Truncator: Time-space Tradeoff of Cryptographic Primitives

Foteini Baldimtsi[1,2], Konstantinos Chalkias[2], Panagiotis Chatzigiannis[3], and Mahimna Kelkar[4]

[1] George Mason University foteini@gmu.edu
[2] Mysten Labs kostas@mystenlabs.com
[3] Visa Research pchatzig@visa.com
[4] Cornell University mahimna@cs.cornell.edu

Abstract. We present mining-based techniques to reduce the size of various cryptographic outputs without loss of security. Our approach can be generalized for multiple primitives, such as cryptographic key generation, signing, hashing and encryption schemes, by introducing a brute-forcing step to provers/senders aiming at compressing submitted cryptographic material.

Interestingly, mining can result in record-size cryptographic outputs, and we show that 5%-12% shorter hash digests and signatures are practically feasible even with commodity hardware. As a result, our techniques make compressing addresses and transaction signatures possible in order to pay less fees in blockchain applications while decreasing the demand for blockchain space, a major bottleneck for initial syncing, communication and storage. Also, the effects of "compressing once - then reuse" at mass scale can be economically profitable in the long run for both the Web2 and Web3 ecosystems.

Our paradigm relies on a brute-force search operation in order to craft the primitive's output such that it fits into fewer bytes, while the "missing" fixed bytes are implied by the system parameters and omitted from the actual communication. While such compression requires computational effort depending on the level of compression, this cost is only paid at the source (i.e., in blockchains, senders are rewarded by lowered transaction fees), and the benefits of the compression are enjoyed by the whole ecosystem. As a starting point, we show how our paradigm applies to some basic primitives commonly used in blockchain applications but also traditional Web2 transactions (such as shorter digital certificates), and show how security is preserved using a bit security framework. Surprisingly, we also identified cases where wise mining strategies require proportionally less effort than naive brute-forcing, shorter hash-based signatures being one of the best examples. We also evaluate our approach for several primitives based on different levels of compression. Our evaluation concretely demonstrates the benefits both in terms of financial cost and storage if adopted by the community, and we showcase how our technique can achieve up to 83.21% reduction in smart contract gas fees at a cost of less than 4 seconds of computation on a single core.

1 Introduction

In blockchain applications like Bitcoin [33] and Ethereum [42], a distributed common ledger is maintained among all participants. As the size of the ledger monotonically increases, most blockchains have large storage requirements for nodes, which can be several hundreds of gigabytes, even after applying techniques to prune or compact the needed storage. In addition, there is typically an upper bound of storage space per block (e.g., in Bitcoin it is 1MB [9])[5], while concretely for Bitcoin at the time of writing, 250 bytes roughly cost $1.5 of transaction fees to be included in next block with high probability, while minting an ERC721 token which implements the Non-Fungible Token Standard (NFT) [35] using a 33-byte IPFS link [13] costs $4.20. Therefore, blockchain space is a scarce resource, and typically there are mechanisms in place to disincentivize posting large amounts of data to the public ledger (e.g., in cryptocurrencies, transaction fees are proportional to the size of that transaction in bytes).

In this work, we present several methods to trade off storage for computation in several cryptographic primitives used in Web2 (i.e., succinct TLS keys and certificates), blockchain applications, and generally in systems that can tolerate sender's work to be more demanding and time-consuming, especially when:

- receivers have limited resources (i.e., mobile, IoT);
- storage or data-size is financially expensive (i.e., blockchains, cloud storage and ingress cost);
- multiple recipients perform verification/decryption/lookup (i.e., blockchains, TLS digital certificates, IPFS lookups).

Our approach is based on the principle that a few extra operations on behalf of the transaction's sender, which could require crafting a valid transaction in a few seconds or minutes instead of milliseconds, would be beneficial for the sender (by lowering transaction fees) as well as all other blockchain participants (by fitting more transactions per block or by reducing the overall blockchain size by a constant factor). Our underlying paradigm for these operations is a brute-force search in the cryptographic primitive's inputs/randomness, in order to craft each primitive's output in a specific way that satisfies the modified system's public parameters, e.g., requiring some specific bits of the output to be constant. This enables us to omit these bits from the output entirely, as these implied constant bits can be "glued back" to the output by the receiver, effectively allocating fewer bits per such output for communication and storage. For each primitive, we argue that security is preserved compared to the standard primitive.

Finally, while in this paper we focus on the basic cryptographic primitives commonly used in the blockchain space (where storage costs are of particularly importance), our techniques can be potentially extended to a much wider spectrum of cryptographic protocols (e.g., zero-knowledge proofs, lattices, multiparty computation etc.) with the level of potential benefits depending on the specific application where these primitives are deployed.

[5] In Ethereum there is no upper bound in block size, but each block has a maximum total gas [1], which has a similar effect.

1.1 Overview of our approach

As discussed above, our approach will be an iterative search of the primitive's input such that the conditions we require for the primitive's output are satisfied. As a first example, in the key generation algorithm for *discrete log*-based keys, we perform an iterative search for a secret key sk such that its derived public key $\mathsf{pk} = g^{\mathsf{sk}}$ has a pre-determined ℓ-bit *prefix.* On other probabilistic primitives, e.g., in a public key encryption scheme, we can simply brute-force the scheme's randomness to achieve the desired truncation. However if we need to truncate a deterministic primitive (e.g., a hash function), a nonce (or salt) must be used. Another possible technique is to introduce randomness within the primitive's payload without altering its semantics, e.g., slightly altering pixels in images to nearby colors, or replacing spaces with non-printable characters in text files. The latter approach is easy to implement, without modifying cryptographic primitives, but is only applicable within certain application scenarios.

Based on the above, we distinguish between the 2 main ways of a brute-force search on a primitive's input: brute-forcing the internal randomness of a primitive (if any) or brute-forcing the primitive's payload (e.g., the message of a signature). Brute-forcing the payload can be implemented in 3 different ways:

- Use a nonce, and send it along with the payload. This method is preferable if the application already includes such a nonce.
- Use a nonce, and have the receiver perform a brute-force operation as well to recover it. A similar method was recently proposed by Pornin for signatures [37], but this is not suitable for our applications, as in those we try to optimize on the verifier's side.
- Brute-force the payload directly without changing its semantics, e.g., slightly altering pixels, use non-printable chars instead of spaces etc.

Randomness search. In the case where we perform a brute-force search on a primitive's randomness, it is particularly important on how this search is algorithmically performed. Simply incrementing the randomness can potentially lead to attacks in some applications, (e.g., in RSA, two random values having a difference of 1 might result in the same key pair from the primality checks) and the safest way is to generate fresh randomness for each iteration [10,11]. While this is costlier in terms of computation (it needs to invoke the system's pseudorandom number generator `/dev/urandom` each time), a potential cheaper alternative is to increment by a large constant instead (although this needs to be carefully considered for each primitive). For the case of hash functions, randomness generation should be performed similarly as in Randomized Hashing for Digital Signatures [24]. We also highlight a recent attack on an Ethereum vanity address generator [6], where the randomness for brute-forcing the prefix on addresses was only 32 bits, making a reverse brute-force search to recover the corresponding private keys feasible, which in turn led to loss of funds [8].

The role of bit security. In our work, we will use a recent bit security framework [41] to analyze the security of our proposed scheme modifications. Bit-security is commonly used to describe the level of security offered by a concrete

instantiation of a cryptographic primitive P and offers a middle ground approach between the common asymptotic proof approach and the concrete security approach. Informally, we say that P has κ-bit security if it takes an adversary 2^κ operations to break it, or alternatively, an efficient adversary breaks the scheme with at most $\epsilon < 2^{-\kappa}$ probability. This implies that for any attack with computational cost T and success probability ϵ, it must hold that $T/\epsilon > 2^\kappa$. Intuitively, bit-security captures that P is as secure as an idealized perfect cryptographic primitive with an κ-bit key.

Our results. We show how to apply our truncation paradigm on some common cryptographic primitives, such as hash digests, ECC public keys and signature outputs, resulting in about 7% compression (2 bytes less) in less than a second for ed25519 signatures, and less than 10 milliseconds for compressed Blake3 digests. We also discuss truncation in ElGamal enryption and Diffie-Hellman based encryption (used for blockchain one-time wallet addresses known as *stealth addresses*, which are used to improve anonymity in cryptocurrency transactions). Using the framework by Watanabe and Yasunaga [41], we show that bit security of the original primitive is preserved after our modifications, and we evaluate the computational overhead compared to the communication/storage savings. In addition, we consider primitives that involve an auxiliary output such as Winternitz one-time signatures, and we show how our paradigm has the potential to be even more efficient when applied in these cases. Note that these "truncated" versions of the primitives we considered only serve as a starting point, as our paradigm can be applied to the whole space of cryptography.

Evaluation. To showcase the usefulness and impact of our truncation paradigm in blockchain applications, we first show that for discrete-log based primitives such as EC public keys and signatures, we can truncate 1 bit in 256-bit curves at the cost of 3ms - 30ms computation depending on the curve, primitive and hardware. In addition, for hash-based primitives, we show that in 256-bit hash functions, 1 bit can be truncated at the cost of 36µs - 166µs computation depending on the hash function and the hardware. We also evaluate the potential for truncating primitives that are based on a combination of elliptic curve and hash operations, which approximate the costs of truncating discrete-log based primitives as expected.

With those costs in mind, we showcase the potential savings in common blockchain applications. We first use Bitcoin as an example, as its transactions include a combination of primitives that can be truncated, namely public keys, signatures and hashes, and show that with modest truncation we can easily achieve savings in transaction fees and block space by 3%. Most importantly however, our approach proves to be highly effective in smart contract applications such as ERC721 contracts, where gas fees can be reduced by over 83% at the computational cost of only 3.84 seconds per contract call on average.

1.2 Related works

Knudsen [29] first observed the possibility of truncating the output of a salted hash function after agreeing on a publicly-known ℓ-bit prefix, by performing a

brute-force computation on the salt, and showed that this increased security against collision attacks by $2^{\ell/2}$ while still maintains security against preimage attacks (without a formal argument for the second). A 2012 publication by NIST [25] discussed the impact of truncating the output of hash function to their security. Note however this publication only considered a “naive” truncation without considering a publicly known fixed prefix as in [29], which naturally reduces the hash function’s security.

A more recent work by Pornin [37] presented techniques to reduce the size of EdDSA and ECDSA signatures, however these techniques required computational work on behalf of the verifier. Kudinov et al. [38] applied a similar technique for Winternitz one-time signature scheme (WOTS), which we compare and discuss in detail in Section A.2. Ethereum developers also proposed the use of addresses with a prefix of many zeroes in order to reduce gas fees (called “gas golfing”) [2]. Also a recent work by Fleischhacker et al. [27] presented algorithms on compressing sparsely-encrypted vectors. A recent work by Blocki and Lee [15] showed how to compress Schnorr signatures, however this type of compression might affect security [21]. Finally, Fregly et al. [28] proposed a new technique called Merkle Tree Ladders for compressing signature schemes. Nevertheless, our approach is orthogonal to the last two approaches and can be applied on top of such techniques.

In the blockchain space, a technique that resembles our paradigm is Bitcoin vanity address generator [7], which attempts to create a new valid Bitcoin public address (i.e., a double-hashed ECDSA public key) given a user-specified address prefix. Later, this approach was leveraged to create slightly shorter signatures in Bitcoin [3,5]. We note that our approach is orthogonal to other blockchain storage compressing techniques (such as zk-SNARKs [4,17] or accumulators [16]), as such techniques can be naturally applied on top of a blockchain that has already compressed cryptographic material through Truncator.

Paper organization. The rest of our paper is organized as follows: In section 2 we provide the necessary cryptographic background. In section 3 we show the methodology of our truncation paradigm on hash functions, public keys and signatures. In section 4 we showcase the concrete benefits of our methods. In section 5 we discuss the potential impact of Truncator, and conclude in section 6. In the Appendix we provide additional examples of truncated primitives (as in some cases the benefits would be even higher), and also provide a “FAQ” section that addresses some common misconceptions on our methodologies.

2 Preliminaries

Notation. We denote a probabilistic polynomial-time (PPT) algorithm B with input a and output b as $b \xleftarrow{\$} B(a)$. We denote the security parameter by λ, the bit security by κ and the truncation parameter (i.e., the number of bits truncated) by ℓ.

2.1 Computational Hardness Assumptions

Definition 1 (Discrete-logarithm problem). *The discrete-logarithm problem for a cyclic group* $\mathbb{G}$ *of order* $q = |\lambda|$ *is hard if* $\forall$ *ppt algorithms* $\mathcal{A}$, $\exists$ *negligible function negl s.t.:*

$$\Pr\begin{bmatrix} g \text{ generator of } \mathbb{G}; \\ h \xleftarrow{\$} \mathbb{G}; \\ x \leftarrow \mathcal{A}(g,h); \\ \text{if } g^x = h \text{ output } 1, \text{else output } 0 \end{bmatrix} = 1 \leq negl(\lambda)$$

2.2 Definitions of Cryptographic Primitives

Definition 2 (Hard Relation). *A relation* R *with a randomized PPT sampling algorithm* Gen *is a hard relation if:*

- *For any* $(x, y) \xleftarrow{\$} \mathsf{Gen}()$ *we have* $(x, y) \in R$.
- $\exists$ *a PPT algorithm that decides if* $(x, y) \in R$.
- $\forall$ *PPT algorithms* $\mathcal{A}$, $\Pr\left[(x, y) \xleftarrow{\$} \mathsf{Gen}(); x^* \leftarrow \mathcal{A}(y); R(x^*, y) = 1\right] \leq negl(\lambda)$

2.3 Bit security of cryptographic primitives

We use the bit-security framework defined in recent work by Watanabe and Yasunaga [41] and provide an brief overview here (the alternative framework from [32] can also be used instead and gives the same results).

Basic intuition. Abstractly, if a cryptographic primitive has κ-bit security, then the intuition is that any adversary would need at least 2^κ operations to break it where the computational cost comes from the security game played by the adversary and the challenger. To precisely quantify bit security, the framework models two adversaries: an *inner* adversary $\mathcal{A}$ which plays the "usual" security game against the challenger, and an *outer* adversary $\mathcal{B}$ which invokes $\mathcal{A}$ a total of $N_{\mathcal{A},\mathcal{B}}$ times in order to amplify its final winning probability $\epsilon_{\mathcal{A},\mathcal{B}}$.

In an κ-bit security game, the challenger chooses a secret $u \in \{0,1\}^\kappa$ uniformly at random, and sends the challenge $X(u)$ to $\mathcal{A}$. The game is classified as a *search* game when $u >> 1$, and as a *decision* game when $u = 1$. For instance in the IND-CPA security game, $\mathcal{A}$'s goal is to distinguish between two encryptions (i.e., u is 0 or 1) while in the simple discrete-logarithm experiment the adversary's goal is to output the value of u for a challenge g^u. Based on this distinction, the framework's structure is somewhat different according to the security game type. In search games, each inner adversary is invoked with a fresh random secret u, and the probability that $\mathcal{B}$ wins is defined as the probability that *some* $\mathcal{A}$ wins (i.e., finds the appropriate search quantity). In contrast, for decision games, each inner adversary plays an independent game with consistent secret u across all invocations. $\mathcal{B}$ can use the outputs from all inner adversaries to produce its output u'; its probability of winning can now be defined as $\Pr[u = u']$.

Definition 3 (Bit Security [41]). *The bit security of an κ-bit game* G *is defined by:*

$$\begin{aligned}\mathrm{BS}^{\mu}_{\mathrm{G}} &= \min_{\mathcal{A},\mathcal{B}} \{\log_2(N_{\mathcal{A},\mathcal{B}} \cdot T_{\mathcal{A}}) : \epsilon_{\mathcal{A},\mathcal{B}} \geq 1-\mu\} \\ &= \min_{\mathcal{A}} \left\{\log_2(T_{\mathcal{A}}) + \log_2\left(\min_{\mathcal{B}}\{N_{\mathcal{A},\mathcal{B}} : \epsilon_{\mathcal{A},\mathcal{B}} \geq 1-\mu\}\right)\right\}\end{aligned}$$

where $N_{\mathcal{A},\mathcal{B}}$ is the number of instances of $\mathcal{A}$ invoked by $\mathcal{B}$, $T_{\mathcal{A}}$ is the computational complexity for playing the inner game by $\mathcal{A}$ and $\mu > 0$ some small constant for $\mathcal{B}$ success probability $1-\mu$.

Based on the above definition and Theorems 1 and 2 provided in [41], the framework for a primitive with a search-type game provides an approximate bit security of $\kappa = \log_2(T_{\mathcal{A}}/\epsilon_{\mathcal{A}})$ where $\epsilon_{\mathcal{A}}$ is $\mathcal{A}$'s success probability in the security game. For primitives with decision-type games (e.g., PRG, encryption, DDH) the framework approximates a lower bound for $\kappa \geq \log_2(T_{\mathcal{A}}/\delta)$ where δ is the advantage of $\mathcal{A}$ when playing a decision game.

3 Truncating cryptographic primitive outputs

Using the bit-security framework from Section 2.3, we now show how a number of different primitives can be compressed or truncated without affecting their concrete security.

3.1 Truncated Hash Functions

We first consider the truncation of simple hash functions. Let $H : \{0,1\}^* \to Y = \{0,1\}^\lambda$ be a (cryptographic) hash function. Suppose that we wish to communicate the λ-bit hash output $H(x)$ of an input x. To compress the amount of communication required, we now define a truncated hash function H'. For truncation parameter ℓ, we define H' as a function from $\{0,1\}^*$ to $\{0,1\}^{\lambda-\ell}$; this will intuitively denote the output of H truncated by ℓ bits.

How to pick the prefix. We start by fixing an ℓ-bit string $s = s_1, \ldots, s_\ell$. We assume that the prefix was sampled uniformly at random from the space of ℓ-bit strings (this could be considered as a part of a setup phase). Let $Y_s \subseteq Y$ denote the subset of Y that contains exactly those outputs that begin with s. We consider the prefix for simplicity, but in general the truncated bits can be in any positions and do not have to be consecutive—this would also support e.g., truncation to a subgroup of the output group. Intuitively, our truncated function $H' = H'_s$ will now sample a nonce r such that $y = H(r \parallel x) \in Y_s$; this will be taken as the output of H'. The upshot is that now the first ℓ bits do not

Truncated Hashing

$H'_s(x)$:
For $r = 0, 1, 2, \ldots$:
  $y \leftarrow H(r \parallel x)$
  If $y \in Y_s$, then output y

Fig. 1. Truncated Hash Function

need to be communicated as part of the hash output; this can be done since the string s, while part of the hash output, will be publicly known to the receiver and therefore assumed to be implicit without needing to be communicated.

We now argue that the bit-security (against hash function preimage attacks) is preserved despite outputting fewer bits of the digest.

Theorem 1. *Let $H : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ a hash function with κ-bit security and computational cost T_H. Then the truncated hash function $H'_s : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ where the first ℓ bits are fixed to string $s \in \{0,1\}^\ell$ is also κ-bit secure.*

Proof. Applying the bit-security paradigm from Definition 3 to the hash function H, let $\mathcal{A}$ denote the inner adversary that minimizes $\log_2(T_\mathcal{A}/\epsilon_\mathcal{A})$ where $T_\mathcal{A}$ denotes the computational complexity of $\mathcal{A}$ and $\epsilon_\mathcal{A}$ denotes its success probability. Then $\kappa = \log_2(T_\mathcal{A}/\epsilon_\mathcal{A})$. We now need to show that the truncated hash function H'_s also provides κ bits of security. For this, first, notice that an evaluation of H'_s takes on expectation 2^ℓ times the cost to evaluate H. However, since the output is also truncated by ℓ, the success probability of the inner adversary to find a preimage will also increase by a factor of 2^ℓ. This means that the expected bit-security of any $\mathcal{A}'$ playing the game for H will be given by $\log_2((T_{\mathcal{A}'}/2^\ell)/(\epsilon_{\mathcal{A}'}/2^\ell) = \log_2(T_{\mathcal{A}'}/\epsilon_{\mathcal{A}'})$. Since this value is minimized for adversary $\mathcal{A}$, we obtain that the bit-security of H' will also be $\log_2(T_\mathcal{A}/\epsilon_\mathcal{A}) = \kappa$.

3.2 Truncated DL-based public keys

Let $\mathbb{G}$ be a cyclic group of prime order p and g be a generator of $\mathbb{G}$. The key generation algorithm, $\mathsf{KeyGen}(1^\lambda)$ for DL based keys works as follows: $\mathsf{sk} = x, \mathsf{pk} = g^x$ where $x \xleftarrow{\$} \mathbb{Z}_p$.

Let $y_1 y_2 \ldots y_\lambda$ denote the binary representation of pk which forms the output space for public keys Y. For truncation parameter $\ell < \lambda$, we fix an ℓ-bit string $s = s_1, \ldots, s_\ell$. Let $Y_s \subseteq Y$ denote the subset of Y that contains exactly those outputs that begin with s. (As with hash functions above, we consider truncation on the prefix for simplicity.) The new key generation algorithm $\mathsf{KeyGen}'(1^\lambda)$ now works as follows: $x \xleftarrow{\$} \mathbb{Z}_p$, compute $\mathsf{pk} = g^x$, if $\mathsf{pk} \notin Y_s$ repeat by picking a new x, else output $\mathsf{pk} \in Y_s$. We note that for the case of public keys, while sampling a "valid" private key requires this overhead of additional repeated "brute-force" style operations, the space savings are *permanent* for the key's lifetime, which makes this compression attractive particularly for blockchain applications.

Theorem 2. *Let $\mathsf{KeyGen}(1^\lambda)$ be a key generation algorithm for the DL hard relation with κ-bit security and computational cost $T_\mathcal{A}$. Then the truncated key generation algorithm $\mathsf{KeyGen}'(1^\lambda)$ where the first ℓ bits are fixed to string $s \in \{0,1\}^\ell$ is also κ-bit secure.*

Proof. The key generation algorithm for DL based keys is a hard relation (as defined in Def. 2) under the DL problem. Following the bit security definitional approach, a hard relation is a search-type game which provides bit security of

$\log_2(T_{\mathcal{A}}/\epsilon_{\mathcal{A}})$ where $\epsilon_{\mathcal{A}}$ is $\mathcal{A}$'s success probability in the security game and $T_{\mathcal{A}}$ is the computational cost of the adversary. We argue that our truncated algorithm maintains the same bit security as the one offered by the underlying group $\mathbb{G}$ [6].

Without loss of generality, assume that the truncation parameter $\ell = \lambda - 1$ and thus the size of the public key is 1 bit. Then, our truncated key sampling method for public key space Y_s creates a secret key space X' where $|X'| = 2$. However, an adversary cannot efficiently compute X' as this would directly reduce to breaking the DL assumption with non-negligible probability.

We now need to show that the truncated key generation algorithm also provides κ bits of security. If x's are sampled uniformly at random, computing the truncated pk will take on expectation 2^ℓ time. However, since the possible secret key space is also truncated by ℓ, the success probability of the inner adversary will also increase by a factor of 2^ℓ. This means that the expected bit-security of any $\mathcal{A}'$ playing the game for KeyGen' will be given by $\log_2((T_{\mathcal{A}'}/2^\ell)/(\epsilon_{\mathcal{A}'}/2^\ell) = \log_2(T_{\mathcal{A}'}/\epsilon_{\mathcal{A}'})$.

3.3 Truncated Schnorr Signatures

$\mathsf{KeyGen}(\mathbb{G}, p, g)$	$\mathsf{Sign}(\mathsf{sk}, m, s)$	$\mathsf{Verify}(\sigma, m, \mathsf{pk})$
$x \xleftarrow{\$} \mathbb{Z}_p$	1. $r \xleftarrow{\$} \mathbb{Z}_p$	Parse $\sigma = (s, e)$
$\mathsf{sk} \leftarrow x$	2. $I \leftarrow g^r$	$I \leftarrow g^s \mathsf{pk}^{-e}$
$\mathsf{pk} \leftarrow g^x$	3. $e \leftarrow H(I \parallel m)$	If $(H(I \parallel m) = e)$ then
return $(\mathsf{pk}, \mathsf{sk})$	4. If $e \notin Y_s$ return to step 1.	**return** 1
	5. $s \leftarrow r + \mathsf{sk} \cdot e \bmod p$	else **return** 0
	6. **return** $\sigma = (s, e)$	

Fig. 2. The Truncated Schnorr Signature Scheme

We now consider compression on DL-based signatures, and we show an application of our approach on Schnorr signatures as an example. In Figure 2 we present a version of our truncated Schnorr signature (we use red font to indicate the differences from standard Schnorr Signatures).

As before, let $\mathbb{G}$ be a cyclic group of prime order p and g be a generator of $\mathbb{G}$. Let $H()$ be a random oracle implemented with a hash function that outputs a uniformly random element $e \in \mathbb{Z}_p$ where p is a prime of size 2λ-bits (in order to achieve λ-bit security)[7]. We truncate the hash function as in Sec. 3.1 and denote

[6] For the case of `secp256k1` discrete log keys, if the group size is λ-bits, then the probability of success for the adversary is roughly $1/2^{\lambda/2}$ and thus the bit security is $\lambda/2$-bits, i.e., in `secp256k1` a 256-bit key roughly offers 128-bit security [34].

[7] We note that one could create a *short* Schnorr signature, by assuming an $H()$ that maps to random λ-bit values, thus the final signature σ is of size 3λ-bits (2λ-bits to encode s and λ-bits to encode e) [15]. Our truncation technique can also apply on top of short Shnorr signatures.

	Ed25519 public keys	secp256k1 public keys	Schnorr Ed25519 signature	ECDSA secp256k1 signature	SHA2-256	SHA3-256	BLAKE3-256
Macbook Pro M1 MAX	3.0589 ms	4.0909 ms	3.2539 ms	15.649 ms	63.033 µs	71.786 µs	36.335 µs
AWS t3.xlarge	5.5355 ms	7.4514 ms	5.5283 ms	30.529 ms	110.11 µs	166.18 µs	43.198 µs

Table 1. Evaluation on truncated primitives for ℓ=8.

the truncation string by s given as input to the signing algorithm. The resulting signature $\sigma = (s, e)$ will save ℓ bits (the truncation parameter).

Note that while our truncation approach could alternatively be applied to the value s, this is not efficient since e is computed first during Sign. Also, attempting to truncate signature values e and s simultaneously would exponentially increase the truncation time. Therefore in general, for primitives where the output consists of two or more elements, it is recommended to truncate the element that is computed first in the algorithm.

Also note that space savings in truncated signature schemes are independent from the existing savings from truncated public keys. The reason is simple - as the set of truncated public keys ${Y^{(\mathsf{pk})}}_s$ is a subset of the whole space of public keys $Y^{(\mathsf{pk})}$, any signature (truncated or not) produced using the respective sk which corresponds to a $\mathsf{pk} \in {Y^{(\mathsf{pk})}}_s$ inherits the same properties as if the sk, pk pair was chosen without applying step 4 in Fig. 2.

The security of Schnorr signatures (existential unforgeability) has been thoroughly analyzed in the literature [36,39]. We now argue that our truncated Schnorr signature scheme maintains the same bit security as the underlying non-truncated version. The theorem below is straightforward given Theorem 1.

Theorem 3. *Let* $\mathsf{SchnorrSign}$ *be a Schnorr signature scheme with* κ*-bit security and computational cost* $T_{\mathcal{A}}$*. Then the truncated signing algorithm as defined in Fig.2 where the first* ℓ *bits are fixed to string* $s \in \{0,1\}^\ell$*, is also* κ*-bit secure.*

4 Evaluation

We performed a series of evaluation experiments[8] to measure the trade-off between the truncation parameter and the computational effort for the primitives we considered. Our evaluation series were performed using fastcrypto library in Rust [22] on a Macbook M1 Pro as well as on an AWS t3.xlarge instance, using a single CPU core (note that our truncation algorithms are naturally parallelizable, but our implementation did not apply multi-threads focusing on the worst case). Note that we used a repeated Monte Carlo simulation over 1 million runs each time and the results were consistent.

In Table 1 we present our results for some of the cryptographic primitives[9] discussed in Section 3, using a sample size of 100 for one byte of truncation

[8] Our evaluation code is available at `https://www.dropbox.com/sh/l3nj4l3ntobbvro/AABfjtwotv_d06NxwF5M3cD4a`

[9] Ed25519 is a deterministic scheme, but we assume analogy with a randomized version, effectively representing Schnorr signature schemes in this list.

(ℓ=8). For more bytes there is a factor of 2^8 computational cost blowup for each additional byte truncated, therefore our results can be naturally extrapolated to derive the expected costs for larger truncation, as shown in Fig. 3. Consequently, the equilibrium between the tolerated computational overhead and the desired truncation benefits ultimately depends on the specific primitive and its application scenario (e.g., even a week's worth of computational work might be tolerable in order to reduce the public address size by 5 bytes in a blockchain application, where the benefit will be permanent). Note that there are additional techniques we can apply to speed up the computation stage, e.g., using pre-computed lookup tables for public key generation and perform elliptic curve additions rather multiplications. We also evaluate truncated hashed public keys as shown in Table 2 (which are a common practice to derive public addresses in cryptocurrencies such as Bitcoin). Although we do not explicitly list evaluation results for encryption schemes, we expect numbers similar to the ones we report already, e.g., DL-based primitives such as ElGamal encryption results will be similar to those of DL public keys.

	Schnorr Ed25519 + SHA2-256	Shnorr Ed25519 + SHA3-256	Schnorr Ed25519 + BLAKE3-256	ECDSA secp256k1 + SHA2-256	ECDSA secp256k1 + SHA3-256	ECDSA secp256k1 + BLAKE3-256
Macbook Pro M1 MAX	3.1746 ms	3.3702 ms	3.1199 ms	4.0191 ms	4.1623 ms	3.9293 ms
AWS t3.xlarge	5.4751 ms	5.4873 ms	5.6650 ms	7.0932 ms	7.0677 ms	7.4188 ms

Table 2. Hashed public key truncation for ℓ=8.

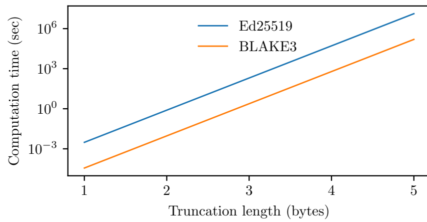

Fig. 3. Truncating primitives for larger truncation parameters by comparing the Schnorr version over Ed25519 curve against Blake3.

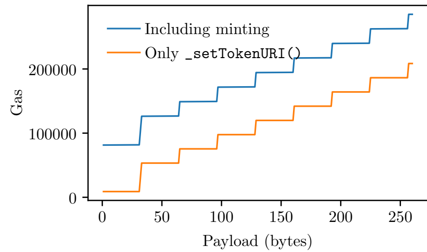

Fig. 4. ERC721 smart contract cost.

Date	Ether per Gas	ETH/USD	Truncation	Savings	Savings % with `_mint()`	Savings % `_setTokenURI()` only
Jan 10 2022	218.55 Gwei	$3156	33 to 32	$15.45	17.7%	41.5%
			33 to 31	$30.73	35.21%	83.21%
			65 to 64	$15.45	15%	29.36%
Jan 10 2023	24.83 Gwei	$1336	33 to 32	$0.74	17.7%	41.5%
			33 to 31	$1.48	35.21%	83.21%
			65 to 64	$0.74	15%	29.36%

Table 3. Truncation savings in OpenZeppelin ERC721 smart contract.

5 Impact

Truncation of cryptographic outputs without loss of security can have tangible impact in a variety of business scenarios, especially in resource-constrained environments such as blockchains. Below we discuss some indicative scenarios.

5.1 Lookup identifiers and certificates

One class of applications is large collections of collision-free unique identifiers or certificates, which could be benefited directly by reducing both storage and network cost. An example is IPFS [13] hash-links or database unique IDs derived by hash digests. Typically, these fields are both queryable and cloned in multiple database tables and indexers as primary keys. Additionally, the aforementioned field types are the most common parameters in application programming interfaces (APIs), especially for lookups before accessing the actual mapped content. In the IPFS example, shorter file hashes would result to:

1. shorter URLs; thus, less bytes transferred per request, slightly saving bandwidth for users which might important for constraint environments where internet bandwidth is expensive.
2. shorter indexer table fields for each IPFS node; thus saving memory and hard disk space.
3. slightly faster serialization and encoding (i.e., base58) performance due to shorter length.
4. shorter IPFS URLs are usually pinned in external applications too (i.e., blockchain); thus, less storage and network requirements for systems that interact with IPFS nodes.

Another application is compressing digital certificates, commonly used in TLS communication. Shorter certificates, both at the public key and signature layers could reduce the footprint of exchanged messages across the web and potentially, slightly reduce website response times.

5.2 Set reconciliation

It is very common that some distributed systems require set reconciliation to synchronize their states. For example, in consensus systems and for indexer services, a potentially large list of all elements (i.e., transaction, certificates or

file hashes) is exchanged between nodes to allow for synchronization in order to identify differences, i.e., when recovering from an outage. Applying our proposed truncation techniques for signatures and hash functions can lead to compressed identifiers. Thus we can get latency improvements by having reconciliations lists that are shorter by a non-negligible percentage.

5.3 Blockchain applications

As discussed in the introduction, truncating cryptographic primitives can have a substantial impact in a blockchain setting. On-chain storage is a particularly scarce resource, since full nodes first need to download the full blockchain for initial synchronization, which can be in the order of several hundreds of Gigabytes, while all this data requires to be maintained and stored by *all* full nodes participating. Naturally, it becomes particularly expensive to store any cryptographic primitive in the blockchain (public keys, signatures etc.) and even truncating those primitives by a few bytes with minimal computational effort implies substantial cumulative savings. We show those potential savings in two typical use-cases, namely in cryptocurrenies and smart contracts.

Cryptocurrency transactions. As a first example, in Bitcoin, a common pay to public key hash (P2PKH) transaction with 1 input and 2 outputs needs about 70 bytes for the input ECDSA signature, 33 bytes for the input ECDSA public key and 20 bytes for each of the two RIPEMD-160 hash outputs[10]. By applying our paradigm we can shorten the ECDSA signature by 2 bytes with 2^8 effort, each of the hashes can be shortened by 1 byte with $2^{8.5}$ effort, and we also make a conservative assumption that the ECDSA public key is already truncated by 2 bytes, as the required 2^{16} effort is only paid once when deriving the key pairs (and as discussed previously, one might be willing to invest even more computation as a one-time cost in order to save storage costs throughout the public key's lifetime). Therefore, with effort of roughly $2^{10.5}$ total operations, we compress a standard Bitcoin transaction by at least 6 bytes, which implies an fee reduction by 3% and an equivalent increase of available space in Bitcoin's blocks.

Smart contract applications. Our truncation paradigm can have a much more significant impact in smart contracts. As an example, we applied our truncation paradigm to a standard ERC721 Token Contract [35], which awards a Non-Fungible Token (NFT) to a blockchain address. In this contract, its NFT payload can be derived by several types of truncated cryptographic primitives, for instance an InterPlanetary File System (IPFS) link. Through a series of experiments, we observed that gas transaction costs vs. contract storage is not a linear relation, but increase sharply every 32 bytes, as shown in Figure 4. We distill the contract transaction costs between a transaction that includes minting a token through the `awardItem()` function, and a transaction that only sets a token's Uniform Resource Identifier (URI). We observe that minting costs are

[10] The transaction also includes the necessary `OP` codes which here we do not take into account as they cannot be truncated.

almost constant w.r.t. token payload length, therefore in smart contract transactions which only set the `tokenURI` (and the token is already minted), such savings can cumulatively make a huge impact.

Another important observation from our experiments is that the change in gas costs becomes more significant in the first 32-byte block, i.e., for a payload of 31 bytes in the contract's `awardItem()` function, the total contract transaction costs are 81962 gas, for 32 bytes the contract costs 104109 gas and for 33 bytes the contract costs 126508 gas (for `_setTokenURI` only, the costs are 8949, 31152 and 53322 respectively). For every additional byte, the gas costs then increase by 12 gas per byte, until 64 bytes with a cost of 126821 gas, while 65 bytes cost 149220 and then again increasing by 12 gas per byte until an additional 32-byte block is needed. Given the above observations, truncating by even a single byte can potentially lead to substantial savings in gas fees, while in common payloads with 33 bytes (e.g., ECDSA keys in compressed format) the savings can be even more significant when truncated to 31 bytes. As an additional comparison, creating a 65-byte item through the `awardItem()` function costs 149220 gas, while a 64-byte item costs 126821 gas. All these 1-byte or even 2-byte truncations can be achieved by minimal computational effort as shown in Tables 1 and 2. While the gas savings for a each contract call are similar when truncating a 33-byte payload to 32 bytes, (i.e., from 126508 gas down to 104109 gas), by truncating further to 31 bytes achieves even less transaction cost of 81962 gas. The latter implies the savings can lower gas fees by 35.21% compared to the original transaction. When considering only the `_setTokenURI` function, assuming the token minting has already occurred, the savings are 83.21% compared to the original! These significant savings can all be achieved at the cost of only 3.84 seconds average computational time on a single core of Macbook Pro, after extrapolating the results of Table 1 for $\ell = 16$. In addition to the relative savings, in Table 3 we also show the absolute dollar-amount savings for different truncation levels and for different dates of 1 year span, as Ethereum's average gas prices and Ethereum to USD exchange rates can fluctuate significantly over time.

5.4 New cryptographic schemes

It seems that there is an interesting opportunity to wisely construct new cryptographic schemes in such a way that they take advantage of Truncator's techniques at the protocol design phase. That said, future schemes could take into account the possibility and feasibility of mining, and then securely alter any of the key generation or other cryptographic operation functions to accommodate that.

An interesting direction is optimising hash-based signatures at the key derivation level aiming at high-performant mining with by far better results than brute forcing. The following example could be a starting point for future research and food for thought for rewriting existing schemes achieving record compression levels for various primitive families.

In the following we examine how one can revisit Lamport [30] signatures by rewriting the key generation flow in order to be "mining-friendly".

In the traditional Lamport scheme, the private key consists of 256 independent pairs of 256-bit random values (seeds), thus 512 elements and 16 KiB in total. Each of these sub-private keys has a corresponding public key, its hash, thus the total public key is also requiring 512 elements.

Typically we sign hashed messages and in Lamport, for each bit in the hash we pick the corresponding sub-private value. For example, if the first bit in the hash is a 0, we pick the first value in the first pair, otherwise if it was 1, then we use the second value in the first pair etc. The total output for a 256-bit hashed message results in 256 revealed elements which can be verified against their corresponding public keys.

One would wonder if we could somehow compress Lamport signatures without using the Wintenitz hash-chain variant and it seems that there is an elegant solution that could take advantage of potential mining. The trick is to derive the private parts in a tree fashion and not pick them independently. Figure 5 shows an example where if we wanted to sign a message that is all zeros we use the top key and verifiers can derive all sub-keys via Merkle tree operations. Similarly, if we have some adjacent similar bits we can use the corresponding tree path to reduce how many keys one should submit. Obviously, the same applies for adjacent set bits. The benefits of this approach, accompanied with message hash mining, can be significant as one can retry hashing the message in order to maximize the number of adjacent bits and thus reduce the signature payload, resulting in a more optimized Lamport verification and shorter proofs.

6 Conclusion and Future Directions

We presented Truncator, a paradigm to truncate the output size of cryptographic primitives with a computational trade-off. As a starting point, we showed how our approach can be applied on basic cryptographic primitives, while showing an additional benefit in certain types of primitives (e.g., in primitives with auxiliary outputs such as checksums). We also demonstrated that our Truncator paradigm can achieve significant reduction in smart contract gas costs with only a few seconds of additional computation per contract call. Although this would require changes in the existing smart contract specifications, the overall benefits of our approach enjoyed by the whole blockchain ecosystem would greatly outweigh this small computational cost, since this cost is only paid by the transaction creator, while blockchain verifiers (e.g., miners or validators) are *not required* to perform *any* additional computation, as they would simply "glue back" the omitted ℓ bits which are implied and not communicated. In addition, our paradigm opens many possibilities for exploration, such as its implications in cryptoeconomics (e.g., the equilibrium of the trade-off when quantifying the benefits and the initial investment in computation), or the ways of applying it (e.g., delegating the computational effort to external services). Another potential avenue for future exploration is towards applying our truncation paradigm into more advanced cryptographic primitives (e.g., for deriving truncated non-interactive zero-knowledge proofs) and formally prove their security.

Another future work direction is proposing novel primitives specifically handcrafted to utilize mining techniques on sender's side, towards improved efficiency for communication and receiver's work.

Finally, as this work is partly inspired by the recent unfortunate buggy "gas golfing" software in Ethereum, where weakly implemented functions incorrectly generated addresses (hashes) with "prefixed zeroes for gas optimization" resulting in millions of losses [2,31,8], we expect our *Truncator* approach to be naturally applied in the blockchain space as a secure solution towards more succinct transactions, addresses and states.

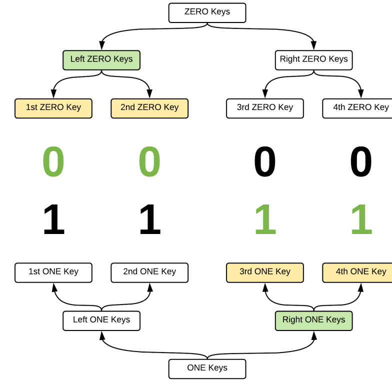

Fig. 5. Binary tree KDF-Lamport, where keys are derived via a Merkle tree structure.

References

1. Ethereum developer resources, `https://ethereum.org/en/developers/docs/blocks/`
2. Ethereum gas golfing, `https://ethereum.org/en/developers/docs/mev/#mev-extraction-gas-golfing`
3. Evolution of the signature size in bitcoin, `https://b10c.me/blog/006-evolution-of-the-bitcoin-signature-length/`
4. An incomplete guide to rollups, `https://vitalik.ca/general/2021/01/05/rollup.html`
5. Length of ecdsa signatures, `https://transactionfee.info/charts/bitcoin-script-ecdsa-length/`
6. Profanity ethereum vanity address tool, `https://github.com/johguse/profanity`
7. Vanitygen, `https://en.bitcoin.it/wiki/Vanitygen`
8. A vulnerability disclosed in profanity, an ethereum vanity address tool, `https://blog.1inch.io/a-vulnerability-disclosed-in-profanity-an-ethereum-vanity-address-tool-68ed7455fc8c`
9. What is the bitcoin block size limit?, `https://bitcoinmagazine.com/guides/what-is-the-bitcoin-block-size-limit`
10. Would it matter if my miner was hashing random vs incremental values?, `https://crypto.stackexchange.com/questions/29318/would-it-matter-if-my-miner-was-hashing-random-vs-incremental-values`
11. Random sampling vs incrementing randomness in cryptographic protocols (2021), `https://crypto.stackexchange.com/questions/93651/random-sampling-vs-incrementing-randomness-in-cryptographic-protocols`
12. Aumasson, J.P., et al.: Sphincs+ PQ NIST competition Round3 presentation, `https://csrc.nist.gov/CSRC/media/Presentations/sphincs-round-3-presentation/images-media/session-1-sphincs-plus-hulsing.pdf`
13. Benet, J.: IPFS - content addressed, versioned, P2P file system. CoRR **abs/1407.3561** (2014), `http://arxiv.org/abs/1407.3561`

14. Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: SPHINCS: Practical stateless hash-based signatures. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part I. LNCS, vol. 9056, pp. 368–397. Springer, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46800-5_15
15. Blocki, J., Lee, S.: On the multi-user security of short schnorr signatures with preprocessing. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part II. LNCS, vol. 13276, pp. 614–643. Springer, Heidelberg (May / Jun 2022). https://doi.org/10.1007/978-3-031-07085-3_21
16. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to IOPs and stateless blockchains. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part I. LNCS, vol. 11692, pp. 561–586. Springer, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26948-7_20
17. Bonneau, J., Meckler, I., Rao, V., Shapiro, E.: Coda: Decentralized cryptocurrency at scale. Cryptology ePrint Archive, Report 2020/352 (2020), https://eprint.iacr.org/2020/352
18. Buchmann, J., Dahmen, E., Ereth, S., Hülsing, A., Rückert, M.: On the security of the winternitz one-time signature scheme. In: Nitaj, A., Pointcheval, D. (eds.) Progress in Cryptology - AFRICACRYPT 2011 - 4th International Conference on Cryptology in Africa, Dakar, Senegal, July 5-7, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6737, pp. 363–378. Springer (2011). https://doi.org/10.1007/978-3-642-21969-6_23, https://doi.org/10.1007/978-3-642-21969-6_23
19. Bünz, B., Agrawal, S., Zamani, M., Boneh, D.: Zether: Towards privacy in a smart contract world. In: Bonneau, J., Heninger, N. (eds.) FC 2020. LNCS, vol. 12059, pp. 423–443. Springer, Heidelberg (Feb 2020). https://doi.org/10.1007/978-3-030-51280-4_23
20. Cecchetti, E., Zhang, F., Ji, Y., Kosba, A.E., Juels, A., Shi, E.: Solidus: Confidential distributed ledger transactions via PVORM. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 701–717. ACM Press (Oct / Nov 2017). https://doi.org/10.1145/3133956.3134010
21. Chalkias, K., Garillot, F., Kondi, Y., Nikolaenko, V.: Non-interactive half-aggregation of eddsa and variants of schnorr signatures. In: Paterson, K.G. (ed.) Topics in Cryptology - CT-RSA 2021 - Cryptographers' Track. Springer (2021)
22. Chalkias, K., Garillot, F., Lindstrøm, J., Riva, B., Wang, J.: fastcrypto v0.1.3, https://crates.io/crates/fastcrypto
23. Chen, Y., Ma, X., Tang, C., Au, M.H.: PGC: Decentralized confidential payment system with auditability. In: Chen, L., Li, N., Liang, K., Schneider, S.A. (eds.) ESORICS 2020, Part I. LNCS, vol. 12308, pp. 591–610. Springer, Heidelberg (Sep 2020). https://doi.org/10.1007/978-3-030-58951-6_29
24. Dang, Q.: Nist special publication 800-106: Randomized hashing for digital signatures (2009), https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-106.pdf
25. Dang, Q.: Recommendation for applications using approved hash algorithms (2012), https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf
26. Fauzi, P., Meiklejohn, S., Mercer, R., Orlandi, C.: Quisquis: A new design for anonymous cryptocurrencies. In: Galbraith, S.D., Moriai, S. (eds.) ASIACRYPT 2019, Part I. LNCS, vol. 11921, pp. 649–678. Springer, Heidelberg (Dec 2019). https://doi.org/10.1007/978-3-030-34578-5_23

27. Fleischhacker, N., Larsen, K.G., Simkin, M.: How to compress encrypted data. Cryptology ePrint Archive, Paper 2022/1413 (2022), https://eprint.iacr.org/2022/1413, https://eprint.iacr.org/2022/1413
28. Fregly, A., Harvey, J., Jr., B.S.K., Sheth, S.: Merkle tree ladder mode: Reducing the size impact of nist pqc signature algorithms in practice. Cryptology ePrint Archive, Paper 2022/1730 (2022), https://eprint.iacr.org/2022/1730, https://eprint.iacr.org/2022/1730
29. Knudsen, L.R.: Small size hashes with enhanced security. Int. J. Netw. Secur. **2**(1), 41–42 (2006), http://ijns.jalaxy.com.tw/contents/ijns-v2-n1/ijns-2006-v2-n1-p41-42.pdf
30. Lamport, L.: Constructing digital signatures from a one way function (1979)
31. McCurdy, W.: Hackers nab nearly 1m in crypto from ethereum vanity adress exploit, https://decrypt.co/110526/hackers-nab-nearly-1-million-crypto-ethereum-vanity-adress-exploit
32. Micciancio, D., Walter, M.: On the bit security of cryptographic primitives. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part I. LNCS, vol. 10820, pp. 3–28. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78381-9_1
33. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009), http://bitcoin.org/bitcoin.pdf
34. NIST: NIST key-length recommendations (2020), https://www.keylength.com/en/4/
35. OpenZeppelin: Constructing an erc721 token contract, https://docs.openzeppelin.com/contracts/4.x/erc721
36. Pointcheval, D., Stern, J.: Security proofs for signature schemes. In: Maurer, U.M. (ed.) EUROCRYPT'96. LNCS, vol. 1070, pp. 387–398. Springer, Heidelberg (May 1996). https://doi.org/10.1007/3-540-68339-9_33
37. Pornin, T.: Truncated EdDSA/ECDSA signatures. Cryptology ePrint Archive, Report 2022/938 (2022), https://eprint.iacr.org/2022/938
38. Ronen, E., Yogev, E.: SPHINCS+C: Compressing SPHINCS+ with (almost) no cost. Cryptology ePrint Archive, Report 2022/778 (2022), https://eprint.iacr.org/2022/778
39. Seurin, Y.: On the exact security of Schnorr-type signatures in the random oracle model. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 554–571. Springer, Heidelberg (Apr 2012). https://doi.org/10.1007/978-3-642-29011-4_33
40. Tsiounis, Y., Yung, M.: On the security of ElGamal based encryption. In: Imai, H., Zheng, Y. (eds.) PKC'98. LNCS, vol. 1431, pp. 117–134. Springer, Heidelberg (Feb 1998). https://doi.org/10.1007/BFb0054019
41. Watanabe, S., Yasunaga, K.: Bit security as computational cost for winning games with high probability. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021, Part III. LNCS, vol. 13092, pp. 161–188. Springer, Heidelberg (Dec 2021). https://doi.org/10.1007/978-3-030-92078-4_6
42. Wood, G.: Ethereum: A secure decentralized generalised transaction ledger (2021), https://ethereum.github.io/yellowpaper/paper.pdf

A Additional truncation paradigms

A.1 Truncation of Encryption

We now discuss how our truncation techniques can be applied on encryption protocols. We focus on the cases of ElGamal and Diffie-Hellman (DH) based encryption since they are both vital building blocks for various layer-2 blockchain proposals. As an example, consider Zether [19] which uses ElGamal encryption to build private transactions via Ethereum smart contracts, or numerous other proposals which use ElGamal based commitments [20,26,23]. Monero's stealth addresses share techniques with key generation of DH based encryption.

We consider two scenarios: (1) truncation of ciphertexts and (2) truncation of encryption keys.

Truncated of Ciphertexts ElGamal ciphertexts can be truncated using our techniques. As above, let $\mathbb{G}$ be a cyclic group of prime order p and g be a generator of $\mathbb{G}$. Let $y_1 y_2 \ldots y_\lambda$ denote the binary representation of a group element. Let Y be the space of all group elements. For truncation parameter $\ell < \lambda$, we fix an ℓ-bit string $s = s_1, \ldots, s_\ell$. Let $Y_s \subseteq Y$ denote the subset of Y that contains exactly those outputs that begin with s.

In Figure 6 we present how truncated ElGamal encryption works (we use red font to indicate the differences from original ElGamal). There are two places in the encryption protocol where truncation could happen, either in the first part of the ciphertext, c_1, or in the second part, c_2 while the only random value which we could "mine" on, is the random selection of r in Step 2. In Figure 6 we propose to truncate c_1—this is because doing truncation in c_2 would be more costly as computing c_2 before testing if it falls in Y_s takes an extra multiplication operation. This serves as a good example to highlight the fact that when there are different choices on where to truncate parts of a protocol, a careful selection is important for higher computation savings. Of course, parties could also opt to truncate their public keys as explained above.

ElGamal encryption satisfies the security property of indistinguishability under chosen plaintext attacks (IND-CPA) [40] under the DDH assumption. We argue that the bit-security of truncated ElGamal is preserved.

Theorem 4. *Let* ElGamalEnc *be the ElGamal encryption scheme with κ-bit security and computational cost $T_{\mathcal{A}}$. Then, the truncated encryption algorithm as defined in Fig.6 where the first ℓ bits of c_1 are fixed to string $s \in \{0,1\}^\ell$, is also κ-bit secure.*

Proof (Sketch). Let $\mathcal{A}$ denote the CPA adversary, $T_{\mathcal{A}}$ denote the computational complexity of $\mathcal{A}$ and δ is the advantage of $\mathcal{A}$ when playing the IND-CPA game. As discussed in Section 2.3, the framework of [41], approximates a lower bound for $\kappa \geq \log_2(T_{\mathcal{A}}/\delta)$ (since IND-CPA is a decision game).

Our truncation process applies to the random value r. Given that the security of ElGamal relies on DDH, if the truncation process allows the adversary to

recover r with non-neglible probability, then $\mathcal{A}$ could break the DDH assumption and recover m (given that the message m is essentially masked by g^r).

However, for the same reasons as discussed in Theorem 2, an adversary cannot efficiently compute the new space defined for c_1 as this reduces to security of DL. Thus, the bit security for the IND-CPA decision game remains unchanged.

$\mathsf{KeyGen}(\mathbb{G}, p, g)$	$\mathsf{Enc}(\mathsf{pk}, m', s)$	$\mathsf{Dec}((c_1, c_2), \mathsf{sk})$
$\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_p^*$	1. map m' to a group element m	1. Compute $m = c_2 * c_1^{-\mathsf{sk}}$
$\mathsf{pk} = g^{\mathsf{sk}}$	2. $r \xleftarrow{\$} \mathbb{Z}_p^*$	2. Map m to m'
	3. $c_1 = g^r$	
	4. If $c_1 \notin Y_s$ return to step 2.	
	5. $c_2 = m * \mathsf{pk}^r$	
	6. Return (c_1, c_2)	

Fig. 6. The Truncated ElGamal Encryption Scheme

Truncation of Encryption Keys and a Blockchain Application Our techniques can be used for the truncation of encryption keys. Consider the case of Diffie-Hellman (DH) based encryption. The public keys are discrete-log based keys and thus can be truncated using our technique described in Section 3.2. This can have applications in the creation of stealth addresses as we explain below.

For concreteness, we first recall how DH based encryption works.

Basic structure Consider two parties: the sender Alice, and the receiver Bob. The basic DH based encryption structure has as follows: Consider a cyclic group $\mathbb{G}$ of prime order p and with generator g; a key pair $(\mathsf{sk}, \mathsf{pk})$ is given by $\mathsf{sk} = x \xleftarrow{\$} \mathbb{Z}_p$ and $\mathsf{pk} = g^x$. Let $(\mathsf{sk}_A, \mathsf{pk}_A) = (a, g^a)$ and $(\mathsf{sk}_B, \mathsf{pk}_B) = (b, g^b)$ denote (long-term) public keys for Alice and Bob respectively. For DH-based encryption, both Alice and Bob can compute a shared secret $s = g^{ab}$ using their own secret key and the other party's public key. A symmetric encryption key k can now be computed as $k = \mathsf{KDF}(s)$ where KDF is a key derivation function (commonly, a cryptographic hash). Alice and Bob can now communicate by encrypting messages using this shared key.

A commonly used variant of the above protocol—ECDH—makes use of elliptic curve groups. Specifically, given an EC group $\mathbb{G}$ with base point G, under additive notation, keys are now of the form $(x, X = x \cdot G)$. As mentioned above, both parties can truncate their (discrete-log based) public keys through mining as described in Section 3.2.

Blockchain-based payments The above protocol can also enable receiver-private blockchain payments; for concreteness, we will describe this using the ECDH variant.

Let $(a, A = a \cdot G)$ and $(b, B = b \cdot G)$ denote the long-term key-pairs of Alice and Bob respectively. Assume that both parties post their receiving public keys on a blockchain. Now in order to pay Bob, Alice first samples a fresh ephemeral key $(r, R = r \cdot G)$ and computes what would be the shared symmetric key as $k = \mathsf{KDF}(r \cdot b \cdot G) = \mathsf{KDF}(r \cdot B)$ using Bob's on-chain public key B. Alice can then send its payment to the ephemeral address corresponding to $k \cdot G + B$; notice that the payment can be spent using the private key $k + b$ which only Bob knows. Further, since k is known only to Alice and Bob, an observer cannot infer the identity of the receiver. Now, Bob only needs to actively monitor the blockchain to notice any payment sent to such an address.

A simple extension of this protocol splits the "spend" key (i.e., the key used to spend the payment) and the "scan" key (to observe transactions on the blockchain) of the receiver Bob; this is done to enable using a proxy to scan the blockchain for payments since it avoids having to use the spend key for this. The Monero blockchain makes use of this protocol for *stealth addresses*—which provide receiver-private payments.

For both protocol variants, parties can use our mining-based truncation techniques on their receiving public keys as well as the ephemeral address. The truncation here happens analogous to the DL-based truncation described in Section 3.2.

A.2 Truncating Primitives' Auxiliary Outputs

While in the previous section we showed how to truncate the primitive's main output, we also consider special cases where some primitives might include a secondary (or auxiliary) output. As an example, we consider the Winternitz one-time signature scheme (WOTS) [18]. This scheme treats the message in blocks of length specified by a parameter, while generating the respective private keys used to sign the message by iteratively applying a hash function, as shown in Fig. 7. However, to prevent forgery attacks from an eavesdropping adversary, an auxiliary output is necessary to prevent using disclosed hash function preimages (otherwise the adversary would be able to forge a signature on a different message after learning the respective hash preimages). This output consists of a checksum of the number of zero bits, which is appended to the main output in order to prevent this forgery (i.e., flipping 1-bits to 0's).

As the checksum value has a much higher probability to fall within the median of the checksum range, we require the fixed-value checksum to be simply the median of the range, as shown by the dark line in Fig. 8, which provides a way of more efficient compression. This approach was briefly considered in the third round of the NIST Post-Quantum Cryptography Standardization Process [12], while Kudinov et al. [38] recently considered a technique similar to Truncator, which requires a *fixed* checksum, therefore omitting it entirely from the signature. However, in cases where signing cost is also important (e.g., in mobile or other resource-constrained devices), in order to achieve more efficient truncation we can specify a *range* of "valid" checksum values (denoted by the red area in Fig.

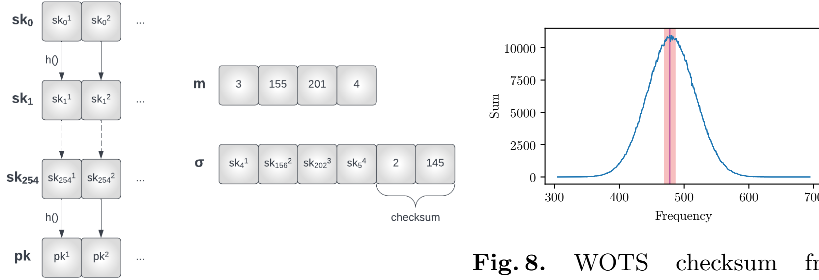

Fig. 7. Winternitz one-time signature example

Fig. 8. WOTS checksum frequency diagram for $w = 16$ and 1 million signatures.

8) which would only require 2 or 3 checksum bits, while exploiting the high probability of those checksum values.

Therefore for primitives with "biased" auxiliary outputs like WOTS, the computation cost paid upfront for truncating is significantly lower compared to the "fixed-bit" approach discussed in Section 3. We evaluate this efficiency benefit in Section 4.

Evaluation. We evaluate truncated WOTS separately, as it involves truncating its auxiliary output instead of truncating the main output in the other primitives we considered (i.e., requiring a fixed checksum instead of fixed bits). By considering the standard Winternitz parameter $w = 16$, the probability of successfully finding an output with the median checksum is roughly 1.1%. This implies that on average, truncated WOTS needs about 90 retries to output a valid one-time signature. Consequently, since WOTS is used in SPHINCS [14], the state-of-the-art stateless post-quantum signature scheme, with 64 hash elements output plus 3 elements as checksum, we can achieve about 4.5% compression with only $2^{6.5}$ effort (because of the bell-like normal distribution frequency curve in Fig. 8), compared to the "fixed-bit" approach used in truncated hash functions which would require 2^{12} effort to achieve the same level of compression. Note that we could also reduce the size of checksum instead of completely eliminating it (i.e., to 1 element instead of 3 for $w = 16$), which would require a lot less effort on the signer's side. For instance, Table 4 shows that saving one WOTS checksum element when $w = 16$ is almost certain per retry, while saving two elements in the same scheme is possible after about 5 retries which make the result really practical for slightly compressing SPHINCS signatures with minimal signer's effort.

B Frequently Asked Questions

In this section we address some common misconceptions that might arise from the reader regarding our methodology.

How to compress a hashed public key (e.g., P2PKH in Bitcoin: RIPEMD-160(SHA256(pk))? Answer: In this case, the brute-forcing is only

saved elements	WOTS (w = 16)	WOTS (w = 256)
1	99.9%	12.5%
2	17.2%	0.1%
3	1.1%	N/A

Table 4. Probability of saving WOTS checksum elements per retry for different modes of WOTS (for $w = 256$ there are only 2 checksum points needed)

happening on the secret key, and the compression algorithm only cares about the final output of RIPEMD-160. There is no need to brute-force both hash functions and the DL-based keys in parallel.

In P2PKH, an attacker can target many preimages at once with a single hash query. Answer: Although a single RIPEMD-160 hash query corresponds to many different SHA256(pk) preimages, the attacker still doesn't get any advantage for deriving pk (let alone the sk).

In Theorem 1 for truncating hash functions, if $\ell = \kappa - 1$ then there are only two possible outputs of the hash function H, so collision happens with 1/2 probability. Answer: While this claim is true, in order for the adversary to perform a $\ell = \kappa - 1$ truncation, this requires $2^{\kappa-1}$ exponential work. This provides the intuition behind our theorems that security of truncated primitives is preserved.

Since the truncated hash requires the sender to brute-force a nonce, the verifier would need to perform a similar brute-force process, leading to increased overhead for the verifier. Answer: The verification costs are not affected by our truncation paradigm. A nonce might not be needed in the first place (e.g., the P2PKH example above), or it will be communicated to the verifier using out of band channels.

The sender's computation cost grows exponentially in the reduced size, and thus only a few bytes can be saved within moderate time, which may confine its real-world applications to some extreme cases. Answer: While this is true, as we showcase in our examples (e.g, blockchain public keys where the savings are permanent, smart contracts, WOTS), the savings can be significant even when truncating primitives by a few bits.