zkStream: a Framework for Trustworthy Stream Processing

Janwillem Swalens
janwillem.swalens@nokia-bell-labs.com
Nokia Bell Labs
Antwerp, Belgium

Lode Hoste
lode.hoste@nokia-bell-labs.com
Nokia Bell Labs
Antwerp, Belgium

Emad Heydari Beni
emad.heydari_beni@nokia-bell-labs.com
Nokia Bell Labs
Antwerp, Belgium

Lieven Trappeniers
lieven.trappeniers@nokia-bell-labs.com
Nokia Bell Labs
Antwerp, Belgium

Abstract

In stream processing, managing sensitive information in a timely manner while ensuring trust remains a significant challenge. When parties without a priori trust cooperate to execute a streaming application, it is difficult to ensure that sensitive data is kept confidential while guaranteeing that every party executes their code honestly.

This paper presents zkStream: a framework that leverages signatures and zero-knowledge proofs (ZKP) to add trust to streaming applications that run in the edge cloud, guaranteeing data confidentiality, provenance, and computational integrity. We introduce two optimizations to minimize the computational overhead associated with ZKPs, making our framework suitable for real-world applications.

We validated our solution with existing benchmarks for streaming applications. Our method achieves an end-to-end latency that is between 6.5 and 15× faster than a naive implementation, demonstrating its potential for industrial adoption where trust is critical.

CCS Concepts

• **Software and its engineering** → **Data flow architectures**; • **Information systems** → **Data streaming**; • **Security and privacy** → **Domain-specific security and privacy architectures**; *Privacy-preserving protocols.*

Keywords

streaming, zero-knowledge proofs, privacy, confidentiality, computational integrity

ACM Reference Format:
Janwillem Swalens, Lode Hoste, Emad Heydari Beni, and Lieven Trappeniers. 2024. zkStream: a Framework for Trustworthy Stream Processing. In *24th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3652892.3700763

MIDDLEWARE '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-0623-3/24/12
https://doi.org/10.1145/3652892.3700763

1 Introduction

Distributed stream processing systems have enabled a new era of data processing. In this paradigm, software code is executed across various locations, strategically placed near data sources to optimize efficiency. Moreover, such a distributed architecture not only improves scalability and efficiency, but also holds the potential to safeguard sensitive data.

For example, consider a household equipped with a smart electricity meter. The electricity supplier wants to aggregate this data in short intervals to adjust pricing according to the dynamic supply and demand within the electricity grid [44]. To minimize computational and network costs, as well as latency, it makes logical sense to delegate parts of the aggregation operations to a location within the households, typically referred to as an "edge cloud".

Two concerns arise in such distributed settings. First, there is the issue of *privacy and confidentiality* for data producers: households share sensitive data, namely their power consumption patterns, throughout the system. Second, data consumers, here the electricity suppliers, must place their trust in multiple intermediaries for the correct execution of their business logic. This is referred to as *computational integrity*. Malicious actors – whether data producers, consumers, or other parties – may violate privacy requirements or manipulate local code execution to game the system to their advantage. Moreover, these systems rely on IoT devices that are operated by non-experts and for which vulnerabilities are rife [3, 16, 57, 60].

In other words, there is a need to establish trust between the different parties that collaboratively execute a streaming application. This need for trust in stream processing extends beyond the energy and utility sector to applications in the Internet of Things, real-time financial analytics, fraud detection pipelines, supply chain management, and more.

Popular streaming platforms, such as Apache Spark [70] or Flink [19], provide no trust guarantees. Hardware-based trust modules such as Trusted Execution Environments can be used to provide computational integrity and data confidentiality, attested to by a trusted hardware vendor [35, 62, 74, 87]. However, in hostile environments at the edge, these solutions are prone to complex attacks [21], exacerbated by the use of heterogenous hardware and the complicated roll-out of firmware patches for non-technical users (discussed in detail in Section 2.4).

In recent years, several software-only techniques have been developed to establish trust in code in the context of blockchain and smart contracts. In particular, Zero-Knowledge Proofs (ZKP) are a strong cryptographic technique that can be used to offer computational integrity while keeping sensitive inputs confidential [9, 83].

Our work applies digital signatures and zero-knowledge proofs to add trust to streaming applications. This comes with two challenges. First, ZKPs require careful consideration to guarantee the expected trust properties. Second, ZKPs introduce a large computational overhead and have a non-obvious performance model. We present these contributions:

- **zkStream: an architecture for trustworthy streaming applications** leveraging signatures and ZKPs. We define a trustworthy system as one that guarantees *confidentiality* of the sensor data and intermediate results, traceability of the *data provenance*, and *computational integrity*.
- **A library of 'gadgets'** of aggregation operations used in streaming systems, implemented efficiently for ZKPs.
- **Two optimizations to achieve practical performance**: (1) *lazy signature verification* to outsource expensive signature verification from the prover to the verifier, combined with (2) the use of *compact multi-signatures* to minimize communication overhead between the prover and verifier.

We have implemented and evaluated six benchmark programs using zkStream, in energy prediction and auctioning. We demonstrate that, while a naive approach results in unacceptable latency for these typical streaming applications, the optimizations lead to a trustworthy system that meets practical latency thresholds. Moreover, zkStream is released as open source[1], and thus is auditable and publicly verifiable.

2 Problem Statement: Trustworthy Stream Processing

In this section, we introduce a running example based on the DEBS 2014 Challenge [39]. Next, we define three trust properties – confidentiality, provenance, and computational integrity – and explain why they are desirable. We also present a generic system and adversary model for the distributed streaming applications we target.

2.1 Running example: electricity load prediction

Annually, the DEBS conference organizes a challenge in which teams compete to implement a streaming application. The challenges have clearly defined requirements and example data, and are representative of typical streaming applications. In 2014, the challenge revolved around the prediction of electricity consumption based on current and historical data from smart electricity plugs installed in households. We selected this challenge as its use case inherently demands trust: such predictions of electricity usage are used to influence production, demand, and trading on electricity markets.

In this application, a prediction of the electricity consumption is generated for the next 15-minute interval. We illustrate how this can be implemented in a streaming system in Figure 1 (without the blue parts). A plug generates a reading approximately every 5 seconds, containing the sensor's ID, a timestamp, and the current load (in Watt). There are two operators forming a dataflow program. First, the `historical` operator calculates historical averages of the same time interval on the previous 30 days, and stores its results in a data store. Next, the `prediction` operator combines real-time data from the sensor with historical averages from the store to compute a final prediction. The result is sent to a consumer: an electricity supplier that uses it to predict future demand.

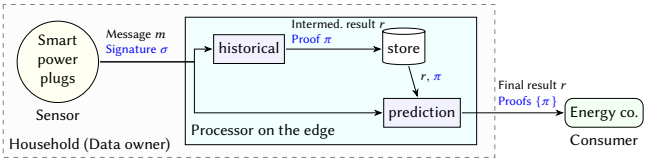

Figure 1: Pipeline for the load prediction case. Sensors generate messages (and signatures). The `historical` operator calculates an average per time window (and proof), saved in a data store. The `prediction` operator uses historical and real-time data to generate a prediction (and proofs), sent to an energy supplier. Elements in blue add trust.

The challenge prescribes that a prediction should be generated every 30 seconds. Note that this is different from the rate of incoming events, which is 1 reading every 5 seconds. Data is processed in tumbling windows of 15 minutes.

This application is one example of a geo-distributed streaming application that processes sensor data. It exhibits several patterns that are typical of such applications. We categorize these according to the programming model features defined by Isah et al. [37]:

- *Stateless* operations on *individual* messages: e.g. extracting the load value from incoming messages.
- *Aggregation* operations on *windows* of messages: calculating the average and median of tumbling windows.
- The *combination of real-time and batched data*: historical data is batched and processed in 15-minute partitions and then combined with real-time data.
- *Pipelining*: the data is processed in a pipeline of operators that may execute concurrently.
- A *latency threshold*: a result is generated every 30 seconds.

2.2 Trust: confidentiality, provenance, and computational integrity

In conventional solutions to the DEBS challenge and similar applications, sensor data is sent to external cloud servers for processing. However, users perceive this as a privacy invasion as it enables detailed tracking of their electricity usage [20, 56]. Similar concerns arise when data originates from a company, as knowledge of their energy usage can allow forecasting production output, resulting in highly sensitive commercial information being exposed [5, 44].

In general, data owners aspire to maintain the **confidentiality** of their data: data consumers may not learn anything beyond allowed leakage profiles, e.g. aggregated results derived using predefined algorithms. Consumers should not have access to the original sensor data, as they are typically sensitive from a privacy or business-strategic point of view.

[1] At https://github.com/Nokia-Bell-Labs/zkstream

An alternative solution is to perform the predictions locally, in an edge device or "edge cloud" that is operated by the individual household (or company). This guarantees that the sensor data never leaves the data owner's premises; only the aggregated final predictions are shared. However, in such a set-up a malicious user can manipulate the results. Moreover, they have a financial incentive to do so: this may allow them to buy electricity at more favorable prices. In this case, the electricity supplier cannot trust the results.

In general, data consumers want a guarantee that the data originated from trusted sensors. We call this **provenance**: a comprehensive record tracing the data's lineage, i.e. its origin, where it moved to, and how it was processed [14]. Moreover, consumers desire **computational integrity**: a proof that the data was processed using a predefined algorithm, honestly and accurately, without tampering with the code or data.

We consider these three properties – confidentiality, provenance, and computational integrity – as the corner stones needed to provide **trust**. To trust that the incoming data was produced correctly, the consumer requires provenance and computational integrity. To trust that the outgoing data cannot be misused, the data owner requires confidentiality.

2.3 System and adversary model

In general, we target geo-distributed applications involving three parties: **sensors** that generate readings, **data owners** (or **processors**) who run a streaming application that processes the sensor readings to results, and **data consumers** who receive the results.[2] We describe a generic system and adversary model.

Sensors. The sensors are IoT devices that emit readings. Typically, they have limited processing power and stringent energy requirements. They are deployed in the local network of the data owner, who has full access to their data.

We assume the sensors function reliably, meaning that their readings are trusted by both the data owner and the consumer. The sensors may be supplied by the consumer, while the data owner installs the sensor in their environment and can verify whether its readings are honest. The sensors can contain secrets, in particular signing keys, which may be safeguarded using hardware-based security modules like Trusted Platform Modules (TPM). (More techniques that provide these guarantees are discussed in Section 7.)

Data owners/processors. The data owners 'own' the incoming sensor data and can choose how it is used. Data owners do not wish to transmit their data to the cloud, to maintain confidentiality. Instead, the data is processed on an edge device or "edge cloud" managed by the respective data owner, which may range from a Raspberry Pi installed in a household to a server at a company. This device executes a dataflow program made up of operators, which continuously takes in sensor data and outputs results to consumers. Hence, we also refer to this party as the "(data) processor".

The data is processed in a fully untrusted environment. The processor can tamper with input data or act maliciously, such as skewing results, selectively omitting data, or even physical attacks of the device, and may have financial incentives to do so. Moreover, the edge device also faces many IoT security threats, including management by non-technical users [55], weak configurations [86], and slow rollout of security patches [67]. This device may also host other applications from untrusted parties.

Consumers. Consumers desire correctness of the output stream produced by the data processor. We interchangeably call them verifiers. We assume a secure communication channel from processor to consumer, e.g. using SSL. From the point of view of the data owner, it is important that the consumer does not learn the values of the individual sensor readings, only aggregated results.

2.4 Applicability of Trusted Execution Environments

Trusted Execution Environments (TEEs), such as Intel TDX and SGX, AMD SEV, and ARM TrustZone, establish their root of trust in hardware. These platforms provide a mechanism to attest to the computational integrity of code, as well as protecting confidentiality of data inside the environment. However, in the context of our system and adversary model, TEEs fall short in several aspects.

First, TEEs have been subject to numerous attacks [1, 21, 81, 82]. This risk is exacerbated when the TEE is deployed at the edge: it is exactly the operator of this device that has the incentive to attack it. (Currently, TEEs are often deployed in the cloud to protect against attacks from the outside, not from the operator of the device.) Moreover, patch management is tricky in our setting: vulnerabilities require firmware updates by non-technical users or may even need new hardware.

Second, TEEs require trust in vendors to implement the hardware correctly, while a software solution can be publicly audited. TEEs also depend on the vendor to provide timely patches for vulnerabilities for the device's whole lifecycle. In contrast, a software-based approach is easier to update.

Finally, IoT set-ups may depend on heterogeneous hardware. TEEs are vendor-specific, each with their own unique requirements, trust assumptions, update management protocols, and vendor-specific attestation reports. This heterogeneity increases development effort, expands the attack surface, and complicates the deployment of security patches.

The aim of this paper is to explore the use of a fully cryptographic approach, using ZKPs, as an alternative for a hardware-based approach using TEEs in this context. We discuss the trade-offs between ZKPs and TEEs in Section 8.

3 Background: Verifiable Computation using Zero-Knowledge Proofs

In this section, we introduce Zero-Knowledge Proofs, in particular zk-SNARKs, and the ZoKrates language.

Zero-Knowledge Proof. A Zero-Knowledge Proof (ZKP) is a technique that allows one party – the prover – to convince another party – the verifier – that a given statement is true, without revealing anything besides the fact that the statement is true [30]. ZKPs are used to prove that a computation was performed correctly (*computational integrity*), without revealing any intermediate values (*confidentiality*). To do so, the prover runs a program on top of

[2] This terminology may be slightly confusing in the context of the electricity prediction use case: the consumer is the electricity supplier that 'consumes' the predictions, *not* the consumer of the electricity.

```
def main(private field a, private field b) -> field {
    assert(a > 1 && b > 1);
    return a * b;
}
```

Listing 1: Proving knowledge of factors.

a proof system (e.g. Groth [32]) that generates a proof along with the program's output. The prover then sends the output and the proof to the verifier. The verifier runs a verification procedure that will convince it that the given output is a valid result of the given program, without needing to re-execute it. Moreover, programs can have public and private (or secret) inputs; the proofs are called zero-knowledge because verifiers only need the public inputs.

More formally, a Zero-Knowledge Proof allows a prover to convince a verifier of the statement that $\exists w : f(x, w) = y$, i.e. that the function f produced an output y for a public input x and a private input w. The proof is denoted π. A ZKP system has the following properties:

- **Completeness**: if the statement is true, an honest prover can convince an honest verifier of this fact.
- **Soundness**: if the statement is false, a cheating prover cannot convince an honest verifier that it is true, except with some extremely small probability.
- **Zero-knowledge**: the verifier learns nothing other than the fact that the statement is true.

Note that a Zero-Knowledge Proof is not a mathematical proof, rather it is a probabilistic 'proof' or *argument of knowledge* that convinces the verifier with a small soundness error. In a secure scheme, this error margin is negligible.

zk-SNARK. One of the most widely used types of ZKP is a Succinct Non-interactive Argument of Knowledge (zk-SNARK) [9]. zk-SNARK systems generate proofs with a small size (e.g. 256 byte, hence "succinct") and constant-time verification costs (in the range of ms), even for arbitrarily large programs. To generate a zk-SNARK, a program is typically transformed to an *arithmetic circuit*: a directed acyclic graph in which the edges or "wires" carry integers and the nodes or "gates" perform addition or multiplication. An arithmetic circuit can be transformed to a set of *constraints* (e.g. an R1CS or Rank-1 Constraint System), i.e. the relations encoded in the circuit that must be satisfied during its execution. If the constraints hold, the circuit was executed correctly, hence the program was executed correctly. There are several proof systems that generate zk-SNARKs [32, 33, 42, 63], as well as other proof systems (e.g. zk-STARK [8]). For detailed descriptions we refer to Parno et al. [63], Walfish and Blumberg [83].

ZoKrates. There are several tools and frameworks to create ZKPs. Developers can manually construct an arithmetic circuit using low-level libraries, or a language like Circom [7].[3] There are also higher-level programming languages that compile to arithmetic circuits, including ZoKrates [25], Noir, Leo, Cairo, and Lurk.[4] As programs are compiled to a fixed circuit, these languages require programs to have a static size. For example, in ZoKrates, `for` loops must have a static bound and are unrolled by the compiler. Variables, including the inputs, must also have a static size.

We use ZoKrates in this paper. A ZoKrates program contains a `main` function that can take private and public arguments and returns a public output. Listing 1 shows a program that proves knowledge of the factors of a large number, without revealing the factors themselves (the "Hello World" of ZKPs). The type `field` refers to an integer in a finite field.

In this program, we must also assert that `a` and `b` are not 1; otherwise, it would be trivial to generate a valid proof without actually knowing the factors. This illustrates how a ZKP must be constructed taking into account the interests of all parties: the keyword `private` is crucial to guarantee confidentiality to the prover, while the assertion is crucial to guarantee integrity to the verifier. In general, ZK programs are tricky to write robustly: a small bug can easily violate either property and such bugs have been found in widely used applications [2, 50].[5]

Phases. Generating a ZKP consists of the following steps:

- **Compilation**: a program is compiled to constraints.
- **Set-up**: the verifier generates a *proving key* that is sent to the prover and a *verification key* for itself.[6]
- **Computing the witness**: the prover evaluates the constraints with the (public and private) inputs and computes a witness. The *witness* is a 'trace' of the program execution that contains the values used in the constraints.
- **Proving**: using the witness and proving key, the prover generates a *proof*. There are several proof systems. We use `groth16` [32] in which the proof is comprised of three elliptic curve points, amounting to only 256 byte.
- **Verification**: the verifier verifies the proof and the public inputs and outputs with its verification key.

For a program that computes the SHA-256 hash of 256 bytes, on a typical laptop, compilation takes 19 s, set-up is 8 s, computing the witness takes 6 s, proving 7 s, and verification only 16 ms. This illustrates the large computational overhead of ZKPs: when computing the witness every operation of the arithmetic circuit is emulated in a finite field, and generating the proof requires constructing high-degree polynomials.

4 Trustworthy Stream Processing using Zero-Knowledge Proofs

In this section, we introduce zkStream. We first describe the general architecture, in which sensors attach signatures to their measurements and operators add proofs to their results (Section 4.1). Next, we describe how operators can run efficiently in the context of ZKPs, on single messages (Section 4.2) and on windows of messages (Section 4.3). Then, we show how real-time and batched data can be combined and how this model supports operator pipelining while preserving confidentiality (Section 4.4). Finally, we summarize the complete protocol (Section 4.5 and Table 1).

[3] https://github.com/filecoin-project/bellperson, http://arkworks.rs, https://docs.circom.io

[4] https://noir-lang.org, https://leo-lang.org, https://cairo-lang.org, https://lurk-lang.org. Lurk supports recursion.

[5] For example, the multiplication in our example program can overflow, which may be problematic depending on the use case.

[6] This is a *trusted set-up*; some proof systems do not require this.

4.1 Architecture

A dataflow program can be represented as a graph: messages originate in sensors (sources), flow through operators, and are eventually sent to consumers (sinks). Figure 1 showed how this applies to the energy prediction case. To add trust, we rely on two cryptographic techniques: signatures and zero-knowledge proofs.

A message $m = (s_{\text{ID}}, t, v)$ is composed of a sensor ID s_{ID}, a timestamp t, and a payload value v. The sensor signs each message using a public-key signature scheme, i.e. it has an embedded secret key sk and a public key pk, and generates the signature $\sigma = \text{SIG.sign}(sk, (s_{\text{ID}}, t, \text{hash}(v)))$. Here, $\text{SIG}(\text{keygen}, \text{sign}, \text{verify})$ is a standard public-key signature scheme. The public key is made available to all parties via standard protocols, and uniquely identifies the sensor.

Sensors send their messages and accompanying signatures to operators. The operators verify incoming messages using the signature and the sensor's public key, i.e. $\text{SIG.verify}((s_{\text{ID}}, t, \text{hash}(v)), \sigma, pk)$. This ensures the messages are authentic and originate from a known and trusted sensor. The operator then performs its computation, resulting in an output value v_o. Both steps – signature verification and actual computation – are executed in a ZKP system but the sensor values are kept secret. This generates a single zero-knowledge proof, thus proving that the computation was performed correctly and that it used authentic sensor readings to do so, while keeping the sensor values secret. The proof π is then sent along with the result $r = (s_{\text{ID}}, t, v_o)$ to the consumer. The consumer thus only receives non-confidential values.

The consumer verifies the proof using the ZKP's verification function, i.e. $\text{ZKP.verify}(v_o, \pi)$. Hence, zkStream already satisfies our three trust properties:

Confidentiality The operator only shares the result and a succinct proof with the consumer. Confidentiality is maintained thanks to the zero-knowledge property of the proof system: from these values alone, other parties cannot deduce the original sensor readings.

Provenance Signatures trace the data provenance, proving that a value originated from a trusted sensor. By verifying the signatures in the proof, the proof in turn certifies its inputs' origins to downstream parties. Hence, it suffices for a verifier to verify the proof to ensure the data origin, without needing access to the individual sensor values.[7]

Computational Integrity Given the result and proof, the consumer can verify the proof to conclude that the operator was executed correctly.

In the rest of this section, we will delve into what is needed to execute these steps in practice and how to apply them to a pipeline of multiple operators.

4.2 Stateless operations on single messages

The approach outlined in the previous section requires the operator's code to be executed on top of a ZKP system. As explained in Section 3, ZKP systems based on SNARKs have limitations on which code can be executed (efficiently). In this section, we describe some factors to consider when adapting code to run in a ZKP.

Data format and operations. Operations on integers are handled efficiently in many ZKP schemes because they natively support arithmetic in (large) finite fields. In contrast, operations on floating-point numbers and arbitrary-length string operations (e.g. JSON) are much less efficient as they require emulation and non-native processing.[8] In fact, ZoKrates only supports integer and boolean types, and arrays, tuples, and structs that are a composed of these.

For example, the data for the energy prediction case is represented as decimal numbers, expressed in Watt (e.g. $1.23\,\text{W}$). In our implementation, we multiply these numbers by 1000 and convert them to integers. We assume the sensors output and sign these integer values. This is necessary to make our implementation feasible.

Hashing and signature verification. Operators must verify the signatures of incoming messages. In our baseline implementation, we use the EdDSA signature scheme, which is based on elliptic curve operations. When the base field of the elliptic curve used for signatures is the same as the scalar field used for arithmetization of the ZKP (in our case the scalar field of the BN128 curve), signature verification can be executed efficiently in the ZKP. However, this assumes the sensors use this particular signature scheme. We address this limitation in Section 5, where we introduce a more versatile approach.

To verify signatures the hash of the value must be recomputed. This is expensive in ZK using a typical hash function. More efficient hash functions have been developed; we use the Poseidon hash function [31], which works natively in the finite field and can be compactly represented as a circuit.

4.3 Aggregation operations on windows of messages

Streaming applications typically partition messages into windows to perform aggregation operations. As mentioned in Section 3, ZKP systems based on arithmetic circuits require the program and input to have a fixed size, which obviously affects aggregation operations. Furthermore, a straightforward translation of the operation can be inefficient, so a ZKP-specific implementation is needed.

We developed a ZK-friendly implementation[9] of the 22 aggregation operations supported by Apache Flink [19] SQL,[10] as well as the median. In analogy with Campanelli et al. [17], Chen et al. [22], we refer to these as *gadgets*: small, specialized and efficient proof components that can be composed and re-used as part of a larger proof. We describe some common patterns below.

Windows and padding. Streaming platforms typically support three types of windowing operations – tumbling, sliding, and session windows [37] – which may include a varying number of messages. As ZKPs require fixed-size inputs, we set the window size to the maximum number of messages that can appear in a window and

[7] The prover may also make some of the messages' metadata public so that the consumer can verify if the appropriate inputs were used, e.g. to check uniqueness (preventing replay attacks) or freshness (timestamps).

[8] For example, zkjson (https://github.com/chokermaxx/zkjson) states that "proof generation takes hours."

[9] Available at https://github.com/Nokia-Bell-Labs/zkstream/tree/master/zkgadgets.

[10] https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/dev/table/functions/systemfunctions/#aggregate-functions. The three operations that use JSON or strings are not supported.

```
def average<N>(u64[N] vals, u32 n) -> u64 {
  u64 mut sum = 0;
  for u32 i in 0..N {
    sum = sum + ((i < n) ? vals[i] : 0); // ignore padding messages
  }
  return sum / cast(n); // floor division; cast n from u32 to u64
}
def median<N>(u64[N] vals, u32 n) -> u64 {
  for u32 i in 1..N { // assert that results are sorted
    assert(i < n ? vals[i - 1] <= vals[i] : true);
  }
  u32 i = n / 2;
  return (n % 2 == 0) ? (vals[i - 1] + vals[i]) / 2 : vals[i];
}
def stddev<N>(u64[n] vals, u64 stddev) -> u64 {
  u64 var = variance(a); // code to calculate variance not shown here
  assert(stddev * stddev <= var && (stddev + 1) * (stddev + 1) > var);
  return stddev;
}
```

Listing 2: Average, median, and std. deviation in ZoKrates.

pad the input with 'zero' messages. During subsequent operations, these padding values are discarded.

Listing 2 shows our `average` gadget. It is calculated as usual, except that a generic parameter `N` indicates the static size of the loop, while the parameter `n` contains the number of actual, non-zero messages. For the load prediction case, data is processed in tumbling windows of 15 minutes of up to 180 messages: by calling `average::<180>(vals, n)`, the ZoKrates compiler will unroll the `for` loop to 180 iterations.

Sorting. Calculating a median requires sorting a list, which is prohibitively expensive using arithmetic circuits as the whole sorting algorithm must be unrolled. Hence, we use a trick: the data is sorted outside the proof, and in the proof we only assert that they are sorted correctly. The `median` gadget is shown in Listing 2. Note that this trick requires a permutation check to ensure all values were included exactly once. This task is delegated to the verifier, which must check the program's inputs anyway. However, this is tricky, as the verifier can only see public inputs (e.g. message IDs) while the actual values are secret. Hence, (1) the verifier checks whether all messages were included once using their public ID, (2) the signature verification in the proof ensures that each message ID was tied to the corresponding value, and (3) the code for the median checks that the secret values were sorted correctly. This trick is also used for other operations that require sorting, like `maximum` and `percentile`.

Non-polynomial functions. Non-polynomial functions, such as calculating a square root, are not natively supported in ZK. One solution is to rely on a polynomial approximation [47]. For the square root however, we calculate the result outside the proof and then inside the proof check that it is correct by squaring. This is shown for `stddev` in Listing 2.

4.4 Combination of real-time and batched data and pipelining using confidential proof chaining

Streaming applications often combine real-time measurements with batched, historical data [48]. They are split into multiple operators that can execute concurrently, with some operators executing more frequently than others. We create one ZKP per operator, allowing us to execute operators independently and concurrently and cache intermediate results and proofs. This was illustrated for our use case in Figure 1.

This means an operator can now accept inputs not only from sensors, but also from previous operators or from a data store. While the sensor readings come with a signature, the intermediate results produced by previous operators come with a proof. Throughout the pipeline, all proofs that contribute to a result are accumulated, and in the end they are all sent to the verifier ($\{\pi\}$ in the figure). This allows the verifier to check that each operator was executed correctly.

However, this introduces an additional challenge: a naive implementation of this idea exposes intermediate results to the verifier, breaking confidentiality. This is unacceptable. Hence, we introduce a technique we refer to as **confidential proof chaining**. We make the following modifications:

- An operator i whose output v is confidential will create a commitment $h = \text{hash}(v, s)$ (with s a random salt) to the output, returning the commitment publicly and the actual result v privately.
- At the start of operator $i + 1$ that takes in the previous operator's result, the commitment is recalculated. In other words, this operator takes both the value v and the salt s as secret inputs and outputs $h = \text{hash}(v, s)$ publicly. The operator then uses v in its subsequent computations.
- The verifier verifies the links between the operators by checking that the commitment created by operator i is the same as was recalculated in operator $i + 1$.

A commitment scheme is a cryptographic primitive that allows one to 'commit' to a value (the *binding* property) while keeping it hidden to others (the *hiding* property). Here, we generate a commitment using a collision-resistant hash function. If a later operator can regenerate the same hash, it must know the same value (binding them). Adding a secret salt is crucial for confidentiality (hiding): without a salt, the hash is vulnerable to dictionary attacks, as the range of v is typically known.

However, we run into a limitation of ZKP systems: all computations must be deterministic so we cannot generate a random number. Hence, the processor generates the salt outside the proof and passes it into the proof as a secret input. It must generate the random salt in a cryptographically secure manner and must keep it secret. Fortunately, it is in its best interest to do so, because it is exactly this party – the prover – that wants to keep the value confidential.

Execution characteristics. This technique enables several features:

Pipelining Operators in a pipeline can execute concurrently, including proof generation.

Parallel execution Multiple invocations of a stateless operator can be executed in parallel, e.g. when processing data from multiple sensors.

Distribution Operators can be distributed across multiple machines; confidential proof chaining ensures that confidential intermediate results are not exposed.

Stateful operators This technique can also be used to create stateful operators, by passing the state as a confidential value from one to the next invocation of the operator.

Caching Results that are re-used can be cached, including proof generation. This does not require special trust or integrity properties from the storage system, so existing databases, caching, or storage infrastructure can be re-used.

In the load prediction case, we apply this to pre-calculate and cache the results of the `historical` operator (cf. Figure 1). Thus, when a new sensor value comes in only the `prediction` operator is executed, and latency is determined solely by this operator.

4.5 Summary

We now have a complete framework to create trustworthy streaming applications, which we refer to as zkStream. Table 1 contains the full protocol that should be applied to the sensors, processors (prover), and consumers (verifier). Sensor values as well as intermediate results are kept **confidential**, **provenance** is provided through signatures and proofs, and **computational integrity** is guaranteed using ZKPs.

Finally, we remark that the verifier must not only verify proofs, but also check their public inputs (e.g. correct timestamps, signed with a public key from a trusted sensor).

5 Optimizations: Lazy and Compact Signature Verification

In this section, we first examine the performance of the approach from the previous section for our running example (Section 5.1). This shows that the latency is too high, mainly due to the cost of signature verification. Hence, we present two optimizations: *lazy signature verification* (Section 5.2) and *compact multi-signatures* (Section 5.3).

5.1 Performance analysis of the naive approach

We implemented the DEBS challenge using the approach of the previous section – the "naive" approach. Results are shown in Table 4 (page 9) and will be discussed in detail in Section 6. In the first column, which shows this approach, we see that the end-to-end latency is 56 seconds. This exceeds the predefined threshold of 30 seconds of the DEBS challenge (cf. Section 2.1).

We now break down the cost of each part of the `prediction` operator. We first describe how we measured these results, as they are approximations, and then examine the results.

Our implementation is built on top of ZoKrates. This environment does not allow profiling code, as during proving the whole set of constraints is processed at once. We therefore 'profile' our program by slicing the `prediction` operator into four subprograms, shown in Table 2. Then, we measured the time to compute the witness and generate the proof of each part. Due to limitations on how we can measure ZoKrates programs, there are some constant costs, hence, the sum of these results is larger than the execution time of the original program. Our aim is not to get an accurate measurement of the absolute cost of each part, but to get an idea of their relative costs, to know which part to optimize.

Table 2 shows the results. We see that the cost of the actual computation is small. This is expected for our application: it consists of a small number (100s) of arithmetic operations, namely to compute the index of a median and to calculate an average. We also see that the cost of hashing is relatively small: this is thanks to the ZK-friendly Poseidon hash.[11] However, signature verification almost completely dominates execution time. Even though the EdDSA signature scheme is executed efficiently in an arithmetic circuit, it is still very costly in practice. Hence, we focus on optimizing that.

Sensor. The sensor with ID s_{ID} and secret key sk generates a reading with value v at timestamp t:
- **Gather message** Set $m \leftarrow (s_{\text{ID}}, t, v)$
- **Compute hash** $h \leftarrow \text{hash}(v)$
- **Compute signature** $\sigma \leftarrow \text{SIG.sign}(sk, (s_{\text{ID}}, t, h))$
- **Output** Output m and σ

Processor. An operator computes f, on sensor readings $\{m_i, ..., m_j\}$ with signatures $\{\sigma_i, ..., \sigma_j\}$ and previous operator outputs $\{r_k, ..., r_l\}$ with hashes $\{h_k, ..., h_l\}$, and with salt s_o:
In proof:
- **Verify signatures** For each $m \in \{m_i, ..., m_j\}$ and corresponding $\sigma \in \{\sigma_i, ..., \sigma_j\}$:
  - Set $(s_{\text{ID}}, t, v) \leftarrow m$
  - Retrieve $PK_s \leftarrow \text{PKI}(s_{\text{ID}})$
  - Compute $h \leftarrow \text{hash}(v)$
  - Check that $1 \stackrel{?}{=} \text{SIG.verify}((s_{\text{ID}}, t, h), \sigma, PK_s)$
- **Commit to inputs from operators** For each $r \in \{r_k, ..., r_l\}$ and corresponding $h \in \{h_k, ..., h_l\}$:
  - Set $(s_{\text{ID}}, t, v, s) \leftarrow r$
  - Check that $h \stackrel{?}{=} h' \leftarrow \text{hash}(v, s)$

  Set $\mathcal{H}_r \leftarrow (h_k, ..., h_l)$
- **Computation** $v_o \leftarrow f(m_i, ..., m_j, r_k, ..., r_l)$
- **Output** If the operator's output is public:
  - Output publicly: $(v_o, \mathcal{H}_r)$

  If the operator's output is confidential:
  - Compute $h_o \leftarrow \text{hash}(v_o, s_o)$
  - Output publicly: $(h_o, \mathcal{H}_r)$ and privately: (v_o, s_o)

The prover generates a proof π.

Consumer. The verifier checks for every operator, with output $(v_o, \mathcal{H}_r)$ or $(h_o, \mathcal{H}_r)$ and proof π:
- **Verify proof** If the output $(v_o, \mathcal{H}_r)$ was public:
  - Check that $1 \stackrel{?}{=} \text{ZKP.verify}((v_o, \mathcal{H}_r), \pi)$

  If the output $(h_o, \mathcal{H}_r)$ was confidential:
  - Check that $1 \stackrel{?}{=} \text{ZKP.verify}((h_o, \mathcal{H}_r), \pi)$
- **Verify commitments to inputs** For each $h_i \in \mathcal{H}_r$:
  - Check that $h_i \stackrel{?}{=} h_{o,i}$ (check that the commitment of the current operator is the one output by the previous operator)

Table 1: Protocol for verifiable stream processing via ZKPs with signature verification in proof. The blue parts are only needed if the operator takes in sensor readings; the green parts if the operator takes in previous operators' (confidential) outputs; and the red part if the output is confidential.

[11] We also performed experiments using the SHA-256 hash (not discussed in this paper), which we found to be 10–100× more costly.

	Compute witness		Prove	
	s	%	s	%
Complete `prediction` operator	20.50	100%	35.81	100%
Compute 180 Poseidon hashes	2.84	14%	6.49	18%
Verify 180 EdDSA signatures	20.03	98%	34.91	97%
Verify 30 hashes of prev. results	2.18	11%	4.84	14%
Predict (median of 30, avg of 180)	2.90	14%	6.58	18%

Table 2: 'Profile' of the `prediction` operator: the time to run and prove subprograms of the operator. Averages of 30 runs. We also show the result relative to the complete program.

5.2 Lazy signature verification

To optimize signature verification, we propose to perform it outside the proof. This requires outsourcing it to the verifier, as code executed by the prover outside a proof cannot be trusted. The complete protocol is described in Table 3 (without the green parts for now). It differs from the previous approach as follows:

(1) The sensor should *generate a random salt* for each message, and signs the salted hash.
(2) For each sensor reading, the prover takes in the value *and salt* as *private* inputs, and recomputes the hash. The hashes are returned as a (public) output.
(3) The hashes computed in the proof and signatures coming directly from the sensor are sent from prover to verifier. (The values and salts are not; they remain confidential.)
(4) The verifier:
   – Verifies the proof: this ensures the program was executed correctly *for the given public inputs and outputs*, thus linking the confidential messages $(m_i, ..., m_j)$ to their public hashes $(h_i, ..., h_j)$.
   – Verifies that each signature σ_i (sent out-of-proof) is valid for the corresponding hash h_i (output of the proof), using the sensor's public key.

This protocol still guarantees trust:

Confidentiality The sensor readings are never sent to the verifier, only a salted hash of their value is. Hence, the verifier cannot deduce the original values. Note that, in contrast to the previous approach, it is crucial that the hash is salted with a secret salt to prevent dictionary attacks.

Provenance The verifier possesses the sensor's public key and uses it for signature verification, ensuring that the data came from a trusted sensor.

Computational Integrity The verifier knows: (1) that values from a trusted sensor were used, by verifying the signatures, (2) that the program was executed correctly, by verifying the proof, and (3) that the signed values from step 1 were actually used as inputs to the program verified in step 2, by using the hashes that were output by the program to verify the signatures.

This optimization eliminates the most expensive step and thus greatly reduces proving time. In exchange, the verification time slightly increases as signatures are now verified in the verifier, although this is much cheaper than before as it can happen outside ZK. Performance measurements will be provided in Section 6.

Sensor. The sensor with ID s_{ID} and secret key sk generates a reading with value v at timestamp t:
– **Generate salt** Set $s \leftarrow$ random value
– **Gather message** Set $m \leftarrow (s_{\text{ID}}, t, v, s)$
– **Compute hash** $h \leftarrow \text{hash}(v, s)$
– **Compute signature** $\sigma \leftarrow \text{SIG.sign}(sk, (s_{\text{ID}}, t, h))$
– **Output** Output m and σ

Processor. An operator computes f, on sensor readings $\{m_i, ..., m_j\}$ with signatures $\{\sigma_i, ..., \sigma_j\}$ and previous operator outputs $\{r_k, ..., r_l\}$ with hashes $\{h_k, ..., h_l\}$, and with salt s_o:
In proof:
– **Commit to sensor inputs** For each $m \in \{m_i, ..., m_j\}$:
  * Set $(s_{\text{ID}}, t, v, s) \leftarrow m$
  * Set $h_i \leftarrow \text{hash}(v, s)$
  Set $\mathcal{H}_m \leftarrow (h_i, ..., h_j)$
– **Commit to inputs from operators** Unchanged.
– **Computation** Unchanged.
– **Output** If the operator's output is public:
  * Output publicly: $(v_o, \mathcal{H}_m, \mathcal{H}_r)$
  If the operator's output is confidential:
  * Compute $h_o \leftarrow \text{hash}(v_o, s_o)$
  * Output publicly: $(h_o, \mathcal{H}_m, \mathcal{H}_r)$ and privately: (v_o, s_o)
Outside proof:
– **Aggregate signatures** $\sigma_{agg} \leftarrow \text{SIG.aggregate}(\sigma_i, ..., \sigma_j)$
The prover generates a proof π, and a list of signatures $\Sigma \leftarrow (\sigma_i, ..., \sigma_j)$ or an aggregate signature σ_{agg}.

Consumer. The verifier checks for every operator, with output $(v_o, \mathcal{H}_m, \mathcal{H}_r)$ or $(h_o, \mathcal{H}_m, \mathcal{H}_r)$, proof π, and Σ or σ_{agg}:
– **Verify proof** If the output $(v_o, \mathcal{H}_m, \mathcal{H}_r)$ was public:
  * Check that $1 \stackrel{?}{=} \text{ZKP.verify}((v_o, \mathcal{H}_m, \mathcal{H}_r), \pi)$
  If the output $(h_o, \mathcal{H}_m, \mathcal{H}_r)$ was confidential:
  * Check that $1 \stackrel{?}{=} \text{ZKP.verify}((h_o, \mathcal{H}_m, \mathcal{H}_r), \pi)$
– **Verify commitments to inputs** Unchanged.
– **Verify signatures** For each $h_i \in \mathcal{H}_m$ and corresponding $\sigma_i \in \Sigma$:
  * Retrieve $PK_s \leftarrow \text{PKI}(s_{\text{ID}})$
  * Check that $1 \stackrel{?}{=} \text{SIG.verify}((s_{\text{ID}}, t, h_i), \sigma_i, PK_s)$
– **Verify aggregate sig.** Create $\mathcal{M} = ()$ and $\mathcal{PK} = ()$.
  For each $h_i \in \mathcal{H}_m$:
  * Retrieve $PK_s \leftarrow \text{PKI}(s_{\text{ID}})$ and add it to $\mathcal{PK}$
  * Add (s_{ID}, t, h_i) to $\mathcal{M}$
  Check that $1 \stackrel{?}{=} \text{SIG.verify}(\mathcal{M}, \sigma_{agg}, \mathcal{PK})$

Table 3: Protocol for verifiable stream processing with lazy signature verification and compact multi-signatures.

A drawback of this optimization is that a lot more metadata needs to be sent to the verifier: the hashes and signatures of all values that were used in the operator. For the load prediction example, this corresponds to approximately 23 kB additional metadata, while the actual ZKP is only 256 B.

5.3 Compact multi-signatures

Now that signature verification is performed natively by the verifier, we are no longer limited to signature schemes that execute efficiently in ZK. This enables a second optimization: we reduce the

metadata that is transferred alongside a result by using an aggregate signature scheme.

An aggregate signature scheme allows the signatures of many messages to be combined in a *single, compact* aggregate signature. We use the Boneh–Lynn–Shacham (BLS) [10, 11, 38] scheme. In particular, the function SIG.aggregate($\sigma_1, ..., \sigma_n$) creates an aggregate signature σ_{agg} that is only as large as a single signature. SIG.verify($\mathcal{M}, \sigma_{agg}, \mathcal{PK}$) verifies the aggregate signature with the messages $\mathcal{M}$ and public keys $\mathcal{PK}$.

In Table 3 (the green parts), this is applied to our protocol: the prover now aggregates all signatures and the verifier verifies the hashes with the aggregate signature. As such, only one signature is sent from prover to verifier.

BLS allows aggregate signatures from different public keys as well, so an operator can combine readings from different sensors. The verifier only needs to be aware of the sensor ID for each message (in the correct order).

This optimization adds a step to the prover, but as this can be done outside the proof, it is fast (16 ms for our running example to aggregate 5400 signatures). In exchange, the communication overhead from prover to verifier decreases.

6 Performance Evaluation

We evaluate the performance of zkStream using six benchmarks: the DEBS 2014 challenge (load prediction), energy flexibility, and four NEXMark queries (auctioning).[12]

6.1 Experimental set-up

Experiments ran on a machine with two Intel Xeon Gold 5318Y CPUs (each has 24 physical cores and 48 threads, with a 2.10 GHz base and 3.40 GHz max frequency) totaling 96 logical cores and 256 GB RAM. We used ZoKrates 0.8.7 [88], the `blst` library 0.3.10 [77], and the `babyjubjub_rs` library dated 2023-05-04 [4].

We measure the time to perform different phases of the application (cf. Section 3), implementing both the naive approach as well as the optimizations described in this paper. We look at the following metrics:

- **Compilation** and **set-up** only happen once per deployment and do not affect performance at run time. We only show them for completeness' sake.
- The prover has to **compute the witness** and **prove** for an operator, and **aggregate signatures**.
- **Verification** includes the time to verify the proofs, the signatures, and the metadata checks.
- Based on these, we calculate the **end-to-end latency**: this is the time between a new sensor reading coming in and the consumer having a verified result.
- We calculate the **data transfer** from prover to verifier, including the proofs, public inputs and outputs[13], signatures, and other metadata (e.g. public key, timestamps).

[12] All code is available at https://github.com/Nokia-Bell-Labs/zkstream.
[13] We calculate this number based on the actual implementation, which is suboptimal in some places. For instance, ZoKrates represents all integer types (u8, u32...) using 256 bit.

	Naive	Lazy sig. ver.	Compact multi-sigs
Hash function	Poseidon	Poseidon	Poseidon
Signature scheme	EdDSA	EdDSA	BLS
Compile historical (s)	327.52±6.10	70.60±0.51	70.45±0.63
Compile prediction (s)	341.49±5.05	79.76±0.47	79.99±0.67
Set up historical (s)	31.78±0.23	3.87±0.04	3.86±0.04
Set up prediction (s)	32.23±0.17	4.35±0.04	4.38±0.05
Comp. witn. historical (s)	19.94±0.08	1.78±0.01	1.80±0.01
Comp. witn. prediction (s)	20.25±0.11	2.05±0.01	2.07±0.02
Prove historical (s)	34.64±0.23	4.17±0.06	4.19±0.06
Prove prediction (s)	35.13±0.20	4.63±0.04	4.66±0.08
Aggregate signatures (s)			0.016±0.006
Verify (s)	0.71±0.00	3.51±0.05	1.92±0.02
End-to-end latency (s)	56.09	10.19	8.67
Data transfer (MB/h)	4.37	8.06	5.93
# constraints historical	3,162,491	295,001	295,001
# constraints prediction	3,207,083	339,593	339,593

Table 4: Performance measurements of the DEBS benchmark, implemented using the naive approach and the two optimizations. Each result shows the average and standard deviation of 30 runs.

- Last, we list the **number of constraints** for the compiled circuits. More constraints increase witness computation and proving times; verification time is constant.

Reported times are averages of 30 runs.

6.2 DEBS 2014 challenge: load prediction

We first look at the DEBS 2014 challenge, which was introduced in Section 2.1. Table 4 shows the performance results of the different phases of this program.

We first remark that the variation on the results is very low: this is logical as arithmetic circuits are deterministic and have a fixed size. Next, we observe that throughout the table, times for the `prediction` operator are slightly higher than those for the `historical` operator. This is explained by the fact that `prediction` performs more work than `historical` (cf. Section 2.1).

End-to-end latency. We focus on `prediction`, as this operator determines the end-to-end latency: this is the sum of the time to compute the witness and prove for the `prediction` operator, aggregate signatures (if applicable), and verify everything. We assume results and proofs of the `historical` operator are pre-calculated and cached (but not yet verified). The components contributing to the end-to-end latency are visualized in Figure 2.

When comparing the naive approach and the first optimization, we see that verifying signatures outside the proof greatly decreases the number of constraints, thus reducing the time to compute the witness (from 20 to 2 s) and to prove (from 35 to 5 s). In exchange, verification is 2.8 s slower. Overall, the end-to-end latency decreases

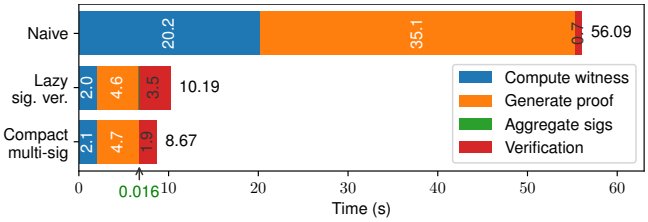

Figure 2: Breakdown of the components of the end-to-end latency, for the naive approach and the two optimizations, for the load prediction case.

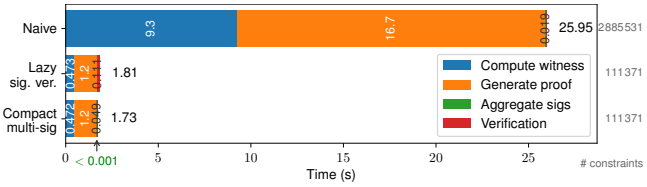

Figure 3: Breakdown of the end-to-end latency for the energy flexibility use case.

from 56 s to 10 s. As the DEBS challenge specified a threshold of 30 s, the naive approach is not acceptable while the optimization is.

For the second optimization, the code in the proofs is unchanged, hence these results are the same. The prover only spends an additional 16 ms to aggregate all signatures. As a result, verification is faster: it is faster to verify one aggregated BLS signature than 4156 EdDSA signatures (of all historical and current readings). As a result, the end-to-end latency is 6.5× lower than the naive approach.

Communication. We calculate the data transfer on an hourly basis based on the fact that the `historical` operator executes every 15 minutes and `prediction` every 5 seconds. Although a zk-SNARK is a succinct proof – only requiring 256 byte per se – there is a much higher communication overhead in a real implementation, due to additional metadata needed by the verifier. For the naive approach, this is comprised of all public inputs and outputs to the programs. Lazy signature verification increases the overhead: it requires a Poseidon hash (32 byte) and an EdDSA signature (96 byte) to be sent to the verifier for all 4156 sensor readings. Aggregated signatures decreases this: this optimization requires all hashes but only a single BLS signature (96 byte). The naive approach requires neither. However, overall, a communication overhead of a few MBs per hour remains very low.

Comparison with conventional solutions. We briefly compare our approach with existing solutions (without trust) to the DEBS 2014 Challenge. These solutions have a wide variation in performance results, with latencies of < 1 ms to > 2000 ms [34, 53] and throughputs of 2500 to 1 million events per second [54, 76], on a large range of hardware. In contrast, our approach, even with modern hardware and only processing data from a single power plug, is orders of magnitude slower. Hence, we do not claim the performance of our solution is within the range of a conventional one. Instead, we argue it is now feasible to add trust based on ZKPs to streaming applications where it is crucial, as the overhead is within practical latency thresholds.

We also note that the architecture is very different. In the existing solutions, all data is sent to and processed by centralized servers. In our approach, data is processed in a decentralized manner on devices in each household (at the edge) and a centralized server merely verifies the results. This allows greater scalability, while of course providing the privacy guarantees we aimed for.

6.3 Energy flexibility

Our second case is based on a real-world need for "energy flexibility": an energy grid operator can request a consumer to increase or decrease their electricity consumption to better balance the grid. For example, British Gas runs a program in which households can get a message that a "saving session" is upcoming, e.g. on a dark winter evening [13, 45]. Households that reduce consumption by 30% get a financial reward.

We implemented this use case using the same sensor data as the DEBS benchmark, but applying a different algorithm. Again, we assume the households do not want to leak their detailed usage, but the grid operator needs proof that the consumer really changed their consumption before handing out a financial reward. Our algorithm takes in 10 minutes of sensor data (120 readings) before the start of the saving session and 5 minutes afterwards (60 readings). We calculate the average electricity usage before and after this time point, and calculate the difference between the two averages. Hence, the algorithm only leaks how much the household increased or decreased consumption, but not their actual usage.

Figure 3 shows the performance. The naive approach takes 26 seconds, while the optimized versions take less than 2 seconds (15× speed-up), again trading a much faster time on the prover's side for a slight increase in verification time. A lower latency means the grid operator can rely on the correctness of this data earlier.

6.4 NEXMark: online auctions

The NEXMark benchmark suite is widely used for streaming systems [80]. It consists of an auction system (e.g. eBay). There are three tables: (1) users, (2) auctions, containing the item name and the seller's ID, and (3) bids, containing the auction ID, the buyer's ID, and a price. Our sample data comprises 25 auctions (in 5 categories) with 500 bids (averaging 20 bids per auction) by 5 users.

We consider the bid's price and auction ID as confidential. In other words, sellers receive bids but want to keep confidential for which auctions and at which price. However, at the same time we want to guarantee to other users that the seller answers queries about its auctions honestly.

The original NEXMark suite consists of eight queries, shown in Table 5. We note that queries 3 and 8 operate only on public data, thus they do not need our system. Queries 1 and 2 leak confidential data by construction; hence we disregard them too. In the end, zkStream is applicable to four queries that operate on and aggregate confidential data. We applied the protocol of Section 4.1 to these.

Figure 4 shows the latency of the queries. We see that our first optimization greatly reduces the latency for all queries, from several minutes to less than 45 seconds (speed-ups of 8 to 11×). For this

	Description	Included?		Description	Included?
1	Currency conversion	✗ (leak)	5	"Hot" auctions	✓
2	Selection of bids	✗ (leak)	6	Avg. price per seller	✓
3	Find local auctions	✗ (public)	7	Highest overall bid	✓
4	Avg. price per category	✓	8	New users	✗ (public)

Queries annotated with "✗ (public)" are not included as they only operate on public data, while queries annotated "✗ (leak)" leak confidential data.

Table 5: Applicability of zkStream to the NEXMark queries.

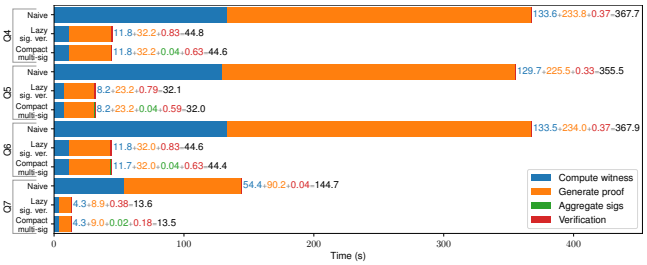

Figure 4: Breakdown of the end-to-end latency (in seconds) for four queries of the NEXMark benchmark.

benchmark, the second optimization has less impact than for the DEBS challenge, as there are only 500 signatures (vs. 4156 for DEBS), reducing the benefit of aggregation.

Queries 4, 5, and 6 consist of two steps, for which the data is partitioned differently. We therefore have split these queries into a pipeline of two operators, which are invoked multiple times. For example, query 4 finds the average price per category: a first operator is invoked for each auction to find the maximum bid, the data is then repartitioned, and for each category a second operator calculates the average. Hence, it consists of 25 invocations of the first operator (taking on average 449 ms to compute the witness and 1223 ms to prove, for optimization 2) followed by 5 invocations of the second one (average of 112 ms to compute the witness and 315 ms to prove). Our benchmark executed these sequentially, hence the results are simply the sum of the execution times of each operator invocation.[14] However, these queries allow pipeline parallelism and concurrent execution, and in a practical system, the different operator invocations could be executed concurrently.

If we compare the four queries, we see that query 4 and 6 are very similar: both first find the maximum price per auction and then average these, either per category or per seller. Consequently, the performance is also very similar. Query 5 first counts the number of bids per auction, and then selects the maximum. This requires more computation in the first operator, but less in the second, leading to an overall performance that is slightly better than queries 4 and 6. For query 7, only a single operator is needed as data does not need to be partitioned: this query fetches the highest bid across all auctions. As a result, it is much faster than the other queries. In general, the fewer times data must be re-partitioned during a query, the better performance will be. In practical systems, it will also be important to structure queries such that the greatest reduction in data occurs in the earliest operator.

[14] E.g. $25 \times 0.449 + 5 \times 0.112 = 11.8$ s and $25 \times 1.223 + 5 \times 0.315 = 32.2$ s, which corresponds to the numbers for the second optimization for Q4 in Figure 4.

6.5 Lessons learned

We share some of the lessons learned while developing trustworthy streaming applications.

First, developers must take into account the specific performance model of ZKPs. Algorithms and data structures should lend themselves to use in an arithmetic circuit and should be implemented with performance in mind. For operations like sorting, searching, or graph traversal, developers must resort to specialized techniques. We developed several gadgets for this, but more may be needed.

Meanwhile, it is tricky to avoid bugs. Existing work [2, 50] focuses mostly on integrity errors, while confidentiality errors are less well studied. However, in our work, we see these are also easy to introduce, e.g. by not salting a hash or not properly considering the leakage profile. Compiler optimizations could be developed to improve performance automatically, as well as static or dynamic analysis techniques to detect integrity or confidentiality bugs.

Moreover, as a commitment must be generated for every confidential input and output, sensors should serialize to formats that are cheap to parse and should use a ZK-compatible hash function.

Finally, application's architecture should be considered carefully. In the DEBS challenge and some NEXMark queries, programs can be split into different operators thus enabling pipelining and parallelism. If this is not possible, performance is severely hindered.

7 Security Analysis

We revisit our adversary model (Section 2.3). Our approach is robust to the following attacks:

- ✓ Manipulation of sensor results after they leave the sensor.
- ✓ The processor using a different algorithm than specified.
- ✓ We can detect if the processor drops inconvenient sensor readings. It is up to the consumer to decide whether this is considered an accident (e.g. network issues) or malicious.
- ✓ If the processor's hardware is compromised, integrity is maintained. We do not require a root of trust in hardware.
- ✓ If the processor's software including the OS, platform libraries, or other programs that run on the same device are compromised, integrity is maintained. Our trusted computing base (TCB) is limited to the ZKP platform (e.g. ZoKrates), our gadget library, and the user's query. This code is open source and publicly verifiable.
- ✓ Physical attacks of the edge device cannot break integrity.
- ✓ After the result has left the processor's site, network adversaries cannot compromise confidentiality or integrity.

The following attacks are not covered by our framework, but must be mitigated using other techniques:

- ✗ Incorrect sensor readings, whether due to a faulty sensor or deliberate manipulation. Some sensors are more vulnerable (e.g. thermometer) than others (e.g. wattage meter). In some cases, it may be possible to correlate with other sensor data, verify results post-hoc, or use hardware-based protection mechanisms [29, 49, 71]. It may also be necessary to transform data before usage using a verifiable technique [24, 41, 59].
- ✗ Physical attacks to acquire a sensor's secrets, in particular its signing key. This may be protected using hardware-based security like Trusted Platform Modules (TPM).

- ✗ Attacks on the ZK platform, e.g. exploits of bugs in the compiler or the program. This code should be open-sourced and verified by both prover and verifier.
- ✗ Reverse engineering sensor readings from final results. E.g. in the load prediction case, by storing subsequent predictions, the energy supplier may be able to derive an approximation of the average usage over the last 30 seconds, as well as detect patterns (e.g. if the user is at home). Even though this is explicitly allowed by our leakage profile, this may be unexpected to some users.
- ✗ If the processor's hardware or software is compromised, integrity is maintained but data confidentiality is no longer guaranteed. Similarly, an adversary that intrudes in the processor's internal network can gain access to confidential sensor readings. It is only possible to protect against this if the sensor encrypts its outgoing values, which are then processed in a TEE or FHE setup.

8 Related Work

ZKP & Streaming. ADSNARK [6] and Fiore-Tucker [27] use ZKPs for a streaming set-up similar to ours. They present a homomorphic signature scheme by combining it with commit-and-prove SNARKs [18] to achieve a system that can prove arbitrary computations. Our system allows sources with different keys, and operator pipelining. Using lazy signature verification, zkStream is also compatible with existing signature schemes and likely offers better performance.

zk-IoT [66] is a framework for distributed computation over IoT devices, using ZKPs to guarantee computational integrity even if devices are malicious or compromised. This approach has a different set-up to ours: it is comprised of only IoT devices that communicate with each other; there is no explicit data owner or consumer.

Distributed and Hardware-accelerated ZKP. Recent work has explored distributing ZK proof generation on clusters [40, 51, 68, 85]. However, these techniques require trust in the cluster owner to maintain confidentiality, while our use cases focus on generating proofs at the edge, where such hardware is not available. There are also techniques that leverage GPUs [36, 52], ASICs, and FPGAs [65] to generate proofs, which could also be applied in our framework.

Trusted Execution Environments (TEEs). There is extensive research applying hardware-based security to streaming. Most approaches run computations server-side (i.e. the consumer in our approach) and thus focus on protecting confidentiality [35, 46, 74, 87]. VC3 [74] applies SGX to MapReduce, but can be susceptible to side-channel attacks. Opaque [87] presents a technique to hide access patterns; however, it requires the Spark master to run in a trusted domain (i.e. at the processor in our terminology). These systems do not necessarily provide public verifiability, and have a relatively large Trusted Computing Base.

Unlike the above approaches, Streambox-TZ [62] processes data at the edge, and proposes optimizations specific to ARM TrustZone. zRA [26] combines hardware-based trust with ZKPs to provide remote attestation. By only requiring limited hardware components, it is vendor-agnostic and likely less vulnerable to hardware exploits.

In contrast, zkStream offers a purely software-based approach, which provides both stronger cryptographic guarantees in theory (relying on mathematical assumptions instead of trust in hardware) and better updatability in case of an exploit. This makes it suitable for deployments at the edge or in other scenarios that demand strong integrity guarantees in a hostile environment, as discussed in Section 2.4. In return, ZKPs impose a hefty performance penalty. Hence, whether TEEs or ZKPs are more suited for a streaming application is a trade-off between the required trust guarantees, performance or latency thresholds, available hardware, cost, and the specific algorithm that is executed. Moreover, a hybrid approach could combine both, alleviating the risks of each individual system.

Finally, we refer to Russinovich et al. [69] for a comparison between TEE and ZKP, and Popa [64] for a comparison between TEE, FHE, and MPC.

Verifiable Data Streaming & Verifiable Computation. Another line of research focuses on Verifiable Data Streaming [43, 72, 73, 84], where a data owner streams data to a (potentially malicious) server. TimeCrypt [15] present a key derivation tree using homomorphic MACs, enabling confidentiality and access control. Unlike these works, zkStream aims to enable the execution of general-purpose functions. They design custom cryptographic tools that are likely more efficient, but not general purpose. The works above also focus on malicious storage providers for data streams, while we allow pipelining of multiple operators, therefore requiring provenance. Hence, we consider these primitives, including Authenticated Data Structures [58, 61, 78], to be interesting techniques that could be incorporated in our framework.

These approaches trace back to Verifiable Computation (VC) [28], in which a client delegates code execution to an untrusted server and can verify computational integrity. However, this violates confidentiality. Consequently, Pantry extends VC with private state stored in untrusted storage, and applies it to MapReduce jobs [12]. This work has been extended in the ZKP community [18, 23].

Homomorphic encryption (HE). MrCrypt [79] and Styx [75] use homomorphic encryption to provide trustworthy stream processing. In this approach, data is encrypted by the client and processed server-side using an HE scheme. HE is an interesting alternative to ZKP, but likely has worse performance (as operations work on ciphertexts) and a more limited set of efficient operations (e.g. multiplication is more complex and multiplicative depth is limited) [64].

9 Conclusion

This paper presents zkStream: a framework for **trustworthy streaming applications** based on signatures and Zero-Knowledge Proofs. A streaming application is trustworthy if it preserves the *confidentiality* of sensor inputs, tracks the data *provenance*, and guarantees the computational *integrity* of results. zkStream consists of an architecture that supports typical features of stream processing systems: aggregation through a set of optimized *aggregation gadgets*, the combination of real-time and historical data using *confidential proof chaining*, and caching and pipelining. It also includes two optimizations – *lazy signature verification* and *compact multi-signatures* – to make streaming applications feasible in practice. As a result, despite their hefty performance overhead, this paper offers a proof point for the feasibility of ZKPs for streaming applications where trust is critical, opening up a path to new use cases for trustworthy stream processing.

References

[1] 2019. CVE-2019-11157: Improper conditions check in voltage settings for some Intel(R) Processors may allow a privileged user to potentially enable escalation of privilege and/or information disclosure via local access. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11157

[2] 0xPARC contributors. 2023. ZK Bug Tracker. https://github.com/0xPARC/zk-bug-tracker/tree/6955d22

[3] Mohamed Abomhara and Geir M Køien. 2015. Cyber security and the internet of things: Vulnerabilities, threats, intruders and attacks. *Journal of Cyber Security* 4, 1 (2015), 65–88.

[4] arnaucube. 2023. *babyjubjub_rs*. https://github.com/arnaucube/babyjubjub-rs/commit/cd36d496e579dc0ff34f0d2c25cc8f013bf50efe

[5] Muhammad Rizwan Asghar, György Dán, Daniele Miorandi, and Imrich Chlamtac. 2017. Smart meter data privacy: A survey. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2820–2835.

[6] Michael Backes, Manuel Barbosa, Dario Fiore, and Raphael M Reischuk. 2015. ADSNARK: Nearly practical and privacy-preserving proofs on authenticated data. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 271–286.

[7] Marta Bellés-Muñoz, Jordi Baylina, Vanesa Daza, and José L. Muñoz-Tapia. 2022. New Privacy Practices for Blockchain Software. *IEEE Software* 39, 3 (2022), 43–49.

[8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046. https://eprint.iacr.org/2018/046

[9] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2012. From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*. Association for Computing Machinery, 326–349.

[10] Dan Boneh, Manu Drijvers, and Gregory Neven. 2018. Compact multi-signatures for smaller blockchains. In *Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part II*. Springer, 435–464.

[11] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *Advances in Cryptology—ASIACRYPT 2001: 7th International Conference on the Theory and Application of Cryptology and Information Security Gold Coast, Australia, December 9–13, 2001 Proceedings 7*. Springer, 514–532.

[12] Benjamin Braun, Ariel J Feldman, Zuocheng Ren, Srinath Setty, Andrew J Blumberg, and Michael Walfish. 2013. Verifying computations with state. In *Proceedings of the twenty-fourth ACM Symposium on Operating Systems Principles*. 341–357.

[13] British Gas. 2023. *Demand flexibility scheme saves 451MWh of electricity*. https://www.britishgas.co.uk/the-source/making-a-difference/peaksave-demand-flexibility-scheme.html

[14] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *Database Theory—ICDT 2001: 8th International Conference London, UK, January 4–6, 2001 Proceedings 8*. Springer, 316–330.

[15] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. 2020. TimeCrypt: Encrypted data stream processing at scale with cryptographic access control. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 835–850.

[16] Ismail Butun, Patrik Österberg, and Houbing Song. 2020. Security of the Internet of Things: Vulnerabilities, Attacks, and Countermeasures. *IEEE Communications Surveys & Tutorials* 22, 1 (2020), 616–644.

[17] Matteo Campanelli, Dario Fiore, and Anaïs Querol. 2019. LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, 2075–2092.

[18] Matteo Campanelli, Dario Fiore, and Anaïs Querol. 2019. Legosnark: Modular design and composition of succinct zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2075–2092.

[19] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[20] Ann Cavoukian, Jules Polonetsky, and Christopher Wolf. 2010. Smartprivacy for the smart grid: embedding privacy into the design of electricity conservation. *Identity in the Information Society* 3 (2010), 275–294.

[21] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1416–1432.

[22] Bing-Jyue Chen, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang. 2024. ZKML: An Optimizing System for ML Inference in Zero-Knowledge Proofs. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, 560–574.

[23] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. 2015. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 253–270.

[24] Trisha Datta, Binyi Chen, and Dan Boneh. 2024. VerITAS: Verifying Image Transformations at Scale. Cryptology ePrint Archive, Paper 2024/1066. https://eprint.iacr.org/2024/1066

[25] Jacob Eberhardt and Stefan Tai. 2018. Zokrates-scalable privacy-preserving off-chain computations. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, 1084–1091.

[26] Shahriar Ebrahimi and Parisa Hassanizadeh. 2024. From Interaction to Independence: zkSNARKs for Transparent and Non-Interactive Remote Attestation. In *NDSS 2024, Network and Distributed System Security Symposium, 26 February-1 March 2024, San Diego, CA, USA*.

[27] Dario Fiore and Ida Tucker. 2022. Efficient Zero-Knowledge Proofs on Signed Data with Applications to Verifiable Computation on Data Streams. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1067–1080.

[28] Rosario Gennaro, Craig Gentry, and Bryan Parno. 2010. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*. Springer, 465–482.

[29] Peter Gilbert, Jaeyeon Jung, Kyungmin Lee, Henry Qin, Daniel Sharkey, Anmol Sheth, and Landon P. Cox. 2011. YouProve: authenticity and fidelity in mobile sensing *(SenSys '11)*. Association for Computing Machinery, 176–189.

[30] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (STOC '85)*. Association for Computing Machinery, 291–304.

[31] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems.. In *USENIX Security Symposium*, Vol. 2021.

[32] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer Berlin Heidelberg, 305–326.

[33] Jens Groth and Mary Maller. 2017. Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs. In *Advances in Cryptology – CRYPTO 2017*, Jonathan Katz and Hovav Shacham (Eds.). Springer International Publishing, 581–612.

[34] Rohit Gupta, Rinku Shah, and Apurva Mhetre. 2014. In-Memory, High Speed Stream Processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. Association for Computing Machinery, 306–309.

[35] Aurélien Havet, Rafael Pires, Pascal Felber, Marcelo Pasin, Romain Rouvoy, and Valerio Schiavoni. 2017. SecureStreams: A reactive middleware framework for secure data stream processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. 124–133.

[36] Karthik Inbasekar, Yuval Shekel, and Michael Asa. 2024. ICICLE v2: Polynomial API for Coding ZK Provers to Run on Specialized Hardware. Cryptology ePrint Archive, Paper 2024/973. https://eprint.iacr.org/2024/973

[37] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. 2019. A Survey of Distributed Data Stream Processing Frameworks. *IEEE Access* 7 (2019), 154300–154316.

[38] Kristof Jannes, Emad Heydari Beni, Bert Lagaisse, and Wouter Joosen. 2023. BeauForT: Robust Byzantine Fault Tolerance for Client-centric Mobile Web Applications. *IEEE Transactions on Parallel and Distributed Systems* (2023).

[39] Zbigniew Jerzak and Holger Ziekow. 2014. The DEBS 2014 Grand Challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. Association for Computing Machinery, 266–269.

[40] Zhuoran Ji, Zhiyuan Zhang, Jiming Xu, and Lei Ju. 2024. Accelerating Multi-Scalar Multiplication for Efficient Zero Knowledge Proofs with Multi-GPU Systems. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 57–70.

[41] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. 2022. ZK-IMG: Attested Images via Zero-Knowledge Proofs to Fight Disinformation. arXiv:2211.04775 [cs.CR] https://arxiv.org/abs/2211.04775

[42] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *Advances in Cryptology – CRYPTO 2022*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer Nature Switzerland, 359–388.

[43] Johannes Krupp, Dominique Schröder, Mark Simkin, Dario Fiore, Giuseppe Ateniese, and Stefan Nuernberger. 2016. Nearly Optimal Verifiable Data Streaming. In *Public-Key Cryptography – PKC 2016*, Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang (Eds.). Springer Berlin Heidelberg, 417–445.

[44] Pardeep Kumar, Yun Lin, Guangdong Bai, Andrew Paverd, Jin Song Dong, and Andrew Martin. 2019. Smart grid metering networks: A survey on security, privacy and open research issues. *IEEE Communications Surveys & Tutorials* 21, 3 (2019), 2886–2927.

[45] Alex Lawson. 2023. Households in Great Britain to be paid to use less electricity: how does it work? *The Guardian* (2023). https://www.theguardian.com/money/2023/jan/23/households-great-britain-paid-use-less-electricity-cut-bills-national-grid

[46] Do Le Quoc, Franz Gregor, Jatinder Singh, and Christof Fetzer. 2019. SGX-PySpark: Secure Distributed Data Analytics. In *The World Wide Web Conference (WWW '19)*. Association for Computing Machinery, 3564–3563.

[47] Eunsang Lee, Joon-Woo Lee, Jong-Seon No, and Young-Sik Kim. 2022. Minimax Approximation of Sign Function by Composite Polynomial for Homomorphic Comparison. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2022), 3711–3727.

[48] Jimmy Lin. 2017. The Lambda and the Kappa. *IEEE Internet Computing* 21, 5 (2017), 60–66.

[49] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. 2012. Software abstractions for trusted sensors. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*. Association for Computing Machinery, 365–378.

[50] Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig, and Yu Feng. 2023. Certifying Zero-Knowledge Circuits with Refinement Types. Cryptology ePrint Archive, Paper 2023/547. https://eprint.iacr.org/2023/547

[51] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. 2023. Pianist: Scalable zkRollups via Fully Distributed Zero-Knowledge Proofs. Cryptology ePrint Archive, Paper 2023/1271. https://eprint.iacr.org/2023/1271

[52] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 2023. GZKP: A GPU Accelerated Zero-Knowledge Proof System. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 340–353.

[53] Ashish Mahendru, Unmesh Deshmukh, and Sandeep Bishnoi. 2014. Smart Plug Monitoring Using Oracle Event Processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. Association for Computing Machinery, 302–305.

[54] Aman Mangal, Arun Mathew, Tanmay Randhavane, and Umesh Bellur. 2014. Predicting Power Needs in Smart Grids. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. Association for Computing Machinery, 298–301.

[55] James Manyika, Michael Chui, Peter Bisson, Jonathan Woetzel, Richard Dobbs, Jacques Bughin, and Dan Aharon. 2015. The Internet of Things: Mapping the value beyond the hype. (2015).

[56] Eoghan McKenna, Ian Richardson, and Murray Thomson. 2012. Smart meter data: Balancing consumer privacy concerns with legitimate applications. *Energy Policy* 41 (2012), 807–814.

[57] Francesca Meneghello, Matteo Calore, Daniel Zucchetto, Michele Polese, and Andrea Zanella. 2019. IoT: Internet of Threats? A Survey of Practical Security Vulnerabilities in Real IoT Devices. *IEEE Internet of Things Journal* 6, 5 (2019), 8182–8201.

[58] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. 2014. Authenticated data structures, generically. *ACM SIGPLAN Notices* 49, 1 (2014), 411–423.

[59] Assa Naveh and Eran Tromer. 2016. PhotoProof: Cryptographic Image Authentication for Any Set of Permissible Transformations. In *2016 IEEE Symposium on Security and Privacy (SP)*. 255–271. https://doi.org/10.1109/SP.2016.23

[60] Nataliia Neshenko, Elias Bou-Harb, Jorge Crichigno, Georges Kaddoum, and Nasir Ghani. 2019. Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations. *IEEE Communications Surveys & Tutorials* 21, 3 (2019), 2702–2733.

[61] Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. 2013. Streaming authenticated data structures. In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*. Springer, 353–370.

[62] Heejin Park, Shuang Zhai, Long Lu, and Felix Xiaozhu Lin. 2019. StreamBox-TZ: Secure stream analytics at the edge with TrustZone. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 537–554.

[63] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2016. Pinocchio: Nearly Practical Verifiable Computation. *Commun. ACM* 59, 2 (jan 2016), 103–112.

[64] Raluca Ada Popa. 2024. Confidential Computing or Cryptographic Computing? Tradeoffs between cryptography and hardware enclaves. *Queue* 22, 2 (may 2024), 108–132.

[65] Alex Pruden. 2022. Announcing the Inaugural ZPrize Competition Results. https://www.zprize.io/blog/announcing-zprize-results

[66] Gholamreza Ramezan and Ehsan Meamari. 2024. zk-IoT: Securing the Internet of Things with Zero-Knowledge Proofs on Blockchain Platforms. arXiv:2402.08322 [cs.CR]

[67] Karen Rose, Scott Eldridge, and Lyman Chapin. 2015. The Internet of Things (IoT): An Overview. (2015). https://www.internetsociety.org/resources/doc/2015/iot-overview/

[68] Michael Rosenberg, Tushar Mopuri, Hossein Hafezi, Ian Miers, and Pratyush Mishra. 2024. Hekaton: Horizontally-Scalable zkSNARKs via Proof Aggregation. Cryptology ePrint Archive, Paper 2024/1208. https://eprint.iacr.org/2024/1208

[69] Mark Russinovich, Cédric Fournet, Greg Zaverucha, Josh Benaloh, Brandon Murdoch, and Manuel Costa. 2024. Confidential Computing Proofs: An alternative to cryptographic zero-knowledge. *Queue* 22, 4 (sep 2024), 73–100.

[70] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. 2016. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics* 1 (2016), 145–164.

[71] Stefan Saroiu and Alec Wolman. 2010. I am a sensor, and I approve this message. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications (HotMobile '10)*. Association for Computing Machinery, 37–42.

[72] Dominique Schöder and Mark Simkin. 2015. VeriStream–a framework for verifiable data streaming. In *Financial Cryptography and Data Security: 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers 19*. Springer, 548–566.

[73] Dominique Schröder and Heike Schröder. 2012. Verifiable data streaming. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 953–964.

[74] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE symposium on security and privacy*. IEEE, 38–54.

[75] Julian James Stephen, Savvas Savvides, Vinaitheerthan Sundaram, Masoud Saeida Ardekani, and Patrick Eugster. 2016. STYX: Stream Processing with Trustworthy Cloud-based Execution. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. Association for Computing Machinery, 348–360.

[76] Abhinav Sunderrajan, Heiko Aydt, and Alois Knoll. 2014. Real Time Load Prediction and Outliers Detection Using STORM. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. Association for Computing Machinery, 294–297.

[77] Supranational. 2023. *blst*. https://github.com/supranational/blst/tree/3b28d0f

[78] Roberto Tamassia. 2003. Authenticated data structures. In *Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11*. Springer, 2–5.

[79] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. 2013. MrCrypt: static analysis for secure cloud computations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. Association for Computing Machinery, 271–286.

[80] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2002. *NEXMark – A Benchmark for Queries over Data Streams (draft)*. Technical Report. OGI School of Science & Engineering at OHSU.

[81] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association.

[82] Stephan Van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX fails in practice. (2020). https://sgaxe.com/files/SGAxe.pdf

[83] Michael Walfish and Andrew J. Blumberg. 2015. Verifying Computations without Reexecuting Them. *Commun. ACM* 58, 2 (jan 2015), 74–84.

[84] Jianghong Wei, Guohua Tian, Jun Shen, Xiaofeng Chen, and Willy Susilo. 2021. Optimal verifiable data streaming protocol with data auditing. In *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part II 26*. Springer, 296–312.

[85] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. 2018. {DIZK}: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security 18)*. 675–692.

[86] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. Association for Computing Machinery, 159–172.

[87] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 283–298.

[88] ZoKrates contributors. 2023. *ZoKrates*. https://github.com/Zokrates/ZoKrates/tree/c537a80