Fastcrypto: Pioneering Cryptography Via Continuous Benchmarking

Kostas Kryptos Chalkias[1], Jonas Lindstrøm[1], Deepak Maram[1], Ben Riva[1], Arnab Roy[1], Alberto Sonnino[1,2], and Joy Wang[1]

[1]Mysten Labs
[2]University College London

March 14, 2024

Abstract

In the rapidly evolving fields of encryption and blockchain technologies, the efficiency and security of cryptographic schemes significantly impact performance. This paper introduces a comprehensive framework for continuous benchmarking in one of the most popular cryptography Rust libraries, `fastcrypto`. What makes our analysis unique is the realization that automated benchmarking is not just a performance monitor and optimization tool, but it can be used for cryptanalysis and innovation discovery as well. Surprisingly, benchmarks can uncover spectacular security flaws and inconsistencies in various cryptographic implementations and standards, while at the same time they can identify unique opportunities for innovation not previously known to science, such as providing a) hints for novel algorithms, b) indications for mix-and-match library functions that result in world record speeds, and c) evidences of biased or untested real world algorithm comparisons in the literature.

Our approach transcends traditional benchmarking methods by identifying inconsistencies in multi-threaded code, which previously resulted in unfair comparisons. We demonstrate the effectiveness of our methodology in identifying the fastest algorithms for specific cryptographic operations like signing, while revealing hidden performance characteristics and security flaws. The process of continuous benchmarking allowed `fastcrypto` to break many crypto-operations speed records in the Rust language ecosystem.

A notable discovery in our research is the identification of vulnerabilities and unfair speed claims due to missing padding checks in high-performance Base64 encoding libraries. We also uncover insights into algorithmic implementations such as multi-scalar elliptic curve multiplications, which exhibit different performance gains when applied in different schemes and libraries. This was not evident in conventional benchmarking practices. Further, our analysis highlights bottlenecks in cryptographic algorithms where precomputed tables can be strategically applied, accounting for L1 and L2 CPU cache limitations.

Our benchmarking framework also reveals that certain algorithmic implementations incur additional overheads due to serialization processes, necessitating a refined 'apples to apples' comparison approach. We identified unique performance patterns in some schemes, where efficiency scales with input size, aiding blockchain technologies in optimal parameter selection and data compression.

Crucially, continuous benchmarking serves as a tool for ongoing audit and security assurance. Variations in performance can signal potential security issues during upgrades, such as cleptography, hardware manipulation or supply chain attacks. This was evidenced by critical private key leakage vulnerabilities we found in one of the most popular EdDSA Rust libraries. By providing a dynamic and thorough benchmarking approach, our framework empowers stakeholders to make informed decisions, enhance security measures, and optimize cryptographic operations in an ever-changing digital landscape.

Keywords— cryptography, cryptanalysis, continuous benchmarking, Rust language, blockchain, crypto audits, supply chain attacks.

1 Introduction

Cryptography plays a pivotal role in safeguarding the integrity and confidentiality of secure communication channels, decentralized applications, digital identity and authentication systems, among the others. In the last 10-15 years, the demand for secure and scalable blockchain solutions caused an exponentially increased need for comprehensive performance evaluations of the underlying cryptographic components, such as digital signatures, zero knowledge proofs, Merkle trees, regular or exotic encryption mechanisms, multi-party computations and randomness beacons. It is believed that blockchain research has advanced the cryptography space rapidly [20], offering some

of the most robust and fastest implementations that are now reused outside web3 as well.

`Fastcrypto` [24] is one of the most recent and modern Rust [23] libraries focusing on high performance implementations of cryptographic primitives, typically required by blockchain applications. Although originally designed to provide all cryptographic functionality for the Sui[1] blockchain, it has been widely adopted by the cryptographic community, and is currently used in at least 167 other projects[2].

A few prominent examples of `fastcrypto`'s community usage include the following: (1) DB3 Network[3], which is a lightweight, permanent JSON document database for Web3. It is designed to store and retrieve data for decentralized applications built on blockchain technology, (2) Rooch Network[4], which is a fast, modular, secured, developer-friendly infrastructure solution for building Web3 Native applications, and (3) Fleek Network[5], which facilitates the deployment and running of performant, geo-aware decentralized web and edge services. These codebases typically use the base64, hashing, and signature algorithms from `fastcrypto`.

In order to meet the performance demands of a scalable blockchain that must process thousands of transactions per second, `fastcrypto` has been continuously and rigorously benchmarked through the entire development process. This has informed decision-making, in particular in the early stages of the development where many crucial and largely irreversible choices had to be made.

This paper gives examples of some actionable insights acquired through our benchmarking efforts while developing the `fastcrypto` library. These insights have been leveraged for both the refinement of the library itself, and the optimization of cryptographic operations within Rust and Move [11] language based blockchains. In some cases this also led to changes in external libraries. It is our hope that these insights may be useful for researchers or developers working on performance critical systems.

2 Method

All cryptographic functions in the `fastcrypto` library are benchmarked continuously as part of the library's continuous integration workflow[6] and a report of the results are published online[7]. The report is generated using the criterion crate [3] and when applicable, functions are benchmarked with various input sizes and grouped together with similar functions to enable comparisons. Benchmarks are run sequentially and each measurement is run 100 times. The report contains the mean and standard deviation of the observed timings for further analysis. At the time of writing (January 2024), the report contains 109 different benchmarks.

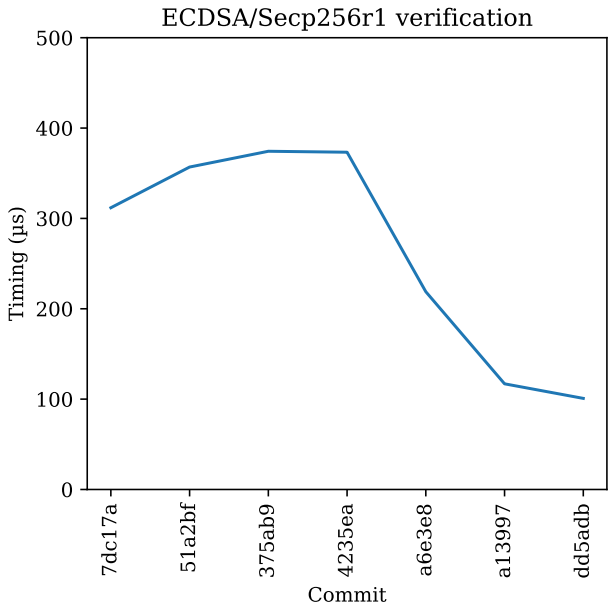

Figure 1: Historic performance of a digital signature verification using the ECDSA signature scheme over the secp256r1 curve.

The report contains historic data, allowing the detection of improvements or regressions in performance. As an example, Figure 1 shows a plot from the published report of the performance of verifying a digital signature using the ECDSA signature scheme over the secp256r1 (aka P-256) curve. This function has been improved several times which is reflected in the graph. These particular improvements are described in detail in section 3.1.3.

The data behind the report is published online in JSON format and may be analyzed using any statistical analysis tool. We have implemented a tool in Python[8] to utilize statistical libraries such as `numpy` [25] for more elaborate statistical analysis and plotting of the data. All plots in this paper were generated using this Python script.

To identify bottlenecks when the cryptographic functions in `fastcrypto` are used elsewhere, we have made *Dummy* implementations of digital signatures and hash functions. These implementations use the same interfaces as the actual cryptographic functions and can be used in place of these. They are not cryptographically secure but are extremely fast, so when they are used in testing they allow a developer to identify where cryptographic operations are a bottleneck in their implementation.

[1] `https://sui.io/`

[2] `https://github.com/MystenLabs/fastcrypto/network/dependents`

[3] `https://db3.network/`

[4] `https://rooch.network/`

[5] `https://fleek.network`

[6] `https://github.com/MystenLabs/fastcrypto/blob/main/.github/workflows/benchmarking.yml`

[7] `https://mystenlabs.github.io/fastcrypto/benchmarks/criterion/reports/`

[8] `https://github.com/jonas-lj/fastcrypto-analyzer`

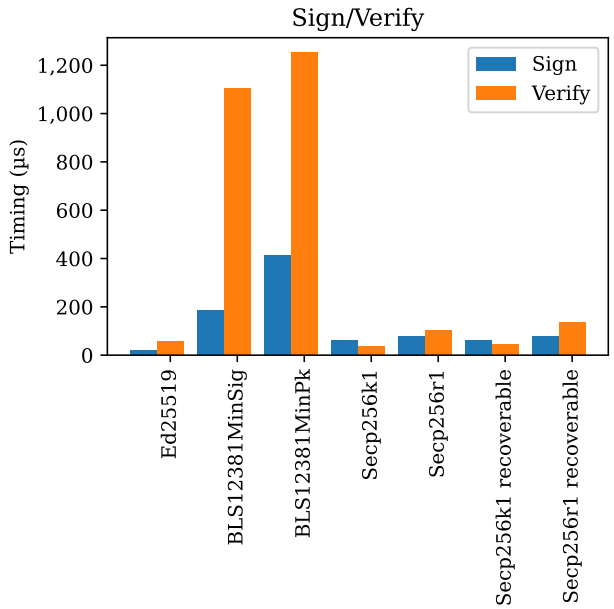

Figure 2: Performance of signing and verifying a message using various digital signature schemes. Secp256k1 and Secp256r1 are variants of ECDSA.

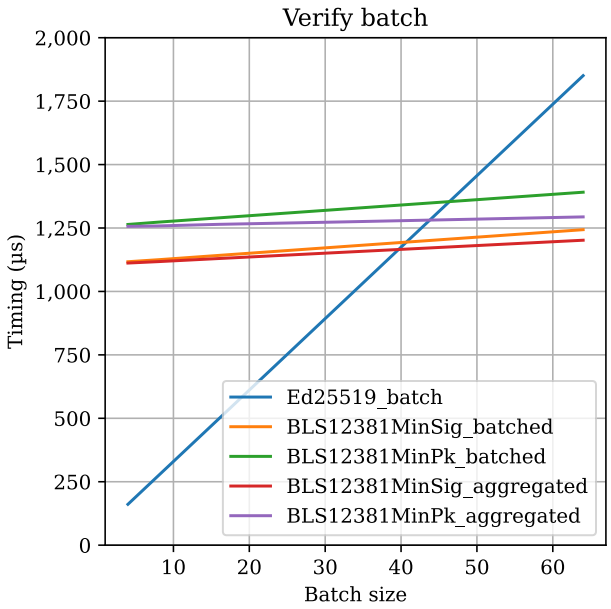

Figure 3: Performance of batched verification of digital signatures using the EdDSA and BLS signature schemes.

3 Case studies

The continuous benchmarks have greatly influenced the decision-making in the development of the `fastcrypto` library and in how it is used in the Sui blockchain and later in other projects. In this section, we outline some of the insights we achieved through the benchmarks and their consequences for the development.

3.1 Picking the right dependencies and specs

3.1.1 Signature aggregation can be catalytic

The BLS signature scheme [13] allows multiple signatures generated under different public keys for the same message to be aggregated into a single signature which is valid only if all the individual signatures are valid [12]. In a blockchain setting, this has the potential to speed up validators' signature verification significantly, as it is possible to aggregate signatures and batch the verification, instead of individually submitting and verifying many independent signature payloads.

Signature schemes such as EdDSA and ECDSA are much faster than BLS for individual signatures (see Figure 2), but do not provide the same performance gain when signatures are batched, so choosing the right signature scheme requires careful assessment of performance [21].

Our benchmarks (see Figure 3) show that there are a number of signatures where verifying an aggregated BLS signature is the fastest option compared to EdDSA, and that using the fastest available implementations of EdDSA [26] and BLS [30], the break-even point is around 40-45 signatures.

Since BLS is used in the Sui blockchain to aggregate validators' signatures, this implies that if there are more than 45 validators, using BLS will be faster than EdDSA. At the time of writing, there are 106 validators in Sui, meaning that verifying aggregated BLS is about 2× faster than EdDSA, when all validators sign.

3.1.2 Hash functions - in the mercy of hardware specs

In blockchains, cryptographic hash functions are arguably the most used cryptographic primitive, so even though they are relatively fast functions they may eventually become a bottleneck.

The performance of all cryptographic hash functions are approximately linear in the input size for sufficiently large inputs, but there are subtle differences in performance because the data is processed in blocks of varying sizes and this difference is more noticeable for small inputs. Sui originally used the Sha3-256 hash function that Meta's Libra [1] project originally utilized, but after benchmarking alternatives it switched to Blake2b [6] which is almost 3× faster and more zero knowledge proof friendly.

A plot of the benchmarks is shown in Figure 4. Note that Sha256 is the fastest hash function here, but this is not the case on all platforms. This is evident, for example, from the benchmarks published by the Blake2 team[9] which

[9] `https://www.blake2.net/`

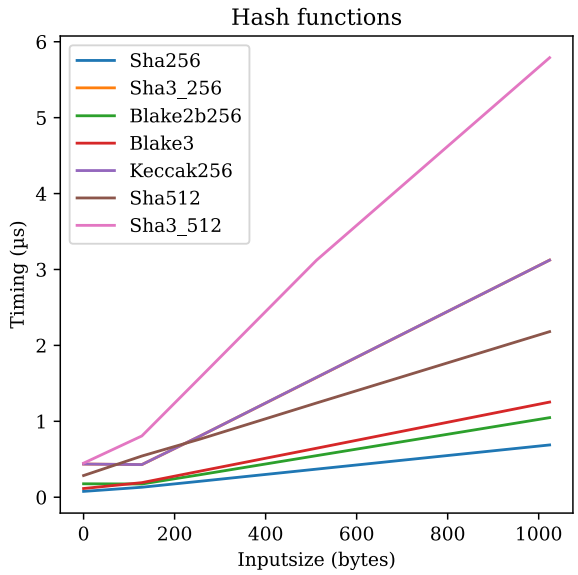

Figure 4: Performance of cryptographic hash functions.

shows that Sha256 is more than 2× slower than Blake2b, but we have also observed this in our benchmarks where the performance of Sha256 suddenly improved significantly between two runs without any changes in the software. We identified that this spike is due to recent updates in hardware for the cloud runner, because some hardware vendors have specialized CPUs to support Sha256 instructions; but running purely in software, Blake2b is faster.

We want to investigate this further and make our benchmarks fairer and more consistent, but it emphasizes the importance of benchmarking on a system similar to the production system because subtle differences (like CPU brand and model) can affect the performance significantly.

3.1.3 Deserialization can be expensive in cryptography

Many modern blockchains enable cryptographic agility for account signature key types. For instance in Sui blockchain, users may choose between a variety of signature schemes to sign their transaction[10]. This allows them to pick their favorite hardware wallet or their smartphone and store their keys securely. The default choice for the Sui blockchain (and many others) is EdDSA [10] over the ed25519 curve which was chosen based on high performance, determinism, adoption and standardization.

There are a few implementations of EdDSA in Rust, and comparing two popular crates (libraries) `ed25519-dalek` [15] and `ed25519-consensus` [26], which are backed by the same crypto arithmetic library, revealed some unexpected results, namely that the prior was much faster. Studying the source code closely showed that the difference was almost exclusively due to the fact that public keys in the latter are given in *compressed* serialized form, which is a representation where only one affine coordinate from the elliptic-curve point is given, meaning that the other coordinate has to be reconstructed before it can be used using a modular square root computation. This decompression operation is not for free. Accounting for this extra computation, the difference between the two libraries were then negligible. Some important lessons from this exercise are that a) we should be careful when comparing similar functions, (de)serialization can be expensive in cryptography, and b) there is a reason why some cryptographic libraries prefer one or the other, for instance the authors of `ed25519-consensus` explained that their method is safer when receiving public keys from the network, because the user does not need to take care of invalid keys before invoking the `signature.verify()` function; this is indeed a valid argument when keys are not cached or are unknown (typically the case in blockchain transactions).

3.1.4 Asymptotic complexity does not always tell the truth

EdDSA has a batched verification mode, where multiple signatures may be verified in a batch, giving some speed-up if enough signatures are verified together (see also section 3.1.1). While benchmarking this, we found an untapped potential optimisation: Typically, batch verification of n EdDSA signatures requires sampling n random scalars, but we found that $n-1$ is actually sufficient. For a small number of signatures, this gives a small speed-up as shown in Figure 5; this might make sense when we verify sponsored or atomic-swap transactions, where two accounts sign over the same transaction bytes.

3.2 Mix and Match Optimizations

3.2.1 Optimize ECDSA over the P-256 curve

As discussed in section 3.1.3 above, clients are allowed to choose among many signature schemes when signing their transactions, but it turns out that some schemes are slower than others so to avoid that verifying signatures of a particular scheme becomes a bottleneck for the entire system, the signature schemes are benchmarked continuously. The information from the benchmarks may be used to encourage users to use the faster schemes, for example by using these schemes as default choice in wallet implementations, but also to identify where optimising an implementation will have the largest effect.

As an example, the ECDSA signature scheme [2] may be realised over different elliptic curves. Two commonly used curves are secp256k1 which is used by the Bitcoin[11] and Ethereum blockchains [31] and the secp256r1 or P-256 curve which was specified by NIST and is used, for example, by the secure hardware on iPhone[12]. Both of

[10] `https://docs.sui.io/guides/developer/sui-101/sign-and-send-txn`

[11] `https://en.bitcoin.it/wiki/Secp256k1`

[12] `https://developer.apple.com/documentation/cryptokit/p256/signing/ecdsasignature`

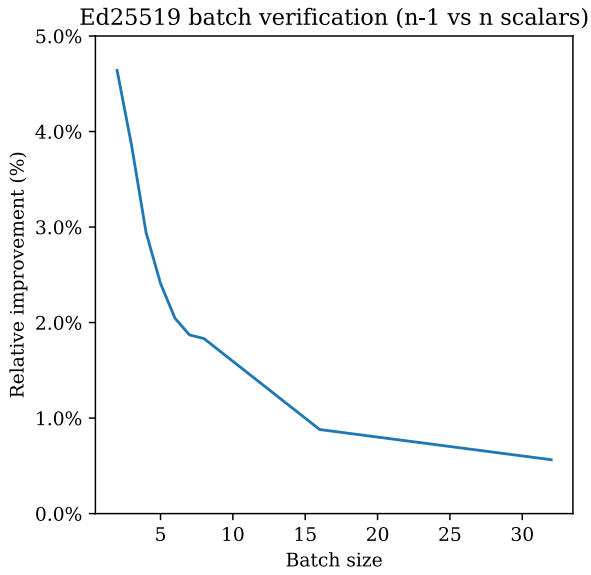

Figure 5: Relative performance improvement between using n and $n-1$ random scalars to verify a batch of n Ed25519 signatures.

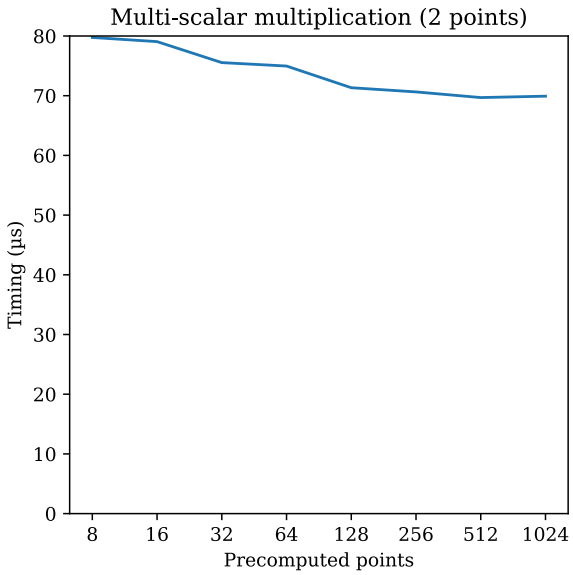

Figure 6: Performance of windowed multi-scalar multiplication with two points on Secp256r1 where one is known in advance over the number of precomputed points. As a reference, a naive computation without any precomputation takes 175 µs.

these are supported by the Sui blockchain and may be used by clients to sign their transactions.

Besides the choice of curve, there is no difference in the protocols for ECDSA over the two curves, but our benchmarks revealed that the fastest implementation of ECDSA over secp256k1 [27] is significantly faster than the fastest implementation of ECDSA over P-256 [28]. This motivated us to develop a new implementation of ECDSA over P-256 which uses a combination of faster elliptic curve arithmetic from Arkworks library [4] with a new, fast multi-scalar multiplication algorithm which requires some pre-computation. The optimised implementation verification for ECDSA over the P-256 curve is 5.5× faster, and is currently the fastest Rust implementation of ECDSA over the P-256 curve available.

Choosing the right number of pre-computed points for the multi-scalar multiplication required careful benchmarking, see figure 6. More pre-computed points (at least up to a certain limit) gives better performance but takes time and space. For our implementation, we use 256 points (each taking up 64 bytes) as default which gives a 68% improvement compared to not using multi-scalar multiplication at all and a 17.5% improvement compared to pre-computing only 16 points. Increasing the precomputation further to, say, 512 points would only give an 1.3% performance improvement, and for 1024 points, performance regresses, so 256 points was chosen as a compromise for our implementation. See Figure 6 for a plot of performance over number of pre-computed points.

3.2.2 A faster Poseidon hash function

The Poseidon hash function [16] is a hash function which is commonly used in zero-knowledge applications because it is easy to compute inside a zero-knowledge circuit. The Poseidon hash function is defined over a specific curve construction, and you need to use the same construction as for the zero-knowledge proof it is used in to get the performance benefit.

There are a few Rust implementations of the Poseidon hash, but not all implementations support all curve constructions. For our purpose, we needed to use the BN254 curve construction for zkLogin[13] [7] and only the poseidon-ark [5] crate supported this construction.

Benchmarking the zkLogin flow end-to-end revealed that computing the Poseidon hash took about 40% of the time so we decided to see if we could optimise it. We found that there are faster implementations of the Poseidon hash function in Rust in particular the neptune [22] crate, but at the time the neptune crate only supported the BLS12-377 curve construction and not the BN254 construction we needed in zkLogin. Using neptune over BN254 required a few changes to the implementation which we contributed by submitting code to the official repository [14] before we could use it. The resulting implementation is almost 70% faster cutting of 25% of the total end-to-end flow for zkLogin (Figure 7).

[13] `https://sui.io/zklogin`
[14] `https://github.com/lurk-lab/neptune/pull/236`

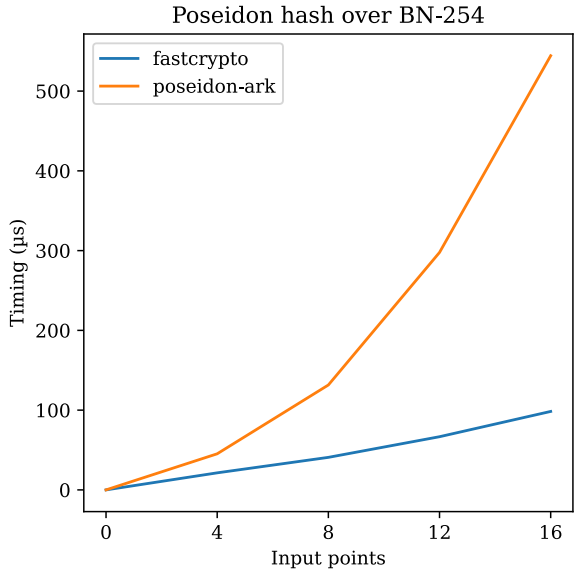

Figure 7: Performance of computing the Poseidon hash over the BN254 curve construction for 0-16 input points using the `fastcrypto` implementation compared with the arkworks-rs crate.

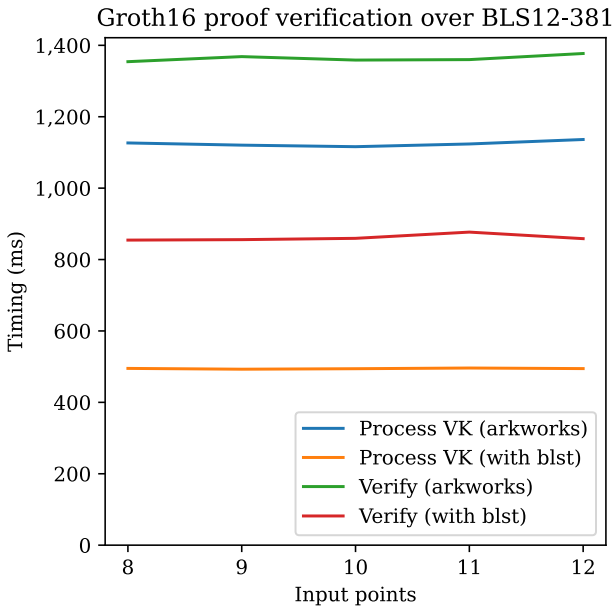

Figure 8: Performance our implementation of Groth16 zk-proof verification vs. Arkworks' implementation. The performance is independent of the input size, as the plot also shows.

3.2.3 Combining dependencies for optimal performance

`Fastcrypto` supports non-interactive zero-knowledge proofs using the Groth16 zk-SNARK construction [18] over two popular curves, namely the BN254 and BLS12-381 [8] curve constructions. Arkworks [4] have implementations of Groth16 for both of these constructions, but for the BLS12-381 construction the blst crate [30] provides a much faster implementation of the curve arithmetic, but does not provide any implementation of Groth16.

In `fastcrypto` we have combined Arkworks' implementation of Groth16 with the elliptic curve arithmetic from the blst crate to create a Groth16 implementation over BLS12-381 that is almost 2× faster than Arkworks implementation. To make this implementation efficient it was important to benchmark all steps of the algorithm independently, in particular the data conversions necessary to combine the blst and Arkworks libraries, to ensure that these conversions did not introduce a significant overhead. A performance comparison of our implementation with Arkworks' implementation is shown in Figure 8. Note that a full verification of a Groth16 zk-proof consists of processing the verification key *and* verifying the proof, but the processing of the verification key only have to happen once per circuit.

3.3 Errors and inconsistencies in dependencies

3.3.1 Bug in base64 implementations

`Fastcrypto` contains functions to encode data to and from base64 which is a very commonly used method to map binary data to ASCII characters, for example for use for serialization purposes. Implementing this, we tested out a few potential Rust crates to wrap in `fastcrypto` and benchmarked them on different input sizes.

The benchmarks revealed unexpectedly significant differences in performance between different libraries, and a closer study found that the difference was caused by some of the libraries not handling padding correctly. This inconsistency causes some libraries for base64 encoding to be incompatible, which is very unfortunate since base64 is often used for serialization and thus depends on portability. It also allows an attack vector on some systems because an attacker may utilize that different base64 strings are decoded into the same data to leverage an attack. This finding and a thorough description of the potential consequences has been published [14].

3.3.2 Exploitable vulnerability in EdDSA libraries

As previously mentioned, `fastcrypto` compares many implementations of the same signature schemes and then wraps the fastest or uses mix and match or applies extra expert optimizations. We realized that some exposed pub-

lic functions for EdDSA *signing* were significantly slower than other implementations even when the libraries where backed by the same back-end arithmetic dependency. A closer look resulted in identifying one of the most spectacular exploitable cryptography vulnerabilities, not only in Rust, but as a domino effect in dozens of cryptographic libraries, a potential vulnerability that was featured in the news [9] and for which a RUSTSEC fix was issued [29]. In short, many libraries, including the popular `ed25519-dalek` expose a `sign` function that additionally takes the public key as an input, and not only the private key and the message, which is the typical architecture in digital signature APIs. The reason behind this implementation design is speculated to be related to performance optimizations, because that addition allowed the function to avoid computing the public key (from the private) internally, and hence it was faster due to avoiding deserialization and other operations we highlighted in section 3.1.3. Note that exploiting such a function could result in private key leakage, an attack that we published as "Double Public Key Signing Function Oracle Attack on EdDSA Software Implementations" [17].

3.3.3 Unwanted parallelization for BLS verification

As with the base64 bug described above, surprising benchmark results are often a hint that some libraries are behaving unexpectedly. In an earlier version of `fastcrypto`, both BLS signature verification [13] over the BLS12-381 and the BLS12-377 constructions [8] were supported. However, BLS12-377, which used the Arkworks [4] implementation, was significantly slower than BLS12-381 which uses the `blst` [30] crate. Analysing this further, we noticed that blst by default allows multi-threaded computations. However, when allowing BLS12-377 to do the same, we got a *regression* in performance. It is unclear why this was the case, but the benefit of using multiple threads for BLS signatures is small (around 25% for `blst`), so if the threads are not managed tightly the small potential improvement from using multiple threads will be lost and performance will regress instead.

In our case, where the primary usage is to verify transaction signatures on the Sui blockchain, we decided to only allow single-threaded verification, because Sui is already a multi-threaded application, and allowing multiple threads for signature verification alone will complicate the thread management for Sui.

3.3.4 Unwanted parallelization for Groth16 proving

An important dependency for zkLogin is rapidsnark [19], a software library that leverages assembly code to speed up the process of generating a Groth16 zero-knowledge proof. It is well known that proving is one of the main remaining bottlenecks for zero-knowledge proofs. In order to optimize as much as possible, rapidsnark provides a server-like interface to process several requests at once. However, our testing revealed that the results returned by the prover under simultaneous requests was often erroneous. This was likely due to improper handling of state between threads resulting in one of the threads over-writing results of another.

Further inspection revealed that rapidsnark already utilizes available parallelism to generate a single zero-knowledge proof. Given this scenario, we decided to modify the library to disable the multi-request feature. We adopt a simpler strategy to handle simultaneous requests: scale the deployment horizontally by adding multiple machines.

We leave it for future work to conduct thorough benchmarks to identify if processing simultaneous requests on a single machine is actually useful. We suspect that it may only be useful on machines with a lot of parallelism or cores. Also note that when the number of cores is not high, then there is a risk of performance regression, that is, processing a request takes more time if there are simultaneous requests than otherwise, which is undesirable in most user-facing applications.

3.4 Continuous benchmarks

The life cycle of our primitives, from initial prototyping to production readiness, extends over several months. The initial implementation is typically unoptimized, emphasizing simplicity and accompanied by basic unit tests. Subsequent cycles focus on refining the primitive until it reaches a state suitable for performance measurements. Various evaluations are integral to this process:

- Local Benchmarks. These involve extensive testing with a diverse range of inputs. These benchmarks serve dual purposes—facilitating rapid development and ensuring progress across optimization cycles.
- Continuous Integration (CI) Tests. These tests are vital for ensuring that any future changes do not introduce performance regressions. They act as a safeguard against unintended setbacks in the optimization journey. This step is crucial as recent changes in sub-components of the library can impact the performance of primitives implemented and benchmarked in the past.

Continuous tests also guarantee accurate and up-to-date benchmark outcomes. They ensure that the latest performance measurements are reported, even in primitives implemented and benchmarked long ago.

4 Conclusion and future work

In the development of the `fastcrypto` library, continuous benchmarking has been a crucial tool in identifying bottlenecks and in qualifying the decision-making, notably when choosing what protocols and software libraries to use, but the benchmarks have in some cases also revealed unexpected insights into the inner workings of dependencies and even revealed critical bugs.

The benchmarks are published online and may also be used by developers to compare implementations or to compare with their own implementations. We have published a Python script to analyse the published data [15], and we

[15] `https://github.com/jonas-lj/fastcrypto-analyzer`

hope to integrate this script with our continuous integration workflow, e.g. to detect performance regressions automatically. The measurements show a large variation, probably because they are run on a cloud service, and we would also like to explore how to make measurements more consistent.

All in all, continuous benchmarks are more than a performance metric tool, it can be an excellent tool to identify vulnerabilities and allow for novel protocol designs and even world record implementations.

References

[1] Zachary Amsden, Ramnik Arora, Shehar Bano, Mathieu Baudet, Sam Blackshear, Abhay Bothra, G Cabrera, C Catalini, K Chalkias, E Cheng, et al. The libra blockchain. *URl: https://developers. libra. org/docs/assets/papers/the-libra-blockchain. pdf*, 2019.

[2] X9 ANSI. 62: public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ecdsa). *Am. Nat'l Standards Inst*, 1999.

[3] Jorge Aparicio and Brook Heisler. criterion.rs: Statistics-driven micro-benchmarking library. `https://github.com/japaric/criterion.rs`, 2024.

[4] Arkworks. arkworks-rs. `https://github.com/arkworks-rs/`, 2024.

[5] arnaucube. poseidon-ark. `https://github.com/arnaucube/poseidon-ark`, 2024.

[6] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, page 119–135, Berlin, Heidelberg, 2013. Springer-Verlag.

[7] Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Yan Ji, Jonas Lindstrøm, Deepak Maram, Ben Riva, Arnab Roy, Mahdi Sedaghat, and Joy Wang. zklogin: Privacy-preserving blockchain authentication with existing credentials, 2024.

[8] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In Stelvio Cimato, Giuseppe Persiano, and Clemente Galdi, editors, *Security in Communication Networks*, pages 257–267, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[9] Ben Dickson. Dozens of cryptography libraries vulnerable to private key theft. The Daily Swig: `https://portswigger.net/daily-swig/dozens-of-cryptography-libraries-vulnerable-to-private-key-theft`, 2022.

[10] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[11] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. Move: A language with programmable resources. *Libra Assoc*, page 1, 2019.

[12] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 416–432, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[13] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 514–532, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[14] Konstantinos Chalkias and Panagiotis Chatzigiannis. Base64 malleability in practice. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 1219–1221, New York, NY, USA, 2022. Association for Computing Machinery.

[15] dalek cryptography. ed25519-dalek. `https://github.com/dalek-cryptography/ed25519-dalek`, 2024.

[16] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *USENIX Security Symposium*, 2021.

[17] Sam Grierson, Konstantinos Chalkias, and William J Buchanan. Double public key signing function oracle attack on eddsa software implementations. *arXiv preprint arXiv:2308.15009*, 2023.

[18] Jens Groth. On the size of pairing-based non-interactive arguments. pages 305–326, 05 2016.

[19] iden3. rapidsnark. `https://github.com/iden3/rapidsnark`, 2024.

[20] Kostas Kryptos. Blockchain research has advanced systems and cryptography. `https://twitter.com/kostascrypto/status/1626983601572302848`, 2023.

[21] Zhuolun Li, Alberto Sonnino, and Philipp Jovanovic. Performance of eddsa and bls signatures in committee-based consensus. In *Workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*, 2023.

[22] lurk-lab. neptune. `https://github.com/lurk-lab/neptune`, 2024.

[23] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[24] Mysten Labs. fastcrypto. `https://github.com/MystenLabs/fastcrypto`, 2024.

[25] NumPy Team. Numpy. `https://numpy.org`, 2024.

[26] Penumbra. ed25519-consensus. `https://github.com/penumbra-zone/ed25519-consensus`, 2024.

[27] Rust Bitcoin Community. rust-secp256k1. `https://github.com/rust-bitcoin/rust-secp256k1/`, 2024.

[28] RustCrypto. p256. https://github.com/RustCrypto/elliptic-curves/tree/master/p256, 2024.

[29] Rustsec. Double public key signing function oracle attack on ed25519-dalek. RUSTSEC-2022-0093: https://rustsec.org/advisories/RUSTSEC-2022-0093, 2022.

[30] Supranational. blst. https://github.com/supranational/blst, 2024.

[31] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger, 2014.