StreamBox-TZ: Secure Stream Analytics at the Edge with TrustZone

Heejin Park and Shuang Zhai, *Purdue ECE;* Long Lu, *Northeastern University;* Felix Xiaozhu Lin, *Purdue ECE*

https://www.usenix.org/conference/atc19/presentation/park-heejin

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

StreamBox-TZ: Secure Stream Analytics at the Edge with TrustZone

Heejin Park[1], Shuang Zhai[1], Long Lu[2], and Felix Xiaozhu Lin[1]

[1]Purdue ECE [2]Northeastern University

Abstract

While it is compelling to process large streams of IoT data on the cloud edge, doing so exposes the data to a sophisticated, vulnerable software stack on the edge and hence security threats. To this end, we advocate isolating the data and its computations in a trusted execution environment (TEE) on the edge, shielding them from the remaining edge software stack which we deem untrusted.

This approach faces two major challenges: (1) executing high-throughput, low-delay stream analytics in a single TEE, which is constrained by a low trusted computing base (TCB) and limited physical memory; (2) verifying execution of stream analytics as the execution involves untrusted software components on the edge. In response, we present StreamBox-TZ (SBT), a stream analytics engine for an edge platform that offers strong data security, verifiable results, and good performance. SBT contributes a data plane designed and optimized for a TEE based on ARM TrustZone. It supports continuous remote attestation for analytics correctness and result freshness while incurring low overhead. SBT only adds 42.5 KB executable to the TCB (16% of the entire TCB). On an octa core ARMv8 platform, it delivers the state-of-the-art performance by processing input events up to 140 MB/sec (12M events/sec) with sub-second delay. The overhead incurred by SBT's security mechanism is less than 25%.

1 Introduction

Many key applications of Internet of Things (IoT) process a large influx of sensor[1] data, i.e. telemetry. Smart grid aggregates power telemetry to detect supply/demand imbalance and power disturbances [76], where a power sensor is reported to produce up to 140 million samples per day [16,17]; oil producers monitor pump pressure, tank status, and fluid temperatures to determine if wells work at ideal operating points [55,60], where an oil rig is reported to produce 1–2 TB of data per day [43]; manufacturers continuously monitor vibration and ultrasonic energy of industrial equipment for detecting equipment anomaly and predictive maintenance [104,120], where a monitored machine is reported to generate PBs of data in a few days [77].

[1]Recognizing that IoT data sources range from small sensors to large equipment, we refer to them all as *sensors* for brevity.

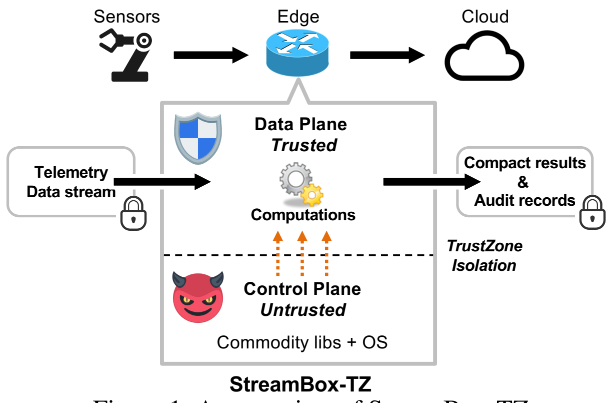


Figure 1: An overview of StreamBox-TZ

The large telemetry data streams must be processed in time. The high cost and long delay in transmitting data necessitate edge processing [98, 100]: sensors send the data to nearby gateways dubbed "cloud edge"; the edge runs a pipeline of continuous computations to cleanse and summarize the telemetry data and reports the results to cloud servers for deeper analysis. Edge hardware is often optimized for cost and efficiency. According to a 2018 survey [45], modern ARM machines are typical choices for edge platforms. Such a platform often has 2–8 CPU cores and several GB DRAM.

Unfortunately, edge processing exposes IoT data to high security threats. i) Deployed in the wild, the edge suffers from common IoT weaknesses, including lack of professional supervision [58, 118], weak configurations [108, 117], and long delays in receiving security updates [58, 114]. ii) On the edge, the IoT data flows through a set of sophisticated components that expose a wide attack surface. These components include a commodity OS (e.g. Linux or Windows), a variety of user libraries, and a runtime framework called *stream analytics engine* [37,42,83]. They reuse much code developed for servers and workstations. Their exploitable misconfigurations [121] and vulnerabilities [23, 35, 109] are not uncommon. iii) With data aggregated from multiple sources, the edge is a high-value target to adversaries. For these reasons, edge is even more vulnerable than sensors, which run much simpler software with narrower attack surfaces. Once attackers compromise the edge, they not only access confidential data but also may delete or fabricate data sent to the cloud, threatening the integrity of an entire IoT deployment.

Towards secure stream analytics on an edge platform, our goal is to safeguard IoT data confidentiality and integrity, support verifiable results, and ensure high throughput with low output delay. Following the principle of least privilege [95], we protect the analytics data and computations in a trusted execution environment (TEE) and limit their interface; we leave out the remaining edge software stack which we deem untrusted. By doing so, we shrink the trusted computing base (TCB) to only the protected functionalities, the TEE, and the hardware. We hence significantly enhance data security.

We face three challenges: i) what functionalities should be protected in TEE and behind what interfaces? ii) how to execute stream analytics on a TEE's low TCB and limited physical memory while still delivering high throughput and low delay? iii) as both trusted and untrusted edge components participate in stream analytics, how to verify the outcome?

Existing solutions are inadequate: pulling entire stream analytics engines to TEE [22, 27, 112] would result in a large TCB with a wide attack surface; the systems securing distributed operators [53, 99, 124] often lack stream semantics or optimizations for efficient execution in a single TEE, which are crucial to the edge; only attesting TEE integrity [65] or data lineages [50, 99, 102, 124] is inadequate for verifying stream analytics. We will show more evidences in the paper.

Our response is StreamBox-TZ (SBT), a secure engine for analyzing telemetry data streams. As shown in Figure 1, SBT builds on ARM TrustZone [2] on an edge platform. SBT contributes the following notable designs:

(1) Architecting a data plane for protection SBT provides a data plane exposing narrow, shared-nothing interfaces to untrusted software. SBT's data plane encloses i) all the analytics data; ii) a new library of low-level stream algorithms called *trusted primitives* as the only allowed computations on the data; iii) key runtime functions, including memory management and cache-coherent parallel execution of trusted primitives. SBT leaves thread scheduling and synchronization out of TEE.

(2) Optimizing data plane performance within a TEE In contrast to many TEE-oblivious stream engines that operate numerous small objects, hash tables, and generic memory allocators [32, 82, 122], SBT embraces unconventional design decisions for its data plane. i) SBT implements trusted primitives with array-based algorithms and contributes new optimizations with handwritten ARMv8 vector instructions. ii) To process high-velocity data in TEE, SBT provides a new abstraction called uArrays, which are contiguous, virtually unbounded buffers for encapsulating all the analytics data; SBT backs uArrays with on-demand paging in TEE and manages uArrays with a specialized allocator. The allocator leverages hints from untrusted software for compacting memory layout. iii) SBT exploits TrustZone's lesser-explored hardware features: ingesting data straightly through trusted IO without a detour through the untrusted OS; avoiding relocating streaming data by leveraging the large virtual address space dedicated to a TEE.

(3) Verifying edge analytics execution SBT supports cloud verifiers to attest analytics *correctness*, result *freshness*, and the untrusted hints received during execution. SBT captures coarse-grained dataflows and generates audit records. A cloud verifier replays the audit records for attestation. To minimize overhead in the edge-cloud uplink bandwidth, SBT compresses the records with domain-specific encoding.

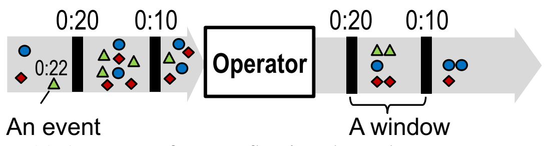
Our implementation of SBT supports a generic stream model [1] with a broad arsenal of stream operators. The TCB of SBT contains as little as 267.5 KB of executable code, of which SBT only constitutes 16%. On an octa core ARMv8 platform, SBT processes up to 12M events (144 MB) per second at sub-second output delays. Its throughput on this platform is an order of magnitude higher than an SGX-based secure stream engine running on a small x86 cluster with richer hardware resources [53]. The security mechanisms contributed by SBT incur less than 25% throughput loss with the same output delay; decrypting ingress data, when needed, incurs 4%–35% throughput loss with the same output delay. While sustaining high throughput, SBT uses up to 130 MB of physical memory in most benchmarks.

The key contributions of SBT are: i) a stream engine architecture with strongly isolated data and a lean TCB; ii) a data plane built from the ground up with computations and memory management optimized for a single TrustZone-based TEE; iii) remote attestation for stream analytics on the edge with domain-specific compression of audit records. To our knowledge, SBT is the first system designed and optimized for data-intensive, parallel computations inside ARM TrustZone. Beyond stream analytics, the SBT architecture should help secure other important analytics on the edge, e.g. machine learning inference. The SBT source can be found at `http://xsel.rocks/p/streambox`.

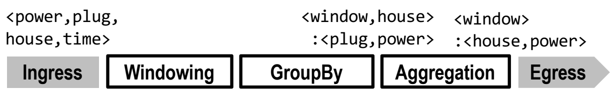
2 Background & Motivation

2.1 ARM for Cloud Edge

As typical hardware for IoT gateways [45], recent ARM platforms offer competitive performance at low power, suiting edge well. Most modern ARM cores are equipped with TrustZone [2], a security extension for TEE enforcement. TrustZone logically partitions a platform's hardware resources, e.g. DRAM and IO, into a normal (insecure) and a secure world. CPU cores independently switch between two worlds. A TEE



(a) A stream of events flowing through an operator.



(b) A simple analytics pipeline that predicts power grid loads

```
/* 1. Declare operators */
Ingress in(/* config info */);
Window w(1_SECOND); GroupBy<house> gb;
Aggregation<house,win> ag; Egress out;
/* 2. Create a pipeline. Connect operators */
Pipeline p; p.apply(in);
in.connect(w).connect(gb)
  .connect(ag).connect(out);
/* 3. Execute the pipeline */
Runner r( /* config */ ); r.run(p);
```

(c) Simplified pseudo code declaring the above pipeline

Figure 2: Example stream data, operators, and a pipeline

atop TrustZone owns dedicated, *trusted IO*, a unique feature that other TEE technologies such as Intel SGX [81] lack.

Trusted IO is a unique feature of ARM TrustZone, implemented through hardware components including TrustZone Address Space Controller (TZASC) and TrustZone Protection Controller (TZPC). TZASC allows privilege software to logically partition DRAM between the normal and the secure worlds. Similarly, TZPC allows to configure IO peripherals accessible to either world. Any peripheral owned by the secure world is completely enclosed in the secure world. We use trusted IO to support the trusted source-edge links on the cloud edge (§3.1).

2.2 Stream Analytics

Stream Model We target stream analytics over sensor data. A data stream consists of sensor events that carry timestamps defined by event occurrence, as illustrated in Figure 2(a). Programmers specify a pipeline of continuous computations called *operators*, e.g. `Select` and `GroupBy`, that are extensively used for telemetry analytics [62, 90]. As data arrives at the edge, a stream analytics engine ingests the data at the pipeline ingress, pushes the data through the pipeline, and externalizes the results at the pipeline egress.

We follow a generic stream model [14, 32, 69, 85, 122]. Operators execute on event-time scopes called *windows*. Data sources emit special events called *watermarks*. A watermark guarantees no subsequent events in the stream will have event times earlier than the watermark timestamp. A pipeline's *output delay* is defined as the elapsed time starting from the moment the ingress receives the watermark signaling the completion of the current window to the moment the egress externalizes the window results [82]. A pipeline may maintain its internal states organized by windows at different operators. See prior work [20] for a formal stream model.

Analytics example: Power load prediction Figure 2(b-c) shows an example derived from an IoT scenario [62]: it predicts future household power loads based on power loads reported by smart power plugs. The example pipeline ingests a stream of power samples and groups them by 1-second fixed windows and by houses. For each house in each window, it aggregates all the loads and predicts the next-window load as an exponentially weighted moving average over the recent windows. At the egress, the pipeline emits a stream of per-house load prediction for each window.

Stream analytics engines Stream pipelines are executed by a runtime framework called a stream analytics engine [37, 42, 46, 82, 83, 90]. A stream analytics engine consists of two types of function: *data functions* for data move and computations; *control functions* for resource management and computation orchestration, e.g. creating and scheduling tasks. The boundary between the two is often blurry. To amortize overheads, control functions often organize data in *batches* and invoke data functions to operate on the batches.

2.3 Security Threats & Design Objectives

The edge faces common threats in IoT deployment. i) IT expertise is weak. Edge platforms are likely managed by field experts [58, 114, 118] rather than IT experts. Such lack of professional supervision is known to result in weak configurations [108, 117]. ii) The infrastructure is weak. Deployed in the field, the edge often sees slow uplinks [84, 114] and hence much delayed software security updates. For cost saving, edge analytics may need to share OS and hardware with other high-risk, untrusted software such as web browsers [114].

Besides the common threats, existing edge software stacks entrust IoT data with commodity OSes, analytics engines, and language runtimes (e.g. JVM). However, these components are incapable of offering strong security guarantees due to their complexity and wide interfaces. Each of them easily contains more than several hundreds of KSLoC [116]. Exploitable vulnerabilities are constantly discovered [3, 6, 23, 35, 38], making these components untrusted in recent research [36, 54, 79, 80]. By exploiting these vulnerabilities, a local adversary as an edge user program may compromise the kernel through the wide user/kernel interfaces [11, 12] or attack an analytics engine through IPC [7]; a remote adversary, through the edge's network services, may compromise analytics engines [4] or the OS [10]. A successful adversary may expose IoT data, corrupt the data, or covertly manipulate the data. Taking the application in Figure 2(b) as an example, the adversary gains access to the smart plug readings, which may contain residents' private information, and injects fabricated data.

Objectives We aim three objectives for stream analytics over telemetry data on an edge platform: i) confidentiality and integrity of IoT data, raw or derived; ii) verifiable correctness and freshness of the analytics results; iii) modest security overhead and good performance.

3 Security Approach Overview

3.1 Scope

IoT scenarios We target an edge platform that captures and analyzes telemetry data. We recognize the significance of mission-critical IoT with tight control loops, but do not target it. Our target scenario includes source sensors, edge platforms, and a cloud server which we dub "cloud consumer". All the raw IoT data and analytics results are owned by one party. The sensors produce trusted events, e.g. by using secure sensing techniques [49, 73, 97]. The cloud consumer is trusted; it installs analytics pipelines to the edge and consumes the results uploaded from the edge. We consider *untrusted* source-edge links (e.g. public networks) which requires data encryption by the source, as well as *trusted* source-edge links (e.g. direct IO bus or on-premise local networks), and will evaluate the corresponding designs (§9). We assume untrusted edge-cloud links, which require encryption of the uploaded data.

In-scope Threats We consider malicious adversaries interested in learning IoT data, tampering with edge processing outcome, or obstructing processing progress. We assume powerful adversaries: by exploiting weak configurations or bugs in the edge software, they already control the entire OS and all applications on the edge.

Out-of-scope Threats We do not protect the confidentiality of stream pipelines, in the interest of including only low-level compute primitives in a lean TCB. We do not defend the following attacks. i) Attacks to non-edge components assumed trusted above, e.g. sensors [111]. ii) Exploitation of TEE kernel bugs [8, 9, 56]. iii) Side channel attacks: by observing hardware usage outside TEE, adversaries may learn the properties of protected data, e.g. key skew [72]. Note that controlled-channel attack [119] cannot be applied to ARM TrustZone as it has separate page management within a separate secure OS unlike Intel SGX. iv) Physical attacks, e.g. sniffing TEE's DRAM access [18, 28]. Many of these attacks are mitigated by prior work [39, 66, 123, 124] orthogonal to SBT.

Note that TEE code authenticity and integrity are already ensured by the TrustZone hardware, i.e. only code trusted by the device vendor can run in TrustZone and its integrity is protected by TrustZone.

3.2 Approach and Security Benefits

As shown in Figure 3, SBT protects its data functions in a trusted *data plane* in TEE. SBT runs its untrusted *control plane* in the normal world. The control plane invokes the data plane through narrow, shared-nothing interfaces. The engine's

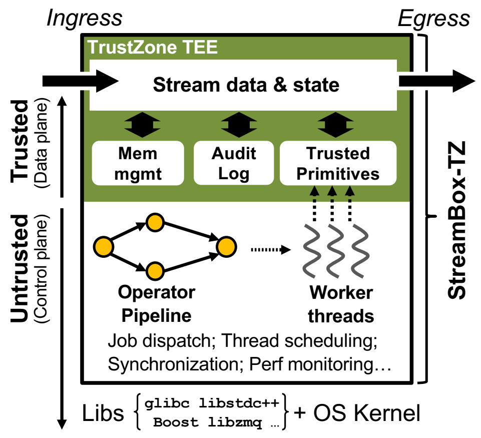


Figure 3: StreamBox-TZ on an edge platform with ARM TrustZone. Bold arrows show the protected data path.

TCB thus only consists of the TEE (including the data plane) and the hardware.

Streaming data always flows in TEE. The data plane ingests the data through TrustZone's trusted IO. After ingestion, it returns *opaque references* of the data batches to the control plane. In turn, the control plane requests computations on the protected data by invoking the data plane with the opaque references. The data plane generates opaque references as long, random integers. It tracks all live opaque references, validates incoming opaque references, and only accepts ones that exist. At the pipeline egress, the data plane encrypts, signs, and sends the result to the cloud.

The analytics execution is continuously attested. SBT captures complete and deterministic dataflows of the stream analytics as well as execution timing, and periodically reports to the cloud server. The cloud server verifies if all ingested data is processed according to the pipeline (correctness), and if the edge incurs low delay (freshness).

Thwarted attacks SBT defeats the following attacks. ***i) Breaking IoT data confidentiality or integrity.*** As the raw and derived data enters and leaves the edge TEE through trusted IOs, adversaries on the edge cannot touch, drop, or inject data. When the data is off the edge transmitted over untrusted networks, it is protected by encryption against network-level adversaries. ii) ***Breaking the data plane integrity.*** Any fabricated opaque reference passed to the data plane will be rejected, since all opaque references are validated before use. Through the data plane's interface, an adversary may exploit bugs in the data plane and compromise it. By minimizing the date plane codebase and hardening its interface, SBT substantially reduces the data plane's attack surface and potential bugs that can be exploited. iii) ***Breaking analytics correctness.*** A compromised control plane may request computations deviating from pipeline declarations or the stream model. For instance, it may invoke trusted computations on partial data, wrong windows, or valid but undesirable opaque references. SBT defeats these attacks through attestation: since the cloud verifier possesses complete knowledge on ingested data and

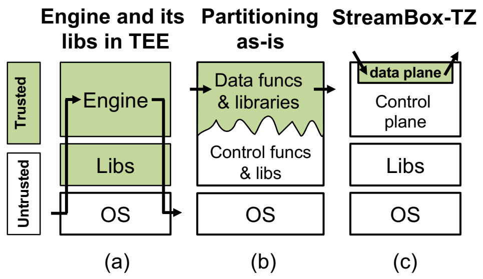


Figure 4: Among alternative architectures for secure stream analytics, StreamBox-TZ (c) leads to the smallest TCB and the most optimized data plane. Arrows indicate data flows.

System	TEE	Analytics	SG	Compute in TEE	Memory	Attestation
VC3 [99]	SGX	Batch	`CIVA-`	Mapper/reducer	Heap	Data lineage
Opaque [124]	SGX	Batch	`CIVAO`	Query plans	unreported	Data lineage
EnclaveDB [91]	SGX	Batch	`CI-A-`	Pre-compiled queries	unreported	TEE integrity
SafeBricks [89]	SGX	Pkt proc.	`CI-A-`	Net func. operators*	unreported	TEE integrity
SecureStream [53]	SGX	Stream	`CI---`	Lua programs	unreported	TEE integrity
StreamBox-TZ	TZ	Stream	`CIV--`	Vectorized primitives*	uArray	Log replay

SG: security guarantees.
`C`: data confidentiality; `I`: data integrity; `V`: verifiability; `A`: analytics confidentiality; `O`: obliviousness
* TEE encloses only low-level computations; otherwise TEE encloses whole analytics.

Table 1: Comparison to existing secure processing systems

Trusted Primitives	Popular Spark Streaming Operators
Sort, Merge, Segment, SumCnt, TopK, Concat, Join, Count, Sum, Unique, FileterBand, Median, ...	GroupByKey, Windowing, AvgPerKey, Distinct, SumByKey, AggregateByKey, SortByKey, TopKPerKey, CountByKey, CountByWindow, Filter, MedianByKey, TempJoin, Union, ...

Table 2: Selected trusted primitives (23 in total) and operators they constitute. These operators cover most listed in the Spark Streaming documentation [103].

pipelines, it detects such correctness violation and rejects the edge analytics results. ***iv) Attacks on analytics performance or availability.*** A compromised control plane may delay or pause invoking of trusted computations, violating the freshness guarantee. As the execution timing of trusted computations is attested, the cloud verifier detects the attacks and can choose to prompt further investigation. ***v) Attempting to trigger data race or deadlock.*** By design, data race and deadlocks will never happen inside the data plane: the trusted computations do not share state concurrently and all locking happens outside of the TEE.

4 Design Overview

4.1 Challenges

Our approach raises three challenges. i) ***Architecting the engine with a proper protection boundary***. This hinges on a key trade-off among TEE functional richness, overhead of TEE entry/exit, and TCB size. ii) ***Optimizing data functions within a TEE***. Processing of high-velocity data in a TEE strongly favors simple algorithms and compact memory. Yet, existing stream engines often operate numerous short-lived objects indexed in hash tables or trees [32, 69, 82, 90, 122], e.g. for grouping events by key. They manage these objects with generic memory allocators [82] or garbage collectors [87, 122]. Such designs poorly fit a TEE's small TCB and limited DRAM portion, e.g. typically tens of MB for TrustZone TEE and up to 128 MB for Intel SGX enclave [31]. iii) ***Verifying stream analytics results***. This requires to track unbounded data flows in stream pipelines, validate if operators respect the temporal properties, e.g. windows, and minimize the resultant overhead in execution and communication.

Why are existing systems inadequate? First, many TEE-based systems [22, 27, 112] pull entire user applications and libraries to the TCB, as shown in Figure 4(a). However, as we described in Section 2.2, a modern analytics engine and its libraries are large, complex, and potentially vulnerable. Second, partitioning applications to suit a TEE, as shown in Figure 4(b) [71, 93, 101], is unsuitable for existing stream engines: partitioning does not change their hash-based data structures and algorithms, which by design mismatch a TEE. Similarly, recent secure processing engines disfavor partitioning [89, 91]. Third, recent systems use TEE to protect data in analytics or in network packet processing. As summarized in Table 1, they lack support for stream analytics, key computation optimizations, or specialized memory allocation, which we will demonstrate as vital to our objective.

Attesting TEE integrity [65, 91] is insufficient to assert analytics correctness. VC3 [99] and Opaque [124] verify correctness of *batch* analytics by checking the history of compute results, i.e. their data lineage [50, 102]. Without tracking data being continuously ingested and lacking a stream model, data lineages cannot assert whether *all* ingested data is processed according to pipeline declarations, watermarks, and temporal windows, which are critical to stream analytics.

4.2 StreamBox-TZ in a Nutshell

SBT builds on TrustZone [2] due to ARM's popularity for the edge and trusted IO benefiting stream analytics (§2).

Programmability Programming SBT is similar to programming commodity engines such as Spark Streaming [122] and Flink [19]. Analytics programmers assemble pipelines with high-level, declarative operators as exemplified in Figure 2(c). SBT provides most of the common operators offered by commodity engines, as summarized in Table 2. These stream operators are widely used for analytics over telemetry data [62, 90]. SBT supports User Defined Functions (UDFs) that are certified by a trusted party, which is a common requirement in TEE-based systems [91].

SBT architecture As shown in Figure 3, SBT's data plane incarnates as a TrustZone module. SBT runs its control plane as a parallel runtime in the normal world. The control plane invokes the data plane through a narrow interface (details in Section 9). The control plane orchestrates the execution of analytics pipelines. It creates plentiful parallelism among and within operators. It elastically maps the parallelism to a pool of threads it maintains. At a given moment, all threads may simultaneously execute one operator as well as different operators over different data.

Data plane & design choices SBT's data plane consists of only the trusted primitives and a runtime for them.

i) Trusted primitives are stateless, single-threaded functions that are oblivious to synchronization. We do not enclose whole stream pipelines in the data plane, because a stream pipeline must be scheduled dynamically for parallelism and handling high-velocity data. We do not enclose whole declarative operators in the data plane, because one operator instance has internal thread-level parallelism and hence requires thread management logic. Our choice keeps the data plane lean, leaving out all control functions including scheduling and threading. This contrasts to many other engines pulling whole analytics to TEE as shown in Table 1.

Although exporting low-level primitives entails more TEE switches, the costs are lower on modern ARM [25, 56] and can be amortized by data batching, as will be discussed soon.

ii) The data plane incorporates minimum runtime functions: memory management and paging, which are critical to TEE integrity; cache coherence of parallel primitives, which is critical to parallelism. The data plane is agnostic to declarative operators and pipelines being executed.

For attestation, the data plane generates audit records on data ingress/egress, watermarks, and primitive executions. It reduces overhead via data batching and record compression.

Coping with secure memory shortage When compute cost or data ingestion rate is high, SBT may run short of secure memory. To avoid data loss in such a situation, SBT adds backpressure to source sensors, slowing down data ingestion. In the current implementation, SBT triggers backpressure when ingestion exceeds a user-defined threshold; we leave as future work automatic flow control, i.e. tuning the threshold online per available secure memory and backlog.

5 Trusted Primitives and Optimizations

Parallel execution inside a TEE SBT exploits task parallelism without bloating the TEE with a threading library. The control plane invokes multiple primitives from multiple worker threads, which then enter the TEE to execute the primitives in parallel. All trusted primitives share one cache-coherent memory address space in TEE, which simplifies data sharing and avoids copy cost. This contrasts to existing secure analytics engines that leave task parallelism untapped in a single TEE [53, 99].

Array-based algorithms to suit TEE Unlike many popular stream engines using hash-based algorithms for lower algorithmic complexity, we make a new design decision. We strongly favor algorithms with simple logic and low memory overhead, despite that they may incur higher algorithmic complexity. Corresponding to contiguous arrays as the universal data containers in TEE, most primitives use sequential-access algorithms over contiguous arrays, e.g. executing Merge-Sort over event arrays and scanning the resultant array to calculate the average value per key.

Trusted primitives and vectorization SBT's trusted primitives are generic. They constitute most declarative stream operators, often referred to as Select-Projection-Join-GroupBy (SPJG) families, shown in Table 2. These operators are considered representative in prior research [44].

To speed up the array-based algorithms inside TEE without TCB bloat, our insight is to map their internal data parallelism to vector instructions of ARM [21]. Despite their well-known performance benefit, vector instructions are rarely used to accelerate data analytics *within* TEEs, to our knowledge. Vectorization incurs low code complexity as the performance gain comes from a CPU feature that is already part of the TCB.

Our optimization focuses on Sort and Merge, two core primitives that dominate the execution of stream analytics according to our observation. Inspired by vectorized sort and merge on x86 [26,64], we build new implementations for SBT by hand-writing ARMv8 NEON vector instructions. Our sort outperforms the ones in the C/C++ standard libraries by more than $2\times$, as will be shown in evaluation. This optimization is crucial to the overall engine performance.

6 TEE Memory Management

Facing high-velocity streams in a TEE, SBT's memory allocator addresses two challenges: *space efficiency*: it must create compact memory layout and reclaim memory timely due to limited physical memory; *lightweight*: the allocator must be simple to suit a low TCB. The challenges disqualify popular engines that organize events in hash tables (e.g. for grouping events by key) and rely on generic memory allocators [32, 69, 82, 90, 122]. The reasons are two: a hash table's principle of trading space for time mismatches TEE's limited memory; generic allocators often feature sophisticated optimizations, adding tens of KSLoC to TCB [41, 59].

SBT specializes memory management for stream computations: it supports unbounded buffers as the universal memory abstraction (§6.1); it places data by using (untrusted) consumption hints and large virtual address space (§6.2).

6.1 Unbounded Array

We devise contiguous, virtually unbounded arrays called *uArrays*, a new abstraction as the universal data containers used by computations in TEE. uArrays encapsulate all the data in a pipeline, including data flowing among trusted primitives as well as operator states traditionally kept in hash tables.

An uArray is an append-only buffer in a contiguous memory region for same-type data objects. Their lifecycles closely map to the producer/consumer pattern in streaming computations. One uArray can be in three states. *Open*: after an uArray is created, it dynamically grows as the producer primitive appends data objects to it. *Produced*: the data production completes and the end position of the uArray is finalized. uArray becomes read-only and no data can be appended. *Retired*: the uArray is no longer needed and its memory is subject to reclamation. The memory allocator places and reclaims uArrays regarding their states, as will be discussed in Section 6.2.

Types uArrays fall into different types depending on their scopes and enclosed data. A *streaming uArray* encapsulates data flowing from a producer primitive to a consumer primitive. A *state uArray* encapsulates operator state that outlives the lifespans of individual primitives. A *temporary uArray* live within a trusted primitive's scope.

Low abstraction overhead An uArray spans a contiguous virtual memory region and grows transparently. The growth is backed by the data plane's on-demand paging that completely happens in the TEE. For most of the time, growing an uArray only requires updating an integer index. Compared to manually managed buffers, this mechanism waives bounds checking of uArray in computation code and hence allows the compiler to generate more compact loops. uArrays always grow in place. This contrasts to common sequence containers (e.g. C++ `std::vector` and `java.util.ArrayList`) that grow transparently but require expensive relocation. We will experimentally compare uArray with `std::vector` in Section 9.

6.2 Placing uArrays in uGroups

Co-locating uArrays The memory allocator co-locates multiple uArrays as a uGroup in order to reclaim them consecutively. Spanning a contiguous virtual memory region, a uGroup consists of multiple *produced* or *retired* uArrays and optionally an *open* uArray at its end, as shown in Figure 5. The grouping is purely physical: it is at the discretion of the allocator, orthogonal to stream computations, and therefore transparent to the trusted primitives and the control plane.

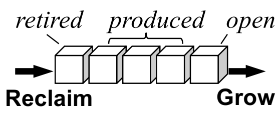


Figure 5: The uArrays in one uGroup

With the grouping, the allocator reclaims *consumed* uArrays by always starting from the beginning of an uGroup, as shown in Figure 5. To place a new uArray, the allocator decides whether to create a new uGroup for the uArray, or append the uArray to an existing uGroup. In doing so, the allocator seeks to i) ensure that each uGroup holds a sequence of uArrays to be consumed consecutively in the future; ii) minimize the total number of live uGroups, in order to compact TEE memory layout and minimizes the cost in tracking uGroups. To this end, our key is to guide placement with the control plane's data consumption plan, as will be presented below.

Consumption hints Upon invoking a trusted primitive T, the control plane may provide two optional hints concerning the future consumption order for the output of T:

- *Consumed-in-parallel* ($\|_k$): the control plane will schedule k worker threads to consume a set of uArrays in parallel.
- *Consumed-after* ($b_1 \Leftarrow b_2$): the control plane will schedule worker threads for consuming uArray b_2 after uArray b_1. The *consumed-after* relation is transitive. uArrays may form multiple *consumed-after* chains.

The control plane may specify these relations between new output uArrays (yet to be created) and existing uArrays.

Hint-guided placement The hints assist the data plane to generate compact memory layout and reclaim memory effectively. Upon allocating a uArray, the allocator examines the existing hints regarding to the uArray.

($\Leftarrow$) prompts the allocator to place the uArrays on the same *consumed-after* chain in the same uGroup. Starting from the new uArray b under question, the allocator tracks back on its consumed-after chain, and places b after the first uArray that is both in state *produced* (i.e. its growth has finished) and is located at the end of an uGroup. If no such uArray is available on the chain, the allocator creates a new uGroup for b.

($\|_k$) prompts the allocator to place uArrays $b_{1..k}$ in separate uGroups, so that delay in consuming any of the uArrays will not block the allocator from reclaiming the other uArrays. Our rationale is that despite $b_{1..k}$ are created at the same time, they are often consumed at different moments in the future: i) since SBT's control plane threads independently fetch new uArrays for processing as they become available (§4), the starting moments for processing $b_{1..k}$ may vary widely, especially when the engine load is high; ii) even when k worker threads start processing $b_{1..k}$ simultaneously, straggling workers are not uncommon, due to non-determinism of a modern multicore's thread scheduling and memory hierarchy [24].

The impacts of misleading hints SBT detects misleading hints in retrospect through remote attestation (§7). As the hints only affect TEE memory placement *policy* on the edge, misleading hints never result in data loss (§4.2) or violation of data security and TEE integrity. Yet, such hints may slow down analytics and therefore violate result freshness.

Managing virtual addresses All uGroups grow *in place* within one virtual address space. To avoid collision and expensive relocation, the allocator places them far apart by leveraging the large virtual address space dedicated to a TrustZone TEE. The space is 256TB on ARMv8, 10,000× larger than the physical DRAM (a few GBs). Hence, the allocator simply reserves for each uGroup a virtual address range as large as the total TEE DRAM. We will validate this choice in Section 9.

7 Attestation for Correctness and Freshness

SBT collects evidences for cloud consumers to verify two properties: *correctness*, i.e. all ingested data is processed ac-

Field	Description	Length
Ts	Data plane timestamp	32 bits
Op	Primitive type, including ingress/egress	16 bits
WinNo	Monotonic window sequence number	16 bits
Data	An uArray ID or a watermark value	32 bits
Hint	An optional consumption hint	64 bits
Count	Number of data/hint fields that follow	16 bits

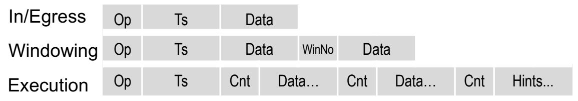


Figure 6: Audit records: fields (top) and layout (bottom)

cording to the stream pipeline declaration; *freshness*, i.e. the pipeline has low output delays.

The above objective has several notable aspects. i) We verify the behaviors of untrusted control plane, i.e., *which* primitives it invokes on *what* data and at *what* time. We do not verify trusted primitives, e.g. if a Sort primitive indeed produces ordered data. ii) Verifying data lineages at the pipeline's intermediate operators or egress [50, 102] is insufficient to guarantee correctness, i.e. all data ingested so far is processed according to the stream pipeline. iii) The windows of stream computations and watermarks triggering the computations must be attested, which are keys to stream model (§2). iv) As the volume of evidences can be substantial, evidences must be compacted to save uplink bandwidth [84, 114].

Therefore, SBT provides the following verification mechanism. Agnostic to the pipeline being executed, the data plane monitors dataflows among primitive instances at the TEE boundary, and then generates audit records. For low overhead, it eschews building data lineages on-the-fly unlike much prior work [50,74,99]. The data plane compresses audit records and flushes to the cloud both periodically and upon externalizing any analytics result. We describe details below.

Audit records As being invoked by the control plane, the data plane generates *audit records*. As illustrated in Figure 6, the records track i) ingested and externalized uArrays, ii) associations between uArrays and windows, and iii) primitive executions (with optional hints supplied by the control plane) which establish *derived-from* relations among uArrays. The records further include ingested watermark values, which are crucial for determining output delays as will be discussed below. The data plane timestamps all the records. It generates monotonically increasing identifiers for recorded uArrays. We will evaluate the overhead of audit records in Section 9.

Attesting analytics correctness The cloud verifier checks if all ingested uArrays flow through the expected trusted primitives. Such dataflows are deterministic given the arrivals of input data (including their windows), the watermarks, and the pipeline declaration. Hence, the verifier replays all ingestion records on its local copy of the same pipeline. It checks if all the records resulting from the replay match the ones reported by the edge (except timestamps). The replay is symbolic without actual computations and hence fast.

Note that the verification works for stateful operators as well. The state of a stream operator (e.g. temporal join) is only determined by all the inputs the operator has ever received. Since the cloud can verify that all the ingested uArrays correctly flow through the expected trusted primitives and thus stream operators, it knows that the operator's current state must be correct, and then all results derived from the operator state must be correct.

Attesting result freshness The key for the verifier to calculate the delay of an output result R is to identify the watermark that triggers the externalization of R, according to the delay definition in Section 2.2. From the egress record of R, the verifier traces *backward* following the *derived-from* chain(s) until it reaches an execution record indicating that a watermark W triggers the execution. The verifier looks up the ingress record of W. It calculates the difference between W's ingress time and R's egress time to be the delay of R.

Example In Listing 1, an uArray with identifier 0xF0 is ingested and segmented into two uArrays (0xF1 and 0xF2) for window 0 and 1 respectively. Sort consumes uArray 0xF1 and produces uArray 0xF3. A watermark with value 100 arrives and completes window 0. Triggered by the watermark, SUM consumes uArray 0xF3 of window 0 and produces uArray 0xF5 as the result of window 0.

```
ts= 1  INGRESS data=0xF0
ts= 5  WND data_in=0xF0 win_no=0 data_out=0xF1
ts=10  SORT data_in=0xF1 data_out=0xF3
ts=15  INGRESS data=0xF4 (watermark=100)
ts=25  SUM data_in=0xF3,0xF4 data_out=0xF5
ts=28  WND data_in=0xF0 win_no=1 data_out=0xF6
ts=30  EGRESS data=0xF5
```

Listing 1: Sample audit records for the pipeline in Figure 2. Format is simplified. ts means processing timestamp.

The cloud verifier replays the ingress records on its local pipeline copy and learns that uArray 0xF1 is processed adhering to the pipeline declaration while uArray 0xF2 is yet to be processed. It will assert analytics incorrectness if 0xF2 remains unprocessed until a future watermark completes window 1 (not shown). To verify result freshness, the verifier traces result 0xF5 backward to find its trigger watermark 0xF4 and calculates the output delay to be 15 (30 − 15).

Columnar compression of records The data plane compresses audit records by exploiting locality within one record field and known data distribution in each field. The data plane produces raw audit records in memory (with the format shown in Figure 6) and in a row order, i.e. one record after the other. Before uploading a sequence of records, it separates the record fields (i.e. columns) and applies different encoding schemes to individual columns: i) Huffman encoding for primitive types and data counts, the two columns likely contain skewed values; ii) delta encoding for timestamps, uArray identifiers, and window numbers, which increment monotonically. Our compression is inspired by columnar databases [107]. We will evaluate the efficacy of compression in Section 9.

8 Implementation

We build SBT for ARMv8 and atop OP-TEE [70] (v2.3). SBT reuses most control functions of StreamBox [82], an open-source research stream engine for x86 servers. Yet, as StreamBox mismatches a TEE (§4.1), SBT contributes a new architecture and a new data plane. SBT communicates with source sensors and cloud consumers over ZeroMQ TCP transport [57] which is known for good performance. The new implementation of SBT includes 12.4K SLoC.

Input batch size, a key parameter of SBT, trades off between delays in executing individual primitives, the rate of TEE entry/exit, and attestation cost. We empirically determine it as 100K events and will evaluate its impact (§9). **Opaque references** for uArrays are 64-bit random integers generated by the data plane. It keeps the mappings from references to uArray addresses in a table, and validates opaque references by table lookup. This incurs minor overhead, as live opaque references are often no more than a few thousands.

9 Evaluation

We answer the following questions through evaluation:

- Does SBT result in a small TCB? (§9.1)
- What is SBT's performance and how is it compared to other engines? What is the overhead? (§9.2)
- How do our key designs impact performance (§9.3)?

9.1 TCB Analysis

TCB size Table 4 shows a breakdown of the SBT source code. Despite a sophisticated control plane, the data plane only adds 5K SLoC to the TCB. SBT's memory management is in 740 SLoC, 9× fewer than glibc's `malloc` and 20× fewer than `jemalloc` [41]. The size of data plane is 42.5 KB, a small fraction (16%) of the entire OP-TEE binary.

TCB interface The SBT's data plane exports only four entry functions: two for data plane initialization/finalization, one for debugging, and one shared by all 23 trusted primitives. The last function accepts and returns opaque references (§4). No state is shared across the protection boundary.

Comparison with alternative TCBs Compared to enclosing whole applications in TCB [22, 27, 112], SBT keeps most of the engine out, shrinking the TCB by at least one order of magnitude. Compared to directly carving out [71,93] the original StreamBox's data functions for protection, SBT completely avoids sophisticated data structures (e.g. AtomicHashMap [47] used by StreamBox) that mismatch TCB. Compared to VC3 [99] that implements Map/Reduce operators in a TCB with ∼9K SLoC, SBT supports much richer stream operators within a 2× smaller TCB.

SoC HiSilicon Kirin 620, TDP 36W		**CPU**	8x ARM Cortex-A53@1.2 GHz
Mem 2GB LPDDR3@800 MHz		**OS**	Normal: Debian 8 (Linux 4.4) Secure: OP-TEE 2.3

Table 3: The test platform used in experiments

Data Plane (Trusted)

Primitives*	Mem Mgmt*	Misc*	**Total**
3.7K (32.5 KB)	0.7K (6 KB)	0.6K (4 KB)	**5K (42.5 KB)**

Control Plane (Untrusted)

Control	Data types*	Operators*	Test*	Misc*	**Total**
23K	1.3K	4.1K	1K	1K	**31K (348 KB)**

Major Libraries (Untrusted)

glibc 2.19	libstdc++ 3.4.2	libzmq 2.2	boost 1.54	**Total**
1135K	110K	13K	37K	**1.3M (3.1 MB)**

* New implementations of this work. Total = 12.4K SLoC.

Table 4: A breakdown of the StreamBox-TZ source, of which 5K SLoC are in TCB. Binary code sizes shown in parentheses

9.2 Performance & Overhead

Methodology We evaluate SBT on a HiKey board as summarized in Table 3. We chose HiKey for its good OP-TEE support [70] and that it is among the few boards with TrustZone programmable by third parties. We built *Generator*, a program sends data streams over ZeroMQ TCP transport [57] to SBT. We run the cloud consumer on an x86 machine. Data streams are encrypted with 128-bit AES.

In the face of HiKey's platform limitations, we set up the engine ingestion as follows. i) Although Gigabit Ethernet on edge platforms is common [5, 88], Hikey's Ethernet interface (over USB) only has 20MB/sec bandwidth. We have verified that the interface is saturated by SBT with 4 cores. Hence, we report performance when SBT and *Generator* both run on HiKey communicating over ZeroMQ TCP, which still fully exercise the TCP/IP stack and data copy. ii) Although HiKey's TEE is capable of directly operating Ethernet interface as trusted IO, our OP-TEE version lacks the needed drivers. Hence, we emulate SBT's direct data ingestion to TEE by running the ingestion in a privileged process in the normal world, and bypassing data copy across the TEE boundary. Our test harness continuously replays pre-allocated secure memory buffers populated with events.

As summarized in Table 5, we test SBT as well as three modified versions: ***SBT ClearIngress*** ingests data in cleartext; this is allowed if source-edge links are trusted as defined in our threat model (§3). ***SBT IOviaOS*** does not exploit TrustZone's trusted IO: the untrusted OS ingested (encrypted) data and copies the data across TEE boundary to the data plane. ***Insecure*** completely runs in the normal world with ingress and egress in cleartext, showing native performance. This is basically StreamBox [82] with SBT's optimized stream computations (§5). We report the engine performance as its maximum input throughput when the pipeline output delay (defined in §2.2) remains under a target set by us.

Benchmarks We employ six benchmarks of processing sensor data streams from prior work [32, 62, 63, 67, 82]. They

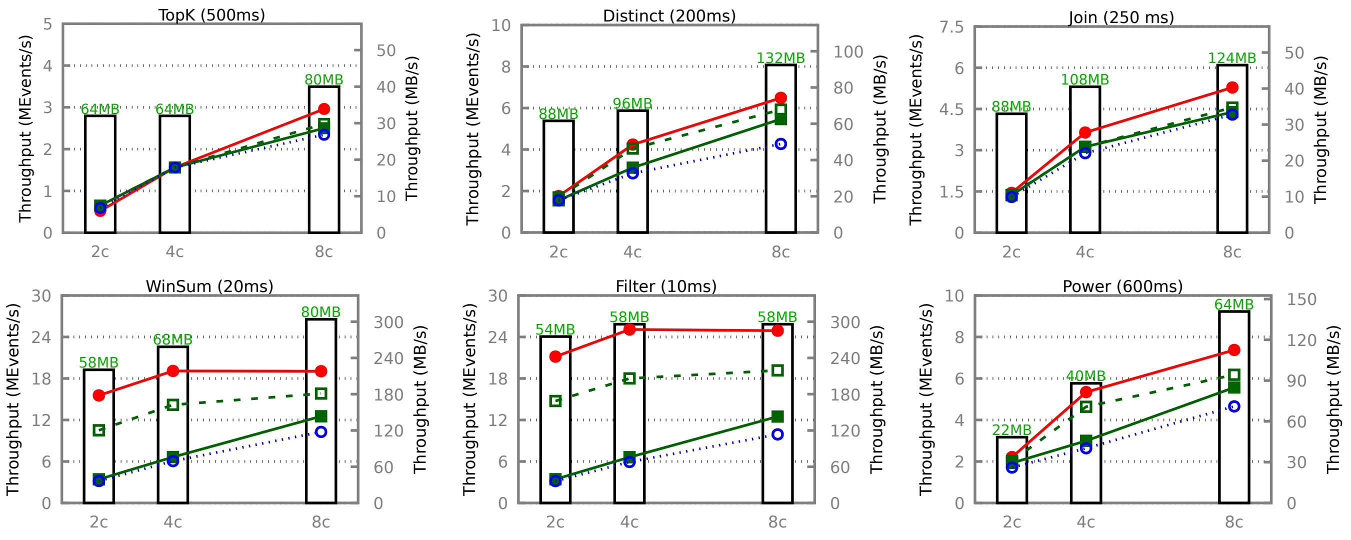


Figure 7: StreamBox-TZ throughput (lines, left/right y-axes) as a function of CPU cores (x-axis) under given output delays (above each plot). Steady consumptions of TEE memory as columns with annotated values. See Table 5 for legends and explanations.

Legend & Version	Data Plane	In/Egress Path	Ingress Data	Egress Data
■ StreamBox-TZ	in TEE	Trusted IO*	Encrypted	Encrypted
□ SBT ClearIngress	in TEE	Trusted IO*	ClearTxt	Encrypted
○ SBT IOviaOS	in TEE	via OS	Encrypted	Encrypted
● Insecure#	out TEE	in OS	ClearTxt	ClearTxt

* Through TrustZone Trusted IO directly to TEE
# Equivalent to a StreamBox invoking StreamBox-TZ's optimized stream compute

Table 5: Engine versions for comparison (plots in Figure 7)

cover major stream operators and a variety of pipelines. We use fixed windows, each encompassing 1M events and spanning 1 second of event time. Each event consists of 3 fields (12 Bytes) unless stated otherwise. (1) **Top Values Per Key (TopK)** groups events based on keys and identifies the K largest values in each group in each window. (2) **Counting Unique Taxis (Distinct)** identifies unique taxi IDs and counts them per window. For input events, we use a dataset of taxi trip information containing 11 K distinct taxi IDs [63]. (3) **Temporal Join (Join)** joins events that have the same keys and fall into same windows from two input streams. (4) **Windowed Aggregation (WinSum)** aggregates input values within each window. We use the Intel Lab Data [75] consisting of real sensor values as input. (5) **Filtering (Filter)** filters out input data, of which field falls into to a given range in each window. We set 1% selectivity as done in prior work [67]. (6) **Power Grid (Power)**, derived from a public challenge [62], finds out houses with most high-power plugs. Ingesting a stream of per-plug power samples, it calculates the average power of each plug in a window and the average power over all plugs in all houses in the window. For each house, it counts the number of plugs that have a higher load than average. It emits the houses that have most high-power plugs in the window. The event for this benchmark is composed of 4 fields (16 Bytes).

Benchmark 2, 4, and 6 use real-world datasets; others use synthetic data sets of which fields are 32-bit random integers. Note that SBT's `GroupBy` operator bases on sort and merge and is insensitive to key skewness [15].

End-to-end performance Figure 7 shows the throughputs of all benchmarks as a function of hardware parallelism. SBT can process up to multiple millions of events within sub-second output delays (labeled atop each plot). For simpler pipelines such as WinSum and Filter, SBT processes around 12M events/sec (140 MB/sec). This throughput saturates one GbE link which is common on IoT gateways [88]. Overall, SBT can use all 8 cores in a scalable manner.

SBT's absolute performance is state of the art. We test three popular, insecure stream engines: Flink [19], designed for distributed environment and known for good single-node performance [68]; Esper [46], designed for a single machine; SensorBee [90], designed for sensor data processing on a single device. As shown in Figure 8, on the same hardware (HiKey) and the same benchmark (WinSum), we have measured that SBT's throughput is at least one order of magnitude higher than the others. This is because i) our *Insecure* baseline has high performance for its rich task parallelism (inherited from StreamBox [82]) and native, vectorized stream computations (new contributions); ii) SBT only imposes modest security overhead, as will be shown later.

Comparison to secure stream engines The comparison is challenged by that TrustZone was rarely exploited for protecting data-intensive computations. To our knowledge, i) no analytics engines use TrustZone for data protection and ii) no systems can partition an insecure stream engine for TrustZone. Note that popular secure analytics engines, e.g. VC3 [99] and Opaque [124], not only require SGX but also target batch processing instead of stream analytics. To this end, we qualitatively compare SBT with SecureStreams [53], the closest system we are aware of. Designed for an x86 cluster, SecureStreams uses SGX to protect stream operators and targets strong data security. On a benchmark similar to WinSum it

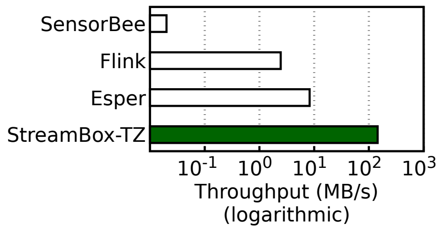


Figure 8: StreamBox-TZ achieves much higher throughput than commodity insecure engines [19, 46, 90] on HiKey. Benchmark: windowed aggregation; target output delay: 50ms.

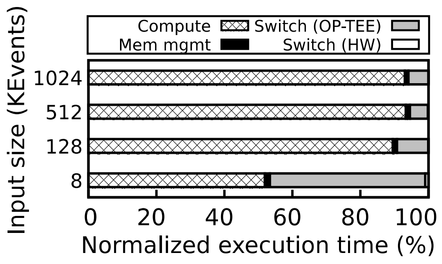


Figure 9: Run time breakdown of operator GroupBy under different input batch sizes. The control plane runs 8 threads to execute GroupBy in parallel. Total execution time is normalized to 100%.

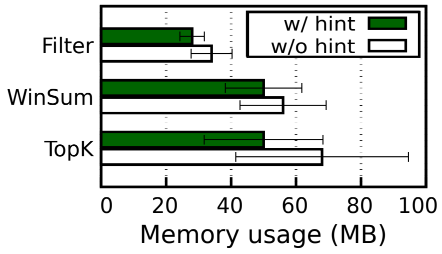


Figure 10: Without consumption hints, the allocator uses more TEE memory. Since memory usage fluctuates at run time, the error bars show two standard deviations below and above the average.

was reported to achieve 10 MB/sec, one magnitude lower than SBT on WinSum. Furthermore, SecureStreams achieved such performance on a small x86 cluster which has much richer resource than HiKey: the former has faster CPUs (8x i7-6700@3.4GHz versus 8x Cortex-A53@1.2GHz), larger DRAM (16 GB versus 2 GB), higher power (130W versus 36W), and higher cost ($600 versus $65).

SBT's advantage comes from i) data exchange via coherent memory inside one TEE, instead of exchanging encrypted messages among workers; ii) memory management specialized for streaming, and iii) vectorized computations.

Security overhead We investigate the overhead of the new security mechanism contributed by SBT – its isolated data plane. We assess the overhead as the throughput loss of *SBT ClearIngress* as compared to *Insecure* (i.e. native performance as StreamBox [82] invoking SBT's stream computations), both paying same costs for data ingress. The target output delays are the same (labeled atop each plot in Figure 7). The security overhead is less than 25% in all benchmarks. This is similar to or lower than the reported overhead (20–70%) in recent TEE systems [22, 71, 112]. ***Overhead analysis:*** The security overhead mostly comes from world switch, among operators and inside each operator. To understand the switch cost within an operator, we profile GroupBy, one of the most costly operators. We test different input batch sizes, which have a strong impact on TEE entry/exit rates and hence isolation overhead (§4). Figure 9 shows a run time breakdown. When each input batch contains 128K (close to the value we set for SBT) or more events, more than 90% of the CPU time is spent on actual computations in TEE. The CPU usage of TEE memory management is as low as 1–2%. In the extreme case where each input batch contains as few as 8K events, the overhead of world switch starts to dominate. Most of the world switch overhead comes from OP-TEE instead of the CPU hardware (a few thousand cycles per switch), suggesting room for OP-TEE optimization.

Impact of decrypting ingress data Decrypting ingress data is needed if source-edge links are untrusted (§3) and source must send encrypted data. It has substantial performance impact. By comparing SBT to *SBT ClearIngress*, turning on/off ingress decryption leads to 4% – 35% throughput difference when all 8 cores are in use. The performance gap is more pronounced for simple pipelines, which has higher ingestion throughput leading to higher decryption cost.

TEE memory usage While sustaining high throughput, SBT consumes a moderate amount of physical memory, ranging from 20 MB to 130 MB as shown in Figure 7. The memory usage is as low as 1–6% of the total system DRAM. The virtual memory usage is also low, often 1–5% of the entire virtual address space in OP-TEE. The memory usage increases with the throughput, since there will be more in-flight data. On the same platform, Flink's memory consumption is 3× higher, due to its hash-based data structures and the use of JVM. This validates our choice of uArrays.

Attestation overhead Attestation incurs minor overhead to both the edge and the cloud. We measured that SBT produces 300–400 audit records per second across all our benchmarks, and spends a few hundred cycles on producing each record. Compressing such record streams on HiKey consumes 0.2% of total CPU time. Our consumer written in Python on a 4-core i7-4790 machine replays 57K records per second with a single core, suggesting a capability of attesting near 500 SBT instances simultaneously. We will evaluate the efficacy of record compression in Section 9.3.

9.3 Validation of Key Design Features

Exploitation of trusted IO As shown in Figure 7, a comparison between SBT and *SBT IOviaOS* demonstrates the advantage of directly ingesting data into TEE and bypassing the OS: SBT outperforms the latter by up to 20% in throughput due to reduction in moving ingested data.

Trusted primitive vectorization (§5) Our optimizations with ARM vector instructions are crucial. To show this, we examine GroupBy, one of the top hotspot operators. When we replace the vectorized Sort that underpins GroupBy with two popular implementations (qsort() from the the OP-TEE's libc and std::sort() from the standard C++ library), we measured the throughput of GroupBy drops by up to 7× and 2×,

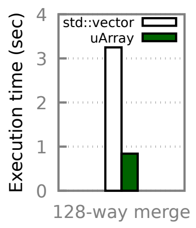
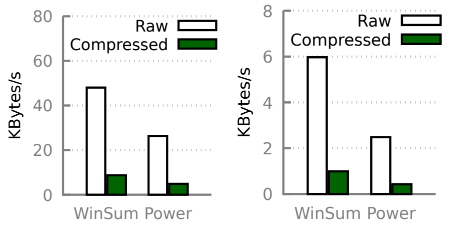


Figure 11: On-demand growth of uArrays vs. std::vector



(a) 10K events/batch (b) 100K events/batch

Figure 12: Compression of audit records saves uplink bandwidth substantially.

respectively. We have similar observation on other operators.

Efficacy of hint-guided memory placement (§6.2) We compare to an alternative design: the modified allocator acts based on the heuristics that all the uArrays produced by the same primitive belong to the same *generation* and are likely to be reclaimed altogether. Accordingly, the modified allocator places these uArrays in the same uGroup. As shown in Figure 10, in three benchmarks, the modified allocator increases memory usage by up to 35%. This is because, without hints, it cannot place uArrays based on future consumption.

uArray on-demand growth (§6.1) We compare uArray to std::vector, a widely used C++ sequence container with on-demand growth. We run a microbenchmark of N-way merge, an intensive procedure in trusted primitives. It iteratively merges 128 buffers (uArrays or vectors), each containing 512 KB (128K 32-bit random integers) until obtaining a monolithic buffer; as merge proceeds, buffers grow dynamically. As shown in Figure 11, uArrays is 4× faster than std::vector, because the allocation and paging in TEE that back uArray growth is much faster than that of a commodity OS.

Compression of audit records (§7) The compression significantly saves the uplink bandwidth. We test two benchmarks (WinSum and Power) on two extremes of the spectrum of computation cost, and test two very different input batch sizes. This is because simpler computations and smaller batch sizes generate audit records at higher rates. Figure 12 shows that SBT compresses audit records by 5×–6.7×. In an offline test using gzip to compress the same records, we find our compression ratios are 1.9× higher than gzip. 2–40 KB/sec of uplink bandwidth is saved, which is significant compared to the uploaded analytics results, which are 144 bytes/sec for WinSum and 400 bytes/sec for Power.

10 Related Work

Secure data analytics DARKLY [61] protects sensor data by isolating computations in an OS process, resulting in a large TCB. VC3 [99] and SecureStreams [53] use SGX to protect the operators in distributed analytics. They lack optimizations for parallel execution in one TEE on the edge. To process data confidentiality, STYX [106] computes over encrypted data, a method likely prohibitively expensive to edge platforms. Opaque [124] protects data access patterns of distributed operators, targeting a threat out of our scope.

TCB minimization Minimizing TCB is a proven approach towards a trustworthy system. Flicker [80] directly executes security-sensitive code on baremetal hardware. Trustvisor [79] shrinks its TCB to a specialized hypervisor. Sharing a similar goal, SBT addresses unique challenges in supporting data-intensive computation on a minimal TCB.

Trusted Execution Environments Much work isolates security-sensitive software components. Terra [48] supports isolation with a virtual machine. Many systems used TrustZone and SGX [81] for TEE. Some systems enclose in TEE whole applications [22, 27, 51, 112], while others partition existing programs for TEE [71, 93, 101]. These approaches often result in larger TCBs and/or higher overhead than SBT and are thus less desirable for SBT. TEE also sees various novel usage, including protecting mobile app classes [96], enforcing security policies [30], remote attestation of application control flows [13], and controlling data access [34]. None addresses data-intensive computation as SBT does.

Edge processing evolves from a vision [98, 100] to practice [37, 42, 83]. Most works focused on programming paradigms [94], developing and deploying application [29, 52, 114], and resource management [86]. Complementary to them, SBT focuses on secure analytics on the edge.

Stream processing systems, in response to big data challenges, evolve from single-threaded [33, 40, 78, 105, 110] to massive parallel systems [14,69,85,92,92,113,122]. The existing systems focus on challenges, such as fault tolerance [122], fast reconfiguration [115], high parallelism [32, 82], and the use of GPUs [67]. Few systems achieve data security and performance simultaneously as SBT does.

11 Conclusions

This paper presents StreamBox-TZ (SBT), a secure stream analytics engine designed and optimized for a TEE on an edge platform. SBT offers strong data security, verifiable results, and competitive performance. On an octa core ARM machine, SBT processes up to tens of millions of events per second; its security mechanisms incur less than 25% overhead.

Acknowledgments

The authors thank the anonymous reviewers and our shepherd, Eyal de Lara, for their insightful comments. For this project: the authors affiliated with Purdue ECE were supported in part by NSF Award #1718702, #1619075, Purdue University CPS/IoT Seed Grant Program, and a Google Faculty Award; the author affiliated with Northeastern University was supported in part by NSF Award #1748334.

References

[1] Apache Beam. https://beam.apache.org/.

[2] ARM TrustZone. http://www.arm.com/products/processors/technologies/trustzone/index.php.

[3] CVE-2010-3190: Untrusted search path vulnerability in the microsoft foundation class (mfc) library. https://nvd.nist.gov/vuln/detail/CVE-2010-3190.

[4] CVE-2017-12629: Remote code execution occurs in apache solr. https://nvd.nist.gov/vuln/detail/CVE-2017-12629.

[5] Marvell Armada 8K family processors. http://www.marvell.com/embedded-processors/armada-80xx/.

[6] CVE-2008-0171: Boost.regex allows context-dependent attackers to cause failed assertion and crash. https://nvd.nist.gov/vuln/detail/CVE-2008-0171s, 2008.

[7] CVE-2009-2493: Active template library does not properly restrict use of oleloadfromstream in instantiating objects from data streams, which allows remote attackers to execute arbitrary code. https://nvd.nist.gov/vuln/detail/CVE-2009-2493, 2009.

[8] CVE-2015-4421: in huawei mate7. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4421, 2015.

[9] CVE-2015-4422: in huawei mate7. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4422, 2015.

[10] CVE-2016-10229: udp.c in the linux kernel before 4.5 allows remote attackers to execute arbitrary code. https://nvd.nist.gov/vuln/detail/CVE-2016-10229, 2016.

[11] CVE-2017-11176: The mq_notify function in the linux kernel allows attackers to cause a denial of service or possibly have unspecified other impact. https://nvd.nist.gov/vuln/detail/CVE-2017-11176, 2017.

[12] CVE-2018-8822: Incorrect buffer length handling in the ncp_read_kernel function could be exploited by malicious ncpfs servers to crash the kernel or execute code. https://nvd.nist.gov/vuln/detail/CVE-2018-8822, 2017.

[13] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[14] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endow.*, 8(12):1792–1803, 2015.

[15] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endow.*, 5(10):1064–1075, 2012.

[16] M. P. Andersen and D. E. Culler. Btrdb: Optimizing storage system design for timeseries processing. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[17] M. P. Andersen, S. Kumar, C. Brooks, A. von Meier, and D. E. Culler. Distil: Design and implementation of a scalable synchrophasor data processing system. In *2015 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, 2015.

[18] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. In *International Workshop on Security Protocols*, 1997.

[19] Apache. Apache flink: Scalable stream and batch data processing. https://flink.apache.org/, 2017.

[20] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *Proceedings of the VLDB Journal*, 15(2):121–142, 2006.

[21] Arm. Arm neon technology. https://developer.arm.com/technologies/neon, 2018.

[22] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[23] Art Manion. CERT/CC Blog – Anatomy of Java Exploits. https://insights.sei.cmu.edu/cert/2013/01/anatomy-of-java-exploits.html/, 2013.

[24] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. *Proceedings of Commun. of the ACM*, 55(5):111–119, 2012.

[25] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[26] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endow.*, 7(1):85–96, 2013.

[27] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[28] A. Becher, Z. Benenson, and M. Dornseif. Tampering with motes: Real-world physical attacks on wireless sensor networks. In *Proceedings of the 3rd International Conference on Security in Pervasive Computing*, 2006.

[29] K. Bhardwaj, M. W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan. Fast, scalable and secure onloading of edge functions using airbox. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016.

[30] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A.-R. Sadeghi. Regulating arm trustzone devices in restricted spaces. In *Proceedings of the 14th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[31] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza. Securekeeper: Confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, 2016.

[32] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endow.*, 8(4):401–412, 2014.

[33] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003.

[34] F. Chen. *Cross-platform data integrity and confidentiality with graduated access control.* PhD thesis, The University of British Columbia, 2016.

[35] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys)*, 2011.

[36] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[37] CISCO. White paper: The cisco edge analytics fabric system. http://www.cisco.com/c/dam/en/us/products/collateral/analytics-automation-software/edge-analytics-fabric/eaf-whitepaper.pdf, 2016.

[38] R. Clapis. Go get my/vulnerabilities: an in-depth analysis of go language. https://www.blackhat.com/docs/asia-17/materials/asia-17-Clapis-Go-Get-My-Vulnerabilities-An-In-Depth-Analysis-Of-Go-Language-Runtime-And-The-New-Class-Of-Vulnerabilities-It-Introduces.pdf, Blackhat Asia 2017.

[39] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[40] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003.

[41] J. E. David Goldblatt, Dave Watson. Jemalloc memory allocator. http://http://jemalloc.net/, 2017.

[42] Dell. Dell further democratizes advanced analytics with latest release of statistica. http://www.dell.com/learn/us/en/uscorp1/press-releases/2016-04-14-dell-further-democratizes-advanced-analytics, 2016.

[43] Documentation. "a new reality for oil & gas". https://www.cisco.com/c/dam/en_us/solutions/industries/energy/docs/OilGasDigitalTransformationWhitePaper.pdf, 2017.

[44] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos. The mondrian data engine. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017.

[45] Eclipse IoT Working Group. IoT Developer Survey 2018. https://https://blogs.eclipse.org/post/benjamin-cab%C3%A9/key-trends-iot-developer-survey-2018, 2018.

[46] EsperTech. Esper. http://www.espertech.com/esper/, 2017.

[47] Facebook. Folly. https://github.com/facebook/folly#folly-facebook-open-source-library, 2017.

[48] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[49] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharkey, A. Sheth, and L. P. Cox. Youprove: Authenticity and fidelity in mobile sensing. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2011.

[50] B. Glavic, K. Sheykh Esmaili, P. M. Fischer, and N. Tatbul. Ariadne: Managing fine-grained provenance on data streams. In *Proceedings of the 7th ACM International Conference on Distributed and Event-based Systems (DEBS)*, 2013.

[51] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[52] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014.

[53] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni. Securestreams: A reactive middleware framework for secure data stream processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS)*, 2017.

[54] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[55] hortonworks. "iot and predictive big data analytics for oil and gas". https://hortonworks.com/solutions/oil-gas/, 2017.

[56] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vtz: Virtualizing ARM trustzone. In *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security)*, 2017.

[57] iMatix Corporation. Zeromq. http://zeromq.org/, 2018.

[58] M. G. Institute. The internet of things: Mapping the value beyond the hype.

[59] Intel. Intel threading building blocks. https://software.intel.com/en-us/intel-tbb, 2017.

[60] Intel. "iot solutions for upstreamoil and gas". https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/oil-and-gas-iot-brief.pdf, 2017.

[61] S. Jana, A. Narayanan, and V. Shmatikov. A scanner darkly: Protecting user privacy from perceptual applications. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[62] Z. Jerzak and H. Ziekow. The debs 2014 grand challenge. In *Proceedings of the 8th ACM International Conference on Distributed and Event-Based Systems (DEBS)*, 2014.

[63] Z. Jerzak and H. Ziekow. The debs 2015 grand challenge. In *Proceedings of the 9th ACM International Conference on Distributed and Event-Based Systems (DEBS)*, 2015.

[64] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proceedings of the VLDB Endow.*, 2(2):1378–1389, 2009.

[65] S. Kim, J. Han, J. Ha, T. Kim, and D. Han. Enhancing security and privacy of tor's ecosystem by using trusted execution environments. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[66] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and

S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[67] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[68] A. Krettek and M. Winters. "the curious case of the broken benchmark: Revisiting apache flink® vs. databricks runtime". https://data-artisans.com/blog/curious-case-broken-benchmark-revisiting-apache-flink-vs-databricks-runtime, 2017.

[69] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: Continuous reliable distributed processing of big data streams. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI)*, 2016.

[70] Linaro. Op-tee: Open portable trusted execution environment. https://www.op-tee.org/, 2017.

[71] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch. Glamdring: Automatic application partitioning for intel sgx. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2017.

[72] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Armageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Conference on Security Symposium (USENIX Security)*, 2016.

[73] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[74] S. Ma, X. Zhang, and D. Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *Proceedings of 23rd Network and Distributed System Security Symposium (NDSS)*, 2016.

[75] S. Madden. Intel lab data. http://db.csail.mit.edu/labdata/labdata.html, 2004.

[76] Magazine. "smart grids: Everything you need to know". https://www.cleverism.com/smart-grids-everything-need-know/, 2014.

[77] Magazine. "internet of things: A data-driven future for manufacturing". https://www.themanufacturer.com/wp-content/uploads/2017/01/IoT_FutureofManuf_ebook_final.pdf, 2017.

[78] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos. Semantics of data streams and operators. In *Proceedings of the 10th International Conference on Database Theory (ICDT)*, 2005.

[79] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, 2010.

[80] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM European Conference on Computer Systems (EuroSys)*, 2008.

[81] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[82] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. Streambox: Modern stream processing on a multicore machine. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2017.

[83] Microsoft. Microsoft azure iot edge– extending cloud intelligence to edge devices. https://azure.microsoft.com/en-us/services/iot-edge/, 2017.

[84] Mohammad Marashi, Tech Crunch. Satellites are critical for IoT sector to reach its full potential. https://techcrunch.com/2017/06/08/satellites-are-critical-for-iot-sector-to-reach-its-full-potential/, 2017.

[85] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[86] S. Nastic, H. L. Truong, and S. Dustdar. A middleware infrastructure for utility-based provisioning of iot cloud systems. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016.

[87] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[88] NXP Semiconductors. i.MX 7Dual Family of Applications Processors Datasheet, howpublished = https://www.nxp.com/docs/en/data-sheet/imx7dcec.pdf, year = 2017.

[89] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy. Safebricks: Shielding network functions in the cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[90] Preferred networks. Sensorbee: Lightweight stream processing engine for iot. http://sensorbee.io/, 2017.

[91] C. Priebe, K. Vaswani, and M. Costa. Enclavedb: A secure database using SGX. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

[92] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013.

[93] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[94] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016.

[95] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[96] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[97] S. Saroiu and A. Wolman. I am a sensor, and i approve this message. In *Proceedings of the 11th Workshop on Mobile Computing Systems & Applications (HotMobile)*, 2010.

[98] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos. Edge analytics in the internet of things. *Proceedings of IEEE Pervasive Computing*, 14(2):24–31, 2015.

[99] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[100] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *Proceedings of IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[101] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *Proceedings of 24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[102] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *Proceedings of SIGMOD Rec.*, 34(3):31–36, 2005.

[103] A. Spark. "spark streaming programming guide". https://spark.apache.org/docs/latest/streaming-programming-guide.html, 2016.

[104] S. Sponseller. "the importance of the edge for the industrial internet of things in the energy industry". https://www.datascience.com/blog/predictive-analytics-in-industrial-iot, 2017.

[105] M. C. Stanley Zdonik, Michael Stonebraker. Streambase systems. http://www.tibco.com/products/tibco-streambase, 2017.

[106] J. J. Stephen, S. Savvides, V. Sundaram, M. S. Ardekani, and P. Eugster. Styx: Stream processing with trustworthy cloud-based execution. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[107] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, 2005.

[108] Symantec. Internet Security Threat Report. https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf, 2017.

[109] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Proceedings of Empirical Software Engineering*, 19(6):1665–1705, 2014.

[110] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, 2002.

[111] T. Trippel, O. Weisse, W. Xu, P. Honeyman, and K. Fu. WALNUT: Waging doubt on the integrity of MEMS accelerometers with acoustic injection attacks. In *Proceedings of the 2nd Annual IEEE European Symposium on Security and Privacy*, 2017.

[112] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2017.

[113] Twitter. Heron. https://twitter.github.io/heron/, 2017.

[114] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman. Farmbeats: An iot platform for data-driven agriculture. In *Proceedings of 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[115] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[116] J. W. Voung, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007.

[117] Wind River. SECURITY IN THE INTERNET OF THINGS – Lessons from the Past for the Connected Future. https://www.windriver.com/whitepapers/security-in-the-internet-of-things/wr_security-in-the-internet-of-things.pdf, 2017.

[118] X. Wu, R. Dunne, Q. Zhang, and W. Shi. Edge computing enabled smart firefighting: Opportunities and challenges. In *Proceedings of the 5th ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2017.

[119] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[120] D. Yarmoluk and C. Truempi. "predictive analytics in industrial iot". https://www.datascience.com/blog/predictive-analytics-in-industrial-iot, 2018.

[121] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[122] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[123] N. Zhang, K. Sun, W. Lou, and Y. T. Hou. Case: Cache-assisted secure execution on arm processors. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[124] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.