Practical Key-Extraction Attacks in Leading MPC Wallets

Nikolaos Makriyannis* Oren Yomtov* Arik Galansky*

February 26, 2025

Abstract

Multi-Party Computation (MPC) has become a major tool for protecting hundreds of billions of dollars in cryptocurrency wallets. MPC protocols are currently powering the wallets of Coinbase, Binance, Zengo, BitGo, Fireblocks and many other fintech companies servicing thousands of financial institutions and hundreds of millions of end-user consumers.

We present four novel key-extraction attacks on popular MPC signing protocols showing how a single corrupted party may extract the secret in full during the MPC signing process. Our attacks are highly practical (the practicality of the attack depends on the number of signature-generation ceremonies the attacker participates in before extracting the key). Namely, we show key-extraction attacks against different threshold-ECDSA protocols/implementations requiring 10^6, 256, 16, and *one signature*, respectively. In addition, we provide proof-of-concept code that implements our attacks.

*Fireblocks. E-mails: {nikos, oyomtov, arik}@fireblocks.com

Contents

1 Introduction

In distributed ledger systems, commonly known as blockchains, digital signatures are crucial for maintaining integrity and authenticity. Each participant in these networks possesses a public key that allows them to receive digital assets, and the corresponding private key enables them to transfer assets to other participants. The secure management of this secret material is facilitated by a 'wallet', which provides the owner with a secure means to sign transactions.

MPC Wallets. In recent years, *Multi-Party Computation* (MPC) [GMW87; Yao86] has emerged as the gold standard for safeguarding digital assets. It is arguably the preferred tool for institutional players, protecting hundreds of billions of dollars via 'MPC wallets'. In a typical setting, several geographically dispersed machines[1] (referred to as *parties*) engage in an interactive key-generation protocol to calculate the public key associated with the wallet, as well as to obtain their individual private key shares which they will use for calculating signatures in the future. Then, when prompted to sign a message (e.g. at the request of some authenticated user), the parties engage in an interactive signing protocol for calculating the desired signature. In this document, we refer to the interactive signing process as *threshold signing* [Des87; DF89].

Informally speaking, MPC ensures that the underlying 'master' private key is never exposed, and the only information revealed by the MPC protocol is the computation output (i.e. the signatures), *even in the presence of corrupted parties*. Essentially, MPC may be viewed as a *last line of defense* protecting against malicious *insiders*, provided that the system is not compromised in its entirety.

ECDSA. The most popular signature scheme in the blockchain space is the *Elliptic Curve Digital Signature Algorithm* (ECDSA) [Nat23]. Unlike other signature schemes that can be 'thresholdized' in a natural way (e.g. Schnorr signatures [Sch91] or BLS [BLS04]), ECDSA requires the full power of MPC to support threshold signing. That is, protocols for threshold ECDSA employ the full spectrum of cryptographic techniques for multiparty computation. These techniques include, but are not limited to, *oblivious transfer* (OT [Rab05]), *pseudorandom correlation generators* (PCGs [Boy+19]), *additive homomorphic encryption* (such as Paillier encryption [Pai99], hereafter referred to as 'Paillier') and *zero-knowledge proofs* (ZKPs [GMR85; GMW91]).

Protocols for threshold ECDSA are fairly abundant in the literature and they are widely implemented for digital wallet applications. It is worth noting that the transaction volume of MPC wallets is estimated to be in the *trillions* of US dollars.[2]

In this paper, we present attacks on Paillier-based threshold-ECDSA protocols. It is important to note that our attacks can be extrapolated to any MPC protocol or implementation that utilizes Paillier as a foundational building block. Below, we mention the three relevant signing protocols potentially affected by these attacks.

1. *Lindell17*: The *Lindell17* protocol (CRYPTO'17 [Lin17a] and J. Cryptol.'21 [Lin21]) is arguably the most popular ECDSA protocol tailored to *two parties*. It is utilized by vendors such as ZenGo (see ZenGo) and Coinbase WaaS (Coinbase WaaS).

[1] Typically, certain machines are located with a vendor, or wallet provider, while others are situated with the client, or end user.

[2] Fireblocks press release, quoting: "... surpassing $2 trillion in assets transferred." Retrieved from `fireblocks.com` (January 2024).

2. *GG18/20*: The '*GG*' family of protocols (CCS'18 [GG18], Manuscript'20 [GG20][3]) are widely implemented by various vendors and open-source projects, including e.g. Binance bnb-chain, ING Bank (Open Source Project), ZenGo (Open Source Project), Safeheron.

3. *BitGoTSS*: Lastly, we mention *BitGoTSS*, a custom protocol by BitGo, a leading cryptocurrency custodian managing assets worth billions.[4] *BitGoTSS* does not closely adhere to the protocols in the literature. It is essentially a bare-bones version of [GG18], where all the zero-knowledge proofs are omitted, c.f. BitGoJS library—(version 16.1.0).

MPC Wallet Threat Model. Following the usual convention, we assume that a single adversary, denoted as Adv, controls a subset of parties in the multi-party protocol, potentially all-but-one of the parties, and Adv can send any maliciously-crafted message on their behalf. In the context of MPC wallets, attack outcomes include:

1. *Denial-of-service*: Preventing the parties from signing.

2. *Signature forgery*: Obtaining a signature on a different message than the prescribed message.

3. *Key Extraction*: Extracting the honest parties' secret shares—eventually the entire key.

For signature forgery and key extraction, a critical metric for evaluating the efficiency of the attack is the number of signature-generation sessions the attacker has access to before the attack outcome occurs. Next, we discuss various ways in which attacks can arise.

Vulnerabilities & Attacks. Vulnerabilities that give rise to attacks in cryptographic protocols typically emerge from one of three sources. The first and most common are implementation issues, where the actual code deviates from the protocol's intended specification. These discrepancies can create critical security gaps. The second source is rooted in insecure assumptions—cases where the security proof of a protocol is sound, but the underlying hardness assumption is weak, often seen in 'experimental' protocols. The third, albeit rarer, source of vulnerabilities comes from mistakes or omissions in the security proof of the protocol itself. Furthermore, protocol-related vulnerabilities are uncommon in well-established, commercially-used protocols, and usually only of theoretical interest.

Related Work. Previous research has explored various attacks on Paillier-based threshold-ECDSA protocols and implementations. Namely, [AS20a; AS20b; Ngu+23a; Ngu+23b; TS21] have identified several implementation-related vulnerabilities potentially affecting most of the *multiparty* Paillier-based protocols, depending on the implementation. Notably, some of these attacks allow for very efficient key extraction.

In a different vein, [MP21] found a protocol-related attack affecting both '*GG*' protocols, where a malicious attacker could gain partial leakage into ephemeral secret randomness. However, this attack was largely theoretical and not applicable to practical scenarios as the authors could not leverage the leakage into a meaningful forgery and/or key extraction. Finally, we mention that [TS21] show an assumptions-related attack on the lightweight, and experimental variant of [GG18] that was based on non-standard assumptions.

[3] A variant of the main protocol from [GG20] was published in CCS'20 [Can+20] (as part of a merge with [CMP20]).
[4] BitGo press release. Retrieved from `bitgo.com` (January 2024).

Remark 1.1 (Old & New *GG18/20*). In 2021, in response to the findings of [MP21] regarding inadvertent leakage, both '*GG*' protocols from [GG18] and [GG20] were revised. The update included a proposed fix involving the modification of a critical protocol parameter. Specifically, the size of the beta parameter in the OLE was adjusted, as detailed in Section 2. However, most implementations (e.g., Binance bnb-chain, ING Bank (Open Source Project), ZenGo (Open Source Project)) did not incorporate this modification, leading to a situation where, in recent years, nearly all implementations deviated from the updated paper(s). In this document, our attack(s) apply equally to [GG18] and [GG20], so we will be referring to [GG18; GG20] as a single protocol '*GG18/20*' with two regimes of parameters, old and new.

1.1 Our Contributions

We present four novel key-extraction attacks; Our attacks exfiltrate the private key in full, and it suffices for the attacker to corrupt a single party in the MPC. The unifying element of our attacks is the leveraging of misimplementations or design flaws within the protocol, specifically targeting the Paillier encryption process. Depending on the scenario, this may involve encrypting a maliciously-chosen plaintext or choosing a maliciously crafted Paillier public key for the compromised party. In the latter case, the adversary selects the Paillier private key in a manner that deviates from the standard Paillier key-generation process.

Furthermore, our attacks are highly practical, requiring 10^6, 256, 16, and *one signature*, respectively. Additionally, two of our attacks have the potential for stealthiness, where the signature-generation process is error-free and it appears benign. Finally, we provide proof-of-concept (PoC) code [fir23a; fir23b; fir23c] implementing three of our attacks (the most practical ones).

The attack cost, specifically the number of signatures required for the adversary to extract the key, along with the stealthiness of each attack, is summarized in Table 1. To our knowledge, all the attacks presented here are novel, and, as previously noted, they are applicable to any MPC protocol utilizing Paillier encryption, extending beyond threshold-ECDSA.

Attack	Protocol	# Parties	# Signatures	Stealthiness
Broken Record	*Lindell17*-Implementations	2	256	✗
6ix1een	*GG18/20* (New)	n	16	✓
Death by 1M Cuts	*GG18/20* (Old)	n	$\approx (n-1)\cdot 10^6$	✗
Zero Proof	*BitGoTSS*	n	1	✓

Table 1: Summary of our key-extraction attacks for each protocol. We note that the attacker is assumed to compromise a single party in the MPC.

Validating the Attacks against Real-World Systems. In the process of validating Broken Record and Zero Proof against actual systems, we established MPC wallets two well-known wallet providers, ZenGo and BitGo. This entailed creating a shared key between our device and the wallet provider's server, specifically for an address on the Ethereum *mainnet* network. Utilizing the PoC-code linked in this document, we were able to extract the wallet provider's share of the key. This extraction allowed us to combine it with our local share, effectively reconstructing the complete private key. In our test against ZenGo, the Broken Record attack successfully retrieved

the key after 256 signatures. For BitGo, using the Zero Proof attack, we managed to retrieve the key with just a single signature.

Ease of Mounting the Attack(s). From a purely algorithmic perspective, compromising just a single party in the MPC is sufficient for the attacker, and there is no inherent obstacle to executing the attack as long as the malicious party goes undetected (see Section 1.2). As detailed in Table 1, two of our attacks, 6ix1een and Zero Proof, have the potential for stealthiness, meaning that the difficulty of those attacks solely depends on the initial corruption of a single MPC party.

In the context of cryptocurrency wallets, this typically involves compromising a mobile device or a server. For other threshold signing applications, such as blockchain bridges, any participant in the protocol can initiate an attack. Given that participation in the protocol is often permissionless, staking some capital may be the sole requirement to launch an attack.

Remark 1.2. As indicated above, the first vulnerability, affecting *Lindell17*, is implementation-specific. However, it important to note that we did not find a single implementation that was not affected. Furthermore, as we discuss in the following section, rather than strictly adhering to the existing protocol, we believe that the most effective way to remediate the issue is to augment the protocol with additional safeguards.

1.2 Detection, Mitigation & Remediation

Each of our attacks presents unique challenges in terms of detection, mitigation, and remediation. The common thread in remediation is the use of ZK proofs to ensure the integrity of key components in the signing process.

In case it is not already known or simply as a reminder to the reader, the Paillier public key, also referred to as a Paillier 'modulus', is essentially an RSA number, $N = pq$, which is the product of two large prime numbers such that $\gcd(N, (p-1)(q-1)) = 1$, *when generated correctly.*

Broken Record (*Lindell17*).

- *Detection & Mitigation*: The signature ceremony for *Lindell17* results in an invalid signature. This is a clear indication that something is amiss in the process. The immediate response, as instructed in *Lindell17* [Lin17a], should be to halt all signing activities as soon as an invalid signature is detected.

- *Remediation*: Stopping the signing process in the event of invalid signatures can be somewhat unrealistic and susceptible to misimplementation. For one, it may be hard to differentiate between a benign software bug and an actual attack. Secondly, even if the protocol is strictly adhered to and the wallet is locked after a failed signature, this approach could inadvertently lead to an easy denial-of-service (DoS) attack: merely flipping a single bit of the payload could lead to the wallet being locked indefinitely. This is in contrast to other protocols where the adversary must continuously disrupt the signing process to achieve a similar DoS effect. To address this issue, it is recommended to add a ZKP of well-formedness to the encrypted partial signature (see Section 2). Such a ZKP can be found in [BMP22].

6ix1een & Death by 1M Cuts (*GG18/20*).

- *Detection*: In this case, the detection involves identifying a corrupted Paillier modulus which admits small factors, making it susceptible to factorization with advanced algorithms. Factors of size 2^{80} or smaller are considered critical vulnerabilities. For the Death by 1M Cuts attack, the attack can further be the detected by the presence of invalid signatures.
- *Mitigation*: Attempting to factor the moduli before the signing process can help identify the vulnerability. For the Death by 1M Cuts attack, the attack can also be mitigated by ceasing all signing activities as soon as an invalid signature is detected.
- *Remediation*: Implementing a ZK proof of well-formedness for the Paillier Modulus is essential. This proof should verify that the modulus is a product of exactly two suitable primes and does not contain small factors, thereby ensuring its integrity. Such a ZKP can be found in [CMP20].

Zero Proof (*BitGoTSS*). For *BitGoTSS*, the situation is more complex. The protocol has fundamental flaws that render it entirely broken; it is essentially an honest-but-curious protocol devoid of any safeguards against active adversaries. Any attempt to patch the protocol to counter our specific attack would be merely superficial. For instance, Death by 1M cuts can also be used against *BitGoTSS*. We note that *BitGoTSS* was quickly deprecated as far as we know.

Remark 1.3. One may wonder why we include an attack against a custom protocol in our findings. We believe we have valid reasons for this choice. First, the protocol itself is not particularly exotic; it is a simplified version of the (old) *GG18/20* protocol and it may be viewed as the honest-but-curious version of *GG18/20* where all the ZKPs are omitted. Second, the fact that we exfiltrate the key in *one*[5] signature is technically noteworthy as, to the best of our knowledge, extracting more than a few bits of the key (in any number of signatures) was previously unknown.

1.3 Responsible Disclosure

We followed the standard 90-day responsible disclosure process for all the vulnerabilities and we made best efforts to ensure all potentially affected parties were informed and had adequate time to address the identified vulnerabilities.

1. *Identification of Vulnerable Open Source Libraries*: Initially, we identified potentially impacted open-source libraries using the following list compiled by ZenGo.
2. *Github Repository Analysis*: We carefully analyzed forks of these repositories on github. Our goal was to infer whether any companies were using or promoting vulnerable code.
3. *Online Search for MPC Marketing Materials*: We conducted online searches to identify companies using marketing materials that mentioned 'MPC', 'TSS', '*Lindell17*' or '*GG18/20*'. This helped us to further pinpoint organizations that might be unknowingly at risk.
4. *Compilation of a Suspected Companies List*: Based on our research, we compiled a detailed spreadsheet cataloging all companies we suspected might be using one of the vulnerable protocols. This list was instrumental in guiding our outreach efforts.

[5]To be precise, the key is extracted in less than one signature. Namely, it is extracted in full in the first round of the first signing session.

5. *Initiation of the 90-Day Disclosure Window*: The 90-day responsible disclosure window was initiated by sending our initial emails in early May 2023. A 90-day window is aligned with industry best practices, providing companies with sufficient time to respond and remediate the issues. As we initiated contact with the identified entities, we tracked progress in a detailed tracking system. This system categorized companies into several stages: 'Contacted', 'Responsive', 'Vulnerable', 'Mitigation Under Process', and 'Fixed'. It is worth mentioning that no company requested an extension beyond the 90-day window.

6. *Alternative Communication Channels*: In cases where we did not receive a response through the primary channel, typically the Chief Information Security Officer (CISO), we employed alternative methods, including reaching out via social platforms, e.g. LinkedIn, or mutual connections.

7. *Follow-up on Severity Misunderstandings*: Occasionally, affected parties did not initially grasp the severity of the vulnerabilities. In these instances, we engaged in follow-up communications to clarify the risks and urge prompt action.

8. *Expanding Notification Based on Recommendations*: On a few occasions, affected parties recommended we contact third companies potentially affected by the vulnerability. We acted promptly on these suggestions, expanding our notification efforts to these additional entities.

GG18/20. The vulnerability was discovered in early May, 2023. Over 10 vendors and/or open-source libraries were impacted. The 6ix1een attack was then demonstrated on SafeHeron's open-source library.

Our findings on *GG18/20* were publicly disclosed in August 2023, and a CVE has been assigned to the identified vulnerability: CVE-2023-33241.

***Lindell17*-Implementations.** We validated the vulnerability by extracting the secret share from ZenGo's servers (associated with our own mainnet account) in late March 2023. Five vendors and/or open-source libraries were impacted.

Our findings on *Lindell17* were publicly disclosed in August 2023, and a CVE has been assigned to the identified vulnerability: CVE-2023-33242.

BitGoTSS. We validated the vulnerability by extracting the secret share from BitGo's servers (related to our mainnet account) in December 2022. BitGo was informed of this issue immediately. Public disclosure followed in March 2023.

1.4 Paper Organization

Section 2 contains a high-level overview of our attacks. The purpose of Section 2 is to provide intuition and give the gist of the attacks, while minimizing non-essential details about the targeted protocols (*Lindell17*, *GG18/20*, *BitGoTSS*). Section 3 introduces notation and definitions for the subsequent technical sections. In Sections 4, 5 and 6, we give the formal description of, respectively, the *Lindell17* protocol and the Broken Record attack, the *GG18/20* protocol and the 6ix1een and Death by 1M Cuts attacks, and, finally, the Zero Proof attack on *BitGoTSS*.

2 High-Level Overview

We assume some familiarity with elementary number theory and basic group theory. For the purposes of the current section, we will be using the following facts about Paillier encryption: (i) The Paillier public key $N = pq$ is an RSA number such that N and $\varphi(N)$ are coprime, where φ denotes Euler's totient function. (ii) The plaintexts and ciphertexts exist in the spaces $\mathbb{Z}_N$ and $\mathbb{Z}^*_{N^2}$, respectively. (iii) Paillier is additive-homomorphic as long as N and $\phi(N)$ are coprime, meaning that using $\mathcal{C} = \mathrm{Enc}_N(x) \in \mathbb{Z}^*_{N^2}$ corresponding to a plaintext $x \in \mathbb{Z}_N$, one can easily compute $\mathcal{D} = \mathrm{Enc}_N(\gamma x - \beta) \in \mathbb{Z}^*_{N^2}$ for any $\beta, \gamma \in \mathbb{Z}_N$.

We start our high-level overview by describing the threshold-ECDSA protocols from [Lin17a] and [GG18; GG20].

2.1 Background

Lindell17. The *Lindell17* protocol crucially relies on Paillier as follows: First, during key-generation, letting x denote B's secret share of the ECDSA key, party B sends $\mathrm{Enc}(x)$ to A encrypted under a Paillier key that B owns, i.e. only B can decryt the ciphertext, but A can homomorphically operate on it. Then, during signing, party A sends $\mathrm{Enc}(s')$ to B where s' is a *partial* ECDSA signature, and $\mathrm{Enc}(s')$ is calculated by homomorphically evaluating $\mathrm{Enc}(x)$. In the end, B finalizes the signature by decrypting $\mathrm{Enc}(s')$ and performing some lightweight data-processing.

GG18/20. The *GG18/20* protocol crucially relies on Paillier encryption as well, but it is employed for a different function: to realize pairwise *oblivious linear evaluation* (OLE), which is defined as follows. OLE takes input γ from A and x, β from B, and returns α to A such that $\gamma \cdot x = \alpha + \beta \bmod N$, where N is the Paillier public key associated with A (looking ahead, x corresponds to the B's ECDSA key share and β, γ are random ephemeral values, and all parties in the MPC play the roles of A and B with every other party in separate instances of the OLE).

To instantiate the OLE, party A sends $\mathrm{Enc}(\gamma)$ to B who returns $\mathrm{Enc}(\gamma \cdot x - \beta)$ by homomorphically evaluating $\mathrm{Enc}(\gamma)$. To conclude the OLE, A decrypts the received ciphertext to outputs $\alpha = x \cdot \gamma - \beta \bmod N$ (the modulo N reduction occurs implicitly when B homomorphically computes $\mathrm{Enc}(x \cdot \gamma - \beta)$. We note that the roles of A and B have been reversed compared to *Lindell17*, and the owner of the Paillier key is now A, and only A can decrypt the ciphertext).

After concluding all the OLE instances (requiring two rounds of interaction), the parties run an additional seven rounds for [GG18] (or five rounds for [GG20]) in order to produce the final output, i.e. the signature.

2.2 Broken Record

Our first attack is against implementations of the *Lindell17* protocol. We show how the adversary may craft a *malicious partial signature* that will cause the signature process to fail or succeed depending on the value of a targeted bit of the honest party's share. In the remainder, recall that A sends $\mathrm{Enc}(s')$ where s' is a partial signature that depends on the honest party's share.

The Attack. Adv corrupts A and sends $\mathrm{Enc}(\sigma')$ to B such that $\sigma' = s'$ if and only if the least significant bit of x (B's private share) is zero. Thus, the signature is valid at the end of the signature

ceremony if and only if x's least significant bit is zero and this value is inadvertently leaked to A when it is notified that the signature failed or succeeded.

The attack can then be iterated (with suitable adjustments) to leak the higher-order bits, and, after approximately two hundred signatures, the key can be recovered in full. We provide PoC code implementing our attack at the referenced github repository [fir23c].

Remark 2.1. Our attack does not challenge the security analysis from [Lin17a] because [Lin17a] assumes that failed signatures terminate signature operations and it specifically instructs parties to stop signing once an invalid signature is detected by the honest party.[6] In fact, [Lin17a] even suggests the theoretical plausibility that bits of the key may be leaked with the success/failure of the signature acting as an oracle (bottom paragraph, p. 11 of the full-version document [Lin17b]). In this paper, our contribution lies in demonstrating that such an attack exists, as well as implementing the attack against real-world systems.

2.3 6ix1een

Our second attack targets the post-update *GG18/20* protocol (under the new regime of parameters). In the *6ix1een*[7] attack, a single corrupted party extracts the private key in full after sixteen signature attempts *for any number of honest parties*.

OLE Parameters. Our attack targets the OLE phase of the protocol (so the solitary corrupted party uses malicious inputs in the pairwise OLE instances with each of the other parties). Recall that the OLE takes input γ from A and x, β from B, and returns α to A such that $\gamma \cdot x = \alpha + \beta \bmod N$, where N is the Paillier public key associated with A. Our attack specifically leverages the size of the inputs and outputs in the OLE, so we note that x and γ are 256 bits and β is a random number of roughly 1024 bits and so $\alpha = x \cdot \gamma - \beta$ is also 1024 bits (in the 'old' regime of parameters, β is chosen from the range $\{1, \ldots, N\}$, so $\beta \approx 2^{2048}$, cf. Remark 1.1). Adversary Adv corrupting A extracts B's secret x as follows.

The Attack. Adv chooses Paillier key $N = p_1 \cdot \ldots \cdot p_{16} \cdot q$ where $p_1 \ldots p_{16}$ are sixteen random primes of size 2^{16} and q is a large prime chosen randomly to match the expected size of the Paillier public key.

Then, in the OLE, Adv sets $k = N/p_i$ for a fixed $p_i \in \{p_1, \ldots, p_{16}\}$, cheats in the zero-knowledge proof (this is the crux of the attack that we defer to the technical sections), and obtains the value of $x \bmod p_i$ because $\alpha = x \cdot (N/p_i) - \beta \bmod N$ and $\beta < N/p_i$ and thus $x \bmod p_i \approx \alpha/(N/p_i)$, i.e. the closest multiple of (N/p_i) to α leaks $x \bmod p_i$ (the exact calculations are deferred to Section 5).

Iterating the above for each prime yields $x \bmod p_i$ for all 16 possible values of p_i. In the end, we reconstruct x using Chinese Remainder Theorem. We provide PoC code implementing our attack at the referenced github repository [fir23b].

Remark 2.2. The primary source of the vulnerability is that the zero-knowledge proofs relating to the Paillier moduli only check for square-freeness (i.e. that N and $\varphi(N)$ are coprime). So, the malicious N described above will go undetected. The secondary aspect of the vulnerability, crucial for our attack, arises from a flaw in the range-proof check. Specifically, the proof of soundness of the ZKP breaks down when the verifier's random challenge in the ZKP is a proper divisor of N;

[6] In practical terms, this assumption means that the wallet must be locked (at least temporarily).

[7] '6ix1een' because it involves sixteen small primes of size sixteen bits, as well as sixteen signatures.

this happens with noticeable probability (and thus can be brute-forced in our attack) because the malicious N has small factors.

2.4 Death by 1M cuts

Our third attack targets the old version of *GG18/20* where the beta parameter is chosen from $\{1, \ldots, N\}$. In this regime of parameters, the 6ixteen attack is no longer relevant because A's output in the OLE, α, completely hides x, for any value of γ. Instead, we will obtain information leakage through the success/failure of the signature process, akin to the broken record attack.

The Attack. The first few steps of the attack are identical to the 6ix1een. Namely, the attacker chooses a Paillier modulus with 16 small prime factors and it sets $\gamma = N/p_i$ during signing, where p_i is one of the small factors. When obtaining α, the attacker reassigns $\alpha := \alpha - y \cdot N/p_i \bmod N$ where y denotes a *random guess* of the value $x \bmod p_i$, and the attacker proceeds with the remaining steps of the protocol as if it had selected $\gamma = 0$. By noticing that

$$\alpha - y \cdot N/p_i \bmod N = ((x \bmod p_i) - y) \cdot N/p_i - \beta$$
$$\begin{cases} = \gamma \cdot x - \beta & \text{if } y = x \bmod p_i \\ \neq \gamma \cdot x - \beta & \text{otherwise} \end{cases}$$

it follows that the adversary's *reassigned* α is consistent with $\gamma = 0$ if and only if the adversary guessed $x \bmod p_i$ correctly, and thus the execution will result in a valid signature only when $y = x \bmod p_i$.

When all the remainders have been extracted (in 16 different signing sessions resulting in 16 *valid* signatures[8] ceremonies), the attacker can reconstruct the x in full using Chinese remainder theorem. We note that the complexity of the attack, i.e. the number of signatures it requires, depends on the size of the chosen primes as well as the number of parameters. For a single corrupted party in a n-party protocol, choosing the smallest possible primes, our attack retrieves the private key with probability $1/2^{n-1}$ after approximately $(n-1) \cdot 10^6$ signatures (cf. Section 5.3.1).

2.5 Zero Proof

Our last attack targets the *BitGoTSS* protocol which does not adhere closely to any paper from the literature. *BitGoTSS* follows the same template as *GG18/20*, except that it devoid of any ZKPs. This allows us to choose a malicious Paillier public key for the corrupted party that is even more distorted than the malicious public key in the 6ix1een and the Death by 1M cuts attack. Namely, we will not only choose N to have small factors, but we will make sure that N and $\varphi(N)$ admit a non-trivial GCD.

The Attack. The first step of our attack is choosing a Paillier key of the form

$$N = b \cdot \prod_{i=1}^{16} p_i \cdot q_i \tag{1}$$

[8] Failed signatures do not result in leakage (we have omitted the details of why in this initial presentation)

where q_i, p_i are 16-bit primes such that $q_i = 2p_i + 1$ (i.e. q_i is a strong prime with inner prime p_i) and b is a large prime chosen to compensate for the expected size of N.

In the first signature session's OLE, we take advantage of the absence of a zero-knowledge proof validating that the message from A is a legitimate ciphertext. We do this by sending the value 4, which is an invalid ciphertext for the chosen key. Consequently, when B processes this message, it outputs a value $\mathcal{D}$. Reduced modulo N, this value equates to $4^x \bmod N$. (The value -4^x is also possible, but we ignore this case in this initial presentation). Finally, to extract the x, the attacker obtains $x \bmod p_i$ by brute forcing $4^x \bmod q_i$ (because 4 generates a group of size p_i in $\mathbb{Z}^*_{q_i}$). In the end, the secret x is reconstructed using Chinese remainder theorem. We provide PoC code implementing our attack at the referenced github repository [fir23a].

Remark 2.3 ([TS21] does *not* apply to *BitGoTSS*). It is important to highlight that the attack outlined in [TS21] is not applicable to *BitGoTSS*. The reason for this is the attacker in [TS21] leverages the value g^β (where β denotes the honest party's additve input in the OLE and g denotes the generator of the ECDSA group), shared by the honest party during the OLE. In *BitGoTSS*, however, such values are not exchanged among the parties, rendering the attack ineffective.

3 Preliminaries

3.1 Notation

Basic notation. Throughout the paper $\mathbb{Q}$, $\mathbb{Z}$ and $\mathbb{N}$ denote the set of rational, integer and natural numbers, respectively, and we write $x \bmod n$ (or $[x]_n$ for conciseness) for the remainder of x modulo n. Further, $\mathbb{Z}_n^*$ denotes the multiplicative group of inverses modulo $n \in \mathbb{N}$, i.e. $\mathbb{Z}_n = (\mathbb{Z}/n\mathbb{Z})^*$. We let $\varphi : \mathbb{N} \to \mathbb{N}$ denote Euler's totient function and we write $\gcd(a, b)$ for the greatest common divisor of a and b.

Algorithms & Protocols. We use roman font $(\mathrm{Enc}, \mathrm{Dec}, \mathrm{PailKeys}, \ldots)$ for algorithms and we write $x = \mathrm{Algo}(m; \rho)$ for computing x according to (probabilistic) algorithm Algo on prescribed input m and *randomizer* (random input) ρ. When the randomizer is omitted, we write $x \leftarrow \mathrm{Algo}(m)$ and it is assumed the randomizer is chosen as prescribed. We use sans-serif letters $(\mathsf{Orc}^*, \mathsf{Prot}, \ldots)$ to denote oracles and protocols. Oracles are distinguished from protocols using a star $(^*)$ identifier. All oracles except the single-party IntCom^* (Definition 5.6) are two-party oracles, i.e. they receive input and deliver output to both parties during the oracle-call.

Groups & ECDSA. We write $(\mathbb{G}, g, q)$ for the group-generator-order tuple associated with the ECDSA algorithm and we use multiplicative notation for the relevant operations. For an arbitrary set $\boldsymbol{S}$, we write $x \leftarrow \boldsymbol{S}$ for x chosen uniformly at random from $\boldsymbol{S}$. Finally, $\mathrm{HASH} : \{0,1\}^* \to \mathbb{Z}_q$ denotes the cryptographic hash function associated with ECDSA (and is instantiated with SHA2), and we recall the ECDSA signing formula: for private key $x \in \mathbb{Z}_q$ and message $\mathrm{msg} \in \{0,1\}^*$, ECDSA signatures consist of pairs (R, s) such that

$$\begin{cases} R = g^k & \text{s.t. } k \in \mathbb{Z}_q \\ s = [k^{-1}(\mathrm{HASH}(\mathrm{msg}) + r \cdot x)]_q & \text{s.t. } r = R|_{\mathrm{proj}} \end{cases},$$

where $(\cdot)|_{\mathrm{proj}} : \mathbb{G} \to \mathbb{Z}_q$ is the so-called 'conversion function' associated with ECDSA.

3.2 Paillier Encryption & CRT

Definition 3.1 (Paillier Enc.)**.** Define $(\text{PailKeys}, \text{Enc}, \text{Dec})$ as the three-tuple of algorithms below.

1. Let $(N, \sigma) \leftarrow \text{PailKeys}$ where $N = p \cdot p'$ is the public key and $\sigma = (p-1)(p'-1)$ is the private key such that p, p' are random primes of bit-length 1024.
2. For $m \in \mathbb{Z}_N$, let $\text{Enc}_N(m; \rho) = (1+N)^m \cdot \rho^N \bmod N^2$, where $\rho \leftarrow \mathbb{Z}_N^*$.
3. For $\mathcal{C} \in \mathbb{Z}_{N^2}^*$, letting $\mu = \sigma^{-1} \bmod N$, $\text{Dec}_\sigma(\mathcal{C}) = \left(\frac{[\mathcal{C}^\sigma]_{N^2} - 1}{N}\right) \cdot \mu \bmod N$.

(We use calligraphic letters to denote Paillier ciphertexts)

Fact 3.2. *Paillier encryption is additively homomorphic. Namely, for every $N \in \mathbb{N}$ such that $\gcd(N, \varphi(N)) = 1$, it holds that $\text{Enc}_N(m; \rho)^\alpha = \text{Enc}_N(\alpha \cdot m; \rho^\alpha) \bmod N^2$ and $\text{Enc}_N(m; \rho) \cdot \text{Enc}_N(m'; \rho') = \text{Enc}_N(m + m'; \rho \cdot \rho') \bmod N^2$, for every $m, m' \in \mathbb{Z}_N$ and $\rho, \rho' \in \mathbb{Z}_N^*$.*

Theorem 3.3 (Chinese Remainder Theorem.)**.** *Let $p_1, \ldots, p_n$ denote n distinct primes and write $M = \prod_{i=1}^n p_i$ and $u_i = [(M/p_i)^{-1}]_{p_i} \cdot M/p_i$. For $x \in \mathbb{N}$ such that $x < M$, it holds that $x = \sum_{i=1}^n u_i \cdot [x]_{p_i} \bmod M$.*

4 Our Attack on Implementations of *Lindell17*

In this section, we present our attacks on implementations of *Lindell17*. Specifically those implementations that ignore failed signatures. To simplify the presentation, we have opted to describe *Lindell17* (Protocol 4.3) in the presence of oracles that help the parties calculate certain correlated values. These oracles don't impact the attack and are solely for presentation purposes (the oracles are defined below together with the protocol).

In our attack (Attack 4.5), corrupted A extracts B's secret share share, x_B, in 256 separate signature sessions, and the adversary uses prior knowledge of the secret share to advance to the next iteration of the overall attack. That is, in the ℓ-th attack, the adversary uses the bits of x_B that it extracted in the first $\ell - 1$ attacks in order to craft a malicious partial signature that will leak the ℓ-th bit of x_B via the failure/success of the signature-generation process.

4.1 Protocol Description

Definition 4.1 (KeyGen^*)**.** Define KeyGen^* on input $(\mathbb{G}, g, q)$ from A and B such that KeyGen^* returns the tuple $(X, x_\mathsf{A}, N, \mathcal{C}) \in \mathbb{G} \times \mathbb{Z}_q \times \mathbb{Z} \times \mathbb{Z}_{N^2}^*$ to A and the tuple $(X, x_\mathsf{B}, N, \sigma) \in \mathbb{G} \times \mathbb{Z}_q \times \mathbb{Z} \times \mathbb{Z}$ to B where $x_\mathsf{A}, x_\mathsf{B} \leftarrow \mathbb{Z}_q$ are uniformly random and $(X, N, \sigma, \mathcal{C})$ are set as follows

$$(N, \sigma) \leftarrow \text{PailKeys} \quad \text{and} \quad \begin{cases} X = g^{x_\mathsf{A} + x_\mathsf{B}} \\ \mathcal{C} \leftarrow \text{Enc}_N(x_\mathsf{B}) \end{cases}.$$

(PailKeys and Enc denote the key-generation and encryption algorithms from Definition 3.1)

Definition 4.2 ($\mathsf{MulShare}^*$)**.** Define $\mathsf{MulShare}^*$ taking common input $(\mathbb{G}, g, q)$ and secret inputs k_A and $k_\mathsf{B} \in \mathbb{Z}_q$ from A and B respectively such that $\mathsf{MulShare}^*$ returns $R = g^{k_\mathsf{A} \cdot k_\mathsf{B}} \in \mathbb{G}$.

Protocol 4.3 (*Lindell17* (A,B))**.**

Oracles: KeyGen^*, $\mathsf{MulShare}^*$

Operations: (Key-Generation)

Upon activation, parties call $\mathsf{KeyGen}^*(\cdot)$ and obtain the following output:

(a) Common Output: ECDSA pk $X = g^{x_\mathsf{A}+x_\mathsf{B}} \in \mathbb{G}$ and Paillier pk $N \in \mathbb{Z}$.

(b) Secret output: A gets $x_\mathsf{A} \in \mathbb{Z}_q$ and $\mathcal{C} = \mathrm{Enc}_N(x_\mathsf{B})$

(c) Secret output: B gets $x_\mathsf{B} \in \mathbb{Z}_q$ and Paillier private key $\sigma \in \mathbb{Z}$.

Operations: (Signing) When prompted on message msg, set $m = \mathrm{HASH}(\mathrm{msg})$ and do:

1. A, B, sample $k_\mathsf{A} \leftarrow \mathbb{Z}_q$ and $k_\mathsf{B} \leftarrow \mathbb{Z}_q$ respectively.
2. Parties call $\mathsf{MulShare}^*(\mathbb{G}, g, q)$ on input k_A and k_B and obtain $R = g^{k_\mathsf{A}\cdot k_\mathsf{B}}$.
3. A sets $r = R|_{\mathrm{proj}}$ and sends $\mathcal{D} \in \mathbb{Z}^*_{N^2}$ to B where
$$\mathcal{D} = \mathrm{Enc}_N\big([k_\mathsf{A}^{-1}(m + rx_\mathsf{A})]_q\big) \cdot \mathcal{C}^{[r\cdot k_\mathsf{A}^{-1}]_q} \bmod N^2$$
4. B outputs (R, s) where $s = k_\mathsf{B}^{-1} \cdot \mathrm{Dec}_\sigma(\mathcal{C}) \bmod q$ iff (R, s) is a valid signature.

4.2 Broken Record Attack

Claim 4.4. *Under Attack 4.5, party* B *finalizes the signature correctly if and only if*

$$x_\mathsf{B} - y_\mathsf{B} \bmod 2^\ell = 0.$$

Proof. Recall that the s-part of the signature in an honest execution of Protocol 4.3 satisfies $s = (2^\ell k_\mathsf{B})^{-1}(m + r(x_\mathsf{A} + x_\mathsf{B})) \bmod q$ when A chooses $k_\mathsf{A} = 2^\ell$. Next. we express $\mathrm{Dec}(\mathcal{D}')$ as a function of s. Namely, for $\zeta = [2^{-\ell}(m + rx_\mathsf{A})]_q$ and $\zeta' = y_\mathsf{B} \cdot r' \cdot [2^{-\ell}]_q$,

$$\begin{aligned}
\mathrm{Dec}(\mathcal{D}') &= \big(\zeta + y_\mathsf{B} \cdot r' \cdot \varepsilon\big) + [x_\mathsf{B} \cdot r' \cdot 2^{-\ell}]_N \bmod N \\
&= \zeta + \zeta' + (x_\mathsf{B} - y_\mathsf{B}) \cdot r' \cdot [2^{-\ell}]_N \bmod N \\
&= \begin{cases} \zeta + \zeta' + \frac{x_\mathsf{B}-y_\mathsf{B}}{2^\ell} \cdot r' & \text{if } [x_\mathsf{B} - y_\mathsf{B}]_{2^\ell} = 0 \\ \zeta + \zeta' + \frac{x_\mathsf{B}-y_\mathsf{B}-2^{\ell-1}}{2^\ell} \cdot r' + \frac{r'-1}{2} + \frac{N+1}{2} & \text{otherwise} \end{cases}
\end{aligned}$$

and thus $\mathrm{Dec}(\mathcal{D}') = \begin{cases} s \cdot k_\mathsf{B} \bmod q & \text{if } [x_\mathsf{B} - y_\mathsf{B}]_{2^\ell} = 0 \\ s \cdot k_\mathsf{B} + N \cdot 2^{-1} \bmod q & \text{otherwise} \end{cases}$. □

Note that $x_\mathsf{B} - y_\mathsf{B} \bmod 2^\ell = 0$ if and only if ℓ-the least significant bit of x_B is zero. Thus, in conclusion, B recovers the happy-flow formula and obtains s if $x_\mathsf{A} - y_\mathsf{B} = 0 \bmod 2^\ell$. (Otherwise, s is offset by $[k_\mathsf{B}^{-1} \cdot N \cdot (q+1)/2]_q$ and the resulting signature is invalid).

Attack 4.5 (Broken Record: Corrupted A in Protocol 4.3)**.**

Auxiliary input: Adv holds $y_{\mathsf{B}} = x_{\mathsf{B}} \bmod 2^{\ell-1}$
(the attack is initialized with $\ell = 1$ and $y_{\mathsf{B}} = 0$, and y_{B} is updated after each attack.)

Operations:

1. Call $\mathsf{MulShare}^*$ on inputs $k_{\mathsf{A}} = 2^\ell$ (chosen by Adv) and k_{B} (chosen by B).
   All parties obtain $R = g^{2^\ell \cdot k_{\mathsf{B}}} \in \mathbb{G}$ from $\mathsf{MulShare}^*$.
   Adv sets $\varepsilon = [2^{-\ell}]_q - [2^{-\ell}]_N$ and
   $$r' = \begin{cases} r & \text{if } r \text{ is odd (recall } r = R|_{\text{proj}}) \\ r + q & \text{otherwise} \end{cases}.$$
2. Adv sends $\mathcal{D}' \in \mathbb{Z}^*_{N^2}$ to B where, for $\zeta = [k_{\mathsf{A}}^{-1}(m + r x_{\mathsf{A}})]_q$,
   $$\mathcal{D}' = \mathrm{Enc}_N(\zeta + y_{\mathsf{B}} \cdot r' \cdot \varepsilon) \cdot (\mathcal{C}^{r'})^{[2^{-\ell}]_N} \bmod N^2$$
3. Adv deduces that
   $$x_{\mathsf{B}} \bmod 2^\ell = \begin{cases} y_{\mathsf{B}} & \text{if sig succeeds} \\ y_{\mathsf{B}} + 2^{\ell-1} & \text{if sig fails} \end{cases}$$

5 Our Attack(s) on *GG18/20*

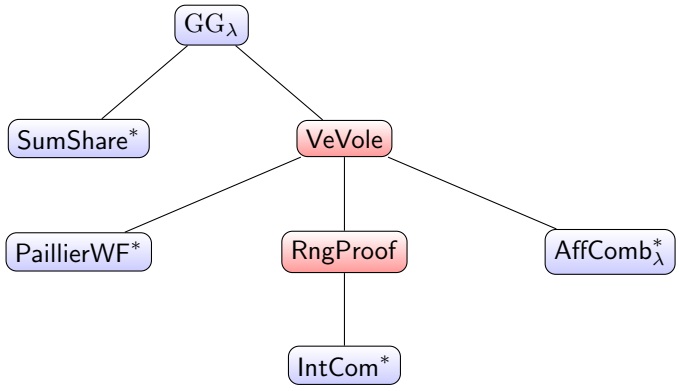

Figure 1: Illustration of the GG protocol dependencies. We note that both of our attacks target the VeVole and $\mathsf{RngProof}$ subprotocols. Specifically, corrupted party A uses maliciously-chosen values in VeVole and $\mathsf{RngProof}$ when calculating the messages to be sent to B.

In this section, we present our attacks on *GG18/20*. To simplify the presentation, we describe our attacks against a generic two-party protocol (Protocol 5.1)[9] and we only briefly mention how

[9]Protocol 5.1 is expressed using the [DKLs23] framework, but it incorporates custom processes from *GG18/20*.

the attack generalizes to the multiparty case. We note that each of our attacks is applicable to specific parameter choices within the protocol, detailed further below. As before, the adversary corrupts A.

In order to keep the description of Protocol 5.1 simple, we use a number of different oracles and subprotocols according to the diagram in Figure 1 (where each edge denotes an oracle or subprotocol invocation).

5.1 Protocol Description

As shown in Figure 1 the signing protocol invokes two subprotocols VeVole (*Verifiable VOLE*) and $\mathsf{RngProof}$ (*Range Proof*), and four oracles $\mathsf{AffComb}^*_\lambda$ (*Affine Combination*), $\mathsf{PaillierWF}^*$ (*Paillier Well-Formedness*), $\mathsf{SumShare}^*$ (*Additively Share*), and IntCom^* (*Integer Commitment*).We describe each (sub)protocol starting from the root of the tree to the leaves, and each oracle is described together with the relevant protocol.

Protocol 5.1 (*GG18/20* (A,B))**.**

Oracle & Sub-protocol: $\mathsf{SumShare}^*(\cdot)$ and $\mathsf{VeVole}(\cdot)$

Common Input: Group-generator-order tuple $(\mathbb{G}, g, q)$

Operations: (Key-Generation)

Each $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$ samples $x_\mathsf{P} \leftarrow \mathbb{Z}_q$.

Parties call $\mathsf{SumShare}^*(\mathbb{G}, g, q)$ on x_A and x_B and obtain:

(a) Common Output: ECDSA pk $X = g^{x_\mathsf{A}+x_\mathsf{B}} \in \mathbb{G}$.

(b) Secret Output: P gets $x_\mathsf{P} \in \mathbb{Z}_q$.

Operations: (Signing) When prompted on message msg, set $m = \mathrm{HASH}(\mathrm{msg})$ and do:

1. Each $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$ samples $k_\mathsf{P} \leftarrow \mathbb{Z}_q$.
2. Parties call $\mathsf{SumShare}^*$ on input k_P from $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$ and obtain $R = g^{k_\mathsf{A}+k_\mathsf{B}}$.
3. Parties execute $\mathsf{VeVole}(\ldots)$ on input $(R, X, k_\mathsf{P}, x_\mathsf{P})$ from $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$ and obtain:
   (a) Common Output: Empty
   (b) Secret Output: P gets random $\alpha_\mathsf{P}, \beta_\mathsf{P}, \hat{\alpha}_\mathsf{P}, \hat{\beta}_\mathsf{P}, \gamma_\mathsf{P} \in \mathbb{Z}_q$ such that: letting $\mathsf{Q} = \{\mathsf{A},\mathsf{B}\} \setminus \mathsf{P}$
   $$\begin{cases} \alpha_\mathsf{P} + \beta_\mathsf{Q} = \gamma_\mathsf{P} \cdot x_\mathsf{Q} \bmod q \\ \hat{\alpha}_\mathsf{P} + \hat{\beta}_\mathsf{Q} = \gamma_\mathsf{P} \cdot k_\mathsf{Q} \bmod q \end{cases}$$
4. Each $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$ sets $r = R|_{\mathrm{proj}}$ and sends $(\hat{s}_\mathsf{P}, \delta_\mathsf{P}) \in \mathbb{Z}_q^2$ to $\mathsf{Q} \in \{\mathsf{A},\mathsf{B}\} \setminus \{\mathsf{P}\}$ where
   $$\begin{cases} \hat{s}_\mathsf{P} = \gamma_\mathsf{P} \cdot m + r \cdot (x_\mathsf{P}\gamma_\mathsf{P} + \alpha_\mathsf{P} + \beta_\mathsf{P}) \bmod q \\ \delta_\mathsf{P} = k_\mathsf{P}\gamma_\mathsf{P} + \hat{\alpha}_\mathsf{P} + \hat{\beta}_\mathsf{P} \bmod q \end{cases}$$
5. Each $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$ outputs (R, s) where $s = (\delta_\mathsf{A}+\delta_\mathsf{B})^{-1} \cdot (\hat{s}_\mathsf{A}+\hat{s}_\mathsf{B}) \bmod q$ iff (R, s) is a valid signature.

None of the attacks in this paper apply to the [DKLs23] protocol itself.

Definition 5.2 ($\mathsf{SumShare}^*$)**.** Define $\mathsf{SumShare}^*$ taking secret inputs v_A and $v_\mathsf{B} \in \mathbb{Z}_q$ from A and B respectively such that $\mathsf{SumShare}^*$ returns $R = g^{v_\mathsf{A}+v_\mathsf{B}} \in \mathbb{G}$. (This oracle simply returns a random a random group element to the parties as well as an additive share v_P of the discrete log to each $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$.)

Each of our attacks is relevant to a certain value of parameter λ in $\mathsf{AffComb}^*_\lambda$, and λ is hardcoded as one of two possible values: q^5 (where q is the order of the ECDSA group) or 2^{2048} (the size of the Paillier modulus). Specifically, when $\lambda = q^5$, Attack 5.8 applies which recovers the key in sixteen signatures, regardless of the number of parties, and, when $\lambda = 2^{2048}$, Attack 5.11 applies which recovers the key in approximately one million signatures per honest party because each share is extracted separately (so each additional honest party incurs an additional 1M signatures to recover their share).

5.1.1 Verifiable VOLE

Protocol 5.3 ($\mathsf{VeVole}\ (\mathsf{A}, \mathsf{B})$)**.**

Oracles & Sub-protocol: $\mathsf{PaillierWF}^*(\cdot)$, $\mathsf{AffComb}^*_\lambda(\cdot)$ and $\mathsf{RngProof}(\cdot)$

Common Input: Group Elements $(R, X) \in \mathbb{G}^2$

Secret Input: Each $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ holds field elements $(x_\mathsf{P}, k_\mathsf{P}) \in \mathbb{Z}_q^2$

Operations: (One-time setup)

1. Each $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ samples Paillier key pair $(N_\mathsf{P}, \sigma_\mathsf{P}) \leftarrow \text{PailKeys}$.
2. Parties call $\mathsf{PaillierWF}^*$ on inputs $(N_\mathsf{A}, \sigma_\mathsf{A})$ and $(N_\mathsf{B}, \sigma_\mathsf{B})$ and obtain $(N_\mathsf{A}, N_\mathsf{B})$. (If $(N_\mathsf{A}, N_\mathsf{B}) = \perp$, abort)

Operations: (VOLE)

1. Each $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ samples $\gamma_\mathsf{P} \leftarrow \mathbb{Z}_q$ and $\rho_\mathsf{P} \leftarrow \mathbb{Z}_{N_\mathsf{P}}$ and sets $\mathcal{C}_\mathsf{P} = \text{Enc}_\mathsf{P}(\gamma_\mathsf{P}; \rho_\mathsf{P})$.
2. Parties execute $\mathsf{RngProof}$ on inputs $(\mathcal{C}_\mathsf{A}, \gamma_\mathsf{A}, \rho_\mathsf{A})$ and $(\mathcal{C}_\mathsf{B}, \gamma_\mathsf{B}, \rho_\mathsf{B})$. Obtain:
   (a) Common Output: $(\mathcal{C}_\mathsf{A}, \mathcal{C}_\mathsf{B}) \in \mathbb{Z}^*_{N_\mathsf{A}^2} \times \mathbb{Z}^*_{N_\mathsf{B}^2}$. (If $(\mathcal{C}_\mathsf{A}, \mathcal{C}_\mathsf{B}) = \perp$, abort)
   (b) Secret Output: N/A
3. Call $\mathsf{AffComb}^*_\lambda$ on input $(\mathcal{C}_\mathsf{A}, N_\mathsf{A}, \mathcal{C}_\mathsf{B}, N_\mathsf{B}, X, R)$ and secret input $(x_\mathsf{P}, k_\mathsf{P})$ from $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$.
   (a) Common Output: $(\mathcal{D}_\mathsf{A}, \hat{\mathcal{D}}_\mathsf{A}) \in \mathbb{Z}^{*2}_{N_\mathsf{A}^2}$ and $(\mathcal{D}_\mathsf{B}, \hat{\mathcal{D}}_\mathsf{B}) \in \mathbb{Z}^{*2}_{N_\mathsf{B}^2}$.
   (b) Secret Output: Each $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ gets $\beta_\mathsf{P}, \hat{\beta}_\mathsf{P}$.
4. Each $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ sets $\begin{cases} \alpha_\mathsf{P} = \text{Dec}_\mathsf{P}(\mathcal{D}_\mathsf{P}) \bmod q \\ \hat{\alpha}_\mathsf{P} = \text{Dec}_\mathsf{P}(\hat{\mathcal{D}}_\mathsf{P}) \bmod q \end{cases}$ and outputs $(\gamma_\mathsf{P}, \alpha_\mathsf{P}, \hat{\alpha}_\mathsf{P}, \beta_\mathsf{P}, \hat{\beta}_\mathsf{P})$.

Definition 5.4 ($\mathsf{PaillierWF}^*$)**.** Define $\mathsf{PaillierWF}^*$ taking secret input $(N_\mathsf{P}, \sigma_\mathsf{P})$ from each $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ such that $\mathsf{PaillierWF}^*$ returns $(N_\mathsf{A}, N_\mathsf{B})$ to A and B if $\varphi(N_\mathsf{P}) = \sigma$ and $\gcd(N_\mathsf{P}, \sigma_\mathsf{P}) = 1$. Else, return $\perp$. (This oracle takes a Paillier public key and the corresponding private key from each party and validates that the private key is coprime to the public key.)

Definition 5.5 ($\mathsf{AffComb}^*_\lambda$)**.** Define $\mathsf{AffComb}^*_\lambda$ taking common input $(\mathcal{C}_\mathsf{A}, N_\mathsf{A}, \mathcal{C}_\mathsf{B}, N_\mathsf{B}, X, R)$ and secret input $(x_\mathsf{P}, k_\mathsf{P}) \in \mathbb{Z}_q^2$ from each $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ such that

1. If $g^{x_\mathsf{A}+x_\mathsf{B}} \neq X \in \mathbb{G}$ or $g^{k_\mathsf{A}+k_\mathsf{B}} \neq R \in \mathbb{G}$, $\mathsf{AffComb}^*_\lambda$ returns $\perp$.
2. Else, $\mathsf{AffComb}^*_\lambda$ returns $(\mathcal{D}_\mathsf{P}, \hat{\mathcal{D}}_\mathsf{P}, \beta_\mathsf{P}, \hat{\beta}_\mathsf{P}) \in \mathbb{Z}^{*2}_{N_\mathsf{P}^2} \times \mathbb{Z}_q^2$ to $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ where (for $\mathsf{Q} \in \{\mathsf{A}, \mathsf{B}\} \setminus \{\mathsf{P}\}$)

$$\begin{cases} \mathcal{D}_\mathsf{P} = \mathcal{C}_\mathsf{P}^{x_\mathsf{Q}} \cdot \mathrm{Enc}_{N_\mathsf{P}}(\nu_\mathsf{Q}) \bmod N_\mathsf{P}^2 & \text{for } \nu_\mathsf{Q} \leftarrow \mathbb{Z}_\lambda \\ \hat{\mathcal{D}}_\mathsf{P} = \mathcal{C}_\mathsf{P}^{k_\mathsf{Q}} \cdot \mathrm{Enc}_{N_\mathsf{P}}(\nu_\mathsf{Q}) \bmod N_\mathsf{P}^2 & \text{for } \hat{\nu}_\mathsf{Q} \leftarrow \mathbb{Z}_\lambda \end{cases} \text{ and } \begin{cases} \beta_\mathsf{P} = -\nu_\mathsf{P} \bmod q \\ \hat{\beta}_\mathsf{P} = -\hat{\nu}_\mathsf{P} \bmod q \end{cases}.$$

   (This oracle finalizes the VOLE operation in a verifiable way)

5.1.2 Range Proof

Definition 5.6 (IntCom^*)**.** Define IntCom^* to be a single-party oracle such that:

1. On input $(\mathtt{com}, \alpha)$, IntCom^* returns $\mathtt{X} \leftarrow \{0,1\}^{256}$ and stores $(\mathtt{X}, \alpha)$ in memory.
2. On input $(\mathtt{eval}, z, \mathtt{X}, e, \mathtt{Y})$, IntCom^* retrieves $(\mathtt{X}, \alpha)$ and $(\mathtt{Y}, \beta)$ from memory and returns $\mathtt{true}$ if $z = \alpha \cdot e + \beta$. Else, return $\mathtt{false}$.

For conciseness, we write $\mathtt{X} \leftarrow \mathsf{IntCom}^*(\alpha)$ and $\mathtt{true}/\mathtt{false} \leftarrow \mathsf{IntCom}^*(\mathtt{eval}, z, \mathtt{X}, e, \mathtt{Y})$ respectively. (This single-party oracle serves as an integer commitment scheme that verifies linear combination of commited values over $\mathbb{Z}$—rather than some finite algebraic structure)

Protocol 5.7 ($\mathsf{RngProof}$ (A, B))**.**

Oracles: $\mathsf{IntCom}^*(\cdot)$

Common Input: Paillier pks $(N_\mathsf{A}, N_\mathsf{B}) \in \mathbb{N}^2$ and Ring Elements $(\mathcal{C}_\mathsf{A}, \mathcal{C}_\mathsf{B}) \in \mathbb{Z}^*_{N_\mathsf{A}^2} \times \mathbb{Z}^*_{N_\mathsf{B}^2}$

Secret Input: Each $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ holds $(\gamma_\mathsf{P}, \rho_\mathsf{P}) \in \mathbb{Z}_q \times \mathbb{Z}^*_{N_\mathsf{C}}$ s.t. $\mathcal{C}_\mathsf{P} = \mathrm{Enc}_\mathsf{P}(\gamma_\mathsf{C}; \rho_\mathsf{P})$

Operations:

1. Each $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ does
   (a) Call $\mathtt{X}_\mathsf{P} \leftarrow \mathsf{IntCom}^*(\mathtt{com}, \gamma_\mathsf{P})$.
   (b) Call $\mathtt{Y}_\mathsf{P} \leftarrow \mathsf{IntCom}^*(\mathtt{com}, u_\mathsf{P})$ where $u_\mathsf{P} \leftarrow \mathbb{Z}_{q^3}$.
   (c) Sample $\mu_\mathsf{P} \leftarrow \mathbb{Z}^*_{N_\mathsf{P}}$, and set $\mathcal{F}_\mathsf{P} = \mathrm{Enc}_\mathsf{P}(u_\mathsf{P}; \mu_\mathsf{P}) \in \mathbb{Z}^*_{N_\mathsf{P}}$.
   (d) Send $(\mathcal{C}_\mathsf{P}, \mathcal{F}_\mathsf{P}, \mathtt{X}_\mathsf{P}, \mathtt{Y}_\mathsf{P}, z_\mathsf{P}, w_\mathsf{P})$ to $\mathsf{Q} \in \{\mathsf{A}, \mathsf{B}\} \setminus \{\mathsf{P}\}$, where

$$\begin{cases} z_\mathsf{P} = u_\mathsf{P} + e_\mathsf{P} \cdot \gamma_\mathsf{P} \quad \text{(no modulo reduction)} \\ w_\mathsf{P} = \mu_\mathsf{P} \cdot \rho_\mathsf{P}^{e_\mathsf{P}} \bmod N_\mathsf{P} \\ e_\mathsf{P} = \mathrm{HASH}(\mathsf{P}, \mathcal{C}_\mathsf{P}, \mathcal{F}_\mathsf{P}, \mathtt{X}_\mathsf{P}, \mathtt{Y}_\mathsf{P}) \end{cases}$$

2. When $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ obtains $(\mathcal{C}_\mathsf{Q}, \mathcal{F}_\mathsf{Q}, \mathtt{X}_\mathsf{Q}, \mathtt{Y}_\mathsf{Q}, z_\mathsf{Q}, w_\mathsf{Q})$ from $\mathsf{Q} \in \{\mathsf{A}, \mathsf{B}\} \setminus \{\mathsf{P}\}$, do:
   (a) Set $e_\mathsf{Q} = \mathrm{HASH}(\mathsf{P}_\mathsf{Q}, \mathcal{C}_\mathsf{Q}, \mathcal{F}_\mathsf{Q}, \mathtt{X}_\mathsf{Q}, \mathtt{Y}_\mathsf{Q})$ and $b \leftarrow \mathsf{IntCom}^*(\mathtt{eval}, z_\mathsf{Q}, \mathtt{X}_\mathsf{Q}, e_\mathsf{Q}, \mathtt{Y}_\mathsf{Q})$.
   (b) Verify that $\mathrm{Enc}_\mathsf{Q}(z_\mathsf{Q}; w_\mathsf{Q}) = \mathcal{F}_\mathsf{Q} \cdot \mathcal{C}_\mathsf{Q}^{e_\mathsf{Q}} \bmod N_\mathsf{Q}^2$ and $z_\mathsf{Q} \in \{1, \ldots, q^3\}$
   (c) If no error was detected and $b = \mathtt{true}$, output $(\mathcal{C}_\mathsf{A}, \mathcal{C}_\mathsf{B})$.
   Else, output $\perp$.

5.2 6ix1een Attack

We now describe our attack on GG_{q^5} that extracts the key in 16 signatures. Recall that the attacker corrupting A chooses a malicious Paillier modulus comprising of sixteen small primes. Then, in the j-th signature ceremony, the attacker sets $\mathcal{C}_{\mathsf{A}} = \mathrm{Enc}_{N_{\mathsf{A}}}(N_{\mathsf{A}}/p_j)$ where p_j is the j-th small factor of N_{A}. Observe that $N_{\mathsf{A}}/p_j > 2^{2000}$, suggesting that the verification in Item 2b of Protocol 5.7 or the integer commitment $\mathsf{RngProof}$ in Item 2a of Protocol 5.7 should detect such a malicious input. As we shall see next, however, there is a way for the $\mathcal{C}_{\mathsf{A}}$ to slip through without detection.

Cheating in $\mathsf{RngProof}$. The attacker cheats in $\mathsf{RngProof}$ by brute-forcing e_{A} until $e_{\mathsf{A}} = 0 \bmod p_j$ (such an e_{A} will be found with overwhelming probability because p_j is small). We note that, for such e_{A}, the attacker is able to cheat because $\mathcal{F}_{\mathsf{A}} \cdot \mathcal{C}_{\mathsf{A}}^{e_{\mathsf{A}}} = \mathcal{F}_{\mathsf{A}} \cdot \mathrm{Enc}_{N_{\mathsf{A}}}(0) \bmod N_{\mathsf{A}}^2$, and thus the range proof will not yield an error for $z_{\mathsf{A}} = u_{\mathsf{A}} + e_{\mathsf{A}} \cdot 0$ when using $\gamma_{\mathsf{A}} = 0$.

Then, when the honest party operates on C_{A} and returns $\mathcal{D}_{\mathsf{A}}$, simple data processing will yield the value of $x_{\mathsf{B}} \bmod p_i$. Iterating the attack sixteen times with different primes allows the attacker to obtain x_{B} in full, using CRT (Theorem 3.3).

Attack 5.8 (6ix1een: Corrupted A in Protocol 5.1 – $\lambda = q^5$)**.**

Operations:

1. Sample $p_1, \ldots, p_{16}$ primes of size 2^{16} and prime b such that $b \cdot \prod_{j=1}^{16} p_j \approx 2^{2048}$.

   Set $N_{\mathsf{A}} = b \cdot \prod_{j=1}^{16} p_j$ and $\sigma_{\mathsf{A}} = \varphi(N_{\mathsf{A}})$

2. In the VeVole execution of the j-th signature session do:

   (a) Set $\gamma_{\mathsf{A}} = 0$ and $\mathcal{C}_{\mathsf{A}} = \mathrm{Enc}_{\mathsf{A}}(N_{\mathsf{A}}/p_j)$ (all other values are sampled as prescribed).

   (b) When executing $\mathsf{RngProof}$ do:

   – Brute force $u_{\mathsf{A}} \leftarrow \mathbb{Z}_q$ until $e_{\mathsf{A}} = \mathrm{HASH}(\ldots) = 0 \bmod p_j$.
   (if no such value is found after 2^{32} tries, output `NoCheat`)

   (c) When obtaining $\mathcal{D}_{\mathsf{A}}$, set

$$y_j = \frac{\mathrm{Dec}_{\sigma_{\mathsf{A}}}(\mathcal{D}_{\mathsf{A}}) - [\mathrm{Dec}_{\sigma_{\mathsf{A}}}(\mathcal{D}_{\mathsf{A}})]_{N_{\mathsf{A}}/p_j}}{N_{\mathsf{A}}/p_j}$$

Multi-Party Case. In multiparty *GG18/20*, each pair of parties essentially execute Protocol 5.1, except that they need to adjust the last round messages. Specifically, letting $\alpha_{i,j}$ denote P_i's output in the OLE with P_j (when playing A) and $\beta_{i,j}$ denote P_i's ephemeral input in the OLE with P_j (when playing B), party P_i sends $\hat{s}_i, \delta_i$ where

$$\begin{cases} \hat{s}_i = \gamma_i \cdot m + r \cdot (x_{\mathsf{P}_i}\gamma_i + \sum_{j \neq i} \alpha_{i,j} + \beta_{i,j}) \bmod q \\ \delta_i = k_{\mathsf{P}_i}\gamma_i + \sum_{j \neq i} \hat{\alpha}_{i,j} + \hat{\beta}_{i,j} \bmod q \end{cases}$$

The signature is set as (r, s) where $s = (\sum_j \hat{s}_j) \cdot (\sum_j \delta_j)^{-1} \bmod q$. It is not hard to see that Adv can perform Attack 5.8 on all counterparties simultaneously, thus obtaining the key in full after sixteen signatures.

Stealthiness. To avoid causing failures, Attack 5.8 can be made stealthy as follows. The attacker reassigns $\mathcal{D}_{\mathsf{A}} := \mathcal{D}_{\mathsf{A}} \cdot (y_j \cdot N_{\mathsf{A}}/p_j)^{-1} \bmod N_{\mathsf{A}}^2$ and $\hat{\mathcal{D}}_{\mathsf{A}} := \hat{\mathcal{D}}_{\mathsf{A}} \cdot (\hat{y}_j \cdot N_{\mathsf{A}}/p_j)^{-1} \bmod N_{\mathsf{A}}^2$ where

$$\hat{y}_j = \frac{\mathrm{Dec}_{\sigma_{\mathsf{A}}}(\hat{\mathcal{D}}_{\mathsf{A}}) - [\mathrm{Dec}_{\sigma_{\mathsf{A}}}(\hat{\mathcal{D}}_{\mathsf{A}})]_{N_{\mathsf{A}}/p_j}}{N_{\mathsf{A}}/p_j}$$

and subsequently proceeding with the protocol as prescribed. That is, the attacker 'corrects' $\mathcal{D}_{\mathsf{A}}$ and $\hat{\mathcal{D}}_{\mathsf{A}}$ by the offset it caused with the malicious input, and then proceeds normally.

5.2.1 Quality of the Attack

We conclude this section by estimating the quality of Attack 5.8. Namely, we heuristically model the hash function as a random oracle and we find that Adv recovers the key after sixteen signatures, almost surely (cf. Claims 5.9 and 5.10).

Claim 5.9. *For fixed j, attacker Adv outputs $\texttt{NoCheat}$ with probability at most 2^{-1000}.*

Proof. Modelling the hash function as a random oracle, it holds that $\Pr[e_{\mathsf{A}} \neq 0 \bmod p_j] = (1 - 1/p_j)$ in a single trial. Thus, $\Pr[\texttt{NoCheat}] = (1 - 1/p_j)^{2^{32}} \leq \exp(-2^{32}/p_j) < 2^{-1000}$ since $p_j \approx 2^{16}$. □

Claim 5.10. *For fixed j, it holds that $x_{\mathsf{B}} \bmod p_j = \frac{\mathrm{Dec}_{\sigma_{\mathsf{A}}}(\mathcal{D}_{\mathsf{A}}) - [\mathrm{Dec}_{\sigma_{\mathsf{A}}}(\mathcal{D}_{\mathsf{A}})]_{N_{\mathsf{A}}/p_j}}{N_{\mathsf{A}}/p_j}$.*

Proof. By the definition of the $\mathsf{AffComb}^*_\lambda$ oracle, notice that $\mathcal{D}_{\mathsf{A}} = \mathrm{Enc}_{N_{\mathsf{A}}}([x_{\mathsf{B}} \cdot N_{\mathsf{A}}/p_j + \nu_{\mathsf{B}}]_{N_{\mathsf{A}}})$ where $\nu_{\mathsf{B}} \in \{1, \ldots, q^5\}$. Thus, since $\nu_{\mathsf{B}} < N_{\mathsf{A}}/p_j$ (because N_{A} is 2048 bits and p_j is roughly 16 bits) it follows that

$$\begin{aligned} \mathrm{Dec}_{N_{\mathsf{A}}}(\mathcal{D}_{\mathsf{A}}) - [\mathrm{Dec}_{N_{\mathsf{A}}}(\mathcal{D}_{\mathsf{A}})]_{N_{\mathsf{A}}/p_j} &= [x_{\mathsf{B}} \cdot N_{\mathsf{A}}/p_j + \nu_{\mathsf{B}}]_{N_{\mathsf{A}}} - \nu_{\mathsf{B}} \\ &= [x_{\mathsf{B}}]_{p_j} \cdot N_{\mathsf{A}}/p_j \end{aligned}$$

□

5.3 Death by 1M Cuts Attack

Attack 5.11 (Death by 1M Cuts: Corrupted A in Protocol 5.1 for $\lambda = 2^{2048}$)**.**

Operations:

1. Same as items Item 1 in Attack 5.8.
2. In the VeVole execution of the j-th signature session sample $(\alpha, \beta) \leftarrow \mathbb{Z}_{p_j}$ and do:
   (a) Same as items Items 2a and 2b in Attack 5.8.
   (b) When obtaining $\mathcal{D}_{\mathsf{A}}, \hat{\mathcal{D}}_{\mathsf{A}}$, reasign (continue the process as the protocol prescribes hereafter)

$$\begin{cases} \mathcal{D}_{\mathsf{A}} & := \mathcal{D}_{\mathsf{A}} \cdot \mathrm{Enc}_{N_{\mathsf{A}}}(-\alpha \cdot (N_{\mathsf{A}}/p_j)) \bmod N_{\mathsf{A}}^2 \\ \hat{\mathcal{D}}_{\mathsf{A}} & := \hat{\mathcal{D}}_{\mathsf{A}} \cdot \mathrm{Enc}_{N_{\mathsf{A}}}(-\beta \cdot (N_{\mathsf{A}}/p_j)) \bmod N_{\mathsf{A}}^2 \end{cases}$$

   (c) If the process terminates in a valid signature deduce that $(x_{\mathsf{B}}, k_{\mathsf{B}}) = (\alpha, \beta) \bmod p_j$.

We now describe our attack on $\mathsf{GG}_{2^{2048}}$ that extracts the key in 16 *successful* signatures, i.e. leakage is only obtained when the signature process yields a valid signature. The attack proceeds almost identically to Attack 5.8 except that, when the attacker obtains $\mathcal{D}_{\mathsf{A}}$ and $\hat{\mathcal{D}}_{\mathsf{A}}$, it reassigns those value according to a (random) guess of the pair $(x_{\mathsf{B}}, k_{\mathsf{B}}) \bmod p_j$ and it continues the execution as the protocol prescribes.

In the end, if the the execution leads to valid signature, the attacker deduces the value of $(x_{\mathsf{B}}, k_{\mathsf{B}}) \bmod p_j$, In the event that the signature is invalid, no leakage is obtained on x_{B}, because of the k_{B} which is a fresh random value every time, and thus the attack must be repeated with the same p_j.

5.3.1 Quality of the Attack

We conclude this section by estimating the quality of Attack 5.11.

Claim 5.12. *For fixed j, attacker* Adv *outputs* `NoCheat` *with probability at most 2^{-1000}.*

Proof. Same as Claim 5.9. □

Claim 5.13. *Using the notation from Attack 5.11, in the j-th iteration, if $(x_{\mathsf{B}}, k_{\mathsf{B}}) \neq (a, b) \bmod p_j$ and $m \neq -(x_{\mathsf{A}} + x_{\mathsf{B}})r \bmod q$ then the protocol yields an invalid signature with probability at least $1 - p_j^2/q \approx 1$.*

Proof. Let $\varepsilon = -[x_{\mathsf{B}} - a]_{p_j} \cdot N/p_j \bmod q$ and $\hat{\varepsilon} = -[k_{\mathsf{B}} - b]_{p_j} \cdot N/p_j \bmod q$. Write s' for the signature string reconstructed by the parties, and note that

$$\begin{aligned} s' &= (\delta_{\mathsf{A}} + \delta_{\mathsf{B}} + \hat{\varepsilon})^{-1} \cdot (\hat{s}_{\mathsf{A}} + \hat{s}_{\mathsf{B}} + \varepsilon) \\ &= (\gamma \cdot (m + rx) + \varepsilon) \cdot (k\gamma + \hat{\varepsilon})^{-1} \\ &= (s + \varepsilon \cdot (k\gamma)^{-1}) \cdot (1 + \hat{\varepsilon} \cdot (k\gamma)^{-1})^{-1} \bmod q, \end{aligned}$$

where $x = x_{\mathsf{A}} + x_{\mathsf{B}}$, $\gamma = \gamma_{\mathsf{A}} + \gamma_{\mathsf{B}}$ and $k = k_{\mathsf{A}} + k_{\mathsf{B}} \bmod q$. So, assuming $m = \mathrm{HASH}(\mathrm{msg}) \neq -xr \bmod q$, note that the signature verifies if $s' = s \bmod q$ which, letting $\rho = (k\gamma)^{-1}$, is equivalent to $s \cdot (1 + \hat{\varepsilon}\rho) = (s + \varepsilon \cdot \rho) \bmod q$ and thus $s = \varepsilon \cdot (\hat{\varepsilon})^{-1} \bmod q$ (assuming $\rho, \hat{\varepsilon} \neq 0$). For random $k \leftarrow \mathbb{Z}_q$, $\varepsilon \cdot (\hat{\varepsilon})^{-1}$ has at most p_j^2 possible values whereas s has q. Thus, with probability at least $1 - p_j^2/q$, it holds that $s \neq s'$ and the signature invalid. □

On the number of required signatures for extracting the key. We conclude this section by estimating the number of signature sessions required in order to extract the key. Recall that Attack 5.11 yields a valid signature (and thus useful leakage) only when $(a, b) = (x_{\mathsf{B}}, k_{\mathsf{B}}) \bmod q$, i.e. the attacker correctly guesses the remainders of both x_{B} and k_{B} modulo p_j. The claim below relates the key-extraction probability to the the number of signatures, with respect to parameter $\tau \in [0, 1]$ (which captures the probability that a single remainder was extracted successfully).

Claim 5.14. *For fixed $\tau \in [0, 1]$, letting $\ell \in \mathbb{N}$ denote the number of primes, Attack 5.11 successfully extracts the key with probability at least τ^ℓ after $\sum_{i=1}^{\ell} f_\tau(p_i)$ signatures, where $f_\tau(p) = \lceil \log(1 - \tau) / \log(1 - 1/p^2) \rceil$.*

Proof. We know that the attack yields a valid signature for a given p_j with probability $1/p_j^2$. Thus, after $f_\tau(p_j)$ tries, our attack does not yield leakage with probability $(1 - 1/p_j^2)^{f_\tau(p_j)} \leq 1 - \tau$ (by the definition of f_τ), and the claim follows immediately. □

In conclusion, when combining with brute-force techniques, Attack 5.11 extracts the key with probability 0.44 after 1.4×10^6 signatures (choosing $\{p_1, \ldots, p_\ell\} = \{3, 5, 7, \ldots, 173\}$, i.e. the first 39 odd primes). If the Paillier moduli are checked for very small factors (as many implementations do), then the malicious modulus can be suitably chosen to avoid detection (though it makes the attack more expensive in terms of signatures required to extract the key). For instance, choosing $\{p_1, \ldots, p_\ell\} = \{6481, 6491, \ldots, 6653\}$, Attack 5.11 extracts the key with probability 0.15 after 1.8×10^9 signatures.

6 Our Attack on *BitGoTSS*

In this section, we present the Zero Proof attack on *BitGoTSS*. Since the protocol is quite similar to Protocol 5.1 (in fact it is a bare-bones version), we only explain how it differs from Protocol 5.1.

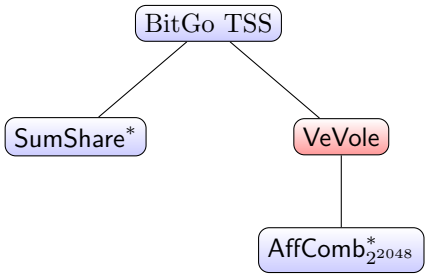

Figure 2: Illustration of the BitGo TSS protocol dependencies. Zero Proof targets the VeVole subprotocol, and the beta parameter is chosen from the maximum range ($\lambda = 2^{2048}$).

BitGoTSS. In a nutshell, *BitGoTSS* protocol is the same as Protocol 5.1 except that there is no range proof subprotocol (RngProof) or well-formedness check (PaillierWF*) when invoking the Verifiable VOLE protocol. Instead, each party simply sends the relevant values over the communication channel, namely the Paillier public key N_A and ciphertext $\mathcal{C}_\mathsf{A}$, which are used to carry out the protocol to its conclusion.

6.1 Zero Proof Attack

Claim 6.1. *Using the notation from Attack 6.2, it holds that* $y_j = x_\mathsf{B} \bmod p_j$ *for all* $j \in \{1, \ldots, 16\}$.

Proof. For some $\mu, \nu \in \mathbb{Z}_N^*$ chosen by B, note that

$$\begin{aligned}\mathcal{D}_\mathsf{A} &= 4^{x_\mathsf{B}} \cdot (1 + \mu \cdot N) \cdot \nu^{N_\mathsf{A}} \bmod N_\mathsf{A}^2 = 4^{x_\mathsf{B}} \cdot \nu^{N_\mathsf{A}} \bmod N_\mathsf{A} \\ &= 4^{x_\mathsf{B}} \cdot (\nu^{p_j})^{q_j b \prod_{\ell \neq j} p_\ell q_\ell} \bmod q_j = \begin{cases} 4^{x_\mathsf{B}} & \text{if } \nu \text{ is a square mod } q_j \\ -4^{x_\mathsf{B}} & \text{otherwise} \end{cases},\end{aligned}$$

where the last equality holds by Lagrange's theorem (because $\mathbb{Z}_{q_j}^*$ has order $2p_j$). In conclusion, since 4 has order p_j in $\mathbb{Z}_{q_j}$, we deduce that $\mathcal{D}_\mathsf{A}^{2 \cdot [2^{-1}]_{p_j}} = 4^{2 \cdot [2^{-1}]_{p_j} \cdot x_\mathsf{B}} = 4^{x_\mathsf{B}} \bmod q_j$ and $y_j = x_\mathsf{B} \bmod p_j$, as desired. □

Attack 6.2 (Zero Proof: Adv corrupting A in *BitGoTSS*)**.**

Operations:

1. Sample $q_1, \dots, q_{16}$ strong primes of size 2^{17}, i.e. such that q_j and $p_j = (q_j - 1)/2$ are both primes, for all $j \in \{1, \dots, 16\}$. Sample arbitrary prime b s.t. $b \cdot \prod_{j=1}^{16} q_j p_j \approx 2^{2048}$.

   Set $N_{\mathsf{A}} = b \cdot \prod_{j=1}^{16} q_j p_j$.

2. When signing, do: (only one signature ceremony)
   (a) Set $\mathcal{C}_{\mathsf{A}} = 4$. After obtaining $\mathcal{D}_{\mathsf{A}}$, do:
   (b) For $j \in \{1, \dots, 16\}$, brute force $y_j \in \{1, \dots, p_j\}$ such that

$$4^{y_j} = \mathcal{D}_{\mathsf{A}}^{2 \cdot [2^{-1}]_{p_j}} \bmod q_j.$$

   (c) When obtaining $y_j = x_{\mathsf{B}} \bmod p_j$ for all j, reconstruct x_{B} using CRT (Theorem 3.3).

Multi-Party Case. Similarly to Attack 5.8, Attack 6.2 retrieves the key in a single signature regardless of the number of parties, because all the honest parties fall into the same trap and calculate $\mathcal{D}$ in the same way.

Stealthiness. 'Stealthifying' Attack 6.2 is somewhat tricky because the Pallier key is so distorted that it is not obvious how use it in order to 'decrypt' $\mathcal{D}_{\mathsf{A}}$ (since $\mathcal{D}_{\mathsf{A}}$ is not even a ciphertext for the chosen Paillier key). However, with the knowledge of x_{B}, the attacker can compute $\mathcal{D}_{\mathsf{A}} \cdot 4^{-x_{\mathsf{B}}} = \mathrm{Enc}_{N_{\mathsf{A}}}(\nu_{\mathsf{B}}) \bmod N_{\mathsf{A}}^2$ and subsequently infer ν_{B}. By following a similar process for $\hat{\mathcal{D}}_{\mathsf{A}}$, the attacker can extract k_{B} and deduce $\hat{\nu}_{\mathsf{B}}$, and, in conclusion, the attacker sets $\gamma_{\mathsf{A}} = 0$ and

$$\begin{cases} \hat{s}_{\mathsf{A}} = \nu_{\mathsf{B}} + \beta_{\mathsf{A}} \bmod q \\ \delta_{\mathsf{A}} = \hat{\nu}_{\mathsf{B}} + \hat{\beta}_{\mathsf{A}} \bmod q \end{cases}$$

which yields an error-free signature process.

Acknowledgments

We thank Dr. Ruan Pingcheng of OKX for identifying minor errors and suggesting fixes.

References

[AS20a] Jean-Philippe Aumasson and Omer Shlomovits. "Attacking Threshold Wallets". In: *IACR Cryptol. ePrint Arch.* (2020), p. 1052. URL: https://eprint.iacr.org/2020/1052 (cit. on p. 2).

[AS20b] Jean-Philippe Aumasson and Omer Shlomovits. "Multiple Bugs in Multi-Party Computation: Breaking Cryptocurrency's Strongest Wallets". Briefings, Blackhat USA 2020. 2020 (cit. on p. 2).

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. "Short Signatures from the Weil Pairing". In: *J. Cryptology* 17.4 (2004), pp. 297–319 (cit. on p. 1).

[BMP22] Constantin Blokh, Nikolaos Makriyannis, and Udi Peled. "Efficient Asymmetric Threshold ECDSA for MPC-based Cold Storage". In: *IACR Cryptol. ePrint Arch.* (2022), p. 1296. URL: https://eprint.iacr.org/2022/1296 (cit. on p. 4).

[Boy+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. "Efficient Pseudorandom Correlation Generators: Silent OT Extension and More". In: *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11694. Lecture Notes in Computer Science. Springer, 2019, pp. 489–518 (cit. on p. 1).

[Can+20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. "UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts". In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM, 2020, pp. 1769–1787 (cit. on p. 2).

[CMP20] Ran Canetti, Nikolaos Makriyannis, and Udi Peled. *UC Non-Interactive, Proactive, Threshold ECDSA*. Cryptology ePrint Archive, Paper 2020/492. 2020 (cit. on pp. 2, 5).

[Des87] Yvo Desmedt. "Society and Group Oriented Cryptography: A New Concept". In: *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*. 1987, pp. 120–127 (cit. on p. 1).

[DF89] Yvo Desmedt and Yair Frankel. "Threshold Cryptosystems". In: *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*. 1989, pp. 307–315 (cit. on p. 1).

[DKLs23] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. "Threshold ECDSA in Three Rounds". In: *IACR Cryptol. ePrint Arch.* (2023), p. 765. URL: https://eprint.iacr.org/2023/765 (cit. on pp. 13, 14).

[fir23a] fireblocks-labs. *bitgo-tss-exploit-poc*. https://github.com/fireblocks-labs/bitgo-tss-exploit-poc. Accessed: 20-06-2024. 2023 (cit. on pp. 3, 10).

[fir23b] fireblocks-labs. *safeheron-gg20-exploit-poc*. https://github.com/fireblocks-labs/safeheron-gg20-exploit-poc. Accessed: 20-06-2024. 2023 (cit. on pp. 3, 8).

[fir23c] fireblocks-labs. *zengo-lindell17-exploit-poc*. https://github.com/fireblocks-labs/zengo-lindell17-exploit-poc. Accessed: 20-06-2024. 2023 (cit. on pp. 3, 8).

[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, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018, pp. 1179–1194 (cit. on pp. 2, 3, 7).

[GG20] Rosario Gennaro and Steven Goldfeder. *One Round Threshold ECDSA with Identifiable Abort*. Cryptology ePrint Archive, Paper 2020/540. 2020 (cit. on pp. 2, 3, 7).

[GMR85] S Goldwasser, S Micali, and C Rackoff. "The knowledge complexity of interactive proof-systems". In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*. STOC '85. Providence, Rhode Island, USA: Association for Computing Machinery, 1985, pp. 291–304. ISBN: 0897911512. DOI: 10.1145/22145.22178. URL: https://doi.org/10.1145/22145.22178 (cit. on p. 1).

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*. 1987, pp. 218–229 (cit. on p. 1).

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems". In: *J. ACM* 38.3 (July 1991), pp. 690–728. ISSN: 0004-5411. DOI: 10.1145/116825.116852. URL: https://doi.org/10.1145/116825.116852 (cit. on p. 1).

[Lin17a] Yehuda Lindell. "Fast Secure Two-Party ECDSA Signing". In: *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II*. 2017, pp. 613–644 (cit. on pp. 1, 4, 7, 8).

[Lin17b] Yehuda Lindell. "Fast Secure Two-Party ECDSA Signing". In: *IACR Cryptol. ePrint Arch.* (2017), p. 552 (cit. on p. 8).

[Lin21] Yehuda Lindell. "Fast Secure Two-Party ECDSA Signing". In: *J. Cryptol.* 34.4 (2021), p. 44. DOI: 10.1007/s00145-021-09409-9 (cit. on p. 1).

[MP21] Nikolaos Makriyannis and Udi Peled. "A Note on the Security of GG18". 2021 (cit. on pp. 2, 3).

[Nat23] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. Federal Information Processing Publication 186-5. 2023 (cit. on p. 1).

[Ngu+23a] Duy Hieu Nguyen, Anh Khoa Nguyen, Huu Giap Nguyen, Thanh Nguyen, and Anh Quynh Nguyen. "New Key Extraction Attacks on Threshold ECDSA Implementations". 2023 (cit. on p. 2).

[Ngu+23b] Duy Hieu Nguyen, Anh Khoa Nguyen, Huu Giap Nguyen, Thanh Nguyen, and Anh Quynh Nguyen. "TSSHOCK: Breaking MPC Wallets and Digital Custodians for *BILLION* Profit". Briefings, Blackhat USA 2023. 2023 (cit. on p. 2).

[Pai99] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*. Ed. by Jacques Stern. Vol. 1592. Lecture Notes in Computer Science. Springer, 1999, pp. 223–238 (cit. on p. 1).

[Rab05] Michael O. Rabin. *How To Exchange Secrets with Oblivious Transfer*. Cryptology ePrint Archive, Report 2005/187. 2005 (cit. on p. 1).

[Sch91] Claus-Peter Schnorr. "Efficient Signature Generation by Smart Cards". In: *J. Cryptol.* 4.3 (1991), pp. 161–174 (cit. on p. 1).

[TS21] Dmytro Tymokhanov and Omer Shlomovits. “Alpha-Rays: Key Extraction Attacks on Threshold ECDSA Implementations”. In: *IACR Cryptol. ePrint Arch.* (2021), p. 1621. URL: https://eprint.iacr.org/2021/1621 (cit. on pp. 2, 10).

[Yao86] Andrew Chi-Chih Yao. “How to generate and exchange secrets”. In: *IEEE* (1986) (cit. on p. 1).