Snort 3.0

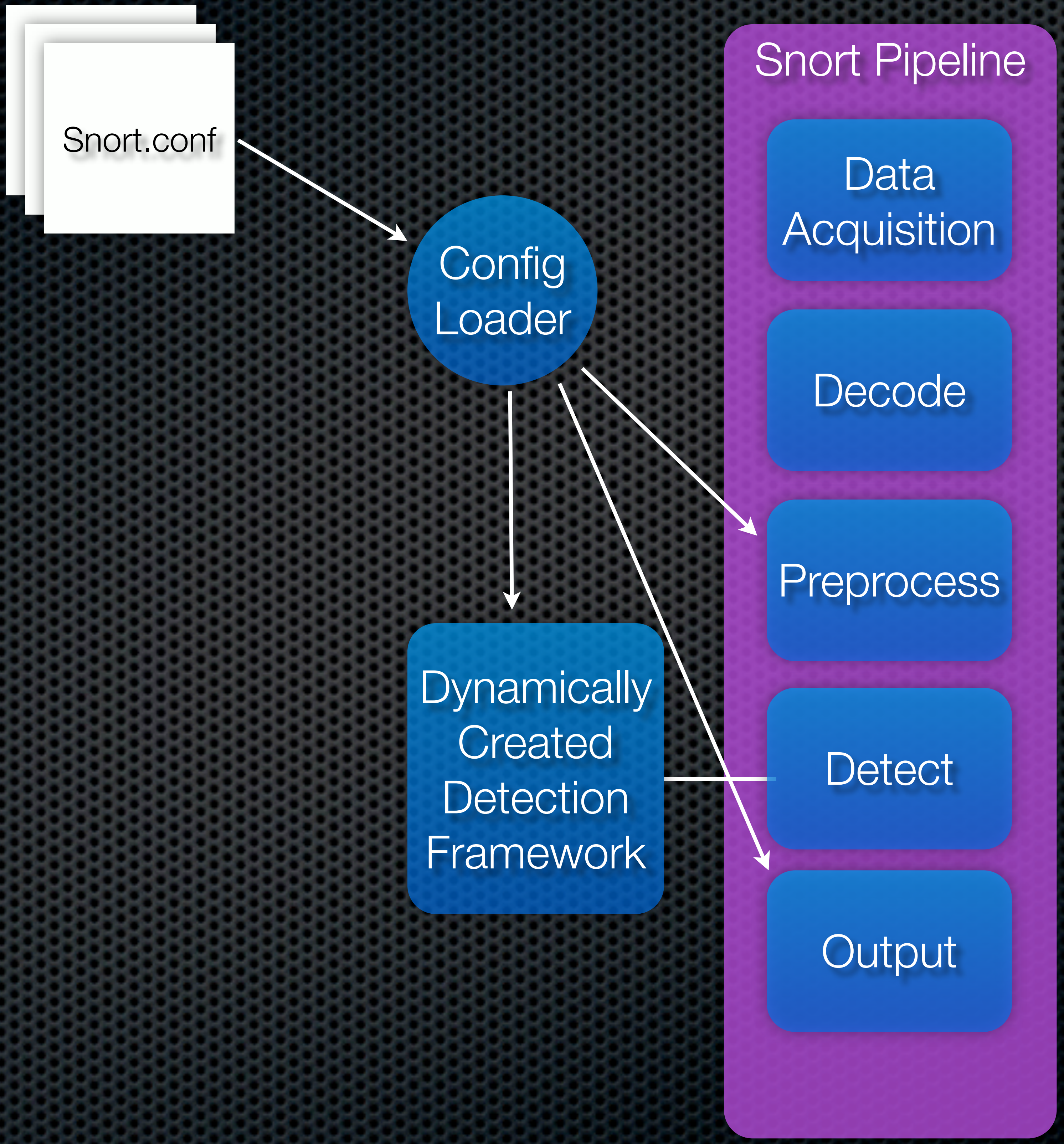
A Brief Overview - 1998

- Snort's Original Goals
  - Learn libpcap library
  - Monitor Home Network
  - Network Application Debugger
- Original Open Source release, December 1998
- People started trying to do useful things with Snort so I started working on it more seriously...

1999

- Snort 1.0 - April 1999
  - Rules language in place
  - Stateless
- Snort 1.5 - December 1999
  - Rewrite of 1.0
  - Same fundamental architecture still in use

Snort 1.5 Architecture



Recently...

- Snort 2.8.0.2 Available, 2.8.1 in RC
  - 12000+ rules
  - Highly Stateful
- Industry leading technology
  - 1Gbps+ -> 10Gbps offerings available
  - Advanced research into detection engine design, anti-evasion, self-tuning, etc

Snort 3.0

What’s Driving Development?

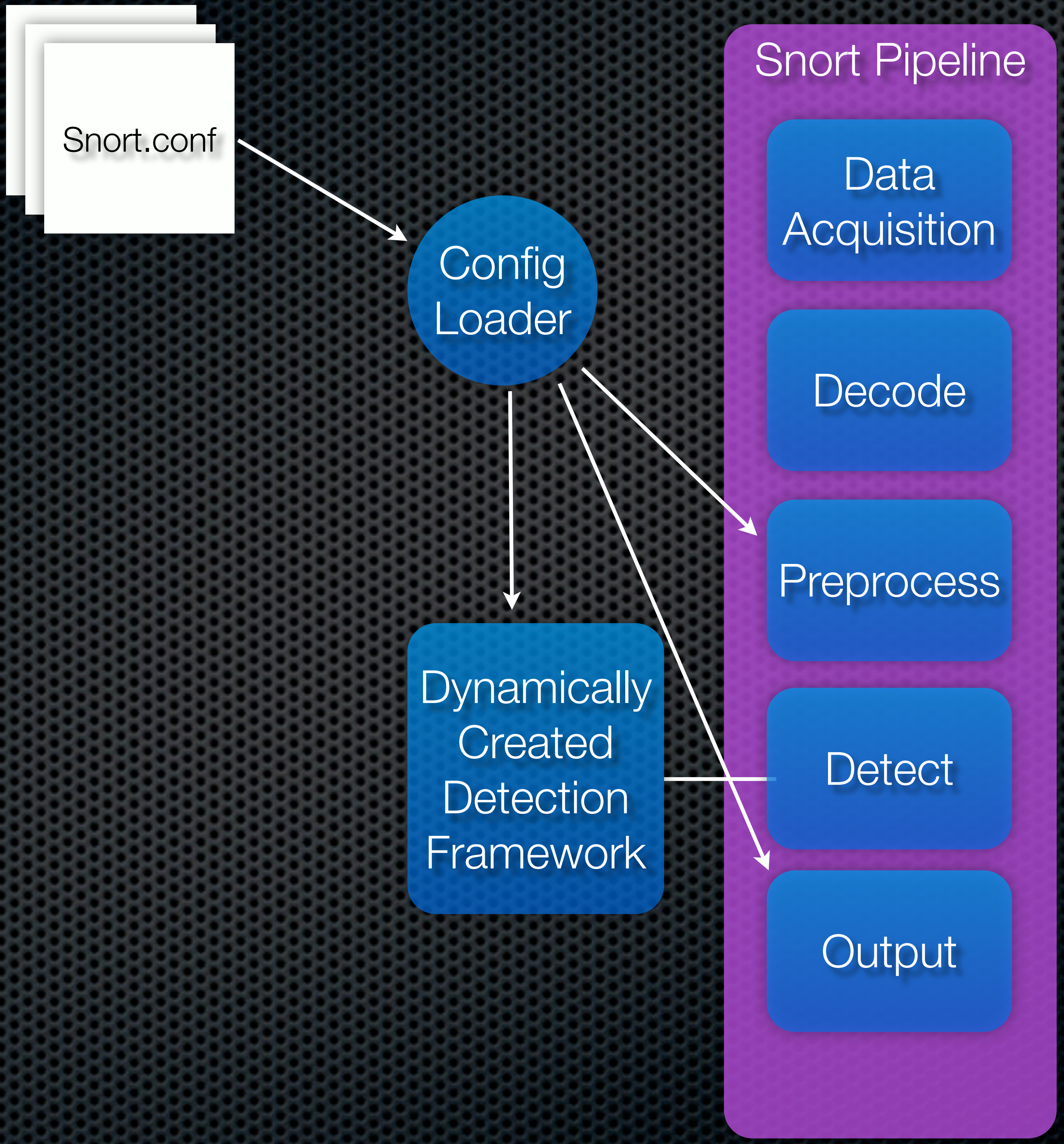
- Completely new code base!
- Architected for high speed in-line operation
- Build a platform that will suit *ANY* network traffic analysis need!
- Operational efficiencies of one code base to advance and maintain

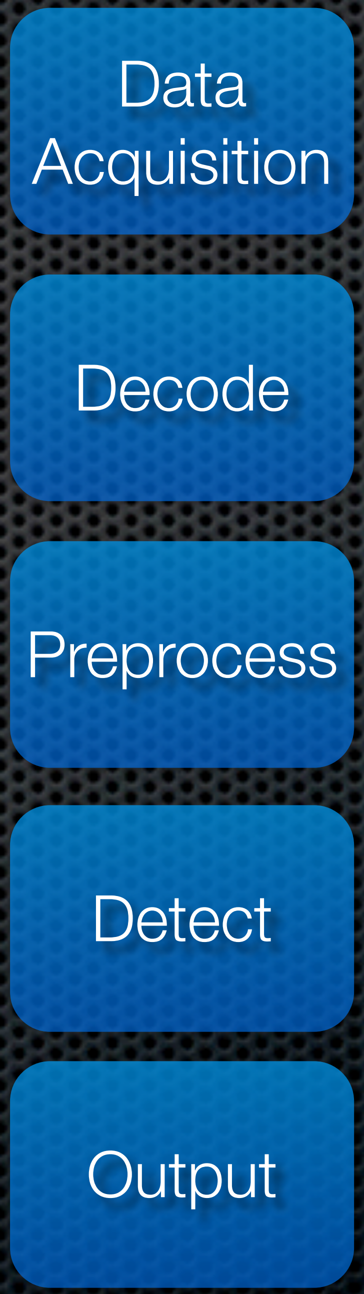
Other Considerations

- Efficiency
  - Snort 3.0 is architected to be accelerated
  - Snort 3.0 is multithreaded
    - Engines can run continuously, reloads unneeded
    - Engines can be parallelized for multi-core CPUs
- Clean code base opportunities
  - Reduce LOC count
  - Eliminate old code, unused features

Snort Lessons

- Users don’t like tuning
    - Users also don’t like false positives...
- Evasion needs to be addressed
- Snort’s language is, well, different
- Prioritization is broken
- Take advantage of modern hardware


Snort.conf
Config Loader
Snort Pipeline
Data Acquisition
Decode
Preprocess
Detect
Output
Dynamically Created Detection Framework



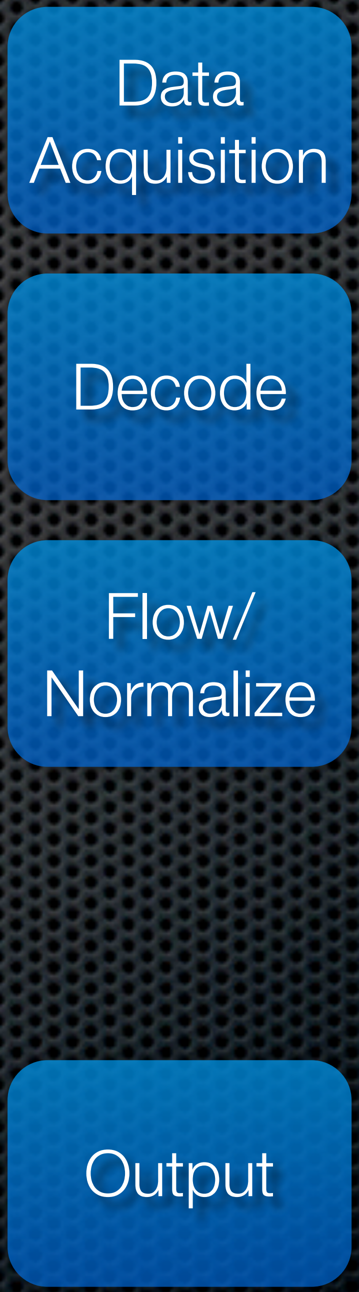
Data Acquisition

Decode

Preprocess

Detect

Output

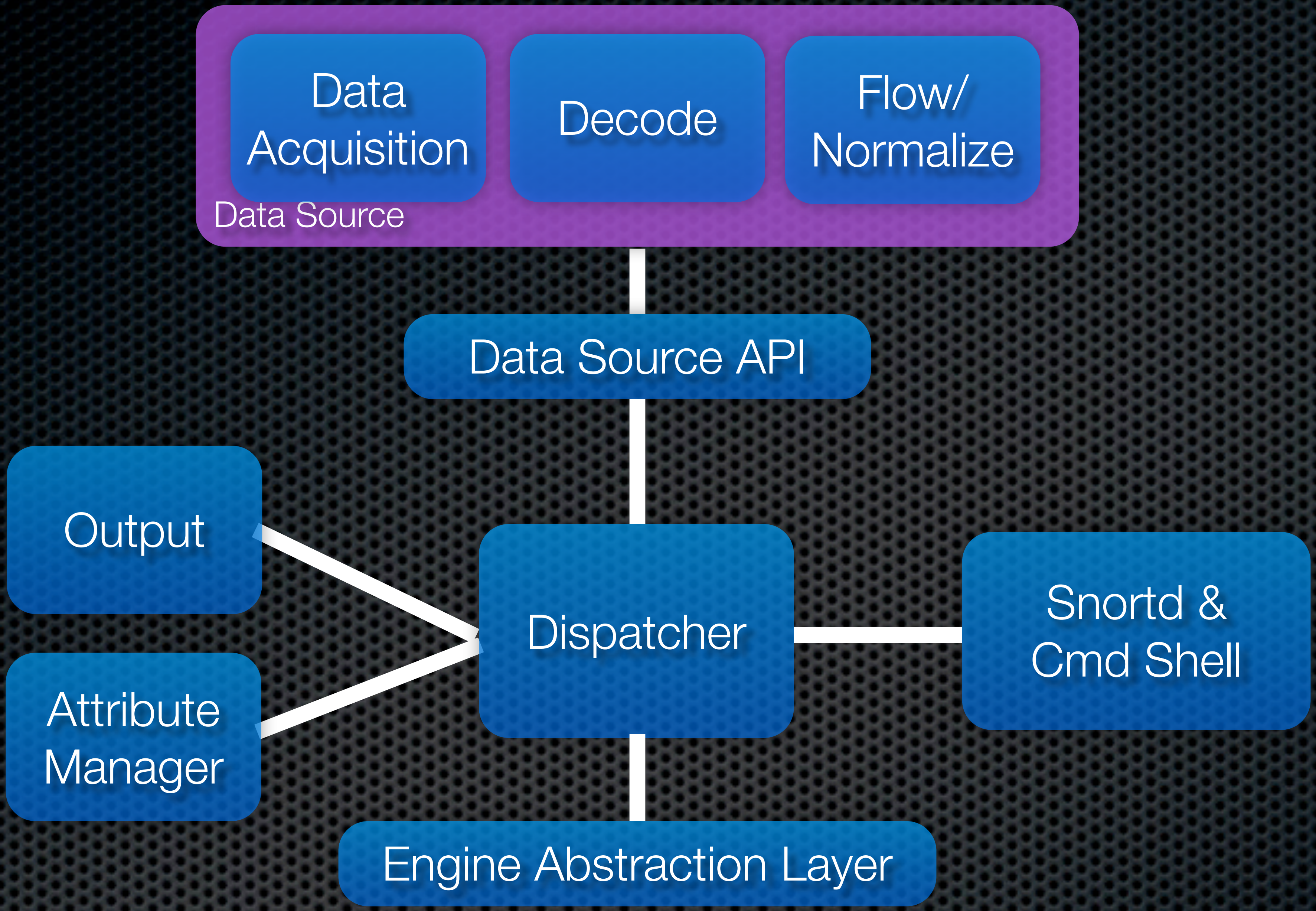


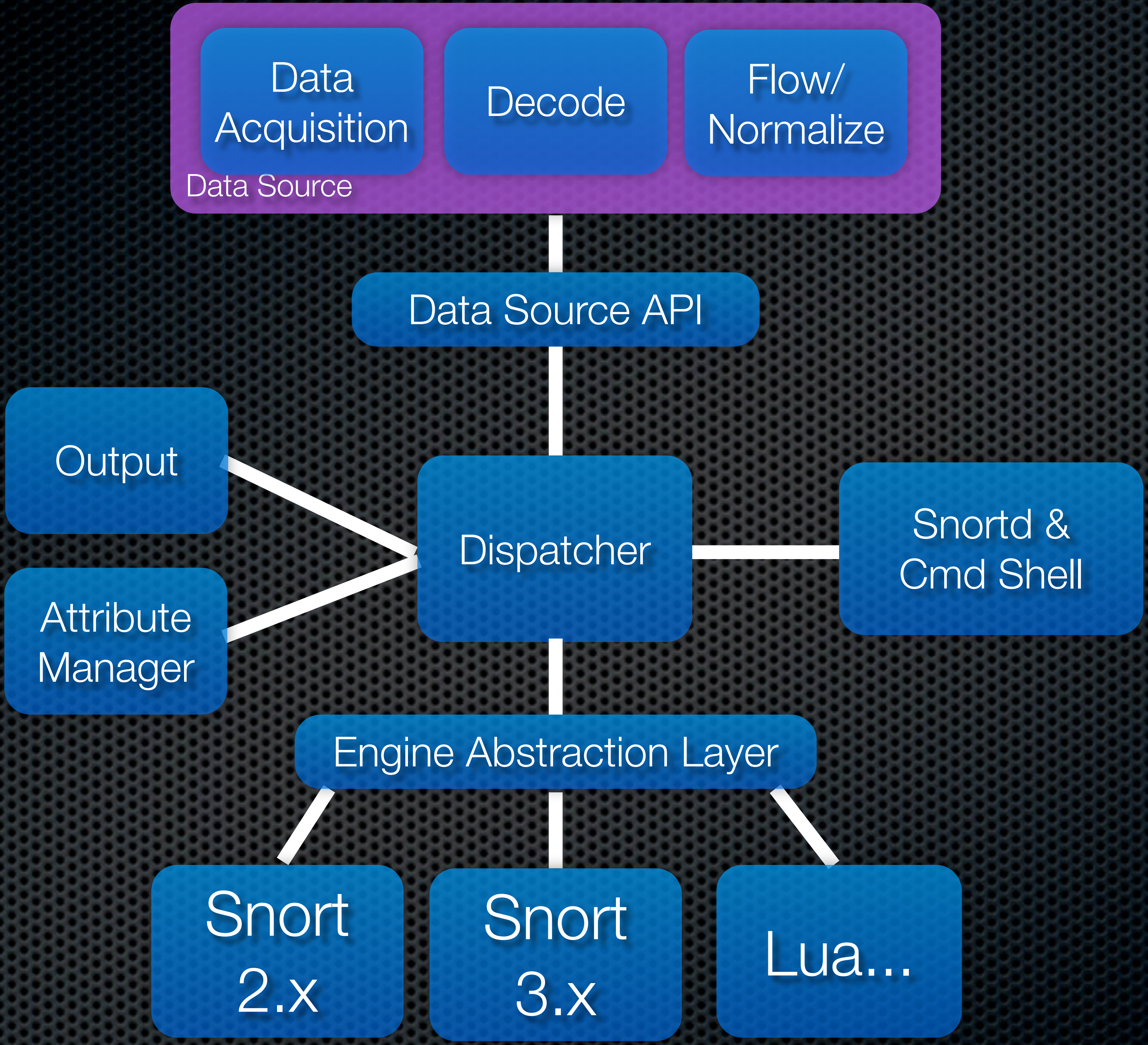
Data Acquisition

Decode

Flow/ Normalize

Output


Data Acquisition
Decode
Flow/ Normalize
Data Source
Data Source API
Output
Dispatcher
Snortd & Cmd Shell
Attribute Manager
Engine Abstraction Layer


Data Acquisition

Decode

Flow/ Normalize

Data Source

Data Source API

Output

Dispatcher

Snortd & Cmd Shell

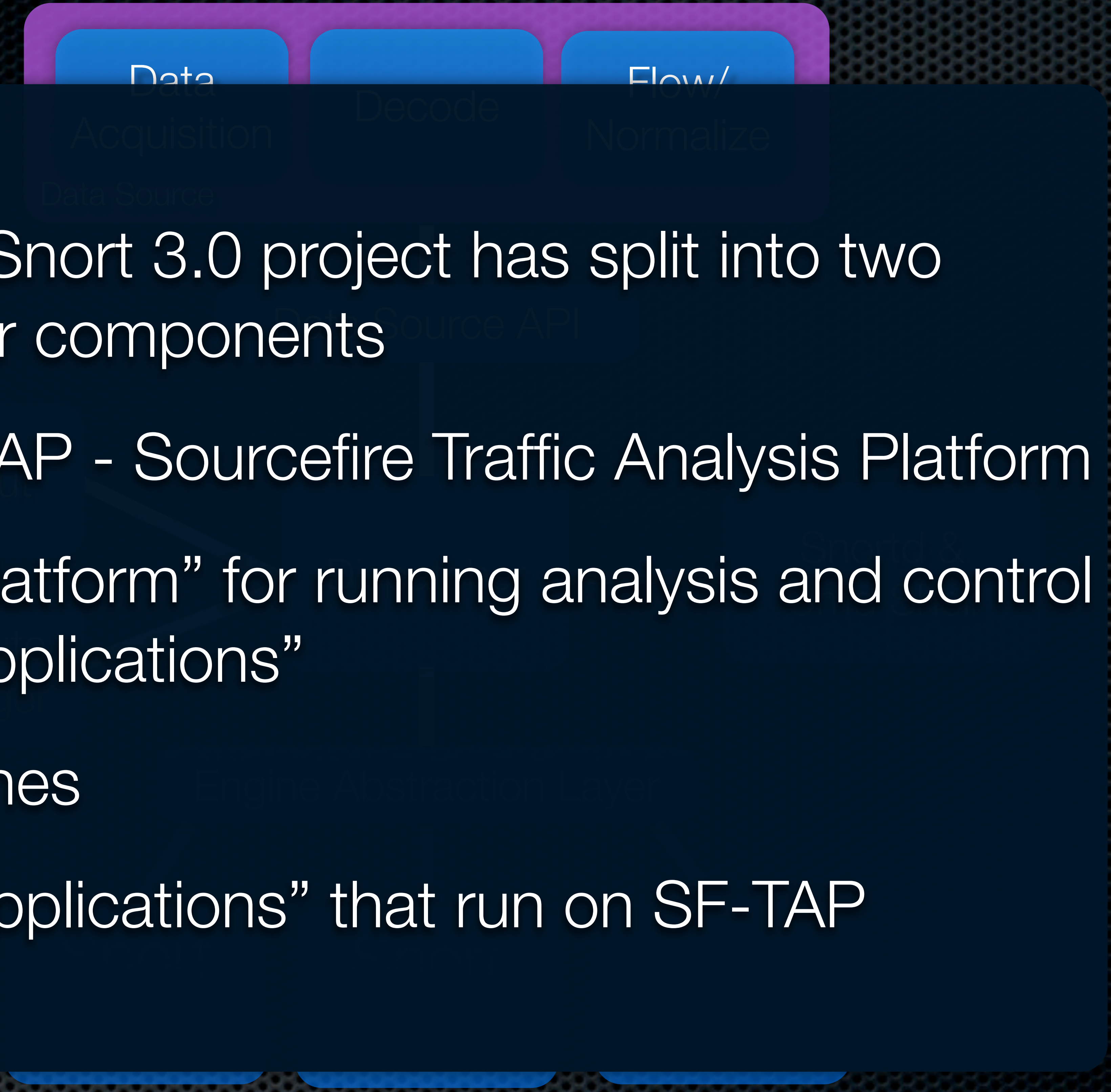
Attribute Manager

Engine Abstraction Layer

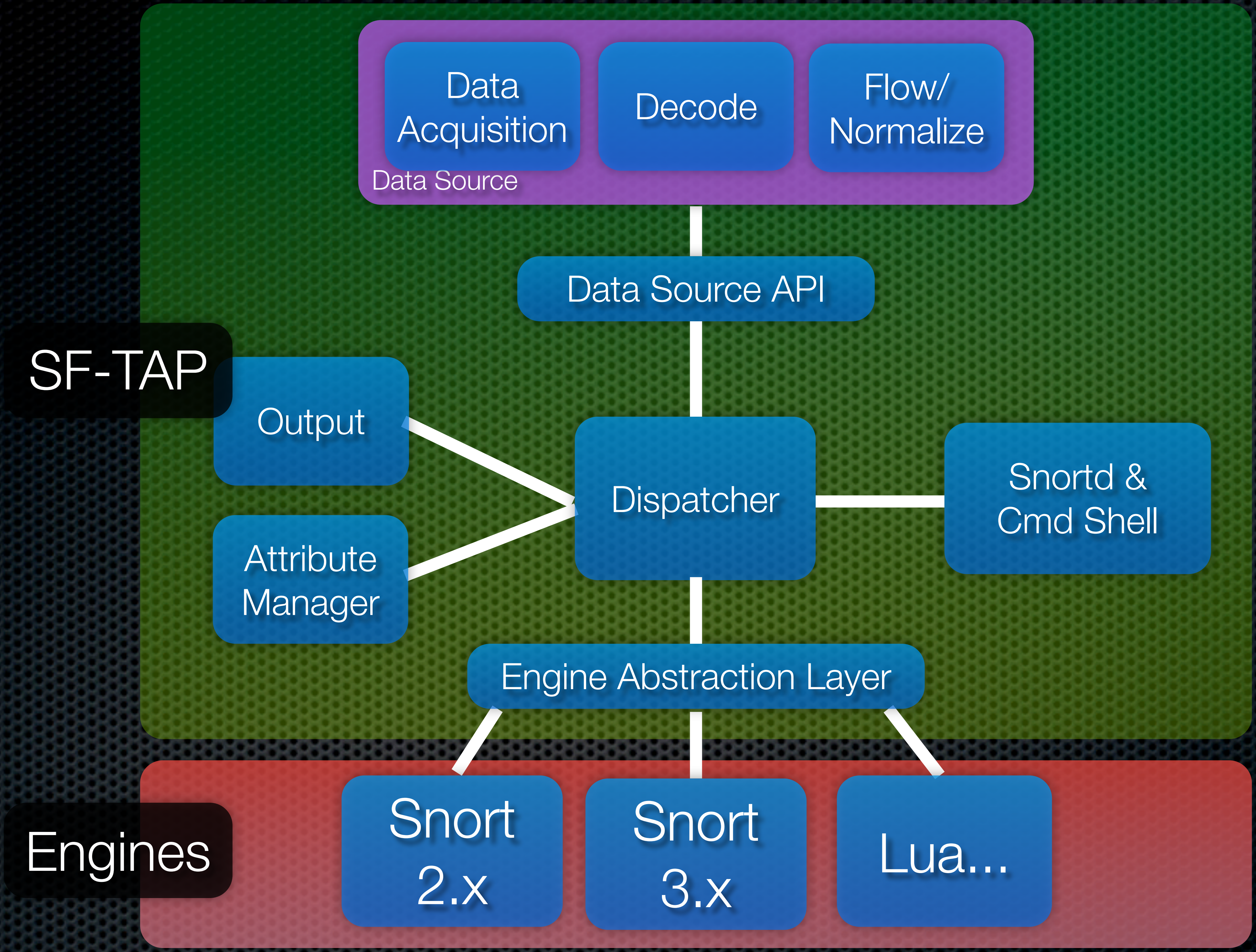
Snort 2.x

Snort 3.x

Lua...



- The Snort 3.0 project has split into two major components
- SF-TAP - Sourcefire Traffic Analysis Platform
  - “Platform” for running analysis and control “applications”
- Engines
  - “Applications” that run on SF-TAP


Data Acquisition
Decode
Flow/ Normalize
Data Source
Data Source API
SF-TAP
Output
Attribute Manager
Dispatcher
Snortd & Cmd Shell
Engine Abstraction Layer
Engines
Snort 2.x
Snort 3.x
Lua...

DAQ Subsystem

- Fully pluggable and extensible via API
- PCAP & IPQ support initially

```
typedef struct _daq_module
{
    char *name;
    u_int32_t type;

    void *(*config)(daq_interface_config_t *);
    int (*init)(void *);
    int (*daq_acquire_cb)(void *);
    int (*close)(void *);
    int (*get_devtype)(void *);
    int (*get_capabilities)(void);
    int (*dump_stats)(void *, u_int32_t *, u_int32_t *);
    int (*register_callback)(void *, daq_data_source_t *, struct _daq_module *, daq_analysis_func_t);
    int (*free_daq)(void *);
    int (*show_config)(void *);
    int (*set_filter)(void *, const char *);
    int (*name_to_index)(void *, const char *, unsigned *);
    int (*index_to_name)(void *, unsigned, const char **);
    int (*finish_packet)(void *, int, void *);
    int (*send_reset)(void *, void *, void *, const u_int8_t *,
                      unsigned, int);
    logging_api_t *logging_api;
} daq_module_t;
```

Decoder Features

```
typedef struct _proto_layer
{
    const u_int8_t  *data;
    u_int16_t       protocol;
    u_int16_t       orig_proto;
    int             size;
    int             length;
    u_int32_t       flags;
    struct _decoder *decoder;
} proto_layer_t;
```

```
typedef struct _decoder
{
    char *name;
    u_int32_t   proto_number;
    u_int32_t   proto_id;

    decoder_init_func   init;
    decoder_decode      decode;
    decoder_print       print;
    decoder_get_ssn     get_ssn_data;
} decoder_t;
```

```
typedef struct _packet
{
    struct _packet          *next;
    size_t                  serial;
    const packet_header_t   pkth;

    proto_layer_t           layer[MAX_LAYERS];
    u_int32_t               flags;
    int                     current_layer;
    int                     encapsulated;
                       :
                       :
```

Decoder Features

- Again, fully pluggable and extensible via API
- Much more natural support of encapsulation
- Supports (today):
    - Ethernet, PPP, PPPoE
    - 802.1Q VLAN, MPLS, GRE, ARP
    - IPv4, IPv6, TCP, UDP, ICMP, ICMPv6

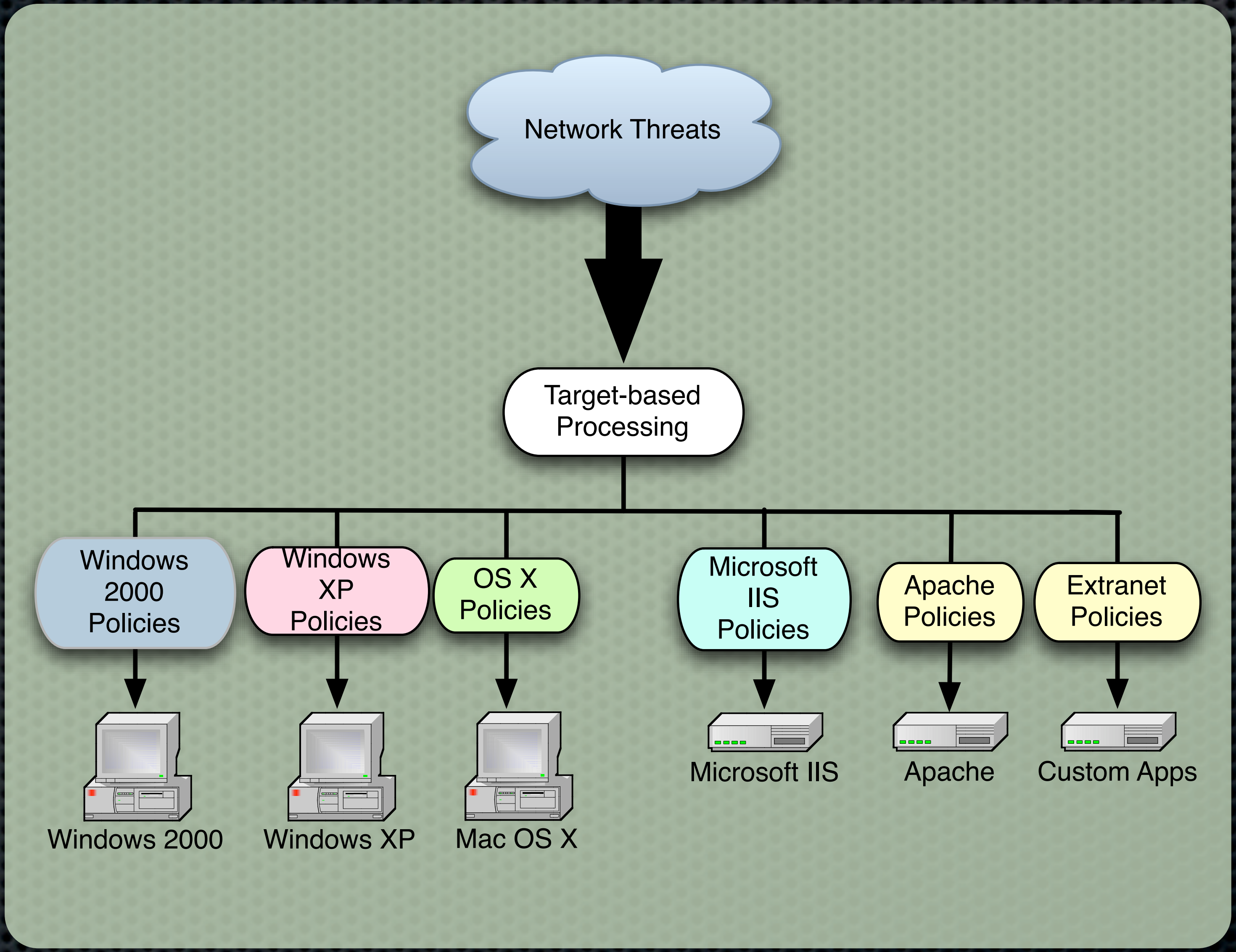
Flow Management

- SF-TAP supports two-level flow acceleration
  - Engines can signal dispatcher to ignore flows
    - Dispatcher stops forwarding traffic to that engine thread for duration of flow
  - Dispatcher can signal flow manager to “fastpath” a flow if all engines “sign off” on it
    - Fastpath flows stay within the data source
- Flow Slots - state data is stored outside engine threads
  - Run-time config can be changed without losing state!

Attribute Manager

- Network map can be kept resident in the engine
- Addressable/updatable in real-time via the snortd command shell
- Enables self-tuning analysis engines
  - Tell the network what it’s defending and it’ll figure out how to defend it!

Adaptive (Target-based) Detection



Data Source API

```
typedef struct _data_source
{
    s_mutex_t               dsrc_mutex;
    data_source_config_t    config;
    volatile int            run;
    int                     inuse;
    void                    *daq_config;
    daq_module_t            *daq;
    int                     daq_flags;
    flow_manager_t          *flow_mgr;
    defrag_manager_t        defrag_mgr;
    decode_instance_t       *decode_instance;
    size_t                  packet_count;
    dsrc_callback_idle_func idlefunc;
    ref_engine_t            *user_context;
    packet_t                *free_packet_list;
    traffic_t               *free_traffic_list;
    s_mutex_t               mutex;
    time_t                  last_packet;
} i_data_source_t;

int           dsrc_init(void);
void          dsrc_cleanup(void);
/*
 * Show list of data source instances
 */
int           dsrc_show_sources();
/*
 * Create/delete a data source
 */
int dsrc_new(data_source_config_t *ds_config);
int dsrc_delete(const char *name);
int dsrc_config_daq(data_source_t *src);
/*
 * Run or stop a configured instance of a data source
 */
int dsrc_start(data_source_t *dsrc);
int dsrc_stop(data_source_t *name);
int dsrc_run(data_source_t *src);
int dsrc_finish_traffic(data_source_t *data_src, struct _traffic *t,
                        ANALYZER_ACTION action);
int dsrc_finish_flow(flow_t *flow);
/*
 * Registration methods for user-provided code
 */
int dsrc_register_idle_function(data_source_t *src,
                                dsrc_callback_idle_func idlefunc);
int dsrc_register_user_context(data_source_t *src,
                               ref_engine_t *context);
```

```
/*
 * Data source lookup function.
 * Must call dsrc_release to release the reference when no longer
 */
data_source_t *dsrc_get_dsrc_byname(const char *name);
/*
 * Data source release function.  Must be called after dsrc_get_d
 */
int           dsrc_release(data_source_t *src);
/*
 * Show a data source's configuration
 */
int           dsrc_show_config(data_source_t *src);
int           dsrc_show_config_byname(const char *name);
/*
 * Show stats
 */
int dsrc_show_stats(data_source_t *src);
int dsrc_get_run_state(data_source_t *src, int * const run);
int dsrc_set_run_state(data_source_t *src, const int run);
int dsrc_get_inuse_state(data_source_t *src, int * const inuse);
int dsrc_set_inuse_state(data_source_t *src, const int inuse);

/*
 * getters/setters for data source config
 */
int dsrc_get_config_name(data_source_t *src, const char ** const
int dsrc_set_config_name(data_source_t *src, const char * const n
int dsrc_get_config_type(data_source_t *src, const char ** const
int dsrc_set_config_type(data_source_t *src, const char * const t
int dsrc_get_config_interface(data_source_t *src, const char ** c
int dsrc_set_config_interface(data_source_t *src, const char * co
int dsrc_get_config_filename(data_source_t *src, const char ** co
int dsrc_set_config_filename(data_source_t *src, const char * con
int dsrc_get_config_snaplen(data_source_t *src, int * const snapl
int dsrc_set_config_snaplen(data_source_t *src, const int snaplen
int dsrc_get_config_flags(data_source_t *src, u_int32_t * const f
int dsrc_set_config_flags(data_source_t *src, const u_int32_t fla
int dsrc_set_config_display(data_source_t *src, const u_int32_t f
int dsrc_get_config_display(data_source_t *src, u_int32_t * const
int dsrc_get_config_verbose_mode_string(int flags, char **modestr
int dsrc_get_config_filter_cmd(data_source_t *src, const char **
int dsrc_set_config_filter_cmd(data_source_t *src, const char * c
int dsrc_get_config_mpls_encap(data_source_t *src, const char **
int dsrc_set_config_mpls_encap(data_source_t *src,
                               const char * const proto_name);
int dsrc_get_config_maxidle(data_source_t *src, time_t * const ma
int dsrc_set_config_maxidle(data_source_t *src, const time_t maxi
int dsrc_get_config_maxflows(data_source_t *src, size_t * const m
int dsrc_set_config_maxflows(data_source_t *src, const size_t max
int dsrc_get_config_flow_memcap(data_source_t *src, size_t * cons
int dsrc_set_config_flow_memcap(data_source_t *src, const size_t
int dsrc_get_config_max_count(data_source_t *src, size_t * const
int dsrc_set_config_max_count(data_source_t *src, const size_t co
int dsrc_get_daq_type(data_source_t *src, int * const type);
int dsrc_get_daq_intf_index(data_source_t *src, const char *, uns
int dsrc_get_daq_intf_name(data_source_t *src, unsigned, const ch
int dsrc_is_inline(data_source_t *src);
```

Data Source API

```
/*
 * Data source lookup function.
 * Must call dsrc_release to release the reference when no longer
 */
data_source_t *dsrc_get_dsrc_byname(const char *name);
/*
 * Data source release function.  Must be called after dsrc_get_d
```

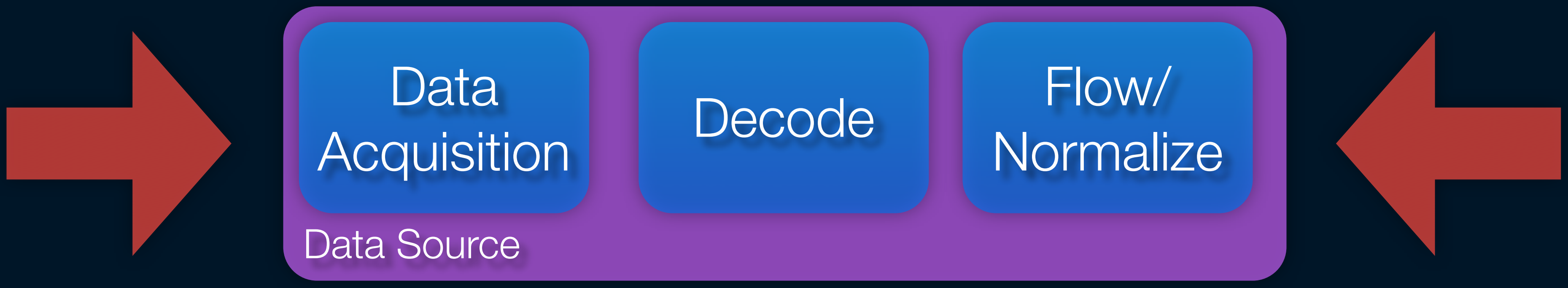
```
typede
{
    s_
    da
    vo
    in
    vo
    da
    in
    fl
    de
    de
    si
    ds
    re
    pa
    tr
    s_
    ti
} i_da

int
void
/*
 * Sho
 */
int
/*
 * Cre
 */
int ds
int ds
int ds
/*
 * Run
 */
int ds
int ds
int ds
int ds

int ds
/*
 * Reg
 */
int dsrc_register_idle_function(data_source_t *src,
                                dsrc_callback_idle_func idlefunc);

int dsrc_register_user_context(data_source_t *src,
                               ref_engine_t *context);
```

- Abstraction layer for the Data Source subsystem
- SF-TAP doesn't care if the Data Source is implemented as software *or hardware*
- Insert hardware accelerator HERE!



Data Acquisition

Decode

Flow/ Normalize

Data Source

```
int dsrc_get_daq_type(data_source_t *src, int * const type);
int dsrc_get_daq_intf_index(data_source_t *src, const char *, uns
int dsrc_get_daq_intf_name(data_source_t *src, unsigned, const ch
int dsrc_is_inline(data_source_t *src);
```

Dispatcher

- Manages data flow between the Data Source subsystem and the Engines
- Engines may analyze traffic in any combination of serial and parallel processing
    - Handy if you want to run Snort + RNA on the same traffic at the same time...
- Per-thread traffic distribution management and fast-pathing

Snort 3.0 Language

- Snort is not a language project!
- Snort's rules and configuration languages are what is know as a "Domain Specific Language" (DSL)
- Embed a language designed for implementing DSL's!
- **Snort 3.0 is using Lua**

Snort 3.0 Langauge FAQs...

- Will I have to throw out my existing rules?
  - No! Snort 2.8.x detection framework is ported!
- Why Lua?
  - Designed for the problem space
  - Used in Nmap, Wireshark, World of Warcraft, Adobe Photoshop Lightroom, BBEdit, etc
- I heard Snort 3.0 has a command shell?!
  - Snort 3.0 is designed to run without stopping...

Progress

- Major components of SF-TAP are in advanced stages of development
- Engines
  - Snort 2.x - Porting underway
  - Snort 3.0 - Prototype under development

Timelines

- Open Source 1st Beta in 2Q08
  - Snort 2.x engine only
  - Open Source initial relase
    - 4Q08
- Snort 3.x engine will debut in 2009

Yes, Snort 3.0 is Open Source

Demo