AutoSketch: Automatic Sketch-Oriented Compiler for Query-driven Network Telemetry

Haifeng Sun[1] *Qun Huang*[1] *Jinbo Sun*[2] *Wei Wang*[3] *Jiaheng Li*[1]
Fuliang Li[3] *Yungang Bao*[2] *Xin Yao*[4] *Gong Zhang*[4]
[1]*National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University*
[2]*Institute of Computing Technology, CAS* [3]*Northeastern University, China* [4]*Huawei Theory Department*

Abstract

Recent network telemetry witnesses tremendous progress in two directions: query-driven telemetry that targets expressiveness as the primary goal, and sketch-based algorithms that address resource-accuracy trade-offs. In this paper, we propose AutoSketch that aims to integrate the advantages of both classes. In a nutshell, AutoSketch automatically compiles high-level operators into sketch instances that can be readily deployed with low resource usage and incur limited accuracy drop. However, there remains a gap between the expressiveness of high-level operators and the underlying realization of sketch algorithms. AutoSketch bridges this gap in three aspects. First, AutoSketch extends its interface derived from existing query-driven telemetry such that users can specify the desired telemetry accuracy. The specified accuracy intent will be utilized to guide the compiling procedure. Second, AutoSketch leverages various techniques, such as syntax analysis and performance estimation, to construct efficient sketch instances. Finally, AutoSketch automatically searches for the most suitable parameter configurations that fulfill the accuracy intent with minimum resource usage. Our experiments demonstrate that AutoSketch can achieve high expressiveness, high accuracy, and low resource usage compared to state-of-the-art telemetry solutions.

1 Introduction

The continuous growth of modern data centers inspires tremendous progress on network telemetry, which forms the basis of various network management tasks, such as anomaly detection [32, 43, 99], performance analysis [74, 78, 92, 100], and fault diagnosis [5, 13, 14, 33, 44, 93, 106]. Recent advances in network telemetry can be categorized into two classes: *query-driven telemetry* and *telemetry algorithms*. Query-driven telemetry [30, 66, 67, 97] considers *expressiveness* as the primary goal. They provide extensive operators for users to build various telemetry tasks, and then automatically deploy the operators into switches. Therefore, users do not need to concern with the implementation details in switch hardware. However, the resource-efficient realization of operators is not well addressed in existing query-driven telemetry solutions. In particular, many operators maintain flow-level *states*. As the number of interested flows increases, maintaining the states easily exhausts switch memory [30] or network bandwidth [67].

The other class is telemetry algorithms. Their common idea is to leverage approximate techniques (e.g., sampling [73, 76, 77], top-k counting [6, 79], and sketch [1, 34, 36, 54, 55, 94, 103]) that sacrifice a small portion of accuracy to relax resource requirements. Even though recent studies show that such approximate techniques can achieve both resource efficiency and bounded errors, it remains non-trivial to utilize them. First, each approximate technique is designed for several specific telemetry applications and fails to support all tasks. Second, an approximate technique needs careful parameter configuration to simultaneously reach high accuracy and resource efficiency. Thus, for each telemetry task, users need to select the appropriate algorithms, carefully realize them, and tune parameters in switches, which needs domain knowledge on the programming models and hardware restrictions.

In this paper, we present AutoSketch that integrates the advantages of both query-driven telemetry and sketch algorithms. Its goal is to fully exploit the resource efficiency of telemetry algorithms while hiding the complicated details of implementing and configuring them in network devices. This is extremely significant because (i) users lack the domain knowledge of underlying telemetry algorithms and architecture; and (ii) the hardware resources of existing network devices are limited. On the one hand, AutoSketch follows recent operator-based telemetry languages. It defines extensive interfaces for users to build telemetry applications with both *built-in* operators and *user-defined* operators, which achieves high expressiveness. On the other hand, AutoSketch automatically converts these operators to sketch algorithms, which achieves high accuracy and low resource usage.

However, it is challenging to bridge the gap between the high-level operators and the sketch algorithms. First, sketch

*Qun Huang is the corresponding author.

algorithms incur errors due to their approximation nature (although the errors are bounded). Users should be capable of perceiving and controlling the incurred errors for telemetry applications. Second, user-defined operators complicate the conversion to sketch. In particular, many user-defined operators maintain numerous states, which are hard to realize as sketch. For some operators, there is even no ready-made sketch algorithm to realize them. Finally, sketch algorithms expose various parameters to configure, which needs to take underlying hardware into account.

AutoSketch addresses the challenges in three aspects:

- First, AutoSketch extends its operator-based interface to allow users to specify the desired accuracy as an *accuracy intent* (§2). Then, AutoSketch employs a *compiler* that automatically generates sketch-based solutions that fulfill the intent (§3). The compiler hides the details on switch hardware and algorithm details. To this end, administrators can control the extent of approximation without concerns with underlying realization.
- Second, AutoSketch constructs efficient sketch instances for the stateful operators (§4). For built-in operators, AutoSketch realizes them by classical sketch algorithms. For user-defined operators, AutoSketch decomposes each of them into several simple operators with the aid of syntax analysis. Then, AutoSketch designs *sketch-like structures* to instantiate the decomposed operators. Each sketch-like structure follows the same idea of sketch that maps traffic into different cells, but employs different methods to aggregate cells (depending on the user-defined functions). The sketch-like structures provide theoretical guarantees to estimate the impact of hash conflicts like classical sketch.
- Finally, AutoSketch formulates the parameter tuning into an optimization problem, which reveals the hardness of parameter tuning (§5). We propose a searching algorithm to find the *most suitable* configuration that fulfills the accuracy intent and incurs minimum resource usage. The algorithm employs various optimization techniques and eliminates the user burdens of tuning sketch parameters.

We build a prototype of AutoSketch that targets PISA [10] switches and compare AutoSketch with existing query-driven telemetry systems and sketch-based telemetry algorithms. We show that AutoSketch can implement telemetry applications in less than 20 lines of code without extra efforts to tune configurations. Under 200 K active flows per 100 ms, AutoSketch still reaches 99% recall, 97% precision, and 2.8% errors with only 0.84 MB switch memory and 84 KB/s bandwidth on average for 11 applications. We release our source code at **https://github.com/N2-Sys/AutoSketch**.

2 AutoSketch Interface

AutoSketch follows recent telemetry languages (e.g., Sonata [30] and Marple [67])that provide operator-based interface. It abstracts network traffic as a stream of *tuples*, each of which consists of a *key* and several *fields*. An operator performs specific computations on each input tuple and then generates some output tuples if needed. A telemetry application is defined as an operator graph in which vertices are operators. A (directed) edge indicates that the output tuples of the upstream operator are consumed by the downstream operator.

Operator	Description
filter(*bool_expr*)	Check the boolean expression *bool_expr* for each tuple and preserve tuples satisfying conditions.
map(*fields*, [*expr*])	Transform each input tuple into an output tuple consisting of *fields*, whose values are set by *expr*.
distinct(*fields*)	Categorize input tuples based on *fields*, and preserve one tuple for each category (i.e., delete duplicated tuples with the same field values).
reduce(*fields*, *val*)	Categorize tuples according to *fields* and sum up *val* for each category.
zip(*s*, *field*)	Merge the input stream with tuples of another stream *s* containing the same field values in *fields*.
groupby(*states*, *udf*)	Invoke user-defined function *udf* to update *states*.

Table 1: Operators of AutoSketch Interface

Operators. Table 1 summarizes the operators supported by AutoSketch. Currently, AutoSketch provides five *built-in* operators: `filter`, `distinct`, `map`, `reduce`, and `zip`. Among them, `distinct` and `reduce` are *stateful*, i.e., maintaining internal states for tuple processing. Here, an operator state refers to a key-value pair associating with tuples that have the same key. To achieve more expressiveness, AutoSketch further provides a `groupby` operator that allows users to customize the processing logic in two aspects. First, a `groupby` operator can maintain user-defined *states*. Second, each `groupby` operator associates with a *user-defined function* (`UDF`) to define the processing logics on the tuples and states. AutoSketch supports three types of statements in a `UDF`: (1) assignment statements, (2) arithmetic statements, and (3) conditional statements.

Accuracy intents. AutoSketch realizes stateful operators (i.e., `distinct`, `reduce`, and `groupby` with states) using sketch techniques, which inevitably incurs errors. To characterize the errors, AutoSketch provides users with *accuracy intent* interface, which can specify user-desired accuracy. In the context of operator-based processing, an accuracy intent consists of four accuracy metrics:

- *Recall*: the ratio of true output tuples to all true tuples that should be output.
- *Precision*: the ratio of true output tuples to all output tuples.
- *Average relative error (ARE)*: the mean relative error of all output tuples.
- *Confidence*: the probability that all other accuracy metrics (e.g., recall, precision) are met.

Users can selectively use the accuracy metrics for a telemetry application. For a metric, users can specify the minimal/maximal value or the desired range for it (e.g., `ARE_max` = 1%, `recall` = 95% ± 5%). If a metric is not specified, AutoSketch considers that there is no requirement on it. Note that the intents are set for the entire application instead of individual operators. Thus, users do not need to concern with how to set

```
1  def nonmt(tcp.seq):
2    if maxseq < tcp.seq or maxseq == 0:
3      maxseq = tcp.seq
4    else:
5      nm_count += 1
6
7  nm[precision_min=99%,ARE_max=1%,confidence=95%]=
8    PacketStream()
9    .filter(ipv4.protocol==TCP)
10   .groupby({5tuple:(maxseq, nm_count)}, nonmt)
```

Application 1: TCP non-monotonic detection

per-operator errors to form the application-level accuracy. AutoSketch automatically handles the application-level intents by configuring operator-level parameters.

Example. Application 1 presents an example that monitors non-monotonic sequence numbers of TCP flows. The application first selects TCP packets by a `filter` operator (line 9). Then it uses the `groupby` operator to define a `UDF`, namely `nonmt` (line 10). The `groupby` operator maintains two states. The `maxseq` state tracks the maximal sequence number of each flow (line 3). The `nm_count` state counts the number of packets whose sequence numbers are lower than `maxseq` for each flow (line 5). The intent requires that with a probability of 95%, the monitored flows achieve 99% precision, and per-flow ARE is below 1% (line 7).

Generality. The operators in AutoSketch come from recent query-driven telemetry systems [30, 67, 105]. They cover a rich set of telemetry applications according to these studies. We do not claim the novelty of this design. However, to the best of our knowledge, we are the first to define accuracy intent interface that (1) empowers automatic control operator errors introduced by sketch and (2) hides the underlying implementation and configuration details of sketch. Some prior works (e.g., BlinkDB [3]) also define accuracy intent but they control sampling rate of input traffic instead of operator behaviors. On the other hand, some efforts [19, 45, 105] also attempt to apply sketch algorithms to query-driven telemetry, but lack the accuracy intent interface, leaving the error configuration to the users.

Limitations. The programming model has three limitations due to the resource constraints of commodity devices. First, AutoSketch cannot support complicated statements, such as loops and floating-point arithmetic, because commodity switch ASICs only allow limited recirculation and such statements are too expensive [26]. Second, the states in an operator must have limited dependency: users cannot define two states whose processing depends on each other (details in §4.1). Finally, the accuracy intent should not consume more resources than hardware capacity. When a telemetry application compromises any of these restrictions, AutoSketch throws an error as existing data-plane compilers [70]. We will support more compiling modes in future work, such as suggesting accuracy intents and returning multiple configurations within accuracy ranges for users to select from.

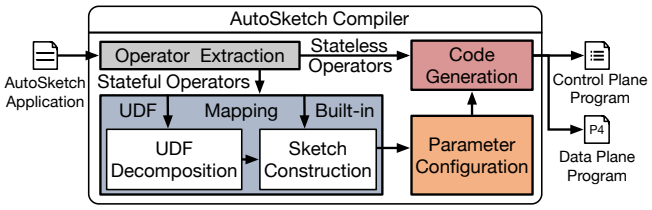


Figure 1: AutoSketch Compiler.

3 AutoSketch Compiler

AutoSketch designs a general compiler in order to bridge the gap between our interface and the underlying hardware. The compiler takes a telemetry application composed of several operators and an accuracy intent as input. It converts the application to a P4 program that runs in the data plane to record per-packet information, and a control plane program that retrieves telemetry results and reports the results to users.

Figure 1 depicts the compiler internals. For an input application, the compiler first extracts its operators. Then it maps the stateful operators into sketch instances. For built-in stateful operators, the compiler directly constructs corresponding sketch instances. For `groupby` operators with complicated `UDFs`, the compiler first decomposes each `UDF` into several simple ones, and then performs the construction. After sketch mapping, AutoSketch configures parameters for the sketch instances to satisfy the accuracy intent. Finally, AutoSketch assembles output programs by integrating the converted sketch instances with the stateless operators.

Although sketch mapping and parameter configuration have been studied in prior works (§3.1 and §3.2), it is non-trivial to combine them because we need to fulfill the application-level accuracy intent. Specifically, when an application is mapped to numerous combinations of different sketch algorithms, it remains an open issue to configure operator-level sketch parameters to satisfy application-level accuracy intent [58].

3.1 Mapping for Stateful Operators

AutoSketch constructs a sketch instance for each stateful operator instead of using a single universal sketch for the entire application. The reason is that existing universal sketch algorithms are also composed of multiple basic structures [36, 50, 55], whose overall resource usage is nontrivial and hard to be optimized. In contrast, per-operator sketch mapping allows more fine-grained sketch selection and tuning.

Specifically, the compiler employs different strategies for built-in operators and `groupby` operators. For built-in stateful operators, since their functionalities are common, the compiler realizes them with well-known sketch algorithms, e.g., Count-Min Sketch (CM) [20] and Bloom Filter (BF) [8].

For `groupby` operators, AutoSketch follows the idea of sketch to construct sketch-like structures. As shown in Figure 2, a sketch-like structure comprises multiple rows of cells, where each cell maintains the state values of the `groupby` operator. To process a tuple, we hash its key to one cell in

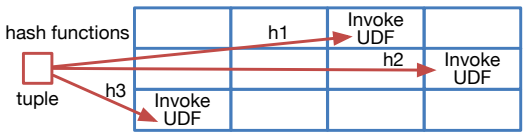


Figure 2: Sketch-like structure in AutoSketch.

each row, and then invoke the UDF to update the states in the hashed cells. When querying for the key, we mitigate the errors caused by hash conflicts. Specifically, we use the same hash functions to select cells, and aggregate the cells based on the linearity or monotonicity of the state. We use the aggregated results to estimate the state value. We also provide theoretical analysis for the sketch-like structures. Thus, compared to existing sketch algorithms, our sketch-like structures support more types of states and user-defined processing logics, while preserving bounded telemetry errors. Note that we choose two-dimensional (2D) matrices for the sketch-like structures because their parameters are simpler and fewer than that of higher-dimensional matrices, which incur more switch resources to handle the additional dimensions.

Challenges. The mapping is non-trivial for both built-in and groupby operators. For each type of built-in operator, there could be multiple possible sketch algorithms that can realize its functionalities. (e.g., reduce operator can be mapped to Count-Min Sketch [20] or Count Sketch [12].) The algorithms achieve different levels of resource-accuracy trade-offs. We need to choose suitable algorithms to meet application accuracy intents. For groupby operators, the challenge comes from the complexity of UDFs. If we map a complicated UDF to a single sketch instance entirely, every cell includes all operator states. In this case, the incurred computational resources are overwhelming. In particular, a UDF may contain multiple state updates. However, existing switch ASICs only support limited per-cell operations. For example, PISA switches only allow one read-and-modify operation for each memory address and a small number of conditional judgments (i.e., the boolean expression in the if statement) [89].

Solutions (§4). For built-in operators, AutoSketch estimates the efficiency of all combinations of candidate sketch algorithms. Then it chooses the one incurring minimum resource usage while reaching the desired accuracy. For UDFs, AutoSketch decomposes every complicated UDF via abstract syntax trees, a common technique for syntax analysis. Each decomposed UDF contains exactly one state. Thus, AutoSketch can construct sketch instances for decomposed UDFs with limited resource usage, and tune the parameters for each state individually. Although Newton [105] also maps operators into sketches, it does not allow UDFs. Note that AutoSketch is not binding to several specific sketches or sketch-like structures. It is expected for AutoSketch to support newly proposed sketches [87, 101], by (i) adding new sketch candidates for existing stateful operators or (ii) introducing new operators (e.g., top operator). Note that the actual errors are varying depending on specific sketches used for operators.

3.2 Parameter Configuration

AutoSketch automatically tunes the parameters of the constructed sketch instances. The parameter configuration targets both high accuracy and resource efficiency. That is, AutoSketch aims to use minimum resources for achieving a specified accuracy intent.

Challenges. Although existing sketch algorithms provide theoretical analysis to characterize the relationship between parameters and accuracy, it remains challenging to tune parameters for AutoSketch in two aspects. First, as in recent studies [21, 36, 57], the theoretical analysis of existing sketch algorithms typically addresses worst-case scenarios, thereby providing limited guidelines for parameter tuning. Second, a telemetry application usually comprises multiple sketch instances. Existing works on merging sketch [2, 48] only apply to multiple sketch instances of the same type. Thus, it remains challenging to integrate the errors of multiple heterogeneous sketch instances to form the overall accuracy of an application, because the errors are quite different across sketch algorithms. For example, some algorithms produce false positives [8], while others introduce a relative error for each flow [20]. It is hard to characterize the errors in a uniform method.

Solutions (§5). The difficulty of integrating the errors of individual sketch instances motivates us to employ a benchmark-based parameter tuning. Specifically, AutoSketch formulates the parameter configuration into an optimization problem. Then it searches all possible configurations to find out the most efficient configuration. For each configuration, AutoSketch estimates its overall accuracy via benchmark experiments. AutoSketch proposes an efficient searching algorithm with several techniques to reduce the searching overhead. Note that our optimization tunes parameters of each operator to satisfy application-level accuracy intent, while existing configuration methods only focus on planning for placement and refinement [30] or configuration for single data structure [31]. Thus, their optimization objectives are all different.

Discussion. AutoSketch adopts static configuration instead of dynamic adjusting [63, 64] for two reasons. First, dynamic approaches require *precisely* estimating the resulting accuracy after adjusting. This is infeasible since existing theoretical analysis typically works for worst scenarios [21, 36, 57]. Second, recent studies on dynamic reconfiguration are not adopted by commodity switches yet [60, 102]. Thus, we leave this issue in future work.

4 Mapping to Sketch

4.1 Operator Decomposition

AutoSketch exploits Abstract Syntax Tree (AST) to decompose a UDF. An AST is a tree that characterizes the syntax structure of a specific function. AutoSketch utilizes AST to perform decomposition in three steps as follows.

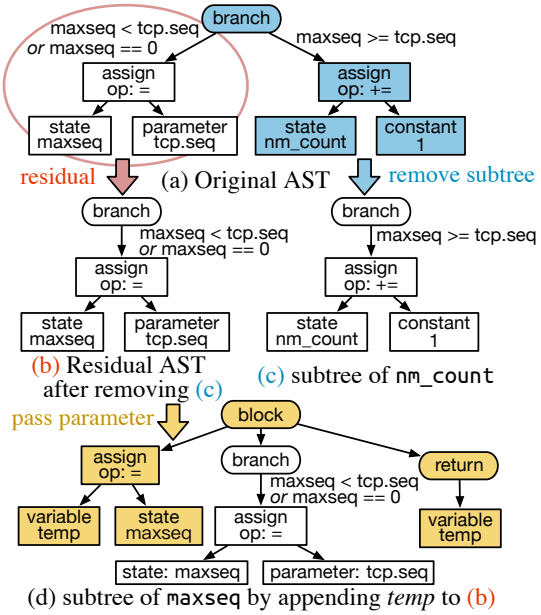


Figure 3: Example of nonmt (Application 1) decomposition.

Step 1: build AST. We first build an AST for the UDF. In an AST, a leaf node represents a state, a parameter, a constant, or a temporary variable. Non-leaf nodes correspond to statements that manipulate their child nodes. Recall that there are three types of statements: assignment statements, arithmetic statements, and conditional statements (§2). For an *assignment* or *arithmetic* node, its children refer to the operands. A *branch* node corresponds to a conditional statement. Its edges to the child nodes are associated with the judgment expressions. The first child refers to the If branch, while the second child indicates the Else branch (if any). For better organization, our AST further introduces a *block* node, which indicates that its children are executed sequentially. Figure 3(a) shows the AST of UDF nonmt in Application 1. The AST contains a branch node as the root. The two children are the assignment nodes for maxseq and nm_count derived from the If and Else branches, respectively. Due to space limitations, we omit the decomposition for maxseq < tcp.seq or maxseq == 0 but the procedure is similar.

Step 2: partition AST. Next, AutoSketch partitions the AST into several small ASTs, such that there is exactly one state being modified in each partitioned AST. Algorithm 1 illustrates the partitioning of an AST. Our basic idea is to repeatedly remove subtrees containing state modifications. Specifically, we enumerate each state and travel the AST (line 3). For each assignment node that modifies the state (lines 5-6), we build a subtree st comprising nodes from the root to this assignment node including its child nodes (line 7). Then, we remove nodes in the subtree from the original AST. The removal travels from the root to the assignment node (line 9). If a node does not appear in other root-to-leaf paths (line 10), we remove this node and all its descendant nodes (lines 11-12). Otherwise, for a node appearing in other root-to-leaf paths,

Algorithm 1 AST Partition

```
Input : The original AST of UDF
 1: function PARTITIONAST(ast)
 2:     partitioned_asts = []
 3:     for each state s maintained by UDF do
 4:         ST = []
 5:         for each assignment node u in ast do
 6:             if u modifies s then
 7:                 Build a subtree st from root to u (with u's children)
 8:                 Add st to ST
 9:                 for each node v from root to u do
10:                     if v does not appear in other root-to-leaf paths then
11:                         Remove v and its descendants from ast
12:                         break
13:         Construct a new_ast by merging the subtrees in ST
14:         Add new_ast to partitioned_asts
15:     return partitioned_asts
```

we keep the node in the AST, such that the connectivity of other paths is preserved. Note that there could be multiple statements modifying a state. Thus, we can partition multiple subtrees from the original AST for one state. In this case, we form a larger AST for this state by merging the duplicate nodes in the multiple subtrees (line 14).

Figure 3(b) and (c) illustrate the partitioning procedure for the AST of nonmt. We start with partitioning modifications on state nm_count. We identify the subtree modifying nm_count as shown in Figure 3(c) (also marked in blue in Figure 3(a)). In the residual AST in Figure 3(b), there is only one state maxseq, and thereby we have no need to remove subtree.

Step 3: pass parameters. Finally, AutoSketch passes state values among the partitioned ASTs because a partitioned AST may need to read state values from another AST. Since there exist multiple versions of the state values, we only pass the needed versions with temporary variables. In Figure 3, updating state nm_count depends on the value of state maxseq. Thus, we extend the AST of maxseq in Figure 3(b) to a new AST in Figure 3(d). Since the value of maxseq before updating is actually needed, we add an assignment statement that saves the value before updating in a variable *temp*. The variable is returned and passed to AST of nm_count after all statements are executed. We put more examples in Appendix.

Discussion. AutoSketch restricts the dependency among states in a groupby operator. For two states A and B, if updating A needs to access B, we say that A depends on B. In this case, the decomposition result must be unique: the stage that deploys partitioned AST of B must be before that of A. Thus, AutoSketch does not allow *mutual dependency*. By mutual dependency, we mean that the two states depend on each other. When mutual dependency occurs, AutoSketch cannot determine which state should be placed on the preceding stage. Then, it throws an error during compilation time.

4.2 Sketch Construction for UDFs

AutoSketch constructs a sketch-like structure for each decomposed UDF. As shown in §3.1, it maps each key into one cell

Algorithm 2 Update procedure of sketch-like structure

```
Input : The packet tuple t
Variables: Sketch-like structures C = map()
Variables: Bloom filter bf, Keyarray = [], l = 0
Variables: Incflags = map(), Decflags = map()        ▷ Initial value true
 1: function UPDATE(t)
 2:     Extract key k from t
 3:     if k ∉ bf then
 4:         Insert k to bf
 5:         if Keyarray is not full then
 6:             Keyarray[l] = k
 7:             l += 1
 8:         else
 9:             Send k to controller
10:     for each UDF in decomposed UDFs do
11:         SL = C[UDF]                  ▷ SL is the sketch-like structure of UDF
12:         for i = 1 to d do
13:             j = h_i(k)
14:             old_value = value of SL[i][j]
15:             Invoke UDF(SL[i][j], t)
16:             new_value = value of SL[i][j]
17:             if old_value > new_value then
18:                 Incflags[UDF]=false          ▷ non-increasing
19:             else if old_value < new_value then
20:                 Decflags[UDF]=false          ▷ non-decreasing
```

with hash functions. Then, AutoSketch updates the mapped cells with the UDF, or queries per-key information by aggregating the mapped cells. We elaborate on the detailed realizations of the update and query procedures as follows.

Update procedure. Recall that a state in a groupby operator is a collection of key-value pairs. Thus, in addition to updating *values* in the sketch-like structures, it also needs to store the keys. One possible solution is to store the keys within each sketch-like structure. However, this incurs overwhelming memory overheads, because there are duplicate keys in the multiple decomposed UDFs. Thus, AutoSketch separates the keys outside the sketch-like structures.

Algorithm 2 details the update procedure for the multiple decomposed UDFs. To separate keys from sketch values, it maintains an array Keyarray to store keys, a counter l indicating the length of Keyarray, and a Bloom Filter bf to check the existence of keys. Upon the arrival of a tuple, we extract its key k and check the occurrence of the key with bf (lines 2-3). If the key never appears, we insert the key into the bf and append the new key to the end of Keyarray (lines 4-7). If Keyarray is full, we send k to controllers (line 9). Then we update the sketch-like structures. Specifically, for each UDF, we hash the key to locate the cells containing the state of the key (line 13). We invoke the function to update the state (line 15). We maintain Decflags and Incflags for each UDF. The two variables indicate whether state values are monotonically decreasing or increasing, respectively (lines 17-20). The two variables will assist the query procedure in the next.

Query procedure. The query procedure is performed in the control plane. The controller periodically collects data plane information, including the sketch-like structures, the Keyarray, and the two variables indicating the monotonicity of the UDF. Then, the controller enumerates the keys in

Algorithm 3 Query procedure of sketch-like structure

```
Input: The key k, The monotonicity flag Decflag, Incflag
1: function QUERY(k, Decflag, Incflag)
2:     rets = []
3:     for i = 1 to d do
4:         SL = C[UDF]                  ▷ SL is the sketch-like structure of UDF
5:         j = h_i(k)
6:         r = SL[i][j]
7:         Add value of r to rets
8:     if Decflag then                  ▷ Monotonically decreasing
9:         return the maximal value in rets
10:    else if Incflag then             ▷ Monotonically increasing
11:        return the minimum value in rets
12:    else                             ▷ No monotonicity
13:        return the median value in rets
```

Keyarray to query their state values in the sketch-like structures (recall that a sketch-like structure corresponds to one state in the original UDF). Specifically, AutoSketch hashes a key to obtain the mapped cells in a sketch-like structure. Then it aggregates the cells to estimate the state value.

AutoSketch mitigates the errors caused by hash conflicts by leveraging the monotonicity of a UDF.

- Monotonically increasing: AutoSketch takes the minimum value among the mapped cells as the estimate, since all cells overestimate the true value of a key.
- Monotonically decreasing: AutoSketch takes the maximum value among the cells, similarly.
- No monotonicity: AutoSketch uses the median of the cells. The rationale comes from the Chernoff Bound [61]. Specifically, for each flow, the occurrences of overestimates and underestimates from the multiple cells are expected to be the same. Thus, using the median yields an accurate result.

Algorithm 3 summarizes the query procedure. The function takes key k and the monotonicity variables of the UDF as input. It hashes k into $SL[i][j]$ in the i-th row with $j = h_i(x)$ and records the value of state $SL[i][j]$ in rets (lines 2-7). Then it uses the monotonicity flags Decflag and Incflag to decide which value in rets to return (lines 8-13). We put the theoretical analysis of sketch-like structures in Appendix C.

Control plane overheads. Although the decomposition increases the number of stateful operators, it does not amplify the query overheads in the control plane. The reason is that the controller only needs to query the operators that form the eventual results. For example, in many applications that follow "Count-Distinct with Threshold" [18], only the last operator is of interest. Since the actually queried operators are more lightweight and much traffic is pruned in previous operators, we do not introduce additional control plane overheads.

4.3 Sketch Selection for Built-in Operators

AutoSketch supports multiple types of sketch algorithms to instantiate built-in operators, so as to adapt to diverse applications and accuracy intents. We leverage a *sampling* technique to select the most efficient algorithms for a given application and accuracy intent. Specifically, we evaluate different com-

binations of sketch algorithms, and select the combination that uses minimum resource usage to fulfill the accuracy. The evaluation uses synthetic workloads to perform benchmark analysis (see §5), so that we can estimate the actual resource-accuracy trade-off of the sketch algorithms. For each built-in operator, we generate several configurations for each potential algorithm that can realize this operator. We not only calculate the theoretical configuration suggested in the literature, but also randomly sample some other configurations.

Note that the sampling only addresses a limited number of configurations, which is acceptable for algorithm selection. Thus, we still need comprehensive configurations in §5. We put more optimizations for sketch mapping in Appendix D.

5 Searching for Configurations

5.1 Problem Formulation

AutoSketch aims to find out a configuration that satisfies input accuracy intent but incurs minimum resource usage. We state the problem as follows (the complete problem formulation is in Appendix E).

Application configuration. We define an application configuration c as a vector of $3n$ variables, $c = (d_1, w_1, s_1, d_2, w_2, s_2, ..., d_n, w_n, s_n)$, given that n is the number of mapped sketch instances. Each variable in the vector is a parameter that needs to be configured. For the i-th sketch instance, the two parameters d_i and w_i indicate the depth and width of the 2D sketch structure, respectively. s_i is the number of stages occupied by the i-th instance, which limits the maximal available resources for each sketch. Here, we omit the size of each element, because it is a constant determined while sketch-based mapping.

Constraints. A configuration needs to satisfy two types of constraints. The first type is about accuracy. It requires that the accuracy of the telemetry application should reach the user-specified accuracy intent. The other type of constraint ensures that the resource usage (stage, ALU, and memory) of a configuration should not exceed the available ones provided by PISA switches. Further, since ALUs and memory are associated with each stage, we also require that a configuration does not compromise per-stage resource restrictions.

Objective. The objective of parameter tuning is to find the configuration that satisfies all the constraints (including accuracy constraints and resource constraints) while incurring minimal resource usage. Currently, AutoSketch considers two types of resources: *register memory* and *stateful ALU*. The resource usage of any configuration c is quantified as a score: $SC(c) = \alpha n_{ALU} + \beta n_{mem}$, where n_{ALU} refers to the number of needed ALUs for the configuration c and n_{mem} refers to the used register memory. Here, α and β are two user-specified parameters to weight ALUs and memory. Note that the number of ALUs also relates to in-switch processing delay because it determines how many match-action tables each packet needs to go through. Thus, if we aim to minimize the processing delay instead of resource usage, we can simply set $\beta = 0$.

Algorithm 4 Searching Algorithm for Configuration

```
Input  : Telemetry application T composed of n mapped sketch instances
Input  : Accuracy intent EB specified by the user
Global variables : S = [], done = [], candidate = []
function SEARCH(T, EB)
    candidate = LHS_INIT(n)
    while candidate not empty do
        Take a configuration c from candidate
        Evaluate T by c
        Add c to done
        if c satisfys EB then
            Add c to S
        neighbors = CALCNEIGHBOR(c)
        for x in neighbors do
            if EXAMINE(x) then
                add x to candidate
    if S is not empty then
        return the configuration whose SC is minimal in S
    else                                   ▷ no configuration satisfies EB
        return the configuration whose accuracy is highest in done
```

Hardness. It remains challenging to adopt classical optimization techniques to solve it. As summarized in §3.2, (1) existing theoretical analysis of sketches considers the worst case; (2) it is challenging to integrate the errors of individual sketch instances into application-level accuracy intent. Thus, we cannot rely on theoretical analysis to estimate actual accuracy for practical workloads or configure parameters. On the other hand, some studies [49, 51, 98] advocate learning the configuration parameters by deep reinforcement learning [62]. However, as telemetry applications vary, it is unacceptable to train a model for each application separately.

5.2 Benchmark-based Searching

AutoSketch performs benchmark analysis to evaluate configuration accuracy and selects configurations accordingly. Specifically, AutoSketch employs several representative workloads. For each configuration, AutoSketch evaluates its accuracy with the workloads. If the resulting accuracy meets the accuracy intent in all workloads, we consider that this configuration can fulfill the intent in practice. Currently, AutoSketch provides several synthetic workloads for the benchmark. The synthetic workloads follow typical distributions of network traffic (e.g., power-law) and are generated with different parameters (e.g., different skewness for power-law distributions) to mimic various scenarios. In addition, we randomly inject additional network traffic in some intervals during the searching to simulate traffic bursts. Users can select the workloads based on their needs or use their own trace. Here, the programs required for the benchmark (including the decomposed `UDF`) are automatically generated by AutoSketch.

Obviously, the benchmark searching is time-consuming because the number of possible configurations is huge. Consider an example in which a sketch instance runs in a switch with four ALUs and 1 MB memory. Even in this simple ex-

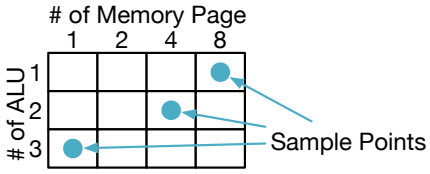


Figure 4: Latin Hypercube Sampling for 2D-space.

ample, the number of possible configurations is above 5 K (the detailed calculation is in Appendix F). Taking an application containing two sketch instances, the time usage of benchmark evaluation for one configuration takes 1-2 minutes under a 4 Gb workload. AutoSketch optimizes the searching procedure in the following paragraphs.

Searching procedure (Algorithm 4). The searching starts with invoking LHS_INIT (see below) to initialize a list of candidate configurations (line 2). Then AutoSketch iteratively pops one configuration, and performs benchmark experiments to evaluate its accuracy (lines 3-6). If this candidate fulfills the intent, it is saved in a set S (lines 7-8). For each candidate, AutoSketch also generates its neighboring configurations that incur similar resource overheads (line 9). Among these neighboring configurations, AutoSketch prunes useless ones and adds the residual to the candidate list (lines 10-12). After all the candidates are evaluated, if there exist configurations fulfilling the intent, AutoSketch returns the one with minimum resource usage (lines 13-14). Otherwise, AutoSketch returns the configuration with the highest accuracy (lines 15-16).

LHS-based configuration initialization. AutoSketch constructs initial configurations based on Latin Hypercube Sampling (LHS) [56]. LHS is a statistical method to generate representative sampled configurations in multi-dimensional spaces. It requires that the generated samples have distinct sampled values in every dimension. Figure 4 shows an example of 2D-space. In this case, LHS generates three sample points. Each row has exactly one sample point. Each column has at most one point. Note that the second column has no sample point because adding any point in this column causes conflicts with other points. In this way, LHS guarantees that the sampled points are evenly distributed in the parameter space. We leave the complete algorithm in Appendix G.

Hardware-aware configuration generation. After evaluating a candidate c, AutoSketch generates the neighboring configurations of c. Function CALCNEIGHBOR leverages the hardware features to reduce the number of configurations. First, existing switches allocate memory at the granularity of 16 KB pages. Thus, we require that the total size of a configuration aligns to the page size. Second, the range of a hash value is a power of two. Thus, we only consider the configurations in which per-row cell count is also a power of two. Specifically, the generation depends on whether c meets the accuracy intent. If c satisfies the intent, AutoSketch generates new configurations with fewer resources, by either decreasing the number of hash functions by one or halving the number of memory pages. Otherwise, if c does not fulfill the intent, AutoSketch either adds one hash function or doubles memory pages to generate configurations with more resources. We leave the pseudo code of CALCNEIGHBOR in Appendix H.

Configuration pruning. AutoSketch applies several rules to prune configurations. AutoSketch first examines unnecessary configurations, including (1) configurations that have been evaluated, (2) configurations whose resource usage exceeds a limit, and (3) configurations that consume more resources than a configuration already satisfying the intent.

In addition, AutoSketch stops when sufficiently good configurations are found. Specifically, if the initial configuration satisfies the intent, AutoSketch stops decreasing resources when a configuration fails to meet the intent. Otherwise, if the initial configuration does not satisfy the intent, AutoSketch stops when a satisfying configuration is found. Therefore, AutoSketch guarantees that it can always find a feasible solution that meets the accuracy intent if such configurations exist. Otherwise, AutoSketch throws an error.

6 Implementation

Interface. We leverage the syntax sugar of Python to implement our interface as a domain-specific language embedded in Python. Specifically, we define a *PacketStream* class and realize all operators as its member functions. For each function, both the input and output have the type PacketStream. This inherently realizes the chaining syntax as shown in §2. We plan to integrate OmniWindow [83] with our interface to support general window mechanisms in our future work.

Compiler. We provide two sketch algorithms for each stateful built-in operator. Specifically, a `reduce` operator can be mapped to a Count-Min Sketch [20] or a Count Sketch [12], while we use Bloom Filter [8] and Counting Bloom Filter [23] for `distinct`. We leverage multi-threading techniques to parallelize the time-consuming benchmark experiments. Currently, AutoSketch supports two types of PISA-based target backends, namely Tofino [89] and BMv2 [9]. Our implementation realizes the idea of this paper (i.e., constructing and optimizing sketch instances). Other program optimization (e.g., Chipmunk [26], BitSense [22]) is out of our scope.

Runtime. We realize a control plane runtime and a data plane runtime to deploy AutoSketch. The data plane runtime installs the output P4 programs of AutoSketch into switches and sends sketch structures to the controller. The control plane runtime invokes the query procedures and reports results to users.

Multiple applications and distributed deployment. To deploy multiple applications over distributed devices, we merge the DAGs of each application into a large application and abstract the switch along the packet transmission path into a big pipeline [16, 25]. We leave the searching for this large application on the big pipeline abstraction with different accuracy intents as our future work.

Telemetry Application	Lines of Code			
	Sonata	Marple	AS	AS-P4
Used in existing query-driven telemetry (see §7.2)				
New TCP Conns.(NC) [97]	6	8	7	192
TCP Incomplete Flows (IF) [97]	12	16	15	237
Port Scan (PS) [40]	6	11	6	285
DDoS (DD) [96]	9	14	6	277
TCP timeouts (TO) [67] *	-	6	7	567
TCP non-monotonic (NM) [67] *	-	7	9	618
TCP out-of-sequence (OOS) [67] *	-	6	7	618
TCP SYN Flood (SF) [97] *	-	17	20	635
Used in existing sketch-based telemetry (see §7.3)				
Superspreader (SP) [96]	6	11	6	271
Cardinality (CD) [34]	3	8	4	227
Heavy Hitter (HH) [20]	5	10	5	271

Table 2: (Exp#1) LoC of telemetry applications (* indicates that the application contains `UDF`).

7 Evaluation

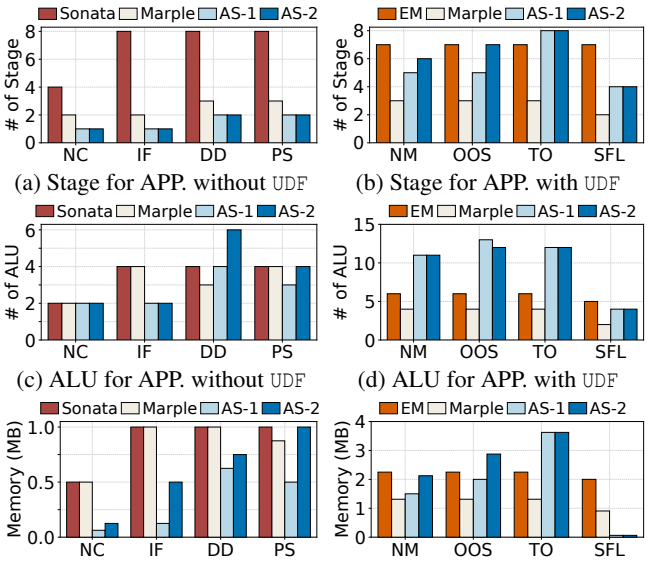
We conduct experiments that compare AutoSketch with state-of-the-art telemetry solutions in various aspects. We summarize our findings for AutoSketch: (1) It implements 11 telemetry applications in <20 lines of code without concerns with parameter configurations (Exp#1 and Exp#5); (2) It satisfies the accuracy intent of at least 95% precision, 97% recall, and at most 2.8% ARE for all the applications with only less than 6 ALUs, 1.06 MB memory and 115 KB/s bandwidth on average (Exp#2, Exp#3 and Exp#4); (3) It consumes around 50% switch resource usage to achieve the intent compared to most classical sketch algorithms (Exp#6 and Exp#7); (4) It achieves better effectiveness than query planning of Sonata and scale to large-volume traffic (Exp#8); (5) It efficiently finds out configurations with high accuracy (Exp#9 and Exp#10 in Appendix I). We put the complete results in Appendix J.

7.1 Setup

Telemetry Applications. Our evaluation considers 11 representative telemetry applications in Table 2. The applications are commonly used to evaluate telemetry solutions, including both query-driven telemetry [30, 67, 105] and sketch-based telemetry. Among these applications, TO, NM, OOS and SFL contain `UDF`s. We configure the applications as described in the literature. We put their realizations in Appendix K.

Workloads. We use CAIDA traces from 2016 to 2019. For stress testing, we simultaneously run PktGen [72] on eight servers to replay the traffic at 40x speed: every 100 ms interval in our result corresponds to 4 s traffic of the origin trace. Thus, each interval contains around 200 K active flows and 6 Gb traffic (60 Gbps). Further, we inject additional 20 K flows at 10% intervals to evaluate AutoSketch under traffic bursts.

Accuracy metrics. For all applications except for CD, we measure both recall and precision. For TO, NM, and OOS that measure per-flow packet count of interest, we additionally evaluate their ARE. For CD, we measure the relative error (RE): $\frac{n-\hat{n}}{n}$, where n and $\hat{n}$ are the true and estimated number of flows, respectively.



(a) Stage for APP. without `UDF` (b) Stage for APP. with `UDF`
(c) ALU for APP. without `UDF` (d) ALU for APP. with `UDF`
(e) Memory for APP. without `UDF` (f) Memory for APP. with `UDF`

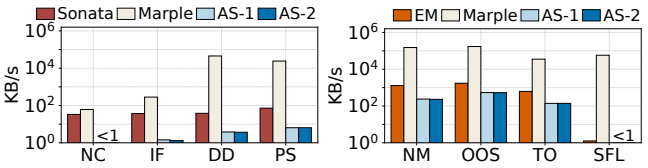
Figure 5: (Exp#2) Switch resource usage against query-driven telemetry.

Resource metrics. We consider three types of switch resources: (1) *on-chip memory* used to maintain the states of stateful operators; (2) stateful ALU used to calculate hash functions and update state values; and (3) stage in the PISA switch associated with its own ALUs and memory. Note that we do not report the results on PHV because the packet tuples passing across stages mainly include the parsed fields in packet headers, whose overheads are limited.

Accuracy intents. We use two accuracy intents for AutoSketch: one aims to achieve 95% recall, 95% precision, and 3% ARE, while the other requires 99% recall, 99% precision, and 1% ARE. Both intents set the confidence as 95%. Therefore, we report the 95-th percentiles among the intervals, with the minimum and maximum values as the error bars. For each application, we apply the two intents to generate their P4 programs, denoted by AS-1 and AS-2, respectively.

7.2 Compare with Query-driven Telemetry

Methodology. We compare AutoSketch with two query-driven telemetry systems, Sonata [30] and Marple [67]. For Sonata, we configure each stateful operator with 2^{16} counters and utilize the refinement plan $* \rightarrow 8 \rightarrow 32$ as in its open-source prototype. For Marple, we implement its key-value cache and evict old keys to handle hash collisions. We configure 2^{16} cache slots as in the paper [67]. For applications with `groupby` operators, we also consider a naive approach, namely *entire mapping* (EM), that converts each `UDF` operator entirely into a sketch-like structure (see §3.1). Since EM does not resolve hash collisions, we configure the structure with 256 K cells to handle the >200 K flows per interval.



(a) Applications without UDF. (b) Applications with UDF.

Figure 6: (Exp#3) Bandwidth usage against query-driven telemetry.

(Exp#1) Expressiveness. We first compare the expressiveness of AutoSketch with Sonata and Marple, both of which regard expressiveness as the "first-class citizens" in their designs. Table 2 shows that AutoSketch needs similar lines of code (LoC) to implement the eleven telemetry applications. On the other hand, it needs hundreds of lines if we directly use P4. Further, AutoSketch allows expressing accuracy intents, while the Sonata and Marple do not have such features.

(Exp#2) Resource usage. This experiment evaluates the resource usage of AutoSketch in switches. We deploy each solution atop a Barefoot Wedge 100B-32X (Tofino) PISA switch [89], and then measure the incurred resource usage. We consider eight telemetry applications in this experiment. Recall that Sonata does not support UDFs, while EM only applies to UDFs. Thus, we present the results of the eight applications separately, according to the existence of UDFs.

Figure 5 presents the results. For applications without UDFs, Sonata occupies much more stages than others due to its refinement mechanism which reduces the traffic sent to the stream processor. AutoSketch consumes less or the same amount of ALUs and memory as Sonata and Marple except for DD under AS-2, but achieves lower bandwidth overhead (Exp#3) and higher accuracy than Sonata (Exp#4). For example, the memory overheads of Sonata and Marple for NC and IF are 8 times higher than AS-1. For applications with UDFs, EM incurs six more stages, nine more ALUs, and 1.6 MB more memory than AS-1 over the total resources of four applications and suffers from serious hash conflicts and incurs low accuracy (see Exp#4). However, AutoSketch consumes two to fine more stages, seven to nine more ALUs and 15%-175% more memory than Marple. The reason is that the user-defined states in UDFs are complicated, thereby AutoSketch enlarges the resource usage to mitigate hash conflicts. However, Marple has to deal with hash conflicts by sending excessive traffic to the control plane (see Exp#3).

(Exp#3) Bandwidth usage. Figure 6 presents the traffic sent to the control plane of each approach. Although the current bandwidth of the control plane is in the magnitude of 100 Gbps, it must hold numerous data plane devices. Thus, it is critical to save control plane bandwidth [15, 30, 46, 67]. AutoSketch incurs 115 KB/s bandwidth for all the applications, while Sonata, EM and Marple consume 45.1 KB/s, 913.2 KB/s, and 485.2 MB/s, respectively. Here, Sonata reduces the bandwidth usage at the cost of higher switch resources (see Exp#2). The reasons why EM and Marple incur higher bandwidth consumption than AutoSketch are as follows. For Marple, it needs to evict conflicting packets to the control plane. For EM, it suffers from severe hash conflicts, so a large number of normal packets are wrongly identified as anomalies and then sent to the control plane.

(Exp#4) Accuracy. This experiment compares the accuracy of AutoSketch with Marple, Sonata, and EM. Figures 7(a) and 7(b) present the results of the applications without UDFs. Marple achieves zero error for all applications because they completely handle hash conflicts. However, it incurs excessive resource overheads and bandwidth consumption (see Exp#2 and Exp#3). Sonata well ensures the precision of all applications but fails at the recall metric. The reason is that Sonata decreases the counting value during the refinement process and many abnormal flows are not wrongly identified as normals. AutoSketch achieves high accuracy. In particular, it satisfies our two accuracy intents for all applications.

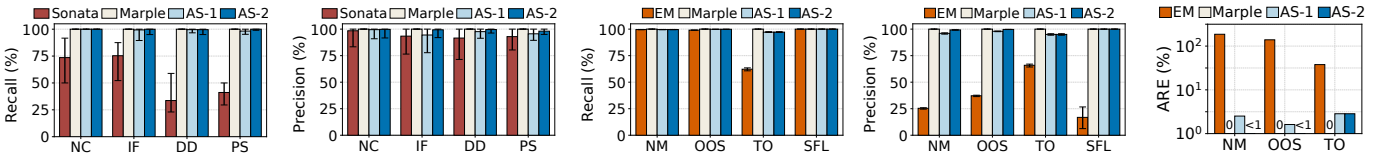
Figures 7(c) to 7(e) show the results on the applications with UDFs. Marple achieves zero error for the four applications. Except for TO, all AutoSketch applications meet both intents. Here, TO also achieves 95% precision and 97% recall, which is close to the intent. However, the recall of TO for EM is only 62% and the average precision for EM is even much lower (only 36%). EM also incurs high ARE: the ARE for NM even exceeds 186%. The reason is that some states are extremely sensitive to hash conflicts. When binding all states in one sketch-like structure, hash conflicts will seriously compromise their accuracy.

7.3 Compare with Sketch-based Telemetry

Methodology. We compare AutoSketch with existing sketch algorithms. In Table 2, we select three applications that have been extensively studied by existing sketch algorithms. We consider two general-purpose sketch-based solutions FlowRadar (FR) [50] and OpenSketch (OS) [96] that can support all three applications. FR proposes a general sketch structure to support various applications. For OS, we build the algorithms in P4 as suggested by the original paper [96].

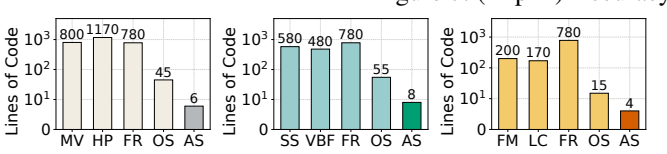
In addition to FR and OS, we also select two state-of-the-art sketch algorithms that are specifically designed for each application: in heavy hitter detection (HH), we compare AutoSketch with MV-Sketch (MV) [87] and HashPipe (HP) [79]; in superspreader detection (SP), we compare AutoSketch with SpreadSketch (SS) [86] and Vector Bloom Filter (VBF) [52]; in cardinality estimation, we compare AutoSketch with FM-Sketch (FM) [24] and Linear Counting (LC) [91].

We realize these algorithms based on their published designs except MV and SS that provide open-source implementation. We also apply AS-1 and AS-2 for AutoSketch and the sketch algorithms. We follow the guidelines in the literature to configure these sketches. Here, we present the results of AS-1. The results of AS-2 are in Appendix J. We do not present the bandwidth usage of sketch-based solutions because they only



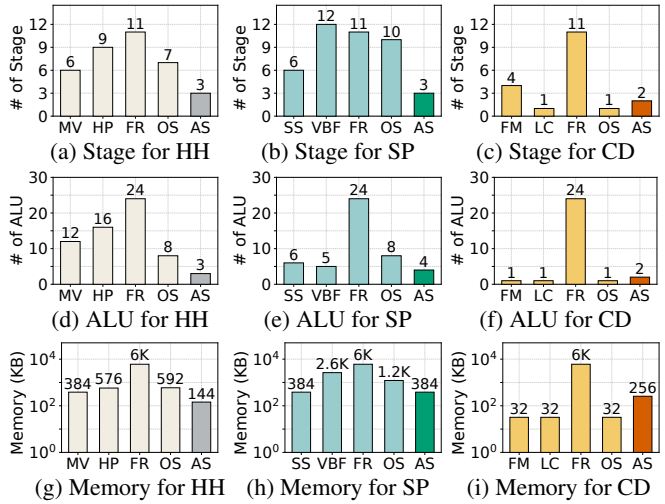
(a) Recall for APP. w/o `UDF` (b) Precision for APP. w/o `UDF` (c) Recall for APP. w/ `UDF` (d) Precision for APP. w/ `UDF` (e) ARE for APP. w/ `UDF`

Figure 7: (Exp#4) Accuracy against query-driven telemetry.



(a) Heavy Hitter (b) Superspreader (c) Cardinality

Figure 8: (Exp#5) Expressiveness against sketch-based telemetry.



(a) Stage for HH (b) Stage for SP (c) Stage for CD

(d) ALU for HH (e) ALU for SP (f) ALU for CD

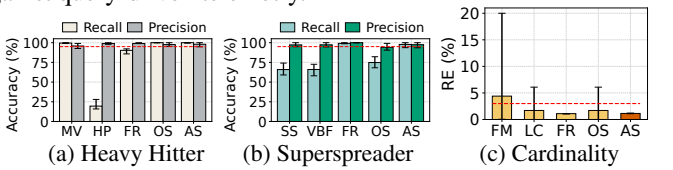
(g) Memory for HH (h) Memory for SP (i) Memory for CD

Figure 9: (Exp#6) Switch resource usage against sketch-based telemetry.

transfer the sketch structures to the controller. Thus, we can infer the bandwidth from their memory usage in Exp#6.

(Exp#5) Expressiveness. We first compare the expressiveness of AutoSketch with the sketch algorithms. Figure 8 shows that AutoSketch only needs less than ten LoC to express each telemetry application. However, existing sketch algorithms require hundreds of lines of P4 code. Note that OS provides one library that includes some pre-built sketches. Therefore, OS can express one telemetry application in dozens of configuration LoC. Further, AutoSketch automatically generates corresponding P4 programs with parameter configurations, while sketch algorithms need many efforts to tune parameters.

(Exp#6) Resource usage. We compare the resource usage of AutoSketch with the sketch algorithms. Figures 9(a) to 9(i) show that AutoSketch only consumes 50% switch resources compared to most existing sketch algorithms except for LC and FM (OS utilizes LC to realize cardinality). The reason is that existing sketch algorithms typically consider worst-case scenarios, while AutoSketch selects the most suitable configuration via benchmarking-based searching. FM and LC use very limited resources because they only support the simple



(a) Heavy Hitter (b) Superspreader (c) Cardinality

Figure 10: (Exp#7) Accuracy against sketch-based telemetry.

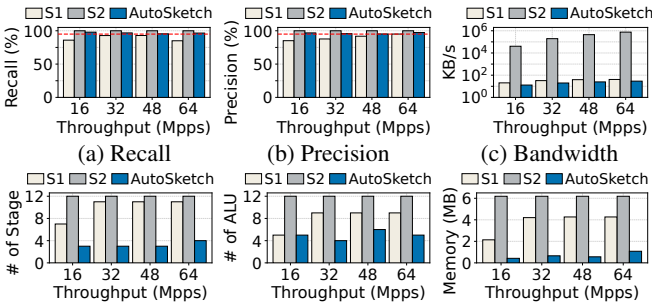
cardinality estimation at the cost of occasional unacceptable error (Exp#7). FR incurs much higher resource usage since it has complicated structures to support general telemetry tasks. HP and VBF take the multi-level data structure that needs more stages than other solutions. OS incurs more resources than AutoSketch to detect heavy hitters and super spreaders, since it adopts reversible sketch [75] to maintain flowkey.

(Exp#7) Accuracy. We compare the accuracy of AutoSketch with the sketch algorithms (the red dotted line in the figure indicates the accuracy intent). For the heavy hitter detection (Figure 10(a)), all solutions can reach 95% precision. However, the recall of HP and FR is only 20% and 90%, respectively. The reason is that their desired resources exceed the hardware capacity. MV achieves similar accuracy as AutoSketch, but needs more resources (see Exp#6). For the superspreader detection (Figure 10(b)), SS, VBF, and OS reach high precision but cannot fulfill the recall goal, since they are sensitive to hash conflicts. For the cardinality estimation (Figure 10(c)), FM cannot satisfy the intent and its RE suffers from terribly high error. AutoSketch meets the accuracy intent in all the telemetry applications for two reasons. First, our parameter configuration avoids worst-case parameters in traditional theoretical analysis. Second, our randomly injecting additional traffic adapts to traffic bursts.

7.4 Micro Benchmarks

(Exp#8) Parameter tuning. We compare our benchmark-based parameter tuning with the query planning of Sonata. We consider two versions of Sonata, denoted by S1 and S2, respectively. S1 is the default setup in Sonata's open-source prototype [69] that does not deal with hash collisions. S2 resolves hash conflicts by mapping each flow to d counters and evicting a flow to the remote stream processor if hash collisions occur (c.f. §3.1 of [30]). We follow the guidelines in [30] to make the partition and refinement decision via the ILP. We stress test all systems with different packet rates.

We present the DDoS detection under AS-1 due to space limitations. Figure 11 shows the results. Even though S1 carefully tunes the data structure size and the refinement plan ($* \rightarrow 16 \rightarrow 32$), it still fails to satisfy AS-1 (Figure 11(a) and



(a) Recall (b) Precision (c) Bandwidth
(d) Stage (e) ALU (f) Memory
Figure 11: (Exp#8) Parameter tuning.

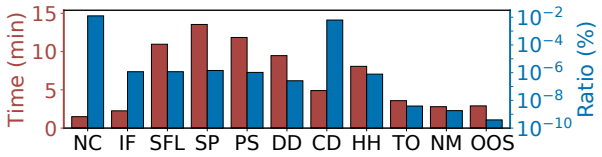


Figure 12: (Exp#9) Searching cost.

11(b)) because it does not resolve hash conflicts. In contrast, S2 achieves zero error at the cost of higher switch resource and bandwidth overhead. Compared to S1 and S2, AutoSketch satisfies the accuracy intent with lower switch resource and bandwidth overhead. The reason is that although Sonata efficiently determines offloaded operators, it lacks the optimization for individual operators. On the opposite, AutoSketch targets parameter tuning for each operator. The two approaches are orthogonal to each other, which can be combined to realize both benefits.

(Exp#9) Searching costs. We evaluate the cost of the benchmark-based configuration searching algorithm. Figure 12 presents the searching time (red bar) and the ratio between the number of configurations actually evaluated and the number of all possible configurations (blue bar). As in §5.2, the number of possible configurations is extraordinarily huge. Our algorithm greatly prunes the number of evaluated configurations by three to nine magnitudes. In particular, our searching algorithm leverages hardware features to prune configurations (see §5), which decrease the candidate configurations by at most six magnitudes. Other techniques further reduce by two magnitudes. All the applications converge a configuration close to the best solution (see Exp#10 in Appendix I) within 15 minutes using a single core, with an average time of 6.5 minutes. It is acceptable because the searching is performed once before an application is deployed.

8 Related Works

Query-driven telemetry. Existing studies on query-driven telemetry can be categorized into two classes. The first class is based on text-based expression [4, 66, 95, 97]. Specifically, PathQuery [66] focuses on the path-based traffic monitoring. NetQRE [97] combines application-level aggregation operations with regular-expression-like pattern matching. dShark [95] leverages the json-based telemetry language to declare fields and then invoke callback functions for further processing. The second class provisions a collection of operators for users to form telemetry applications [19, 29, 30, 38, 45, 47, 67, 105]. AutoSketch follows the line of operator-based telemetry with two enhancements: the accuracy intent in the language, and the automatic compiler to generate sketch-based programs.

Telemetry algorithms. There are three classes of telemetry algorithms. Sampling algorithms [11,41,73,76,77,82] consume limited resources but suffer from high errors. Counter-based approaches [6, 79] only address top-k flows, thereby failing to support generic applications. Sketch algorithms maintain compact data structures that allow multiple flows to share counters [1, 28, 35, 50, 54, 55, 94, 96, 101, 103]. With proper parameters, they achieve both high accuracy and resource efficiency. However, they suffer from heavy implementation and configuration burdens. AutoSketch hides such complexity with its programmability.

Telemetry architectures. Host-based telemetry systems [27, 42,65,84] suffer from limited network visibility. Switch-based systems [7, 17, 39, 59, 68, 71, 80, 81, 88, 90, 104] process traffic with high-speed ASICs but suffer from the limited memory. Recent studies often combine different network entities. SwitchPointer [85] utilizes the switch memory as a directory that points to the telemetry data on end-hosts. Sonata [30] abstracts the switch and end-host into a big streaming processor. Marple [67] treats the switch memory as the cache of the key-value store on end-hosts. MOZART [53] coordinates the switches and end-hosts to select and monitor traffic. OmniMon [37] integrates the capabilities of end-hosts, switches, and the controller. AutoSketch is orthogonal to these works.

9 Conclusion

This paper proposes AutoSketch, a network telemetry solution that bridges the gap between query-driven telemetry and sketch-based telemetry algorithms. AutoSketch extends the existing operator-based interface to allow users to specify the accuracy intent. It designs a compiler to translate the stateful operators in telemetry applications into sketch instances in the data plane. AutoSketch automatically configures parameters to fulfill the accuracy intent. Experiments demonstrate that AutoSketch achieves high expressiveness and high accuracy with limited hardware resources.

Acknowledgements

We thank our shepherd, Alan Zaoxing Liu, and the anonymous reviewers for their valuable comments. The work was supported in part by National Key R&D Program of China (2019YFB1802600), Joint Funds of the National Natural Science Foundation of China (U20A20179), National Natural Science Foundation of China (62172007).

References

[1] Anup Agarwal, Zaoxing Liu, and Srini Seshan. HeteroSketch: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *Proc. of USENIX NSDI*, 2022.

[2] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. In *Proc. of ACM PODS*, 2012.

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proc. of ACM EuroSys*, 2013.

[4] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proc. of ACM SIGCOMM*, 2016.

[5] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *Proc. of ACM SIGCOMM*, 2016.

[6] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C Luizelli, and Erez Waisbard. Constant Time Updates in Hierarchical Heavy Hitters. In *Proc. of ACM SIGCOMM*, 2017.

[7] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proc. of ACM SIGCOMM*, 2020.

[8] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of ACM*, 13, 1970.

[9] BMv2 PISA switch. https://github.com/p4lang/behavioral-model.

[10] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proc. of ACM SIGCOMM*, 2013.

[11] Marco Canini, Damien Fay, David J. Miller, Andrew W. Moore, and Raffaele Bolla. Per Flow Packet Sampling for High-Speed Network Monitoring. In *Proc. of COMSNETS*, 2009.

[12] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312:3–15, 2004.

[13] Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. One primitive to diagnose them all: Architectural support for internet diagnostics. In *Proc. of EuroSys*, 2017.

[14] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proc. of ACM SIGCOMM*, 2016.

[15] Xiang Chen, Qun Huang, Peiqiao Wang, Hongyan Liu, Yuxin Chen, Dong Zhang, Haifeng Zhou, and Chunming Wu. Mtp: Avoiding control plane overload with measurement task placement. In *Proc. of IEEE INFOCOM*, 2021.

[16] Xiang Chen, Hongyan Liu, Qun Huang, Peiqiao Wang, Dong Zhang, Haifeng Zhou, and Chunming Wu. Speed: Resource-efficient and high-performance deployment for data plane programs. In *Proc. of IEEE ICNP*, 2020.

[17] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *Proc. of CoNEXT*, 2019.

[18] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proc. of ACM SIGCOMM*, 2020.

[19] Xin Cheng, Zhiliang Wang, Shize Zhang, Xin He, and Jiahai Yang. Intstream: An intent-driven streaming network telemetry framework. In *Proc. of IEEE CNSM*, 2021.

[20] Graham Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, pages 58–75, 2005.

[21] Xenofontas Dimitropoulos, Paul Hurley, and Andreas Kind. Probabilistic lossy counting: an efficient algorithm for finding heavy hitters. *Proc. of ACM SIGCOMM CCR*, 38, 2008.

[22] Rui Ding, Shibo Yang, Xiang Chen, and Qun Huang. Bitsense: Universal and nearly zero-error optimization for sketch counters with compressive sensing. In *Proc. of ACM SIGCOMM*, 2023.

[23] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. on Networking*, 8:281–293, 2000.

[24] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31:182–209, 1985.

[25] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proc. of ACM SIGCOMM*, 2020.

[26] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *Proc. of ACM SIGCOMM*, 2020.

[27] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. SIMON: A simple and scalable method for sensing, inference and measurement in data center networks. In *Proc. of USENIX NSDI*, 2019.

[28] Junzhi Gong, Tong Yang, Haowei Zhang, Hao Li, Steve Uhlig, Shigang Chen, Lorna Uden, and Xiaoming Li. HeavyKeeper: An accurate algorithm for finding top-k elephant flows. In *Proc. of USENIX ATC*, 2018.

[29] Arpit Gupta, Rüdiger Birkner, Marco Canini, Nick Feamster, Chris Mac-Stoker, and Walter Willinger. Network monitoring as a streaming analytics problem. In *Proc. of ACM HotNets*, 2016.

[30] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *Proc. of ACM SIGCOMM*, 2018.

[31] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular switch programming under resource constraints. In *Proc. of USENIX NSDI*, 2022.

[32] Chi-Yao Hong, Matthew Caesar, Nick Duffield, and Jia Wang. Tiresias: Online anomaly detection for hierarchical operational network data. In *Proc. of IEEE ICDCS*, 2012.

[33] Ningning Hu, Li (Erran) Li, Zhuoqing Morley Mao, Peter Steenkiste, and Jia Wang. Locating internet bottlenecks: Algorithms, measurements, and implications. In *Proc. of ACM SIGCOMM*, 2004.

[34] Qun Huang, Xin Jin, Patrick P C Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust Network Measurement for Software Packet Processing. In *Proc. of ACM SIGCOMM*, 2017.

[35] Qun Huang and Patrick P. C. Lee. A Hybrid Local and Distributed Sketching Design for Accurate and Scalable Heavy Key Detection in Network Data Streams. *Computer Networks*, 91:298–315, 2015.

[36] Qun Huang, Patrick P. C. Lee, and Yungang Bao. SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *Proc. of ACM SIGCOMM*, 2018.

[37] Qun Huang, Haifeng Sun, Patrick P. C. Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proc. of ACM SIGCOMM*, 2020.

[38] Arthur S. Jacobs, Ricardo J. Pfitscher, Rafael H. Ribeiro, Ronaldo A. Ferreira, Lisandro Z. Granville, Walter Willinger, and Sanjay G. Rao. Hey, lumi! using natural language for Intent-Based network management. In *Proc. of USENIX ATC*, 2021.

[39] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. Burstradar: Practical real-time microburst monitoring for datacenter networks. In *Proc. of ACM ApSys*, 2018.

[40] Jaeyeon Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proc. of IEEE Symposium on Security and Privacy*, 2004.

[41] Srikanth Kandula and Ratul Mahajan. Sampling Biases in Network Path Measurements and What To Do About It. In *Proc. of ACM IMC*, 2009.

[42] Khandelwal, Anurag and Agarwal, Rachit and Stoica, Ion. Confluo: Distributed Monitoring and Diagnosis Stack for High-Speed Networks. In *Proc. of USENIX NSDI*, 2019.

[43] Daniel Kopp, Matthias Wichtlhuber, Ingmar Poese, Jair Santanna, Oliver Hohlfeld, and Christoph Dietzel. Ddos hide & seek: On the effectiveness of a booter services takedown. In *Proc. of ACM IMC*, 2019.

[44] Jan Kučera, Ran Ben Basat, Mário Kuka, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. Detecting routing loops in the data plane. In *Proc. of ACM CoNEXT*, 2020.

[45] Paolo Laffranchini, Luis Rodrigues, Marco Canini, and Balachander Krishnamurthy. Measurements as first-class artifacts. In *Proc. of IEEE INFOCOM*, 2019.

[46] Jonatan Langlet, Ran Ben-Basat, Sivaramakrishnan Ramanathan, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. Zero-cpu collection with direct telemetry access. In *Proc. of ACM HotNets*, 2021.

[47] Christopher Leet, Robert Soulé, Yang Richard Yang, and Ying Zhang. Flow algebra: Towards an efficient, unifying framework for network management tasks. In *Proc. of IEEE INFOCOM*, 2021.

[48] Jakub Lemiesz. On the algebra of data sketches. *Proc. VLDB Endowment*, 14:1655–1667, 2021.

[49] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. of VLDB Endow.*, 2019.

[50] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A Better NetFlow for Data Centers. In *Proc. of USENIX NSDI*, 2016.

[51] Zhao Li, Haifeng Sun, Zheng Xiong, Qun Huang, Zehong Hu, Ding Li, Shasha Ruan, Hai Hong, Jie Gui, Jintao He, Zebin Xu, and Yang Fang. Noah: Reinforcement-learning-based rate limiter for microservices in large-scale e-commerce services. *IEEE Transactions on Neural Networks and Learning Systems*, 34(9):5403–5417, 2023.

[52] W. Liu, W. Qu, J. Gong, and K. Li. Detection of superpoints using a vector bloom filter. *IEEE Trans. on Information Forensics and Security*, 11:514–527, 2016.

[53] Xuemei Liu, Meral Shirazipour, Minlan Yu, and Ying Zhang. MOZART: Temporal Coordination of Measurement. In *Proc. of ACM SOSR*, 2016.

[54] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *Proc. of ACM SIGCOMM*, 2019.

[55] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proc. of ACM SIGCOMM*, 2016.

[56] Beckman RJ McKay MD and Conover WJ. A comparison of the three methods for selecting values of input variable in the analysis of output from a computer code. *Technometrics;(United States)*, 1979.

[57] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proc. of ICDT*, 2005.

[58] Ruijie Miao, Fenghao Dong, Yikai Zhao, Yiming Zhao, Yuhan Wu, Kaicheng Yang, Tong Yang, and Bin Cui. Sketchconf: A framework for automatic sketch configuration. In *Proc. of IEEE ICDE*, 2023.

[59] Chris Misa, Ramakrishnan Durairajan, Reza Rejaie, and Walter Willinger. Revisiting network telemetry in coin: A case for runtime programmability. *IEEE Network*, 35:14–20, 2021.

[60] Chris Misa, Walt O'Connor, Ramakrishnan Durairajan, Reza Rejaie, and Walter Willinger. Dynamic scheduling of approximate telemetry queries. In *Proc. of USENIX NSDI*, 2022.

[61] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[62] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 2015.

[63] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Proc. of ACM SIGCOMM*, 2014.

[64] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. SCREAM: Sketch Resource Allocation for Software-defined Measurement. In *Proc. of ACM CoNEXT*, 2015.

[65] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *Proc. of ACM SIGCOMM*, 2016.

[66] Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker. Compiling Path Queries. In *Proc. of USENIX NSDI*, 2016.

[67] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *Proc. of ACM SIGCOMM*, 2017.

[68] Inband network telemetry. http://p4.org/wp-content/uploads/fixed/INT/INT-current-spec.pdf.

[69] Sonata open-source repository. https://github.com/Sonata-Princeton/SONATA-DEV.

[70] p4c: P4 Compiler. https://github.com/p4lang/p4c.

[71] Mengying Pan, Robert MacDavid, Shir Landau-Feibish, and Jennifer Rexford. Memory-efficient membership encoding in switches. In *Proc. of ACM SOSR*, 2020.

[72] PktGen. https://pktgen-dpdk.readthedocs.io.

[73] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *Proc. of ACM SIGCOMM*, 2014.

[74] Arjun Roy, Deepak Bansal, David Brumley, Harish Kumar Chandrappa, Parag Sharma, Rishabh Tewari, Behnaz Arzani, and Alex C. Snoeren. Cloud datacenter sdn monitoring: Experiences and challenges. In *Proc. of ACM IMC*, 2018.

[75] Robert Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter Dinda, Ming Yang Kao, and Gokhan Memik. Reversible Sketches: Enabling Monitoring and Analysis over High-Speed Data Streams. *Proc. of IEEE/ACM Trans. on Networking*, pages 1059–1072, 2007.

[76] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. cSAMP: A System for Network-Wide Flow Monitoring. In *Proc. of USENIX NSDI*, 2008.

[77] Vyas Sekar, Michael K Reiter, and Hui Zhang. Revisiting the Case for a Minimalist Approach for Network Flow Monitoring. In *Proc. of ACM IMC*, 2010.

[78] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with cascara. In *Proc. of USENIX NSDI*, 2021.

[79] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-Hitter Detection Entirely in the Data Plane. In *Proc. of ACM SOSR*, 2017.

[80] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proc. of ACM EuroSys*, 2018.

[81] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *Proc. of USENIX ATC*, 2018.

[82] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. OpenSample: A Low-Latency, Sampling-Based Measurement Platform for Commodity SDN. In *Proc. of ICDCS*, 2014.

[83] Haifeng Sun, Jiaheng Li, Jintao He, Jie Gui, and Qun Huang. Omniwindow: A general and efficient window mechanism framework for network telemetry. In *Proc. of ACM SIGCOMM*, 2023.

[84] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *Proc. of USENIX OSDI*, 2016.

[85] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed Network Monitoring and Debugging with SwitchPointer. In *Proc. of USENIX NSDI*, 2018.

[86] Lu Tang, Qun Huang, and Patrick P. C. Lee. Spreadsketch: Toward invertible and network-wide detection of superspreaders. In *Proc. of IEEE INFOCOM*, 2020.

[87] Lu Tang, Qun Huang, and Patrick PC Lee. Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams. In *Proc. of IEEE INFOCOM*, 2019.

[88] Ross Teixeira, Rob Harrison, Arpit Gupta, and Jennifer Rexford. Packetscope: Monitoring the packet lifecycle inside a switch. In *Proc. of ACM SOSR*, 2020.

[89] Tofino. https://www.barefootnetworks.com/products/brief-tofino/.

[90] Weitao Wang, Xinyu Crystal Wu, Praveen Tammana, Ang Chen, and TS Eugene Ng. Closed-loop network performance monitoring and diagnosis with SpiderMon. In *Proc. of USENIX NSDI*, 2022.

[91] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *Proc. of ACM TODS*, 15:208–229, 1990.

[92] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In *Proc. of USENIX NSDI*, 2019.

[93] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. In *Proc. of ACM SIGCOMM*, 2014.

[94] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Proc. of ACM SIGCOMM*, 2018.

[95] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces. In *Proc. of USENIX NSDI*, 2019.

[96] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *Proc. of USENIX NSDI*, 2013.

[97] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative Network Monitoring with NetQRE. In *Proc. of ACM SIGCOMM*, 2017.

[98] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proc. of ACM SIGMOD*, 2019.

[99] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. *Proc. of NDSS*, 2020.

[100] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of ACM IMC*, 2017.

[101] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proc. of ACM SIGCOMM*, 2021.

[102] Hao Zheng, Chen Tian, Tong Yang, Huiping Lin, Chang Liu, Zhaochen Zhang, Wanchun Dou, and Guihai Chen. Flymon: Enabling on-the-fly task reconfiguration for network measurement. In *Proc. of ACM SIGCOMM*, 2022.

[103] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proc. of ACM SIGMOD*, 2018.

[104] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In *Proc. of ACM SIGCOMM*, 2020.

[105] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. Newton: Intent-driven network traffic monitoring. In *Proc. of CoNEXT*, 2020.

[106] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proc. of ACM SIGCOMM*, 2017.

Appendix A Decomposed TCP non-monotonic

```
def nonmt_cf(tcp.seq):
  temp = maxseq
  if maxseq < tcp.seq:
    maxseq = tcp.seq
  return temp

def nonmt_uf(tcp.seq, maxseq):
  if maxseq >= tcp.seq:
    nm_count += 1

nm[precision_min=99%,ARE_max=1%,confidence=95%]=
  PacketStream()
  .filter(ipv4.protocol==TCP)
  .groupby({5tuple: maxseq}, nonmt_cf)
  .groupby({5tuple: nm_count}, nonmt_uf)
```

Application 2: Decomposed TCP non-monotonic

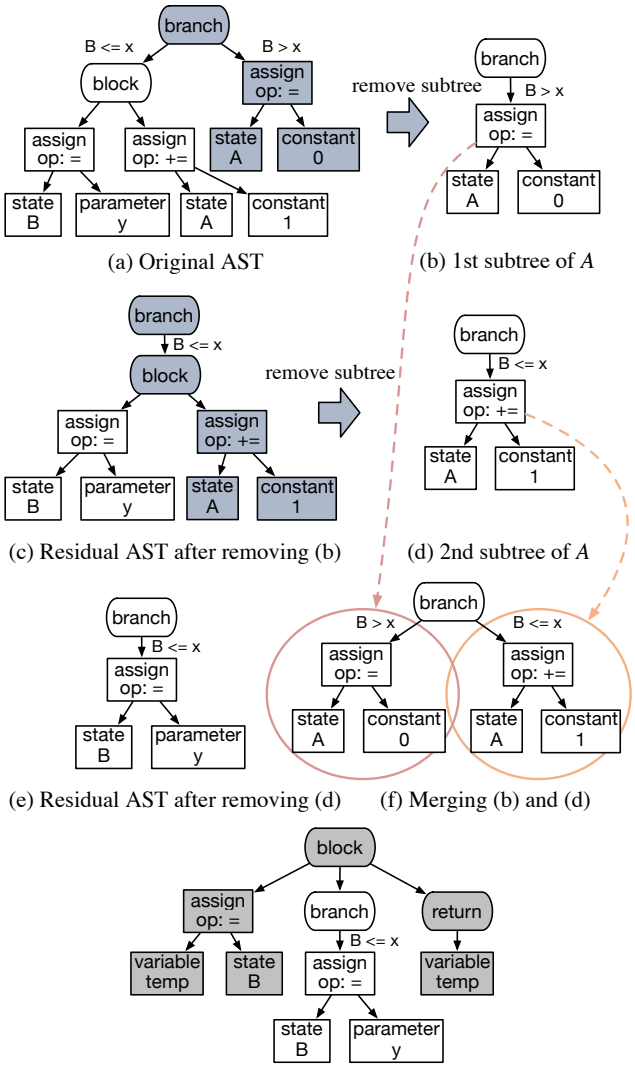
Appendix B Complex Decomposition Example

Figure 13 illustrates the partitioning procedure with a complex example AST. The original AST in Figure 13(a) modifies two states: state *A* and state *B*. We start with partitioning modifications on state *A*. We identify a first subtree modifying *A* as shown in Figure 13(b) (also marked in gray in Figure 13(a)). Note that the root node appears in multiple paths. Thus, the root remains in the residual AST in Figure 13(c), while the assignment node and its children are removed. Since there is still a subtree modifying *A* (marked in gray in Figure 13(c)), we continue to remove it. Figure 13(d) and Figure 13(e) present the second removed subtree and the residual AST. In Figure 13(e), there is only one state *B*, and thereby we do not remove subtrees further. Since there are two subtrees modifying state *A* (in Figure 13(b) and Figure 13(d), respectively), Figure 13(f) integrates them by merging the duplicated root. In Figure 13, updating state *A* depends on the value of state *B*. Thus, we extend the AST of *B* in Figure 13(e) to a new AST in Figure 13(g). Since the value of *B* before updating is actually needed, we add an assignment statement that saves the value before updating in a variable *temp*. The variable is returned and passed to AST of *A* after all statements are executed.

Appendix C Theoretical analysis of Sketch-like structure

We analyze theoretical error bounds according to the behavior of states in sketch-like structures. One type is summable state (i.e., counter), whose sketch-like structure is equivalent to classical CM [20]. The other type is non-summable state (e.g., ingress port number, TCP sequence number).

For non-summable states, we analyze the probability that a flow maintains inaccurate states. Specifically, a flow is accurate if at least one of its hashed cells has no conflicts. Consider



(a) Original AST (b) 1st subtree of A

(c) Residual AST after removing (b) (d) 2nd subtree of A

(e) Residual AST after removing (d) (f) Merging (b) and (d)

(g) Appending temporary variable to (e)

Figure 13: Example of `UDF` decomposition.

N flows in a sketch-like structure with w rows and d columns. We assume that hash functions are perfectly random and independent. The probability that a flow does not collide with others in one row is $(1-\frac{1}{d})^{N-1}$. For w rows, the probability that the flow has at least one row without conflict P is:

$$P = 1-(1-(1-\frac{1}{d})^{N-1})^{w} \approx 1-(1-e^{-\frac{d}{N}})^{w}. \quad (1)$$

The users can utilize the above theoretical analysis to estimate the accuracy of the search configuration.

We can further narrow the error bounds according to the monotonicity-based query method of our sketch-like structure. Take `nonmt_cf` in Application 2 as an example, its sketch-like structure maintains the maximal sequence number of each flow, which is monotonically increasing and thus return the minimum value as the query result. In fact, not the conflicts of all the flows will affect query results. We can first sort the packets with the largest sequence number of each flow in ascending order. Considering the i^{th} largest flow, only $N-i$ flows that collide with it will affect the query result, referred as *interference flows*. Here, Equation 1 is updated to $P = 1-(1-(1-\frac{1}{d})^{i-1})^{w}$. For our sketch-like structure, the query results remain correct as long as one row does not conflict with the interference flows. Therefore, the expectation of the number of flows for which the query results are accurate is

$$E = \sum_{i=1}^{N} 1-(1-(1-\frac{1}{d})^{i-1})^{w} = N-\sum_{i=1}^{N-1}(1-(1-\frac{1}{d})^{i})^{w} \quad (2)$$

This is similar to the sketch-like structure of monotonically decreasing. For the case without monotonicity, all the flows can be interference flows.

Appendix D Optimization for sketch mapping

We optimize the sketch instances with several techniques. First, we allow sketch instances to share hash functions: we compute several hash values for each key, and use them to locate cells in multiple sketch instances. This reduces computational units in switches, particularly the decomposition in §4.1 produces numerous sketch instances. Second, we clear the state values of terminated TCP connections to mitigate hash conflicts. Third, since some applications are interested in partial flows, we only record the keys addressed by the applications instead of all flow keys.

Appendix E Formulation of Benchmark-based parameter tuning

Goal	$min(SC(c))$
Constraints	
C_0	$T(c)$ satisfies EB
C_1	$\sum_{i=0}^{n-1} s_i \le S - s_{stateless}$
C_2	$s_i \ge 1, i \in [1,2,...,n]$
C_3	$d_i \in [1,2,...,s_iA], i \in [1,2,...,n]$
C_4	$\frac{w_i b_i}{8} \le M, i \in [1,2,...,n]$
C_5	$\frac{d_i w_i b_i}{8} \le s_i M, i \in [1,2,...,n]$

Table 3: Formulation for searching.

Table 3 formulates the optimization problem of our benchmark-based parameter tuning. First, the accuracy of the telemetry application $T(c)$ should reach the user-specified accuracy intent EB (C_0). Second, the resource usage is bounded by the hardware capability. C_1 indicates the total stage resources that can be used to deploy sketch instances. Here, S refers to the number of stages in match-action pipeline and $S_{stateless}$ indicates the number of stages used for stateless operators, which can be determined before sketch-based mapping. C_2 indicates that each sketch instance should use at least one

stage. C_3 constraints the maximal number of stateful ALUs cannot exceed sA, where A is the number of stateful ALUs per stage. The register memory size allocated for one stateful ALU cannot exceed the size of register memory (in bytes) per stage M (C_4). Meanwhile, the total register memory of one sketch instance should be less than sM (C_5).

Appendix F The number of Configurations

Consider an example in which we deploy a sketch with a two-dimensional matrix structure in a switch with a single stage. The stage contains four ALUs and 1 MB memory. Assume that each cell occupies fours bytes. Since each hash function needs one ALU, we can support at most four rows. When there is only one single row, we have at most $2^{20}/4 = 262144$ cells in the row, resulting 2550 configurations. Similarly, when the number of rows ranges from two to four, the number of configurations is 1270, 840, and 630, respectively. Therefore, there are 5290 possible configurations in total.

Appendix G LHS-based initialization for searching

Algorithm 5 LHS-based initialization

Input : The number of mapped sketch instances n
1: **function** LHS_INIT(n)
2: m = min(ALU_{max}, $Page_{max}$)
3: Initialize m empty application configurations $c_1, c_2, ..., c_m$
4: **for** i = 0 to n **do**
5: `alus` = [1, 2, ..., ALU_{max}]
6: `pages` = [1, 2, 4, ..., $Page_{max}$]
7: **for** j = 0 to m **do**
8: d = random sample in `alus`
9: Remove d from `alus`
10: w = random sample in `pages`
11: Remove w from `pages`
12: $c_j[2i] = d$; $c_j[2i+1] = w$
13: **return** the list of $c_1, c_2, ..., c_m$

Algorithm 5 details the LHS-based initialization. It takes the number of mapped sketch instances n as input and outputs a list of initial configurations sampled by LHS. The algorithm first initializes m empty configurations (line 3). Here, m depends on the minimal value between the available maximal stateful ALUs and memory pages (line 2), in that LHS has to ensure the parameter values of all samples in each dimension are different. As mentioned in §5.1, the application configuration is a vector of $2n$ variables ($c = (d_1, w_1, d_2, w_2, ..., d_n, w_n)$). Therefore, the algorithm fills in the d and w dimensions of all samples one by one in the order of the sketch instance (lines 4-12). For each sketch instance, the algorithm prepares two lists (`alus` and `pages`) of available resource values (lines 5,6). Then, c_1 to c_m iteratively pops the values in `alus` and `pages` randomly to fill in $c_j[2i]$ and $c_j[2i+1]$ (lines 7-12). In this way, the configurations meet the requirement of LHS.

Algorithm 6 Pseudo Code of Function CALCNEIGHBOR

1: **function** CALCNEIGHBOR(c)
2: `neighbors` = []
3: **for** each parameter p **in** c **do**
4: c' = copy of c
5: **if** c satisfys EB **then** ▷ decrease resource
6: **if** p is a row parameter **then**
7: $p' = p - 1$
8: **else**
9: $p' = p/2$
10: **else** ▷ increase resource
11: **if** p is a row parameter **then**
12: $p' = p + 1$
13: **else**
14: $p' = p * 2$
15: Replace p **in** c' by p'
16: Add c' to `neighbors`
17: **return** `neighbors`

Appendix H Hardware-aware configuration generation

Algorithm6 details the hardware-aware configuration generation process. It generates neighbors for a given configuration by changing one parameter in the configuration at a time (line 3). If the given configuration satisfies the accuracy intent, the algorithm calculates its neighbors by decreasing the resource usage (lines 5-9). Otherwise, it calculates its neighbors by increasing the resource usage (lines 10-14). The algorithm considers hardware characteristics. Specifically, when changing the number of rows, it uses 1 as the step size, and when changing the number of columns, it ensures that the number of columns is a power of 2.

Appendix I Additional Experiment

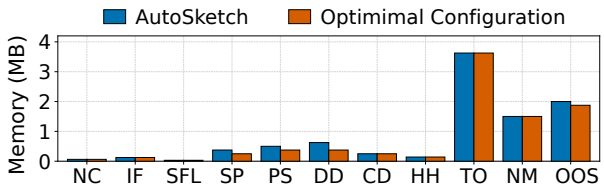


Figure 14: (Exp#10) Memory efficiency of searching results.

(Exp#10) Efficiency of searching results. We compare the resource usage between the configuration searched by AutoSketch and the optimal configuration found by brute-force searching. Figure 14 shows that the memory consumption of AutoSketch is slightly more than the optimal configuration: even the largest difference (DD) is only 0.25 MB. In additional, the searched results consume the same number of stages and at most one more ALU than the optimal ones (not shown in the figure). Such additional costs are acceptable.

Appendix J Complete Experiment Results

Table 4 and Table 5 list the complete experiment results of Exp#2 to Exp#4 in §7.2, including SSH Brute Force detection. Table 6 summarizes the complete switch resources and application accuracies of sketch-based solutions under two accuracy intents (AS-1 and AS-2).

Telemetry Applications	**Resources**																			
	Memory (MB)					**Stage**					**ALU**					**Bandwidth (MB/s)**				
	Marple | Sonata | EM | AutoSketch-1 | AutoSketch-2																			
New TCP Conns.	0.5	0.5	-	0.06	0.13	2	4	-	1	1	2	2	-	2	2	0.06	0.033	-	0.001	0.001
TCP Incomplete Flows	1.0	1.0	-	0.13	0.5	2	8	-	1	1	4	4	-	2	2	0.276	0.036	-	0.001	0.001
DDoS	1.0	1.0	-	0.63	0.75	3	8	-	2	2	3	4	-	4	6	44.24	0.037	-	0.0037	0.0036
Port Scan	0.88	1.0	-	0.5	1.0	3	8	-	2	2	4	4	-	3	4	23.56	0.07	-	0.0063	0.0063
SSH Brute Force	1.25	1.0	-	0.03	0.03	3	8	-	2	2	5	4	-	2	2	0.001	0.001	-	0.001	0.001
TCP non-monotonic	1.31	-	2.25	1.5	2.13	3	-	7	5	6	4	-	6	11	11	147.6	-	1.265	0.233	0.226
TCP out-of-sequence	1.31	-	2.25	2	2.88	3	-	7	5	7	4	-	6	13	12	167.3	-	1.692	0.526	0.517
TCP timeouts	1.31	-	2.25	3.63	3.63	3	-	7	8	8	4	-	6	12	12	34.4	-	0.609	0.136	0.136
SYN Flood	0.91	-	2.0	0.06	0.06	2	-	7	4	4	2	-	5	4	4	56.35	-	0.001	0.001	0.001
Total	10.47	4.5	8.75	8.54	11.11	24	36	28	30	33	32	18	23	53	55	473.8	0.177	3.57	0.909	0.893
Average	1.16	0.9	2.19	0.95	1.23	2.67	7.2	7	3.33	3.67	3.56	3.6	5.75	5.89	6.11	52.64	0.035	0.89	0.101	0.099

Table 4: Complete resource overheads of solutions in Exp#2 and Exp#3.

Telemetry Applications	**Accuracy**														
	Precision					**Recall**					**ARE**				
	Marple | Sonata | EM | AutoSketch-1 | AutoSketch-2														
New TCP Conns.	1.0	0.98	-	0.995	0.998	1.0	0.74	-	1.0	1.0	-	-	-	-	-
TCP Incomplete Flows	1.0	0.93	-	0.95	0.994	1.0	0.75	-	0.993	1.0	-	-	-	-	-
DDoS	1.0	0.92	-	0.976	0.996	1.0	0.34	-	0.996	0.996	-	-	-	-	-
Port Scan	1.0	0.93	-	0.956	0.981	1.0	0.41	-	0.982	1.0	-	-	-	-	-
SSH Brute Force	1.0	1.0	-	1.0	1.0	1.0	1.0	-	1.0	1.0	-	-	-	-	-
TCP non-monotonic	1.0	-	0.25	0.96	0.992	1.0	-	0.996	0.996	0.996	0.0	-	1.86	0.025	0.006
TCP out-of-sequence	1.0	-	0.37	0.98	0.99	1.0	-	0.99	0.998	0.999	0.0	-	1.39	0.016	0.005
TCP timeouts	1.0	-	0.66	0.95	0.95	1.0	-	0.62	0.972	0.972	0.0	-	0.38	0.028	0.028
SYN Flood	1.0	-	0.17	1.0	1.0	1.0	-	1.0	1.0	1.0	-	-	-	-	-

Table 5: Complete accuracy of solutions in Exp#4.

Solutions	**Accuracy Intent 1**						**Accuracy Intent 2**					
	Memory (MB)	**Stage**	**ALU**	**Precision**	**Recall**	**RE**	**Memory (MB)**	**Stage**	**ALU**	**Precision**	**Recall**	**RE**
HashPipe	0.563	9	16	0.99	0.195	-	0.63	12	18	0.989	0.464	-
MV-Sketch	0.375	6	12	0.958	0.995	-	0.63	8	20	0.989	1.0	-
FlowRadar-HH	5.91	11	24	0.989	0.903	-	5.91	11	24	0.989	0.903	-
OpenSketch-HH	0.578	7	8	0.976	1.0	-	0.95	8	8	0.986	1.0	-
AutoSketch-HH	0.14	3	3	0.98	0.999	-	0.27	3	3	0.997	1.0	-
SpreadSketch	0.375	6	6	0.972	0.659	-	0.63	7	10	0.99	0.695	-
Vector Bloom Filter	2.58	12	5	0.973	0.66	-	2.58	12	5	0.973	0.66	-
FlowRadar-SS	5.91	11	24	1.0	0.991	-	5.91	11	24	1.0	0.991	-
OpenSketch-SS	1.2	10	8	0.941	0.748	-	1.75	10	8	0.976	0.728	-
AutoSketch-SS	0.375	3	4	0.975	0.978	-	1.0	3	6	0.983	1.0	-
FM-Sketch	0.031	4	1	-	-	0.044	0.046	4	1	-	-	0.006
Linear Counting	0.031	1	1	-	-	0.017	0.031	1	1	-	-	0.004
FlowRadar-CD	5.91	11	24	-	-	0.011	5.91	11	24	-	-	0.011
OpenSketch-CD	0.031	1	1	-	-	0.017	0.031	1	1	-	-	0.004
AutoSketch-CD	0.25	2	2	-	-	0.011	0.19	2	3	-	-	0.009

Table 6: Complete experiment results of solutions in Exp#6 and Exp#7.

Appendix K Telemetry Applications

We show the function of each application in Table 7 and how to realize these twelve telemetry applications.

#	Application	Description
1	New TCP Connections	Count the number of newly opened TCP connections exceeds threshold.
2	TCP Incomplete Flows	Count the number of incomplete TCP connections exceeds threshold.
3	Port Scan	Detect the end-hosts that send traffic over more than threshold destination ports.
4	DDoS	Detect the end-hosts that receive traffic from more than threshold unique sources.
5	TCP timeouts	Count the number of timeouts for each TCP connection, by checking for packet inter-arrival times around 300 ms (retransmission timer).
6	TCP non-monotonic	Count the number of packets per connection with sequence numbers lower than the maximum so far.
7	TCP out-of-sequence	Count the number of packets per connection arriving with a sequence number that is non-consecutive with the last packet.
8	Superspreader	Detect the end-hosts that contact more than threshold unique destinations.
9	Cardinality	Count the number of flows in the network.
10	Heavy Hitter	Identify large flows that consume more than threshold during a time interval.
11	SYN Flood	Detect the end-hosts whose half-open TCP connections exceeds threshold.

Table 7: Description of telemetry applications.

```
n_syn[precision_min=95%,recall_min=95%,confidence
    =95%] = PacketStream()
    .filter(ipv4.protocol==TCP)
    .filter(tcp.flags==SYN)
    .map((ipv4.dstIP, count), count=1)
    .reduce((ipv4.dstIP), val=count)
    .filter(count>=Thld)
    .distinct((ipv4.dstIP))
```

Application 1: New TCP Connections

```
n_fin = PacketStream()
    .filter(ipv4.protocol==TCP)
    .filter(tcp.flags==FIN)
    .map((ipv4.srcIP, fin_cnt), fin_cnt=1)
    .reduce((ipv4.srcIP), val=fin_cnt)

diff[precision_min=95%,recall_min=95%,confidence
    =95%] = PacketStream()
    .filter(ipv4.protocol==TCP)
    .filter(tcp.flags==SYN)
    .map((ipv4.dstIP, syn_cnt), syn_cnt=1)
    .reduce((ipv4.dstIP), val=syn_cnt)
    .zip(stream=n_fin, (ipv4.srcIP))
    .map((ipv4.dstIP, diff), diff=syn_cnt-fin_cnt)
    .filter(diff>=Thld)
    .distinct((ipv4.dstIP))
```

Application 2: TCP Incomplete Flows

```
port_scan[precision_min=95%,recall_min=95%,
    confidence=95%] = PacketStream()
    .distinct((ipv4.srcIP, tcp.dport))
    .map((ipv4.srcIP, count), count=1)
    .reduce((ipv4.srcIP), val=count)
    .filter(count>=Thld)
    .distinct((ipv4.srcIP))
```

Application 3: Port Scan

```
ddos[precision_min=95%,recall_min=95%,confidence
    =95%] = PacketStream()
    .distinct((ipv4.dstIP, ipv4.srcIP))
    .map((ipv4.dstIP, count), count=1)
    .reduce((ipv4.dstIP), val=count)
    .filter(count>=Thld)
    .distinct((ipv4.dstIP))
```

Application 4: DDoS

```
def to(tin):
    if tin - last_ts > Thld:
        to_count += 1
    last_ts = tin
timeout[precision_min=95%,recall_min=95%,ARE_max
    =3%,confidence=95%] = PacketStream()
    .filter(ipv4.protocol==TCP)
    .groupby({5tuple:(last_ts, to_count)}, to))
```

Application 5: TCP timeouts

```
def nonmt(tcp.seq):
    if maxseq < tcp.seq:
        maxseq = tcp.seq
    else:
        nm_count += 1
nm[precision_min=95%,recall_min=95%,ARE_max=3%,
    confidence=95%] = PacketStream()
    .filter(ipv4.protocol==TCP)
    .groupby({5tuple:(maxseq, nm_count)}, nonmt)
```

Application 6: TCP non-monotonic detection

```
def oos(tcp.seq, payload_len):
    if lastseq != tcp.seq:
        oos_count += 1
    lastseq = tcp.seq + payload_len
tcp_oos[precision_min=95%,recall_min=95%,ARE_max
    =3%,confidence=95%] = PacketStream()
    .filter(ipv4.protocol==TCP)
    .groupby({5tuple:(lastseq, oos_count)}, oos)
```

Application 7: TCP out-of-sequence detection

```
super_spreader[precision_min=95%,recall_min=95%,
    confidence=95%] = PacketStream()
    .distinct((ipv4.dstIP, ipv4.srcIP))
    .map((ipv4.srcIP, count), count=1)
    .reduce((ipv4.srcIP), val=count)
    .filter(count>=Thld)
    .distinct((ipv4.srcIP))
```

Application 8: Superspreader

```
cardinality[ARE_max=3%,confidence=95%] =
    PacketStream()
    .distinct((5tuple))
    .map((5tuple, count), count=1)
    .reduce((), val=count)
```

Application 9: Cardinality

```
heavyhitter[precision_min=95%,recall_min=95%,
    confidence=95%]  = PacketStream()
    .map((5tuple, count),count=ipv4.totalLen)
    .reduce((5tuple), val=count)
    .filter(count>=Thld)
    .distinct((5tuple))
```

Application 10: Heavy Hitter

```
def remap_key(tcp.flag):
    if tcp.flag == SYNACK:
        nkey = ipv4.srcIP
    else:
        nkey = ipv4.dstIP
    return nkey
def sf(tcp.flag, tcp.seq, tcp.ack):
    if tcp.flag == SYNACK:
        nextseq = tcp.seq + 1
        cnt += 1
    else if nextseq == tcp.ack
        cnt -= 1
    return cnt
syn_flood[precision_min=95%,recall_min=95%,
    confidence=95%] = PacketStream()
    .filter(ipv4.protocol==TCP)
    .filter(tcp.flags contains ACK)
    .groupby(null, remap_key)
    .groupby({nkey:(nextseq, cnt)}, sf)
    .filter(cnt > Thld)
    .distinct((nkey))
```

Application 11: SYN Flood