VAST: Visibility Across Space and Time
Architecture & Usage

Matthias Vallentin
matthias@bro.org

BroCon
August 19, 2014
SAFEGUARDS AND SECURITY PROGRAM PLANNING AND MANAGEMENT

U.S. DEPARTMENT OF ENERGY
Office of Security and Safety Performance Assurance

Vertical line denotes change.

AVAILABLE ONLINE AT:  INITIATED BY:
http://www.directives.doe.gov  Office of Security and Safety
Performance Assurance
Table 1. Reportable Categories of Incidents of Security Concern, Impact Measurement Index 1 (IMI-1)

<table>
<thead>
<tr>
<th>Incident Type</th>
<th>Report within 1 hour</th>
<th>Report within 8 hours</th>
<th>Report monthly</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Confirmed or suspected loss, theft, or diversion of a nuclear device or components.</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Confirmed or suspected loss, theft, diversion, or unauthorized disclosure of weapon data.</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Confirmed or suspected loss, theft, or diversion of Category I or II quantities of special nuclear material (SNM).</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. A shipper-receiver difference involving a loss in the number of items which total a Category I or II quantity of SNM.</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Confirmed or suspected loss, theft, diversion, unauthorized disclosure of Top Secret information, Special Access Program (SAP) information, or Sensitive Compartmented Information (SCI), regardless of the medium, method, or action resulting in the incident.</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Confirmed or suspected intrusions, hacking, or break-ins into DOE computer systems containing Top Secret information, SAP information, or SCI.</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. Confirmed or suspected physical intrusion attempts or attacks against DOE facilities containing nuclear devices and/or materials, classified information, or other national security related assets.</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

DOE O 151.1B, Comprehensive Emergency Management System, dated 10-29-03, and facility emergency management plans may require more stringent reporting times for IMI-1 type incidents than listed here. Shorter reporting times should be determined on an individual incident basis and applied accordingly.
MEMORANDUM FOR: ASSOCIATE DIRECTORS  
OFFICE DIRECTORS  
SITE OFFICE MANAGERS

FROM: GEORGE MALOSH  
ACTING CHIEF OPERATING OFFICER  
OFFICE OF SCIENCE

SUBJECT: Office of Science Policy on the Protection of Personally Identifiable Information

The attached Office of Science (SC) Personally Identifiable Information (PII) Policy is effective immediately. This supersedes my July 14, 2006, memorandum providing

• **Incident Reporting**

  Within 45 minutes after discovery of a real or suspected loss of Protected PII data, Computer Incident Advisory Capability (CIAC) needs to be notified (ciac@ciac.org). Reporting of incidents involving Public PII will be in accordance with normal incident reporting procedures.
Outline

1. Introduction: VAST

2. Architecture
   - Overview
   - Example Workflow: Query
   - Data Model
   - Implementation

3. Using VAST

4. Demo
VAST: Visibility Across Space and Time

VAST
A unified platform for network forensics

Goals
- Interactivity
  - Sub-second response times
  - Iterative query refinement
- Scalability
  - Scale with data & number of nodes
  - Sustain high & continuous input rates
- Strong and rich typing
  - High-level types and operations
  - Type safety
VAST & Bro

**Bro**

- Generates rich-typed logs representing summary of activity
  - How to process these huge piles of logs?
- Fine-grained events exist during runtime only
  - Make ephemeral events persistent?

**VAST: Visibility Across Space and Time**

- Visibility across **Space**
  - Unified data model: same expressiveness as Bro
  - Combine host-based and network-based activity
- Visibility across **Time**
  - Historical queries: retrieve data from the past
  - Live queries: get notified when new data matches query
# VAST & Big Data Analytics

## MapReduce (Hadoop)

Batch-oriented processing: *full scan* of data
- Expressive: no restriction on algorithms
- Speed & Interactivity: full scan for each query

## In-memory Cluster Computing (Spark)

Load full data set into memory and then run query
- Speed & Interactivity: fast on arbitrary queries over working set
- Thrashing when working set too large

## Distributed Indexing (VAST)

Distributed building and querying of bitmap indexes
- Fast: only access space-efficient indexes
- Caching of index hits enables iterative analyses
- Reduced computational model (e.g., no joins in query language)
Outline

1. Introduction: VAST

2. Architecture
   - Overview
   - Example Workflow: Query
   - Data Model
   - Implementation

3. Using VAST

4. Demo
Outline

1. Introduction: VAST

2. Architecture
   - Overview
   - Example Workflow: Query
   - Data Model
   - Implementation

3. Using VAST

4. Demo
High-Level Architecture of VAST

**Import**
- Unified data model
- Sources generate events

**Archive**
- Stores raw data as events
- Compressed chunks & segments

**Index**
- Secondary indexes into archive
- Horizontally partitioned

**Export**
- Interactive query console
- JSON/Bro output
Query Language

Boolean Expressions
- Conjunctions `&&`
- Disjunctions `||`
- Negations `!`
- Predicates
  - LHS op RHS
  - `(expr)`

Examples
- `A && B || !(C && D)`
- `orig_h == 10.0.0.1 && &time < now - 2h`
- `&type == "conn" || :string +] "foo"
- `duration > 60s && service == "tcp"`

LHS: Extractors
- `&type`
- `&time`
- `x.y.z.arg`
- `:type`

Relational Operators
- `<`, `<=`, `==`, `>=`, `>`
- `in`, `ni`, `[+, +]`
- `!in`, `!ni`, `[-, -]`
- `~`, `!~`

RHS: Value
- `T, F`
- `+42, 1337, 3.14`
- "foo"
- `10.0.0.0/8`
- `80/tcp, 53/?`
- `{1, 2, 3}`
Outline

1. Introduction: VAST

2. Architecture
   - Overview
   - Example Workflow: Query
     - Data Model
     - Implementation

3. Using VAST

4. Demo
Query

1. Send query string to search
2. Receive query actor
3. Extract results from search

1. Parse and validate query string
2. Spawn dedicated query
3. Forward query to index

1. Receive hits from index
2. Ask archive for segments
3. Extract events, check candidates
4. Send results to client
CLIENT

1. Send query string to SEARCH

Search

Client
1. Send query string to **SEARCH**

```
src == 10.0.0.1
&&
port == 53/udp
```
QUERY

CLIENT
1. Send query string to SEARCH

SEARCH
1. Parse and validate query string
2. Spawn dedicated QUERY

Client
Search
src == 10.0.0.1
&&
port == 53/udp

Index
Partitions
Indexers
Query

src == 10.0.0.1
&&
port == 53/udp
Query

**CLIENT**
1. Send query string to SEARCH
2. Receive QUERY actor

**SEARCH**
1. Parse and validate query string
2. Spawn dedicated QUERY

### Diagram

```
Client
  /     \
 /       \
Search   Query
  \       /  src == 10.0.0.1
   \  &&
    \ port == 53/udp
    \  \
    Index
      /  \
     /   \
    Indexers
      /     \
     /       \
    Partitions
      /         \
     /           \
    Index
```

**Extracts**
1. Send query string to SEARCH
2. Receive QUERY actor
3. Parse and validate query string
4. Spawn dedicated QUERY

**NOTE:**
- Query logic involves parsing, validation, spawning dedicated QUERY, and sending results back to the client.
- The diagram illustrates the flow from client to search, and then to indexers and partitions, ultimately returning results.
**Query**

**CLIENT**
1. Send query string to SEARCH
2. Receive QUERY actor

**SEARCH**
1. Parse and validate query string
2. Spawn dedicated QUERY
3. Forward query to INDEX

```
src == 10.0.0.1 && port == 53/udp
```
**Query**

**CLIENT**
1. Send query string to SEARCH
2. Receive QUERY actor

**SEARCH**
1. Parse and validate query string
2. Spawn dedicated QUERY
3. Forward query to INDEX

---

Diagram:
- **Client**
- **Search**
- **Query**
- **Index**
- **Partitions**
- **Indexers**

Query string details:
- `port == 53/udp`
- `src == 10.0.0.1`
Query

CLIENT
1. Send query string to SEARCH
2. Receive QUERY actor

SEARCH
1. Parse and validate query string
2. Spawn dedicated QUERY
3. Forward query to INDEX

Diagram:
- Client
- Search
- Query
- Index
- Partitions
- Indexers
**QUERY**

**CLIENT**
1. Send query string to **SEARCH**
2. Receive **QUERY** actor

**SEARCH**
1. Parse and validate query string
2. Spawn dedicated **QUERY**
3. Forward query to **INDEX**

**INDEX**
1. Receive hits from **INDEX**
2. Ask **ARCHIVE** for segments
3. Extract events, check candidates
4. Send results to **CLIENT**
**Query**

**CLIENT**
1. Send query string to SEARCH
2. Receive QUERY actor

**SEARCH**
1. Parse and validate query string
2. Spawn dedicated QUERY
3. Forward query to INDEX

**Diagram:**
- Client
- Search
- Query
- Index
- Indexers
- Partitions
- Index

Flow:
1. Client sends query string to Search
2. Search parses and validates query string
3. Search spawns dedicated Query
4. Query forwards query to Index
5. Index returns hits to Query
6. Query asks archive for segments
7. Query extracts events, checks candidates
8. Query sends results to Client
**Query**

**CLIENT**
1. Send query string to SEARCH
2. Receive QUERY actor

**SEARCH**
1. Parse and validate query string
2. Spawn dedicated QUERY
3. Forward query to INDEX

**QUERY**
1. Receive hits from INDEX
**Query**

**CLIENT**
1. Send query string to **SEARCH**
2. Receive **QUERY** actor

**SEARCH**
1. Parse and validate query string
2. Spawn dedicated **QUERY**
3. Forward query to **INDEX**

**QUERY**
1. Receive hits from **INDEX**
2. Ask **ARCHIVE** for segments
**QUERY**

1. Receive hits from **INDEX**
2. Ask **ARCHIVE** for segments
3. Extract events, check candidates

---

**SEARCH**

1. Parse and validate query string
2. Spawn dedicated **QUERY**
3. Forward query to **INDEX**

---

**CLIENT**

1. Send query string to **SEARCH**
2. Receive **QUERY** actor
**Query**

**CLIENT**
1. Send query string to SEARCH
2. Receive QUERY actor
3. Extract results from QUERY

**SEARCH**
1. Parse and validate query string
2. Spawn dedicated QUERY
3. Forward query to INDEX

**QUERY**
1. Receive hits from INDEX
2. Ask ARCHIVE for segments
3. Extract events, check candidates
4. Send results to CLIENT
Outline

1. Introduction: VAST

2. Architecture
   - Overview
   - Example Workflow: Query
   - Data Model
   - Implementation

3. Using VAST

4. Demo
VAST Architecture

10.0.0.1 10.0.0.254 53/udp
10.0.0.2 10.0.0.254 80/tcp

Archive

Import

Export

Index
Data Representation

Terminology

- **Data**: C++ structures (e.g., 64ull)
- **Type**: interpretation of data (e.g., count)
- **Value**: data + type
- **Event**: value + meta data
  - Type with a unique name (e.g., conn)
  - Meta data
    - A timestamp
    - A unique ID \( i \in [1, 2^{64} - 1] \)
- **Schema**: collection of event types
- **Chunk**: serialized & compressed events
  - Meta data: schema + time range + IDs
  - Fixed number of events, variable size
- **Segment**: sequence of chunks
  - Meta data: union of chunk meta data
  - Fixed size, variable number of chunks
Types: Interpretation of Data

- **container types**
  - vector
  - set
  - table

- **compound types**
  - record
    - field 1
    - …
    - field n

- **recursive types**

- **basic types**
  - bool
  - int
  - count
  - double
  - time range
  - time point
  - string
  - regex
  - address
  - subnet
  - port
  - none

- **types**
  - TYPE
  - TYPE
  - TYPE
  - KEY
  - VALUE
  - TYPE
  - TYPE

- **examples**
  - record
  - vector
  - set
  - table
  - KEY
  - VALUE
  - TYPE
  - TYPE

- **visual representation**
  - Diagram showing the relationships between container types, compound types, and basic types.
Index Hits: Sets of Events

**Bitvector**: sets of events
- Query result \(\equiv\) set of event IDs from \([1, 2^{64} - 1]\)
- Model as **bit vector**: \([4, 7, 8] = 0000100110 \ldots\)

**Bitstream**: encoded append-only sequence of bits
- EWAH (no patents unlike WAH, PLWAH, COMPAX)
- Compact, space-efficient representation
- Bitwise operations do not require decoding

**Bitmap**: maps values to bitstreams
- \(\text{push}_{-}\text{back}(T\ x)\): append value \(x\) of type \(T\)
- \(\text{lookup}(T\ x, Op \circ)\): get bitstream for \(x\) under \(\circ\)
Composing Results via Bitwise Operations

Combining Predicates

- Query $Q = X \land Y \land Z$
  - $x = 1.2.3.4 \land y < 42 \land z \in \text{"foo"}$
- Bitmap index lookup yields $X \rightarrow B_1$, $Y \rightarrow B_2$, and $Z \rightarrow B_3$
- Result $R = B_1 \& B_2 \& B_3$
Outline

1. Introduction: VAST

2. Architecture
   - Overview
   - Example Workflow: Query
   - Data Model
   - Implementation

3. Using VAST

4. Demo
Actor Model

**Actor:** unit of sequential execution

- **Message:** typed tuple $\langle T_0, \ldots, T_n \rangle \in \mathbb{T}^n$
- **Behavior:** partial function over $\mathbb{T}^n$
- **Mailbox:** FIFO with typed messages
- Can send messages to other actors
- Can spawn new actors
- Can monitor each actors

**Benefits**

- Modular, high-level components
- Robust SW design: no locks, no data races
- Network-transparent deployment
- Powerful concurrency model
CAF: C++ Actor Framework

**libcaf**

- **Native** implementation of the actor model
- **Strongly typed** actors available → protocol checked at compile-time
- **Pattern matching** to extract messages
- **Transparency**l supports heterogeneous components
  - Intra-machine: efficient message passing with copy-on-write semantics
  - Inter-machine: TCP, UDP *(soon)*, multicast *(soon)*
  - Special hardware components: GPUs via OpenCL

https://github.com/actor-framework
Outline

1. Introduction: VAST

2. Architecture
   - Overview
   - Example Workflow: Query
   - Data Model
   - Implementation

3. Using VAST

4. Demo
Getting Up and Running

Requirements

- C++14 compiler
  - Clang 3.4 (easiest bootstrapped with Robin’s `install-clang`)
  - GCC 4.9 (not yet fully supported)
- CMake
- Boost Libraries (headers only)
- C++ Actor Framework (develop branch currently)

Installation

- `git clone git@github.com:mavam/vast.git && cd vast`
- `./configure && make && make test && make install`
- `vast -h # brief help`
- `vast -z # complete options`
VAST Architecture

10.0.0.1 10.0.0.254 53/udp
10.0.0.2 10.0.0.254 80/tcp

Import

Archive

Index

Export
Deployment

Network Transparency

- Actors can live in the same address space
  → Efficiently pass messages as pointer
- Actors can live on different machines
  → Transparent serialization of messages

Import with 2 Processes

One-Shot Import
## Importing Logs

### One-Shot Import

- `vast -C -I -r conn.log`
- `zcat *.log.gz | vast -C -I`
- `vast -C -I -p partition-2014-01 < conn.log`

### Import with 2 Processes

- `vast -C # core`
- `vast -I < conn.log # importer`
VAST Architecture

10.0.0.1 10.0.0.254 53/udp
10.0.0.2 10.0.0.254 80/tcp

Import

Archive

Index

Export
Synopsis: One-Shot Queries

**JSON Query**
- `vast -C`  # core
- `vast -E -o json -l 5 -q ':addr in 10.0.0.0/8'`

**Bro Query**
- `vast -C`  # core
- `vast -E -o bro -l 5 -q ':addr in 10.0.0.0/8'`
Outline

1. Introduction: VAST

2. Architecture
   - Overview
   - Example Workflow: Query
   - Data Model
   - Implementation

3. Using VAST

4. Demo
Thank You... Questions?

VAST

https://github.com/mavam/vast

IRC at Freenode: #vast