Alpha-Rays: Key Extraction Attacks on Threshold ECDSA Implementations

Dmytro Tymokhanov
Velas
dmytrotym1@gmail.com

Omer Shlomovits
ZenGo-X
omer.shlomovits@gmail.com

Abstract

In this paper we provide technical details on two new attack vectors, relevant to implementations of [GG18] and [GG20] threshold ECDSA protocols. Both attacks lead to a complete secret key extraction by exploiting different parts of the Multiplicative-to-Additive (MtA) sub-protocol the parties run during signing. Our first attack applies to the setting of "fast" MtA, which runs the protocol with no range proofs. We leverage a powerful oracle, much stronger than originally anticipated in [GG18], to reveal a part of the secret key with each signature we run. The number of required signatures depends on the implementation under attack and the number of parties controlled by the attacker. Our proof of concept demonstrates a full key extraction by a single malicious party using eight signatures. Our second attack deals with the more common setting of "full" MtA, that is, including ZK proofs. The only requirement for mounting a successful attack is to use a small Paillier encryption key. The key size check was not specified in the protocol and therefore missing from most existing threshold ECDSA implementations, making them vulnerable. As we show, choosing a small key completely eliminates a specific hiding property in one of the values sent from the victim to the attacker during one of ZK proofs. This allows a single malicious party to extract the full secret key after a single valid signature. We provide a proof of concept for this attack as well.

1 Introduction

In multiparty computation (MPC), a set of distrusting parties jointly run a computation without exposing their private inputs. MPC has been systematically studied for decades and recently began to see major adoption in the industry, contributing to areas of privacy and security[1].

A Threshold Signature Scheme (TSS) is an MPC that enables a group of parties to collectively compute a digital signature without learning any information about the private key. Due to the rising popularity of cryptocurrencies

[1]see www.mpcalliance.org for list of companies adopting MPC

and blockchains which heavily rely on digital signatures, TSS, which offer superior security properties in some settings, have seen a surge in interest by the research community over the past few years. Most notably, solutions for threshold ECDSA, the signing scheme used in Bitcoin and many other popular coins, made a meteoric leap in their practicality, making threshold ECDSA usable for several real-world applications. In this work, we investigate one construction for threshold ECDSA, originally introduced by Gennaro and Goldfeder in 2018 [GG18], which was one of the first protocols to enable efficient threshold ECDSA in the dishonest majority setting. The protocol was later improved [GG20] to support identifiable aborts and one round online signing. Our attack surface concerns the Multiplicative-to-Additive sub-protocol, which is used in both versions, as well as in protocols [CGG$^+$21, FLOP18]. Due to its unique combination of properties, the GG protocol is arguably the most frequently used threshold ECDSA protocol nowadays. A growing number of maintained open-source implementations exist for it, spanning some of the most prominent products in the blockchain industry. For more details about existing libraries as well as competing constructions, we refer to a recent survey [AHS20].

1.1 Related work

Prior to this work, several bugs have been reported on threshold ECDSA implementations in the wild, as summarized in [AS20]. Our second attack follows a similar pattern to the *Golden Shoe* attack from that paper: Some unverified protocol parameters are sent to the victim. These parameters are corrupted but this goes undetected. The parameters are later used by the victim in a ZK proof, revealing the secret key to the attacker. Both attacks require the attacker to control a single malicious party during the distributed key generation and one run of signing. Both attacks are covert in the sense that they produce a valid signature. Deep inspection however reveals that our attacks are in fact different in nature. While previous attacks were enabled by wrong practice of crypto-engineering, i.e. buggy code or misuse, both of our attacks can be traced to erroneous analysis in [GG18] protocol.

1.2 Summary of our attacks

As part of a crypto-system, threshold ECDSA is a low-level primitive for distributing digital signing between different signers. We usually define two parameters: n — for the total number of parties, and t — for some threshold smaller than n. Only $> t$ honest signers will be able to output a valid signature, while any group of t parties and less will not be able to forge a signature[2]. The threshold assumption in a given crypto-system is that an attacker will never control more than t parties at the same time, and hence will never be able to forge a signature. The following is an overview of our attacks and their implications for different implementations:

[2]in the rest of the paper n denotes the number of signers, and not the number of parties that generate keys.

- Our first attack deals with the case where the Multiplicative-to-Additive (MtA) sub-protocol is implemented using the "fast" option, without range proofs. Due to the missing range proof over the attacker's input, we give the attacker the ability to count how many times the victim's output was reduced modulo N. This is done by testing the equality $a \cdot b = \alpha + \beta$ in the exponent. If the attacker carefully crafts their nonce, they are able to focus the search on different bits of the victim's secret key with each signature, until finally revealing the full secret key. In our proof of concept, we extract the key using 16 signatures, but show that it's doable with just 8. ZenGo [Zen] and ING [ING] libraries were found susceptible to this attack.

- Since the malicious party uses it's nonce to control the window and therefore the specific location of the bits it learns, the attack can be improved if the attacker controls multiple parties. For example, for $n = 17$ parties and threshold $t = 8$, if the signing is run with 9 parties, 8 of which are controlled by the attacker, we can learn simultaneously 8 29-bit segments of the ninth honest party. For 256-bit secret key, after a small exhaustive search, this will allow us to learn the full secret key within a single signature.

  It's important to note that signatures generated as part of this attack will fail verification, or abort during the protocol execution. For some of the libraries we looked into, we were able to produce one valid signature as part of the attack.

- Our second attack vector is dealing with the case of running the expensive version of the MtA sub-protocol, where range proofs are included. Conditioned on missing the size check on Paillier encryption key N, simple computation allows the attacker who knows a, α in the equation $a \cdot b = \alpha + \beta$ to learn β and deduce b, victim's secret key. The trick is to choose N to be close in size to $|e|$, where e is the public challenge. Thus, 256-bit N is chosen. This choice will immediately give the attacker β. This means that the attacker controlling a single party can extract the secret key of all honest parties using a single successful (hence covert) signature. We found tss-lib [Bin] and all its forks [Tho, Swi, Kee, Any] susceptible to this attack.

2 Attacks on multiplicative-to-additive protocol

2.1 Preliminaries

MtA protocol. A principal problem in designing a threshold ECDSA protocol is how to multiply two secretly shared values. In protocols [GG18] and [GG20] multiplicative-to-additive sub-protocol (MtA for short) is used. It is described in section 3 of [GG18], but we outline it here for completeness:

Suppose that two parties, Alice and Bob hold multiplicative shares of a secret value $x \in \mathbb{Z}_q$. That is, Alice holds $a \in \mathbb{Z}_q$ and Bob holds $b \in \mathbb{Z}_q$ such that $x = ab \mod q$. The parties want to obtain additive shares of x, i.e. $\alpha \in \mathbb{Z}_q$ (known only to Alice) and $\beta \in \mathbb{Z}_q$ (known only to Bob) such that $x = \alpha + \beta \mod q$. This problem is in fact common in MPC protocols and several techniques can be found in the literature to solve it. In the GG protocols, the authors use an additive homomorphic encryption scheme, instantiated using Paillier cryptosystem. This choice is unimportant for our attack. It is assumed that Alice is associated with a public key E_A of an additively homomorphic encryption scheme over modulus N (i.e. plaintexts in this scheme are elements of $\mathbb{Z}_N$). K is a bound that will be specified later. The MtA protocol works as follows:

1. Alice starts the protocol:
   - Encrypt a with own homomorphic key: $c_A = E_A(a)$
   - Compute a ZK range proof π_A that $\{\exists a' : D_A(c_A) = a' \wedge a' < K\}$
   - Send (c_A, π_A) to Bob
2. After receiving (c_A, π_A) from Alice, Bob does the following:
   - Verify π_A and if it fails to verify, abort immediately
   - Choose β' uniformly at random from $\mathbb{Z}_N$
   - Set own result β to be $-\beta' \mod q$
   - Compute $c_B = b \times_E c_A +_E E_A(\beta')$ where $\times_E$ and $+_E$ are homomorphic operations provided by the encryption scheme
   - Compute a ZK range proof π_B that $\{\exists b', \beta' : c_B = b' \times_E c_A +_E E_A(\beta') \wedge b' < K\}$
   - Send (c_B, π_B) to Alice
3. After receiving (c_B, π_B) from Bob, Alice does the following:
   - Verify π_B and if it fails to verify, abort immediately
   - Set own output α to be $D_A(c_B) \mod q$

If the protocol is successfully completed, Alice's α is equal to $ab + \beta' \mod N$ (assuming parties used correct values of a and b). If $ab + \beta' < N$, the reduction modulo N is never executed, and we have that $ab + \beta' = \alpha$ as integers. This guarantees that $ab \equiv \beta + \alpha \mod q$ as desired. However, if the reduction modulo N happens, this equality no longer holds. This is the reason range proofs are used in the protocol, and the proofs used in [GG18] and [GG20] require that $K \sim q^3$. As a result, in order to statistically prevent reduction modulo N, N has to be chosen at least of order q^7. In practice, parameter q is often the order of Secp256k1 curve, which is 256 bits in size, and N if often chosen to be 2048-bit, which is the current industry standard, equivalent to a security level of 128 bits.

MtA with check. As mentioned above, MtA provides no way of checking whether parties used correct values of a and b during protocol execution. However, if commitments to these values are publicly known, the MtA protocol can be augmented to perform a check against these commitments. Here we describe the so-called MtA with check protocol (MtAwc for short) used in [GG18] and [GG20][3]:

Consider a group of points on an elliptic curve $\mathcal{G}$ and its subgroup of order q and a publicly agreed upon generator g. Suppose that $B = g^b$ is public knowledge where b is Bob's secret multiplicative share. Then the following modifications are made to the MtA protocol to ensure that b used in it is equal to $\log_g B$:

- Bob additionally computes and sends to Alice a ZK proof of knowledge (ZKPoK) of b: $\{\exists b', \beta' : c_B = b' \times_E c_A +_E E_A(\beta') \wedge g^{b'} = B\}$
- After receiving Bob's message, Alice verifies this proof of knowledge and aborts if it does not verify

MtAs in [GG18] and [GG20]. In the setting of the two papers, signing is performed by n parties: $\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_n$. They use additive secret sharing of the following values, where party $\mathcal{P}_i$ holds the i'th term (also often called share):

1. $x = w_1 + w_2 + \cdots + w_n$ is a ECDSA private key
2. $k = k_1 + k_2 + \cdots + k_n$ is a one-time nonce
3. $\gamma = \gamma_1 + \gamma_2 + \cdots + \gamma_n$ is a field element used for blinding, such that the computations on k are performed without revealing it.

All computations are done modulo q, and individual shares of all three values are sampled uniformly at random from $\mathbb{Z}_q$.

In real-world applications, the same private key x is often reused multiple times, but k and γ are freshly uniformly sampled random values per each signature.

During one signing, every pair of parties engage in one MtA protocol and one MtAwc protocol[4]. Parties $\mathcal{P}_i$ and $\mathcal{P}_j$ obtain additive shares of $k_i \cdot \gamma_j$ in the course of MtA and additive shares of $k_i \cdot w_j$ in the course of MtAwc. In the latter, a check is performed against the publicly known value of g^{w_j}.

Removal of range proofs from MtA. Both MtA and MtAwc are proven to be statistically zero-knowledge via a simulation argument [GG18]. However, range proofs over Paillier encryption are heavy, so the authors propose the following simplification of MtA and MtAwc protocols:

[3]multiplicative group notation is used here and throughout the paper for elliptic curve points.

[4]to be clear, we mean ordered pairs of parties here. For example, parties $\mathcal{P}_1$ and $\mathcal{P}_2$ engage in 4 conversions: MtA for k_1 and γ_2, MtA for k_2 and γ_1, MtAwc for k_1 and w_2, MtAwc for k_2 and w_1.

- Alice and Bob drop the range proofs from the protocol.
- In MtAwc, Bob does not prove the correspondence between g^b and his multiplicative share, used in the protocol.
- Instead, both in MtA and MtAwc, Bob reveals $B = g^b$ and $B' = g^{\beta'}$ to Alice, proving the knowledge of $\log_g B$ and $\log_g B'$. Next, Alice verifies the proofs, aborting if they do not verify, and checks if $g^\alpha = B^a B'$. If parties run MtAwc, Alice also verifies if B is equal to the publicly known value of g^b.

The authors conjecture that this version of the protocol leaks almost no information. Quoting section 5 of [GG18]:

> [If range proofs are removed] *the MtA protocol needs to be secure in the presence of an oracle that tells the parties if a reduction* $\bmod N$ *happens during the execution. Note that in reality, the oracle represents the failure of the verification of the signature generated by the protocol, and if that happens the system is reset. So the oracle is a very weak oracle, which stops working the moment it tells you that a reduction* $\bmod N$ *happened.*

The authors assume that after an unsuccessful signature, the private key x is no longer used, which seems to be an impractical assumption. If x is a private key associated with a cryptocurrency wallet, a bare minimum of one more signature is required to move funds from the threatened wallet. And in practice users of threshold ECDSA often do not change the secret key in response to a failed signature at all [Tho].

Our main observation concerns the power of the oracle that is available to an attacker controlling Alice after range proofs are removed. Consider an example when the attacker (here and throughout the paper we will assume it is $\mathcal{P}_1$) chooses $k_1 = \left\lfloor \frac{2N}{q} \right\rfloor$ (computation is over the integers) and uses this value in MtAwc as Alice (as mentioned above, k_1 plays the role of a in MtA and MtAwc for party $\mathcal{P}_1$). Remember that in the simplified version of MtA and MtAwc with party i, the attacker receives g^{w_i} and $g^{\beta'_{i,1}}$ where by $\beta'_{i,1}$ we denote β' from the original description of MtA. Denote the value that the attacker decrypted as $\alpha_{i,1}$, which, as mentioned above, will equal $w_i \cdot k_1 + \beta'_{i,1} \mod N$. There are different options for the value $\alpha'_{i,1} = w_i \cdot k_1 + \beta'_{i,1}$, computed over the integers and unknown to the attacker:

- $\alpha'_{i,1} \in [0; N)$. Thus, $\alpha'_{i,1} \mod q = \alpha_{i,1} \mod q$ which yields:

  $$g^{w_i \cdot k_1} \cdot g^{\beta'_{i,1}} = g^{\alpha_{i,1}}$$

  i.e. MtA protocol was executed successfully, without modulo N reduction.

- $\alpha'_{i,1} \in [N; 2N)$. Thus, $\alpha'_{i,1} \mod q = \alpha_{i,1} + N \mod q$ which yields:

  $$g^{w_i \cdot k_1} \cdot g^{\beta'_{i,1}} = g^{\alpha_{i,1}} \cdot g^N$$

- $\alpha'_{i,1} \in [2N; 3N)$. Thus, $\alpha'_{i,1} \mod q = \alpha_{i,1} + 2N \mod q$ which yields:

$$g^{w_i \cdot k_1} \cdot g^{\beta'_{i,1}} = g^{\alpha_{i,1}} \cdot g^{2N}$$

We can see how the attacker can easily distinguish between the three options, as opposed to distinguishing between the first, successful option, and the rest as claimed in [GG18]. We think the authors might have missed the fact that in their version without range proofs, a party playing as Alice additionally gains access to g^b and $g^{\beta'}$. This implies that the attacker can recover much more information about the size of $\alpha'_{i,1}$ (and, subsequently w_i — party $\mathcal{P}_i$'s secret) than previously assumed.

2.2 Attack on absent range proofs

Hopefully, it is clear how the pattern presented above continues if larger values of k_1 are chosen by the adversary. Specifically, if the attacker chooses $k_1 = \left\lfloor \frac{MN}{q} \right\rfloor$ for some integer $M \in [1; q]$, $\alpha'_{i,1}$ will belong to one of $M + 1$ disjoint intervals of the form $[sN; (s+1)N)$ for $s \in \{0, 1 \dots M\}$. In other words, for exactly one value of s the following equation will hold:

$$g^{w_i \cdot k_1} \cdot g^{\beta'_{i,1}} = g^{\alpha_{i,1}} \cdot g^{sN} \tag{1}$$

Thus, the attacker chooses M and then has access to an oracle that tells them $s \in \{0, 1 \dots M\}$ such that $\left\lfloor \frac{MN}{q} \right\rfloor w_i + \beta'_{i,1} \in [sN; (s+1)N)$. Considering that $\beta'_{i,1}$ is chosen from $\mathbb{Z}_N$, the attacker learns that $w_i \in \left[\frac{(s-1)q}{M}; \frac{(s+1)q}{M}\right]$[5] if $s > 0$ and $w_i \in \left[0; \frac{q}{M}\right]$ if $s = 0$.

This leads us to the algorithm 1 for the attacker. Each j in the outer **for**-loop is one signature (either using protocol [GG18] or [GG20]). The attacker $\mathcal{P}_1$ executes signing protocol correctly except for the choice of k_1 which is given by the algorithm on each iteration. $\text{MtAwc}_i(k_1)$ denotes running MtAwc with party $\mathcal{P}_i$ and receiving the result, which is a part of the protocol execution.

The intuition behind the algorithm is that in the first signature, the attacker learns the most significant bits of everyone's secret. Then, adjusting the range that is searched with that knowledge, they are able to recover subsequent bits in the second signature by choosing larger k_1, and so on. The first signature can even complete successfully because the attacker can recover their "correct" additive shares after both MtA and MtAwc. In subsequent signatures, however, we see no way to do this after MtA, as it is run for a very large k_1 and random γ_i. So, in our attack, these signatures fail.

To keep the attacker's computation efficient, a reasonable value of M might be 2^{29}, leading to the complete leakage of the secret in just 8 signatures. This is assuming the signers who participate are the same each time so that w_i's do not change. Alternatively, 8 signers can recover the shared secret key in just 1 failed signature provided there are at least 8 signers in total.

[5]can be shown using inequalities $\frac{MN}{q} - 1 < \left\lfloor \frac{MN}{q} \right\rfloor \leq \frac{MN}{q}$ and $N > q^2$.

```
Data: $M \in \mathbb{N}$ ;                              /* Attacker's "step" */
Result: $x = \sum_{i=1}^{n} w_i$ ;     /* Result is the shared secret key */
$W \leftarrow [g^{w_2}, g^{w_3}, \ldots, g^{w_n}]$ ;          /* List of parties' public keys
 computed during keygen protocol */
$G \leftarrow [g^0, g^N, g^{2N}, \ldots, g^{2MN}]$ ;     /* List of pre-computed values.
 */
$w_{min} \leftarrow [0, 0, \ldots, 0]$ ; /* List of length $n-1$ where the attacker
 tracks their knowledge about minimum possible values of
 each honest party's $w_i$.  Naturally, initialized by zeros */
for $j \leftarrow 1$ to $\lfloor \log_M q \rfloor$ do
    $k_1 \leftarrow \left\lfloor \frac{2 \cdot M^j \cdot N}{q} \right\rfloor$ ;            /* We need $\lfloor \log_M q \rfloor$ signatures  */
    for $i \leftarrow 2$ to $n$ do
        $(\alpha, B, B') \leftarrow \mathrm{MtAwc}_i(k_1)$ ;  /* The output of MtAwc with $\mathcal{P}_i$
         in which the attacker plays Alice */
        $k_{min} \leftarrow \left\lfloor \frac{k_1 \cdot w_{min}[i-2]}{N} \right\rfloor \cdot N$ ;    /* The attacker is sure that
         the value of $k_1 \cdot w_i$ must be at least $k_1 \cdot w_{min}[i-2]$.
         Thus, the search starts from there */
        $h \leftarrow B^{k_1} \cdot B' \cdot g^{-\alpha} \cdot g^{-k_{min}}$;
        for $s \leftarrow 0$ to $2M$ do
            if $h == G[s]$ then
                $w_{min}[i-2] \leftarrow w_{min}[i-2] + \left\lfloor \frac{s \cdot q}{2 \cdot M^j} \right\rfloor$;
            end
        end
    end
end
$x \leftarrow w_1$;    /* At this point, to find $w_i$, the attacker has to
 search less than $M$ values, which they can do */
for $i \leftarrow 2$ to $n$ do
    for $v \leftarrow 0$ to $M$ do
        if $W[i-2] == g^{w_{min}[i-2]+v}$ then
            $x \leftarrow x + w_{min}[i-2] + v \mod q$;
            break
        end
    end
end
return $x$;
```

Algorithm 1: An attack on missing range proofs

2.3 Small Paillier attack

Bob's ZK proof from [GG18]. If range proofs are used in MtA and MtAwc, [GG18] proposes the proof described in its appendix 2.A. This proof is inspired by the proofs given in [MR01]. Here we only mention the details of it that are relevant to our attack. Namely:

1. Challenge $e \in \mathbb{Z}_q$ is calculated via Fiat-Shamir heuristic;
2. Bob chooses random $\gamma \in \mathbb{Z}_N^*$, where N is Alice's Paillier public key;
3. Bob computes and sends Alice $t_1 = e\beta' + \gamma$ where β' is Bob's secret from MtA between them. Note that t_1 is computed over the integers.

The attack. It is considered a bad practice to generate invalid or short encryption keys. Intuitively, such keys may pose a security risk to anyone that uses them. In a multiparty computation with malicious adversaries, we expect such keys to be detected before usage. In the previous attack, we have seen how incorrect proportions of the Paillier encryption key N, and values that are meant to be elements of $\mathbb{Z}_q$, can lead to a disaster. For instance, if Alice chooses small Paillier N, the probability of a reduction modulo N in MtA may not be negligible anymore. Thus, Alice might get access to some information about the sizes of other parties' secrets. Unfortunately, the first attack cannot be applied as-is to the setting of MtA/MtAwc with ZK proofs since in this setting we no longer have access to Bob's $g^{\beta'}$ as was the case in the "simplified" version of MtAwc protocol. Instead, we propose a different attack:

Suppose that an adversary (again, party $\mathcal{P}_1$) chooses their Paillier encryption key to be slightly less then 2^{256} during the distributed key generation stage of [GG18] or [GG20] protocol. If parties' key sizes are not explicitly validated, this choice might lead to successfully generated keys. Note that such a choice immediately compromises the hiding property of β' inside t_1 from the ZK proof. Dividing t_1 by the public challenge e we get:

$$\frac{t_1}{e} = \beta' + \frac{\gamma}{e}$$

And because the ranges from which γ and e are taken are close to one another, the second term will be less than 1 with probability $\frac{1}{2}$, and not larger than 15 bits with probability around $1 - 2^{-16}$. The strategy of an attacker is then to play as Alice in MtA/MtAwc in the following way:

- Choose $k_1 = 1$. Note that this choice and the fact that Paillier encryption key N is close to q, implies that there are two options for what the result of MtAwc between $\mathcal{P}_1$ and another party $\mathcal{P}_i$ might be:
    1. $\alpha = w_i + \beta'$ if $w_i + \beta' < N$,
    2. $\alpha = w_i + \beta' - N$ if $w_i + \beta' \in [N; 2N)$.

- After MtAwc with party $\mathcal{P}_i$ is finished, try setting $\beta = \left\lfloor \frac{t_1}{e} \right\rfloor \mod q$, $\left\lfloor \frac{t_1}{e} \right\rfloor - 1 \mod q$, $\left\lfloor \frac{t_1}{e} \right\rfloor - 2 \mod q \ldots$ and calculate $w^{(1)} = \alpha - \beta \mod q$ and $w^{(2)} = \alpha - \beta + N \mod q$. Here t_1, e and α are values from the range proof and the decrypted share, all received from Bob in the course of this MtAwc. If $g^{w^{(1)}} = g^{w_i}$ or $g^{w^{(2)}} = g^{w_i}$, then $\mathcal{P}_i$'s secret share is successfully recovered (remember that parties' "public keys" g^{w_i} are known to everyone).

- Having recovered other parties' secrets from MtAwc, the attacker can then set their own output of MtAwc correctly. As to MtA, where parties receive additive shares of $k_i \cdot \gamma_j$, the only difference is that the attacker does not possess the value of g^{γ_j} at the moment MtA is carried out. As previously, denote values received from the ZK proof and the decrypted share as t_1, e, and α respectively. The attacker then calculates the approximate value of β as $\left\lfloor \frac{t_1}{e} \right\rfloor \mod q$ and the result of MtA is set to α if $\beta < \alpha$, and to $\alpha + N \mod q$ otherwise.

Not only does it take just a single signature to recover the full secret key, but the signature is also successfully generated afterward with high probability. This might be important if parties manage several keys (which happens in practical applications [Tho]), and we want to recover them one by one covertly without causing any suspicion prematurely.

Root cause. When analyzing the attack, we can identify two factors that contributed to its success.

First, we notice that the bound for γ, second step above, is simply wrong. To achieve honest verifier zero-knowledge (HVZK), γ must be chosen randomly from at least $\mathbb{Z}_{q^2N}$. Taking γ to be from $\mathbb{Z}_N^*$ as is the case in the paper, will break HVZK — dividing by e will leak some bits from β' regardless of the Paillier modulus size. We note that a similar error, likely due to a typo, happens with τ in the proof as well.

Finally, the missing Paillier size check was leveraged to make N comparable in size to q. Composed with the broken HVZK, we were able to completely remove the hiding of β' from t_1. Since the issue is with HVZK, completeness remains and the output signature will be valid.

3 Implementation and experiments

For our first attack, targeting implementations of MtA and MtAwc with no range proofs, we implemented a proof-of-concept on top of ZenGo's multi-party-ecdsa Rust library [Zen]. To make the attack fast, a value of $M = 2^{16}$ was chosen, leading to secret key leakage in 16 signatures, which can be improved by choosing a larger M.[6]

[6]`https://github.com/velas/multi-party-ecdsa-no-range-proofs-attack`

For the small Paillier attack, a proof-of-concept was implemented for ING bank's threshold-signatures library [ING]. We stress that in this library, Paillier size check is performed by all parties, and so it is not susceptible to the attack. We commented out Paillier size check for the sake of demonstration.[7]

4 Responsible Disclosure

We reached out to maintainers from all existing open source threshold ECDSA libraries.

We present below a table of what we perceive to be possible attack vectors on different implementations. Since the attacks discovered in this work impact multiple projects, one of the projects' leads opened up a private telegram group for all maintainers to coordinate SecOps. At the time of writing, all active projects, that are dependent on the protocols we identified, already pushed the fix. We describe the timeline of events in more detail on a separate document. The Discovery was rewarded a bug bounty equivalent to $500,000$ USD, pooled from several open source projects and led by Thorchain.

Library	Vector of attack	Required number of signatures	Number of signatures that will complete successfully
[ING][a]	Missing range proofs	8	1
[Zen]	Missing range proofs	8	1
[Bin]	Small Paillier	1	1
[Tho]	Small Paillier	1	1
[Kee]	Small Paillier	1	1
[Swi]	Small Paillier	1	1
[Any]	Small Paillier	1	1

[a]Fast version only

5 Recommendations and conclusions

The body of research around threshold ECDSA has grown immensely over the past few years, reflecting a concrete need coming from the industry. As a result, "young" protocols were adopted, implemented, and put into production, securing large sums of money. This, in turn, made threshold ECDSA implementations a lucrative and popular target for attacks. It also means that these crypto-systems enjoy accelerated battle-testing. Our work is simply one part of this fast-paced cycle. It is our belief that putting a spotlight on the technical details of bugs and errors will lead to an even more robust protocol design and implementation.

[7]`https://github.com/velas/threshold-signatures-small-paillier-attack`

We recommend that maintainers of all [GG18] or [GG20] implementations check that in their libraries:

- Range proofs in MtA and MtAwc are in place at least for parties playing as Alice.
- Every party checks that the sizes of other parties' Paillier public keys are correct. For instance, if Paillier keys are generated by multiplying two random primes that are exactly 1024 bit in size, parties might check that every Paillier modulus is either 2047-bit or 2048-bit and abort otherwise.
- In MtA and MtAwc, in Bob's range proofs γ should be sampled from $\mathbb{Z}_{q^2 N}$ and τ from $\mathbb{Z}_{q^3 \tilde{N}}$.

6 Acknowledgements

We would like to thank Claudio Orlandi and Elichai Turkel for reviewing the draft of this paper.

References

[AHS20] Jean-Philippe Aumasson, Adrian Hamelink, and Omer Shlomovits. A survey of ecdsa threshold signing. *IACR Cryptol. ePrint Arch.*, 2020:1390, 2020.

[Any] Anyswap's implementation of TSS. `https://github.com/anyswap/FastMulThreshold-DSA`.

[AS20] Jean-Philippe Aumasson and Omer Shlomovits. Attacking threshold wallets. *IACR Cryptol. ePrint Arch.*, 2020:1052, 2020.

[Bin] Threshold Signature Scheme, for ECDSA. `https://github.com/binance-chain/tss-lib`.

[CGG⁺21] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. Uc non-interactive, proactive, threshold ecdsa with identifiable aborts. Cryptology ePrint Archive, Report 2021/060, 2021. `https://ia.cr/2021/060`.

[FLOP18] Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, and Benny Pinkas. Fast distributed rsa key generation for semi-honest and malicious adversaries. Cryptology ePrint Archive, Report 2018/577, 2018. `https://ia.cr/2018/577`.

[GG18] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ecdsa with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1179–1194, 2018.

[GG20] Rosario Gennaro and Steven Goldfeder. One round threshold ecdsa with identifiable abort. *IACR Cryptol. ePrint Arch.*, 2020:540, 2020.

[ING] Threshold Signature Scheme for ECDSA. `https://github.com/ing-bank/threshold-signatures`.

[Kee] The smart contracts and client behind the Keep ECDSA client. `https://github.com/keep-network/keep-ecdsa`.

[MR01] Philip MacKenzie and Michael K. Reiter. Two-party generation of dsa signatures. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 137–154, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[Swi] Threshold Signature Scheme for Skybridge. `https://github.com/SwingbyProtocol/tss-lib`.

[Tho] Go implementation of TSS for Thorchain. `https://gitlab.com/thorchain/tss/go-tss`.

[Zen] Rust implementation of t,n-threshold ECDSA (elliptic curve digital signature algorithm). `https://github.com/ZenGo-X/multi-party-ecdsa`.