leaked information (k_i, γ_i, $w_i \bmod e$ for $1 \le i \le t$, recall that t is the number of TSS parties participating in the signing ceremony) to obtain the following equation:

$$a\bar{w} = b + \bar{\gamma} \bmod q \tag{1}$$

In which solving for the small unknowns $\bar{w}, \bar{\gamma}$ (bounded by $\lfloor \frac{tq}{e} \rfloor$) would give the TSS private key[**].

[¶] Another workaround is to choose small input $g = h_2$, then brute-force the first 3 steps of Algorithm 1 until a delimiter appears in the byte representation of the output $h = h_1$. However, this approach is only suitable for short delimiters.

[‖] The correctness of Algorithm 1 does not rely on whether $\beta = \texttt{int}(a|D|a)$ (as in the original version) or $\alpha = \texttt{int}(b|D|b)$ (as in the version being described).

[**] Deriving (1) from the ECDSA formula $s = k(m + rd) \bmod q$ requires some arithmetic:

1. Multiply both sides by γ to separate the product of 2 unknowns kd (recall that $\delta = k\gamma$ is a public value).

And also:

$$\bar{w}P = Q \tag{2}$$

For some elements P, Q of the ECDSA group[††].

Next, multiple pairs of message-signature must be collected. Supposing that l pairs are given:

$$a_j\bar{w} = b_j + \bar{\gamma}_j \bmod q \text{ for } 1 \leq j \leq l \tag{3}$$

Note that all w_j need not be the same since each depends on the set of participating parties that may change for each signing ceremony. However, they all equal d mod q. Therefore, a single $\bar{w}$ is used for all j to reflect this fact.

Rewriting (3) as a vector equation, one obtains (4).

This is exactly an instance of the lattice enumeration problem described in Section 2.5. The left-hand side columns are basis vectors for a $(l+1)$-dimensional lattice. The required condition on the result vector v is that its first entry $\bar{w}$ must satisfy (2). Since $0 \leq \bar{w}, \bar{\gamma}_1, \bar{\gamma}_2, ..., \bar{\gamma}_l \leq \lfloor \frac{tq}{e} \rfloor$, the searching region is in fact a hypercube. Its edge has a length of $\lfloor \frac{tq}{e} \rfloor$ and $(0, b_1, b_2, ..., b_l)$ is one of its vertices.

Note that the equation (2) makes it very straightforward to apply MITM in this case. After the lattice basis is reduced with LLL, let ω_i denote the first entry of the new basis's i-th vector, then $\bar{w} = \sum c_i\omega_i$. Let $W_i = \omega_i P$, from (2) we have $\sum c_i W_i = Q$. Decomposing the search space is now as simple as shifting some $c_i W_i$ from the sum on the left-hand side to the right-hand side. To make the two subspaces balanced, one may consider splitting a single c_i, say $c_1 = c_a m + c_b$ for a suitable m, then repartitioning the set of c_i (has now become $\{c_a, c_b, c_2, c_3, ...c_{l+1}\}$) accordingly[‡‡].

The only remaining question is how many message-signature pairs are required (i.e., the minimum value for l) to solve for $\bar{w}$. The answer is that it depends on how much computational power an attacker has or is willing to pay for. With t fixed to 32[§§], Table 2 gives the search space size ($|S|$) under various configuration for e and l. From the cell corresponding to $(e,l) = (2^{56}, 2)$, it can be understood that if the attacker is willing to construct 2^{56} different `dlnproof` challenges and perform about $2 \times 2^{\frac{104.71}{2}}$ elliptic curve group additions, then the TSS private key can be recovered with only 2 signatures.

$$\bar{w}\begin{bmatrix}1\\a_1\\a_2\\...\\a_l\end{bmatrix} + u_1\begin{bmatrix}0\\q\\0\\...\\0\end{bmatrix} + u_2\begin{bmatrix}0\\0\\q\\...\\0\end{bmatrix} + ... + u_l\begin{bmatrix}0\\0\\0\\...\\q\end{bmatrix} = \begin{bmatrix}0\\b_1\\b_2\\...\\b_l\end{bmatrix} + \begin{bmatrix}\bar{w}\\\bar{\gamma}_1\\\bar{\gamma}_2\\...\\\bar{\gamma}_l\end{bmatrix}, u_j \in \mathbb{Z} \tag{4}$$

3.2.3 c-split in practice

The c-split exploitation technique should work well in practice. One can even choose a small e, say 32-bit long, since the required number of signatures is usually not a practical issue. The payload of the attack ($\tilde{N}, h_1, h_2$ and `dlogproof` for $\log_{h_2} h_1$) looks statistically the same as a legit one.

However, there is a case in which Bob respondent range proof in MtA is omitted. As a result, an attacker no longer has access to γ_i and w_i mod e for each signing ceremony. Note that omitting Bob respondent proof comes with its own issues such as unidentifiable abort, i.e., a malicious party may corrupt the signing ceremony without being detected.

In this situation, with enough malicious cooperating parties, it's possible to recover the TSS private key with just one signature. The idea is to have a pairwise relatively prime set $\{e_1, e_2, ..., e_l\}$ (the e values from l malicious parties) so that k_i mod $e_\pi = e_1 e_2 ... e_l$ can be recovered by applying the Chinese remainder theorem (CRT) to k_i mod e_j for $1 \leq j \leq l$. Now, the malicious parties need to solve for k satisfying:

2. Compute $(\bar{\gamma}, \bar{w}) = (\gamma, w)$ mod e by summing up γ_i and w_i mod e respectively.

3. Substitute $\gamma = \bar{\gamma}e + \tilde{\gamma}$ and $d = w = \bar{w}e + \tilde{w}$ mod q into the formula.

Note that a, b are just aliases for $\frac{\delta r}{s}$ and $\frac{\tilde{\gamma}s - \delta m - \delta r\tilde{w}}{es}$ mod q respectively. Since $\gamma, w < tq, \bar{\gamma}, \bar{w} \leq \lfloor \frac{tq}{e} \rfloor$. If $\bar{w}$ is determined, the TSS private key $d = \bar{w}e + \tilde{w}$ mod q can also be recovered.

[††]Let G denote the group generator defined by the ECDSA signature scheme and $D = dG$ denote the public key corresponding to d, then $(\bar{w}e + \tilde{w})G = D$. Therefore, P and Q are just aliases for eG and $D - \tilde{w}G$ respectively.

[‡‡]This idea is similar to the baby-step-giant-step (BSGS) technique.

[§§]There is a known scalability issue with the TSS design as it has time and communication complexity of $O(t^2)$. $t = 32$ is a quite high value in practice.

e $\backslash l$	1	2	3	4	5	6	7	8
2^{32}	202.48	177.24	151.26	126.03	100.40	76.13	51.57	27.56
2^{40}	186.62	152.43	119.23	85.72	52.61	20.06	0.00	0.00
2^{48}	171.01	128.52	87.74	46.29	1.58	0.00	0.00	0.00
2^{56}	154.39	104.71	54.78	4.75	0.00	0.00	0.00	0.00
2^{64}	138.40	80.74	22.96	0.00	0.00	0.00	0.00	0.00

Table 2: The search space size on the log2 scale for different (e,l). t is always 32. The data are generated by simulating the signing and exploiting process with SageMath v9.5 [8]. Each test is repeated 1000 times for increased reliability.

$$\begin{cases} \tilde{k} = k \bmod e_\pi \text{ is known} & \text{(by summing up } k_i \bmod e_\pi\text{)} \\ kR = G & \text{(since } k^{-1}G = R\text{)} \end{cases} \tag{5}$$

Similar to Section 3.3, let $\bar{k} = \left\lfloor \frac{k}{e_\pi} \right\rfloor$ ($k = \bar{k}e_\pi + \tilde{k}$), $P = e_\pi R$ and $Q = G - \tilde{k}R$, then the rest of the attack is about searching for $\bar{k} < \left\lfloor \frac{tq}{e_\pi} \right\rfloor$ such that $\bar{k}P = Q$. [9] is a very well-optimized tool to tackle this kind of problem. It has been used to crack a 114-bit secp256k1 private key for a reward of 1.15 BTC, so one can expect 1.15 BTC to be a reliable upper bound on the cost of solving for $\bar{k}$ when $\left\lfloor \frac{tq}{e_\pi} \right\rfloor \leq 2^{114}$ (assume that the curve in use is also secp256k1).

Again, the only remaining question is how many malicious parties are required. $l = 3$ is quite practical, while $l = 2$ is also possible but the attack will be costly. In practice, making a decision on l should depend on many factors such as the benefit from a successful attack, the cost to become or corrupt a member of the targeted TSS group, ...

3.2.4 Mitigation

In a proof system, the most important factor to determine the number of proof iterations is its soundness error. Deciding to not repeat a proof without proving that the proof has a negligible soundness error is like taking a crazy risk.

It is therefore highly advised that protocols are implemented in compliance with their specifications. Any attempt to optimize the implementation without a proper understanding of its security implications should be avoided, as this might create vulnerabilities exploitable by attackers.

3.3 c-guess Attack

3.3.1 The Flaw

In `dlnproof`, each proof iteration requires the verifier to send a binary challenge $c_i \in \{0,1\}$. The probability for a successful guess on all c_i is $\frac{1}{2^\lambda}$ for λ iterations. It is important to note that applying Fiat-Shamir heuristic allows the prover to not be punished for making an incorrect guess. Therefore, when λ is low enough with regards to the computational power of a malicious prover, the prover can then repeat the proof generation until a correct guess for all c_i is found, thus successfully forge a `dlnproof`.

3.3.2 The Attack

The attack is straightforward: the malicious prover chooses random challenge bits c_i and applies H (a hash function according to the Fiat-Shamir transformation) to the corresponding payload prepared for those guessed c_i to check if the actual output challenge bits are the same as the guessed ones. If they are not, the prover simply makes another guess and retries until a correct one is made. The expected number of trials is 2^λ.

3.3.3 Mitigation

As discussed in 2.2, the `dlnproof` requires a minimum number of iterations to achieve a certain level of security. CGGMP21 recommends the number of iterations to be at least 80 which should be enforced by all implementations.

4 Impact Factors

In this section, we will provide a comprehensive summary of our study's key findings. We will begin by recapping the implementation flaws that enabled our attacks on the `dlnproof` protocol. We will then discuss the detrimental impact of our attacks. Finally, we will provide an estimate of the funds that were saved due to our swift actions in contacting product owners. By highlighting the critical importance of thorough testing and robust security measures in threshold cryptography, this section underscores the significance of our research.

4.1 Weaknesses

During our research, we identified several implementation flaws of `dlnproof` that can be exploited, such as hash collisions resulting from concatenating hash values. To avoid this issue, we recommend using a non-ambiguous encoding

scheme. Moreover, reducing the number of rounds in the protocol can significantly undermine its security. It is imperative to follow the protocol specifications and perform an adequate number of binary challenges, which is 80 as specified in CGGMP21.

4.2 Key Recovery

The attacks we presented in this paper can rebuild the ECDSA private key, despite it not being computed in the Threshold ECDSA protocol. This is achievable because all private values that pass through the MtA protocol can be recovered through our attacks. These values play a significant role in the logic of computing ECDSA signatures. With a series of signatures and secret values, an attacker can quickly re-compute the private key.

This has significant implications for the security of Threshold ECDSA if implemented wrongly, as the reconstruction of the private key would allow an attacker to forge arbitrary signatures and gain complete control over the system. As such, it is critical to identify if the implementation are vulnerable to our attacks and take measures as we suggest to enhance the security of the library.

4.3 Impacts

In our study, we developed proof-of-concept attacks on various open-source projects that implement Threshold ECDSA. Our PoCs demonstrated that, in most cases, a single malicious party and one signing ceremony are enough to recover the private key. The details of the affected implementations can be found in Table 3. In addition to the PoC attacks on open-source projects implementing Threshold ECDSA, we also created demonstrations that successfully withdraw all funds from a deployed development environment. These simulations serve to illustrate the potential impact that such attacks could have on a production level.

5 Conclusions

This paper introduces new attack vectors leveraging common implementation flaws of Threshold ECDSA, a new cryptographic protocol. The paper warns of the dangers of relying on untested code and highlights the gap that often exists between design and implementation. It emphasizes that modifications made to the design to meet specific needs can lead to serious bugs and that the implementation may not always be a direct representation of the design. The paper stresses the importance of thorough testing of implementations to guarantee the security and stability of the technology in use. The findings of this paper have significant consequences for various vendors who use Threshold ECDSA, which is widely adopted in production environments. Our disclosure may have prevented a potential major hack in the blockchain industry.

Implementations	Attack Technique	PoC	Required number of		
			Malicious parties	(Re)sharing ceremonies	Signing ceremonies
Axelar (tofn)	c-split	YES	1	1	2
Binance/BNBChain (tss-lib)	α-shuffle	YES	1	1	1
ING Bank (threshold-signatures)	c-split	YES	1	1	2
Keep Network/Threshold Network	α-shuffle	YES	1	1	1
Multichain (fastMPC)	α-shuffle	YES	1	1	1
	c-guess	YES	1	1	1
Swingby (tss-lib)	α-shuffle	YES	1	1	1
Taurus (multi-party-sig)	α-shuffle	YES	1	1.5526	1
Thorchain (tss-lib)	α-shuffle	YES	1	1	1
ZenGo X (multi-party-ecdsa)	c-split	YES	2	1	1

Table 3: Affected implementations.

The table above was calculated based on the assumption that the attacker could practically break 64-bit security.
This table was last updated on March 2023. For the updated table, please visit verichains.io/tsshock.

References

[1] Rosario Gennaro and Steven Goldfeder. Fast Multiparty Threshold ECDSA with Fast Trustless Setup. Cryptology ePrint Archive, Paper 2019/114, 2019. https://eprint.iacr.org/2019/114.

[2] Rosario Gennaro and Steven Goldfeder. One Round Threshold ECDSA with Identifiable Abort. Cryptology ePrint Archive, Paper 2020/540, 2020. https://eprint.iacr.org/2020/540.

[3] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts. Cryptology ePrint Archive, Paper 2021/060, 2021. https://eprint.iacr.org/2021/060.

[4] Dmytro Tymokhanov and Omer Shlomovits. Alpha-rays: Key extraction attacks on threshold ecdsa implementations. Cryptology ePrint Archive, Paper 2021/1621, 2021. https://eprint.iacr.org/2021/1621.

[5] Protobuf. https://protobuf.dev/.

[6] ASN.1. https://en.wikipedia.org/wiki/ASN.1.

[7] TLV. https://en.wikipedia.org/wiki/Type%E2%80%93length%E2%80%93value.

[8] Sage Math. https://www.sagemath.org/.

[9] Kangaroo. https://github.com/JeanLucPons/Kangaroo.