JOHNS HOPKINS
APPLIED PHYSICS LABORATORY

# BPSec Library (BSL) Preliminary Design

## Prepared by JHU/APL for NASA AMMOS

Brian Sipos

# Development Conventions

- Work products will be hosted on a JPL-controlled GitHub organization NASA-AMMOS
  - A BSL Developers team has been created for access control
  - A BSL project has been created for ticket visibility and milestone planning
  - A BSL and BSL-private (fork) repository has been created for BSL source
    - The BSL source repository wiki will host the BSL plans for AMMOS dynamic release planning
  - A BSL-docs repository has been created for out-of-source docs (Product Guide and User Guide)
- Intermediate work will be kept on the private fork of the source repository
- APL tech-transfer needs to pre-approve an open source development plan
- Library release versioning will be based on SemVer-2.0 conventions
- Source repositories will have CI jobs to verify active branches and pull requests satisfy:
  - Build and unit test in Ubuntu, and RHEL environments
  - Goal to build in cross-compiled environment for RTEMS/LEON
  - Extract API documentation (Doxygen) and publish to associated Pages
- Unit testing includes coverage results for AMMOS DDR reporting

# Source and Build Conventions

- Adhere to C99 language (ISO/IEC 9899:1999)
  - No compiler-specific extension use
  - Use language-specified primitives from <stdint.h>, <stdbool.h>, <inttypes.h>
  - Use `extern "C"` for headers to have symbol compatibility
  - Test fixtures can expand this envelope
- Following NASA cFE/cFS source and symbol conventions
  - Use a source formatting tool `clang-format` to adhere to cFS conventions
- A tool-generated “bsl_config.h” header contains compile-time options (rather than compiler defines)
- Enable inline, machine-readable documentation
  - Using the Doxygen tool to extract and publish documentation
  - Output compatible with GitHub Pages (similar to current DTNMA Tools workflow)
- Use a build automation tool CMake with local modules and “modern CMake“ style
- Use COTS libraries where possible (for *backend* implementation)
  - Streaming CBOR encoding/decoding via QCBOR
  - Crypto functions via OpenSSL and MbedTLS
  - Type safe, dynamic data containers via M*LIB
  - Time keeping via POSIX
  - Unit test fixtures via Unity
  - Coverage accounting via `gcovr`

# API Terminology and Conventions

- Naming follows C99 convention of lowercase names with underscore separators
- All API is prefixed with “bsl_” as a C99 non-namespace
- As much as possible functions follow “<noun>_<verb>” convention
- Terminology for struct lifecycles:
  - An “alloc” function allocates from a memory pool, and a “free” function de-allocates and returns to the pool
  - An “init” function initializes the state of a struct, and a “deinit” function cleans up state as necessary
- Terminology for bundles:
  - To “delete” a bundle is a BPA processing activity
  - To “discard” a bundle drops it silently, and “discard” a block removes it from a bundle
- Function return values indicate status (zero is success, others are error)
- Maintain const-correctness where possible to aid in API clarity and compiler optimization
- Separate public API from compilation-unit-local “static” functions

# Overall Activities of the BSL

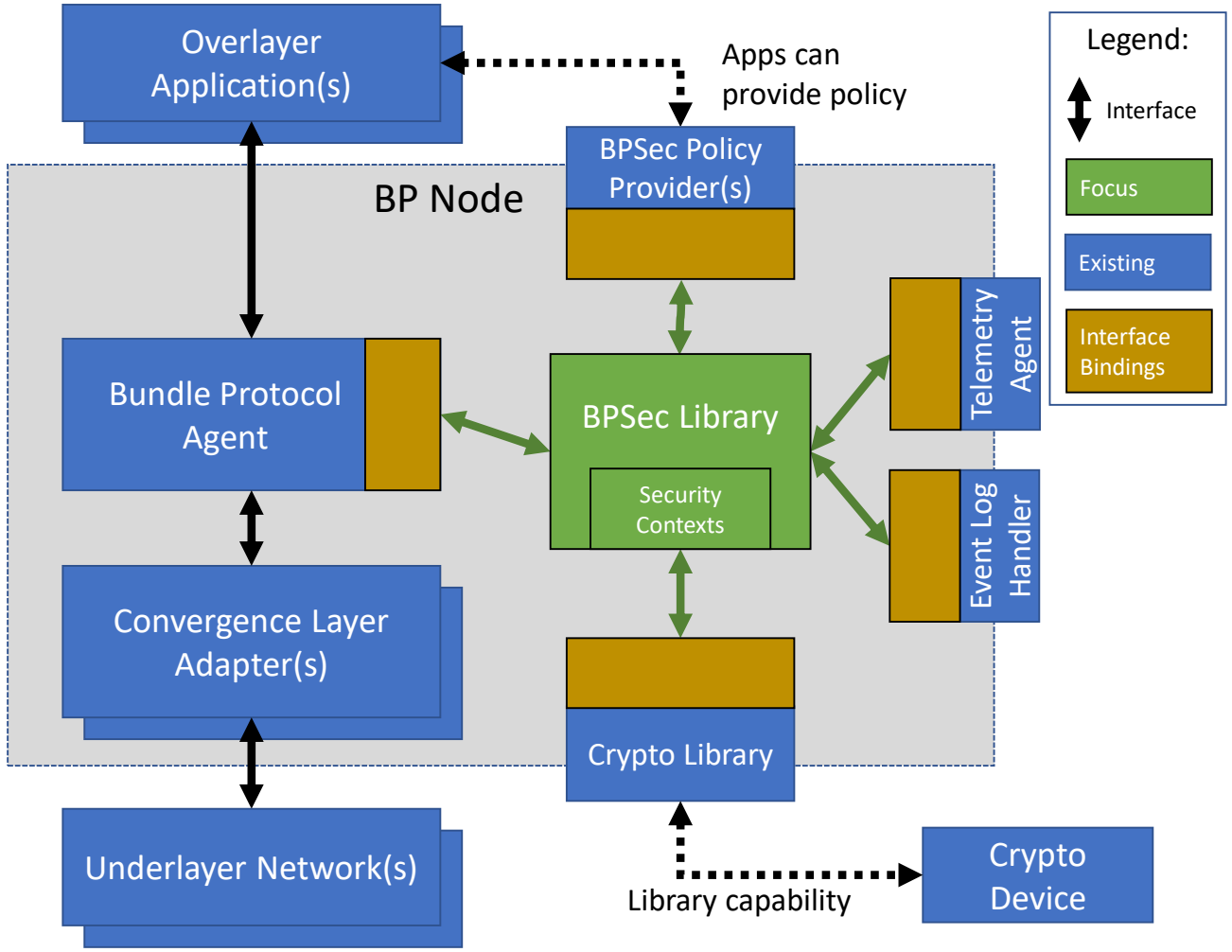
- Policy and bundle processing
- Security Operation and BIB/BCB processing
- Crypto functions delegated to an external library
- Telemetry and event logging with host

# Design Terminology and Conventions

- Using SysML 1.5 for notional diagrams, no requirement mapping or lower-level flowdown
- Top packages are: data, software systems, artifacts
- Data packages correspond with protocol domains
- Systems packages roughly correspond with BSL boundaries to: BPA, Host, or COTS libraries (the term “COTS” is used generically for any third-party library)
- Systems blocks roughly correspond with source files (or pairs of header and compilation-unit files)
  - For C99, abstract base blocks represent header-declared APIs and concrete specializations represent compilation-unit-defined implementations of APIs
  - For the dynamic backend, these are implemented by descriptor structs containing callback pointers
- Artifacts blocks correspond with linked libraries and allocations to libraries

# Context Diagram

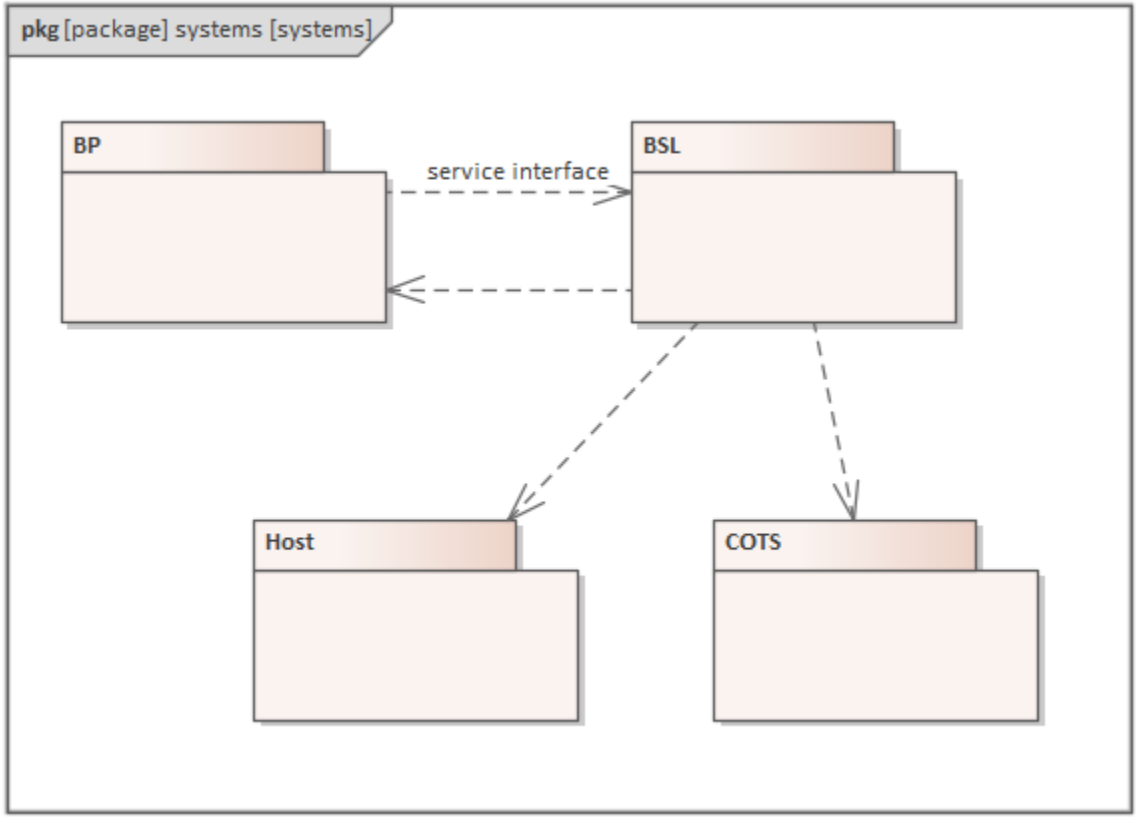
- This graphic was taken from the BSL Software Requirements Document
- Indicates interfaces between BSL and systems outside of the BSL
- The BSL has two types of interface:
  - Interfaces *into* the BSL: the BPA uses the public API to drive the BSL
  - Interfaces *out of* the BSL: the BSL uses external APIs to perform functions (e.g. calls into crypto library or event log)
- The principal interface into the BSL is the Security Service interface used by the BPA



# Logical Packages and Dependencies

For later design diagrams, design artifacts are grouped into four packages:

- **BSL** contains blocks specific to the BSL implementation
- **BP** contains blocks related to the BPA or BP-specific processing
- **Host** contains blocks related to host functions outside of the BPA (e.g. an event log sink)
- **COTS** is used as a basket to contain blocks related to third-party libraries and systems not strictly part of the host functional interface

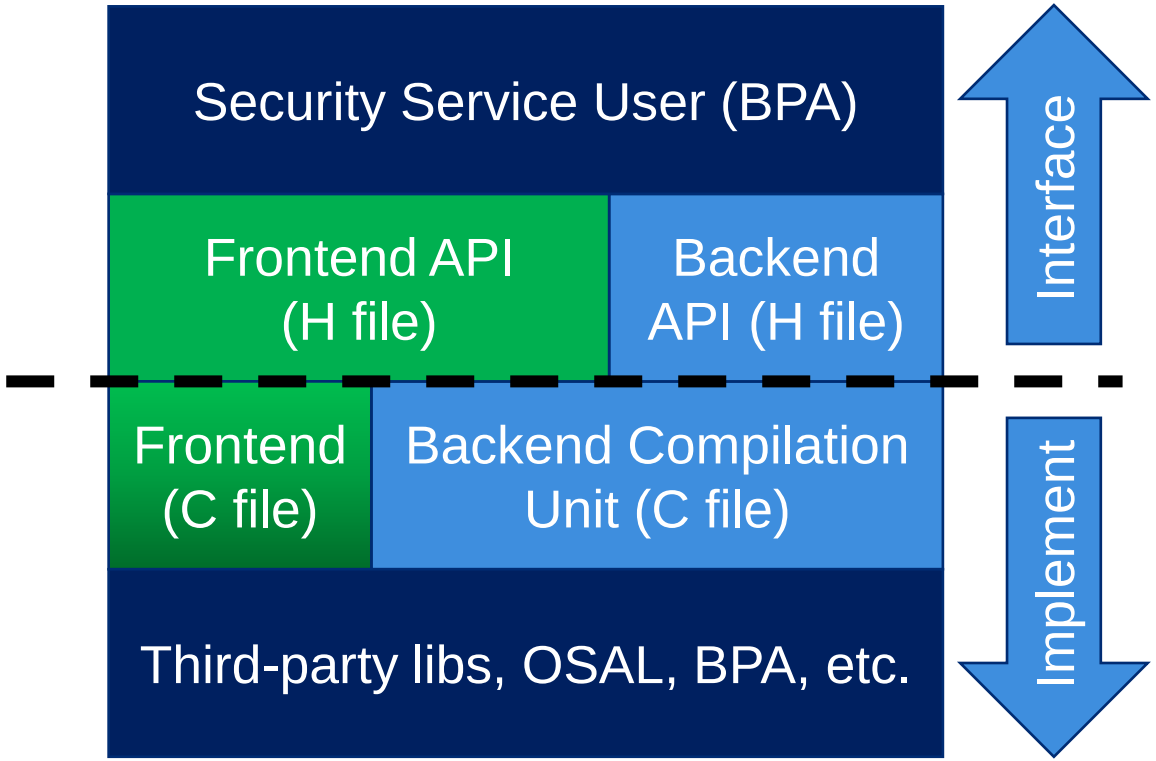


JOHNS HOPKINS
APPLIED PHYSICS LABORATORY

# Structural Breakdown

# Implementation Patterns

- There is a project goal to separate a stable API from specific memory allocation strategies
  - Flight software restricts memory allocation patterns
- BSL separates internal structure into two parts:
  - Frontend API
  - Backend implementation
- Frontend is mostly header contents:
  - Forward-declared (opaque) structs
  - Public API function declarations
  - Possible pure-algorithm definitions
  - No heap allocation
- Backend is mostly compiled definitions:
  - Defined structs (i.e. member visibility)
  - Public API function definitions
  - Possible “private API” header declarations
  - Dynamic backend uses dynamic heap allocation, other backend could use static memory pool(s)



# BSL Artifacts

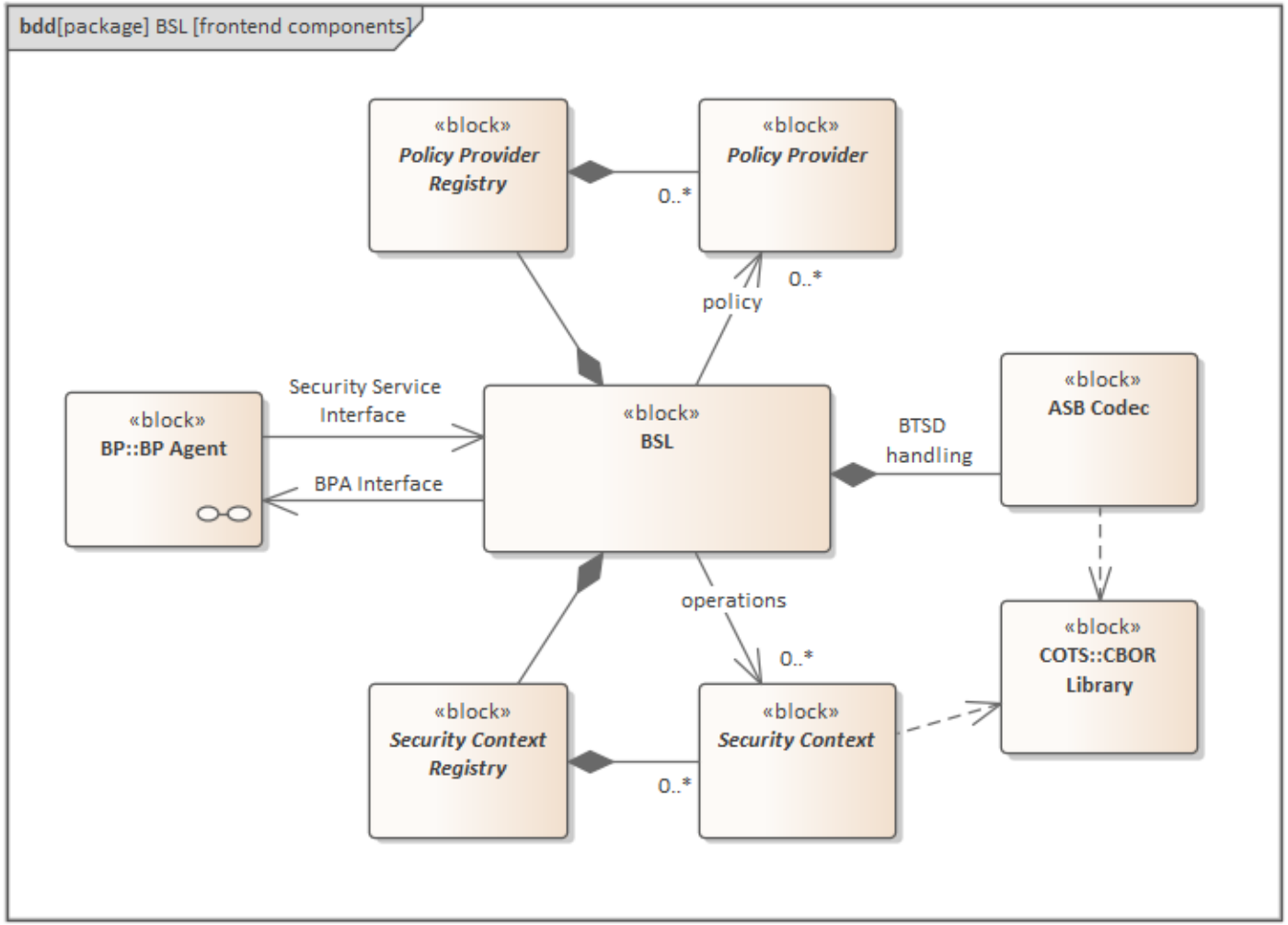
- The BSL will be split into four artifacts (*i.e.* linked libraries):
  - BSL Frontend
  - Dynamic Backend
  - Example Default Security Contexts
  - Example Policy Provider database, which allows using the default Security Contexts
- Each of those will be built and installed as part of the normal CMake configuration
- Additionally, there will be test fixture artifacts:
  - Binding to mock BPA and other external interfaces for unit tests
  - Alternative Policy Provider instances to exercise unit tests
  - Alternative Security Context instances to exercise unit tests

# Artifact Contents

- BSL Frontend:
  - Headers declaring “abstract” interfaces used by the BSL
    - Sequential BTSD
    - Bundle Context
    - BSL Context
  - Definitions of algorithms which use Frontend APIs are independent of backend implementation
    - Overall BSL workflow
    - Security activity and operation sequencing logic
- Dynamic Backend:
  - Compilation units satisfying the BSL Frontend APIs using heap-allocated dynamic storage
  - Definition of a Bundle Context based on fully caching block information
  - Definition of Sequential BTSD based on simple, flat array iteration
  - Definition of a BSL Context with an API to register/deregister Policy Providers and Security Contexts
- Example Default Security Contexts:
  - Implementation of the BIB-HMAC-SHA2 and BCB-AES-GCM contexts within the BSL structures
- Example Policy Provider:
  - Use database and JSON representation from ION BPSec

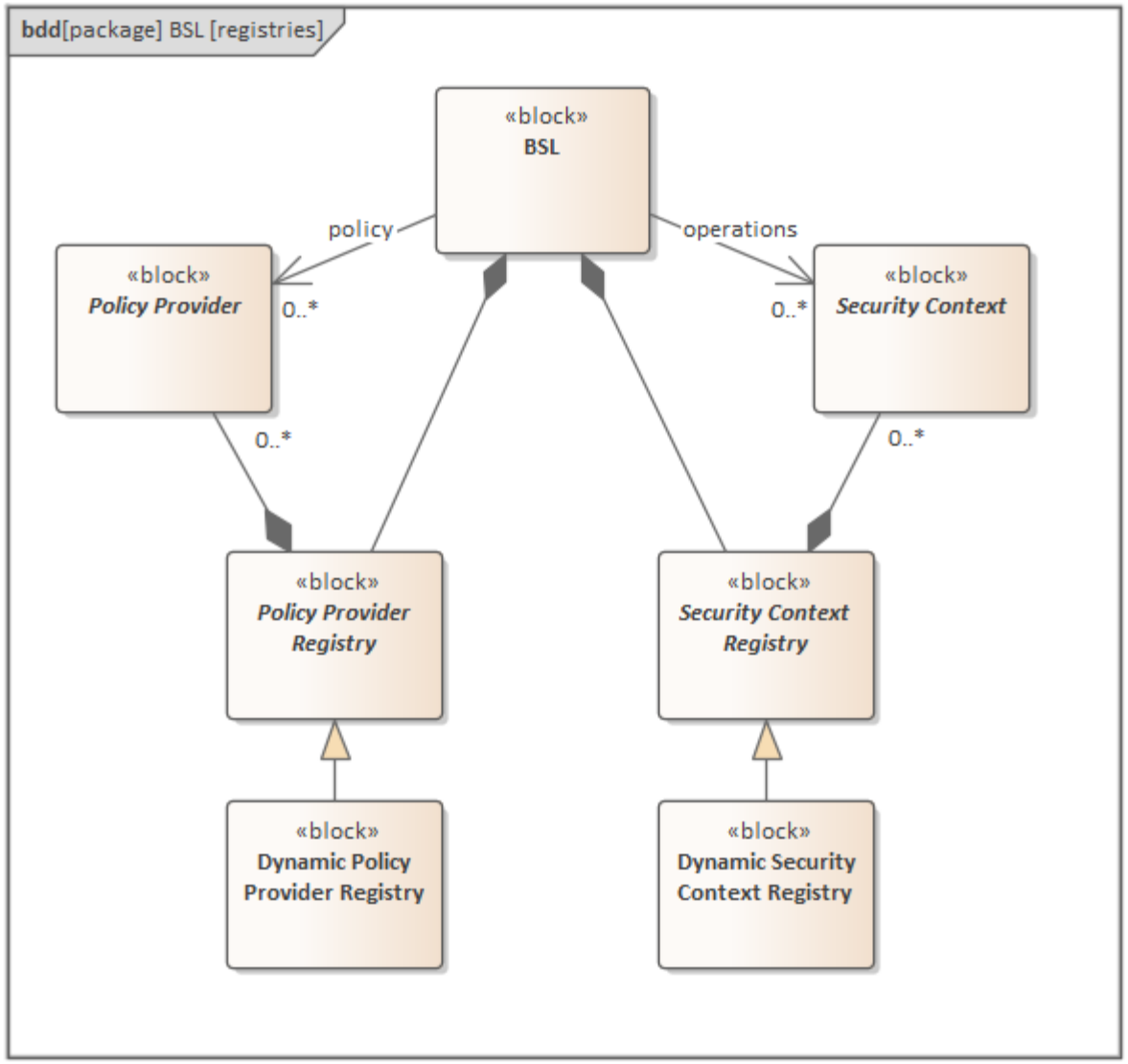
# BSL Frontend Components

- The three major frontend interactions of the BSL are:
  - Policy Provider Registry and instance interaction
  - Security BTSD Codec based on COTS CBOR library and BPA callbacks
  - Security Context Registry and instance interaction
- Common ASB Codec handles common structure:
  - Target block list
  - Context ID
  - Security Source (delegated to BPA)
- Security parameters and results are handled by individual Security Contexts



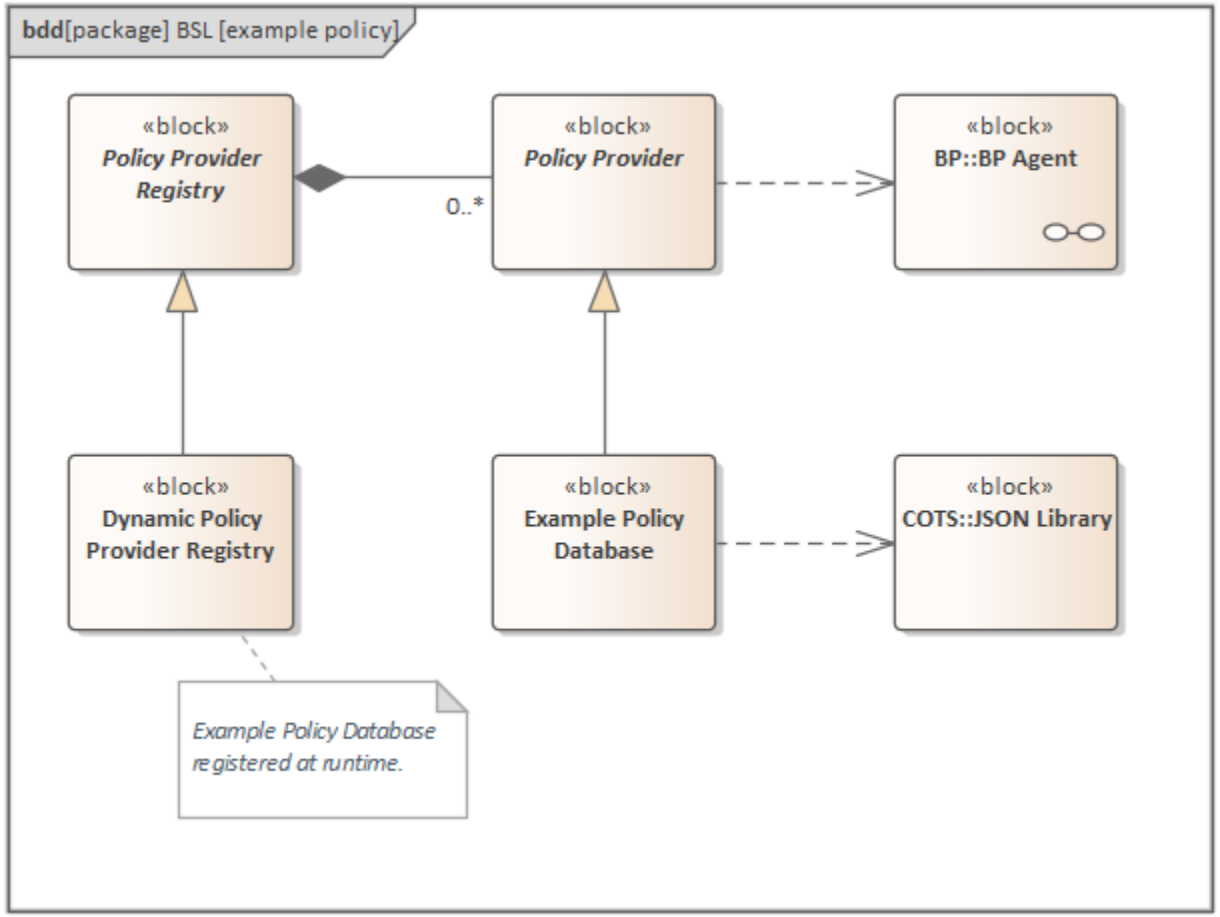
# Abstract Registry Interfaces

- In C99 an abstract interface is an API declared in a header
- The function definitions are deferred to a specific implementation
- In the BSL, this pattern is followed for Policy Providers and for Security Contexts
- An implementation of these interfaces is provided for full “out of the box” capability and for unit testing
  - Based on dynamic data structures and heap allocations
  - Provide separate, specific API for adding and removing registered instances



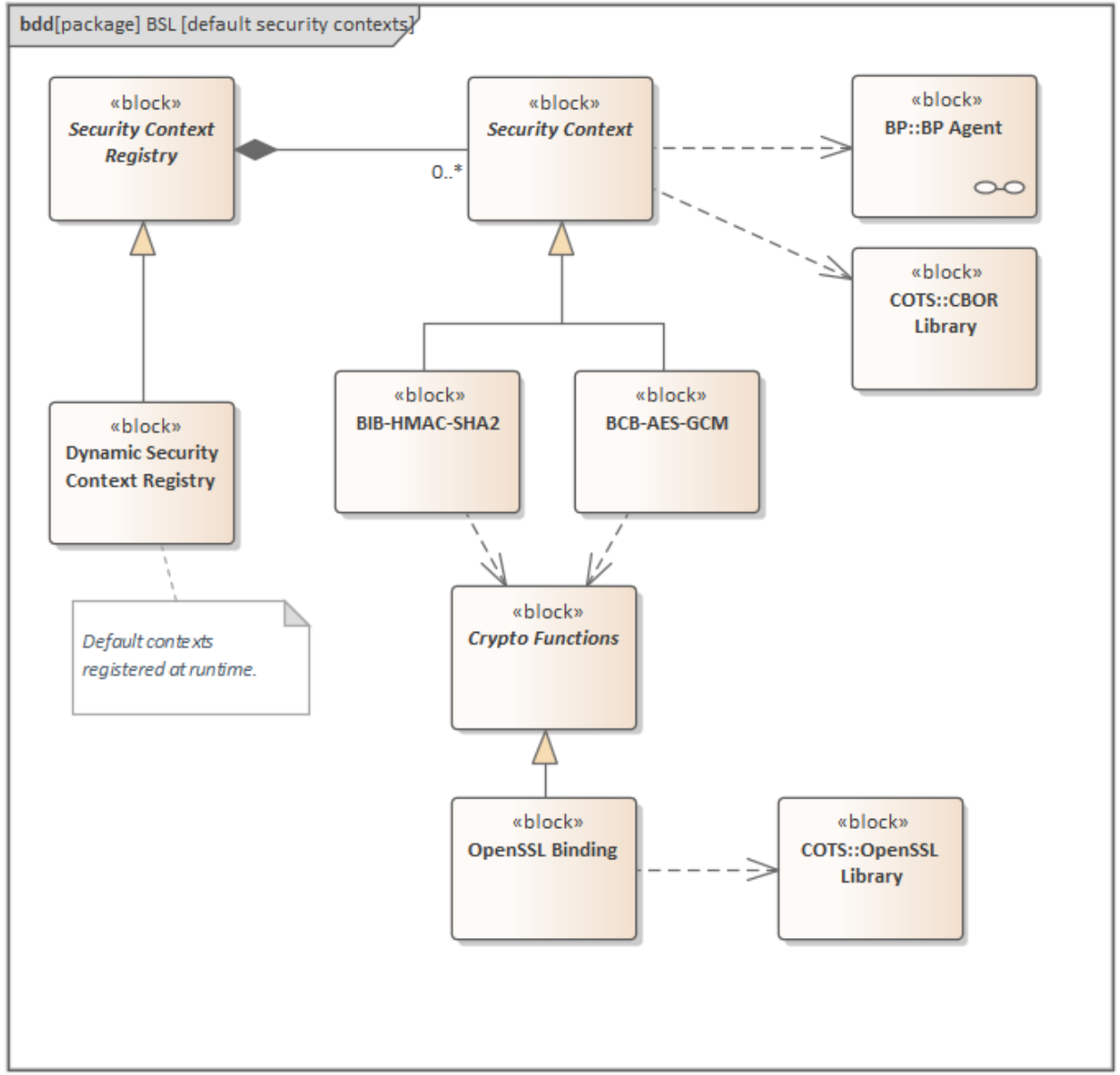
# Example Policy Database

- Example provider based on existing BPSec implementation from ION
- Uses JSON-encoded configuration file
- Implements logic specific to that information model
  - This logic is more specific than the general behavior allowed by Policy Provider API
- Provides a concrete example Policy Provider and provides a transition from existing ION security configuration



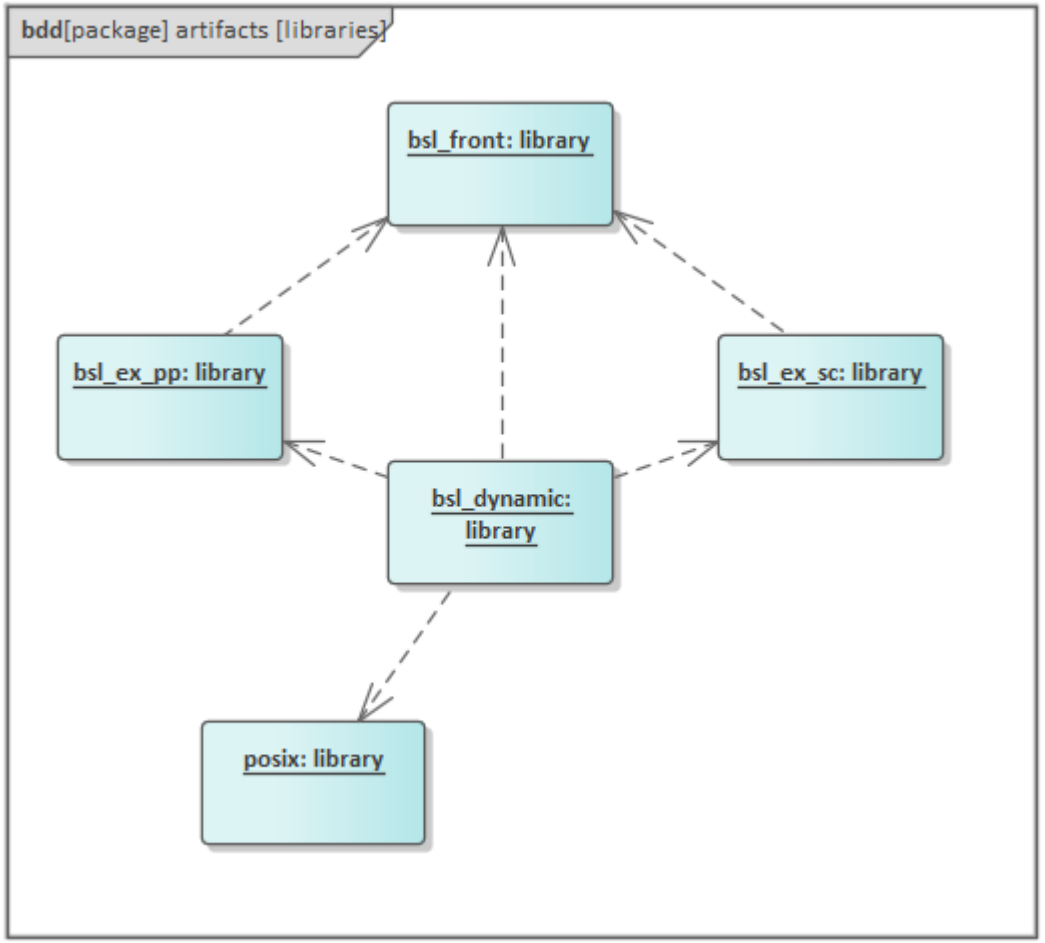
# Default Security Contexts

- Default contexts based on existing implementations of the RFC 9173 default security contexts
- All security contexts need to implement codecs for their parameter and result values
- Example crypto function bindings will be provided for OpenSSL library
  - This enables a path to FIPS 140-2 testing



# Library Structure and Linkage

- Following the logical structure outlined above, the actual built artifacts will follow the same pattern of separating:
  - `bsl_front`: Declarations of public API and implementation of generic algorithms
  - `bsl_dynamic`: Declarations of dynamic API and implementation of public API
    - Library initialization registers example providers
  - `bsl_ex_pp`: Implementation of an example Policy Provider
  - `bsl_ex_sc`: Implementations of the default Security Contexts
- Library contents are human discoverable via Doxygen produced documentation
- The linkage dependencies are *not* the same as logical uses or information flows



# Frontend Interface API Concepts

- Declared **BSL Context** is an opaque struct containing configuration for a single (thread) use of the BSL
  - Logically a container for any memory pools associated with BSL instance
- Declared **Bundle Context** is an opaque struct containing back-references to the BPA for obtaining and manipulating individual bundle's data
  - Logically a container for caching or sharing data between operations on the same bundle
- Declared **Sequential Reader** and **Writer** are opaque structs used to read or write BTSD for a single canonical block within a bundle
- Declared **Policy Action** struct and related functions
- Defined **Security Operation** struct and related algorithms
- Declared security functions specific to the Default Security Context needs

# Dynamic Interface API Concepts

- Defined **Host Descriptor** is a copyable struct containing callbacks to host functions:
  - Memory management
  - Time keeping
  - Event logging
  - Telemetry registration
- Defined **BPA Descriptor** is a copyable struct containing callbacks to access the BPA
- Defined **Policy Provider Descriptor** is a copyable struct containing callbacks to Policy Provider functions and user data pointer
- Defined **Security Context Descriptor** is a copyable struct containing callbacks to Security Context functions and user data pointer
- Defined **BSL Context** which consists of:
  - A *Host Descriptor* and *BPA Descriptor*
  - A dynamic list of *Policy Provider Descriptor* and *Security Context Descriptor*
- Defined **Bundle Context** which consists of:
  - Primary block information and encoded form
  - A list of canonical block metadata and BTSD, along with lookup maps (cached)
- Defined **Sequential Reader** and **Writer** using flat buffers

# Dynamic Backend API Workflows

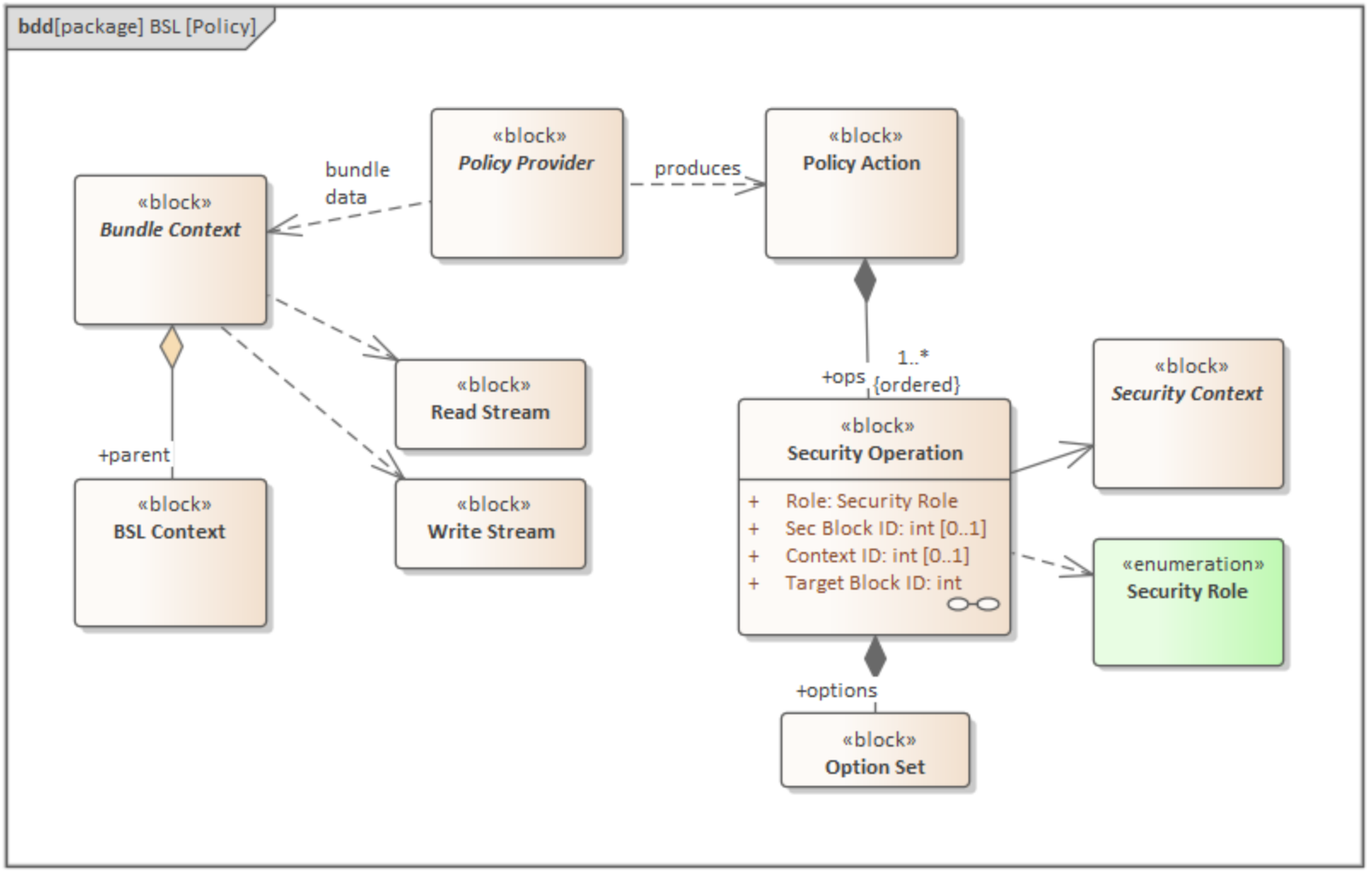
Want to be similar to existing APIs and flows for OpenSSL, MbedTLS, and others

This workflow does not involve the use of internal threads, threads are managed by the BPA

1. Populate a *BPA Descriptor* and a *Host Descriptor* for the runtime configuration
2. Initialize a *BSL Context* to be used per-thread, based on the given descriptors
3. For each bundle being processed:
   A. Initialize a *Bundle Context* for that specific bundle (the BPA may cache data here)
   B. Query each policy provider with that *Bundle Context*, which will likely result in BPA callbacks to obtain bundle/block data, possibly resulting in a Policy Action with some number of Security Operations
   C. Iterate through all *Policy Actions* and process them (in parallel if desired by BPA), which will each result in some combination of:
      - Changes to the bundle (*e.g.* BCB ciphertext substitution, BIB removal)
      - Annotations to the bundle about successful BIB verification or BCB decoding
   D. Inform each used Policy Provider of the result of all of its Security Operations to finalize its behavior
   E. Release the *Bundle Context* after use
4. Release the *BSL Context* when no longer needed

# Relationships Between Policy and Security Contexts

- The output of Policy are references to how security blocks are supposed to be used
  - Each security operation includes concrete “options” for how to use a given security context in a given role
  - These options are not one-to-one with security “parameters”
  - Some options may drive parameters for the security Source role
  - Option vs. parameter is something that current ION policy language is fuzzy about

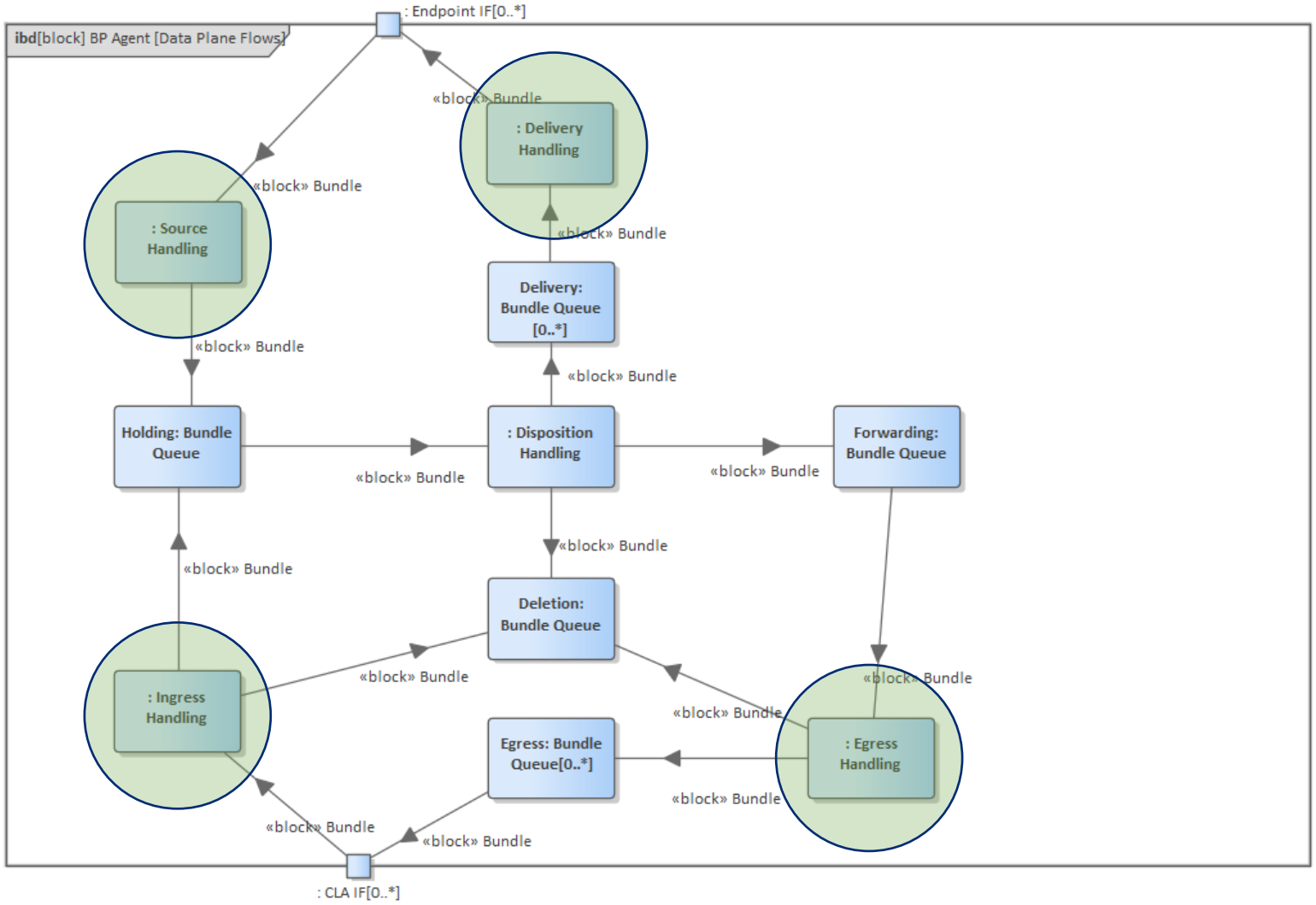


JOHNS HOPKINS
APPLIED PHYSICS LABORATORY

# Behavioral Breakdown

# Security Interaction Points

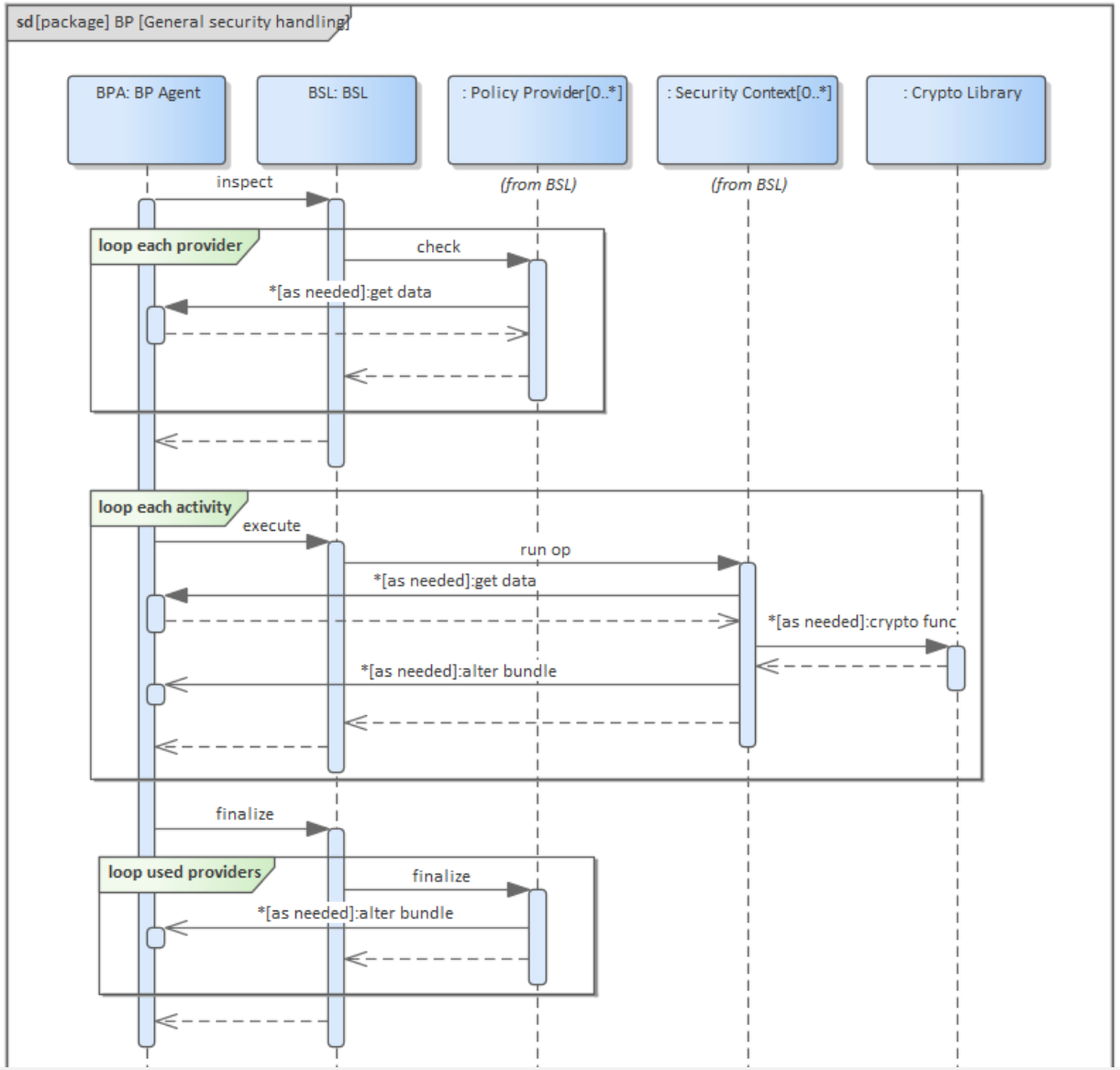
- These four interaction points are at the boundaries of the BPA for south-side (CLA) interface and north-side (App) interface
- Policy processing is identical for the four points except for an enumerated indication of which point is being processed



# Security Interaction Sequencing

At each interaction point the BPA will perform the same sequence:

1. Check with the BSL about what, if anything, needs to be done with the bundle
   - The result of each check will be a (possibly empty) set of security operations to perform, which are aggregated together for the whole bundle
2. The BPA can then perform security operations (in work threads if desired)
   - More detailed design will be needed regarding operations which require specific ordering (e.g. decrypt then verify signature).
3. Finally the result of each operation is provided back to the Policy to perform any follow-on activities (*e.g.,* failure handling).



# Bundle Inspection

This activity is broken down as:

1. The BPA calls the BSL to inspect a specific bundle
2. The BPA initializes a *Bundle Context* with some amount of pre-cached data about the contents of the bundle and its security blocks
3. The BSL iterates across registered Policy Providers and calls each to inspect the bundle
   A. Each Policy Provider will access the *Bundle Context* to obtain information related to that policy
   B. Each Policy Provider will result in some number of security actions, each a sequence of security operations
4. The BSL combines all resulting actions and checks the consistency of their operations (next slide)
5. The combined operations are kept for execution (later slide)

# Policy Action and Security Operation Ordering

- Each Policy Action represents a sequence of Security Operations as the outcome of a single Policy Provider
- Each Security Operation is an atomic activity for a single target of a single security block with a single security role (source, verifier, acceptor) along with options
  - Options of an operation can lead to parameters in a sourced security block, but are not the same

# Security Operation Execution

This activity is broken down as:

1. The BPA calls the BSL to execute specific security actions and operations
   A. Each security operation references a specific Security Context, which is dereferenced by BSL
   B. Each associated Security Context is called into to execute a specific operation
      I. The Security Context will access the *Bundle Context* to obtain information as needed
      II. The Security Context will call into a Crypto Library as needed
      III. The Security Context will access the *Bundle Context* to alter it as needed
   C. The result of each security operation is kept for the finalizing activity (next slide)

# Bundle Finalizing

This activity is broken down as:

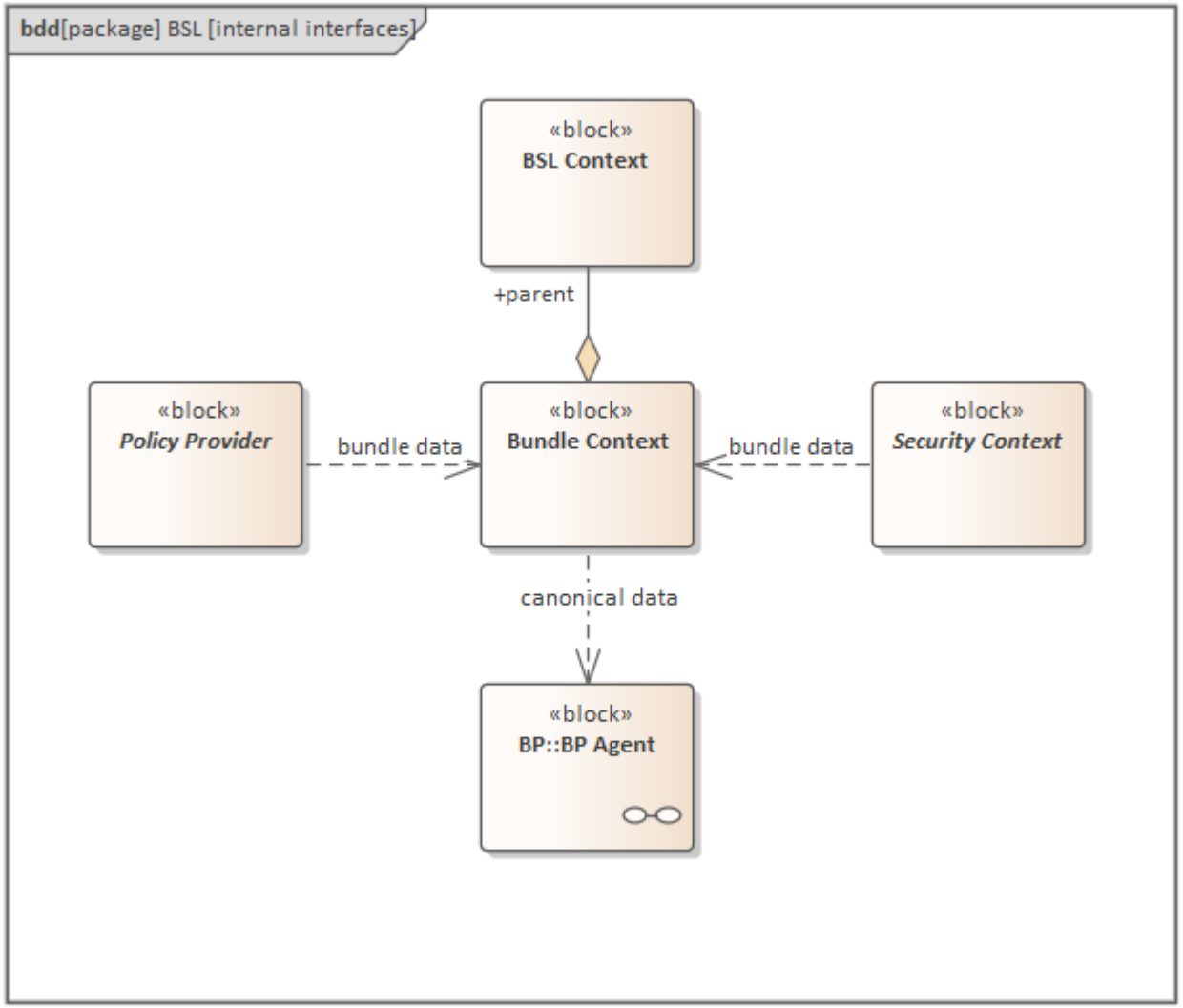
1. After all Policy Actions and their Security Operations have finished execution, the results are given back to each corresponding Policy Provider
2. Each Policy Provider has a chance to perform any follow-on logic such as:
   - Delete the bundle if some combination of operations fails to complete
   - Reprocess security operations (specific ones?) after inspecting the bundle further
   - This satisfies one bootstrapping use case where identity information is contained in payload ADU and needed by security processing

JOHNS HOPKINS
APPLIED PHYSICS LABORATORY

# Notional Interfaces and API Examples

# Logical Data Dependencies

- Within the BSL both Policy Providers and Security Contexts use local Bundle Context as an intermediary to the real bundle state
- This provides the ability to index and cache block information that the BPA may not do itself
- This also allows the BSL to efficiently manage security blocks in the BPA while using units of Security Operations above the Bundle Context
  - Sourcing multiple operations will create one or more aggregated security blocks
  - Accepting an operation will either update or discard a security block depending on other operations



# Notional Host Interface

- Assumptions:
  - `struct timespec` is available via POSIX API (but not part of C language until C11)
  - All host interactions have a “user data” opaque pointer associated with BSL initialization function
- Memory management functions:
  1. Allocate a heap memory pool
  2. Free a memory pool
- Telemetry functions (provided by BSL):
  1. Fetch statistics from a BSL context
  2. Aggregate statistics from multiple BSL contexts
- Event log functions (provided by host):
  1. Get current system timestamp
  2. Accept a log event from the BSL, including: timestamp, BSL type code, message text

# Notional Policy Provider Interface

- Assumptions:
  - The enumerated options available to each security context are known to a policy provider that uses it
  - The BSL may pre-cache some bundle information based on expected policy
- Registration functions (provided by BSL):
  1. Register a Policy Provider
  2. Iterate over registered Policy Providers
- Bundle data functions (provided by BSL): see *Bundle Context* slide
- Security functions (for each provider):
  1. Inspect bundle state based on a *bundle context* handle, resulting in a set of independent *Policy Action* and subordinate *Security Operation* objects
  2. Handle the result of all Security Operations on a bundle to perform policy-specific failure activities

# Notional Security Operation Options

- Non-option parameters are:
  - Security Role
  - Security Context ID (for Source role)
  - Security Block number (for non-Source roles)
  - Target Block number
- For Source role and BIB-HMAC-SHA2 context:
  - Option for what key to use (key ID)
  - Option for AAD scope parameter
- For Verifier role and BIB-HMAC-SHA2 context:
  - Option for BIB block number to verify
  - Option for what key to use (default contexts don’t transport key ID)
- Looking ahead to COSE context and more complex operations:
  - Option to identify specific source or recipient PKIX certificate

# Notional BPA Data Interface (part 1)

- Assumptions:
  - The native form of EID and traffic label is opaque to the BSL, but BPA provides access functions for them
  - All BSL inspection activity operates with a *Bundle Context* handle
  - Access to BTSD in a sequential form
- EID functions:
  1. Get local node administrative EID
  2. Decode EID (with length) from data buffer
  3. Encode EID to existing buffer
  4. Decode EID from text URI (for policy configuration)
  5. Encode EID to text URI (for logging and manual processing)
  6. Decode EID Pattern from text (for policy configuration)
  7. Match an EID against an EID Pattern
- Traffic Label functions:
  1. Convert text to Label (for policy configuration)
  2. Convert Label to text (for logging and manual processing)

# Notional BPA Data Interface (part 2)

- Bundle Context management:
  1. Initialize a Bundle Context (from internal BPA state)
  2. Release a Bundle Context
- Bundle block functions (for a bundle handle) accessed via Bundle Context:
  1. Determine if the bundle is marked with a specific Label
  2. Add a specific Label to a bundle
  3. Get primary block data (source EID, destination EID, bundle flags, creation timestamp, seq. num.)
  4. Get encoded-form of entire primary block
  5. Iterate through canonical blocks (used by Bundle Context indices)
  6. Get block metadata (type code, block flags, CRC Type) from block number
  7. Get BTSD from block number
  8. Add a new (security) block based on metadata using BPA-assigned block number
  9. Replace BTSD from block number
  10. Remove a (security) block from block number

# Notional Bundle Context Interface (part 1)

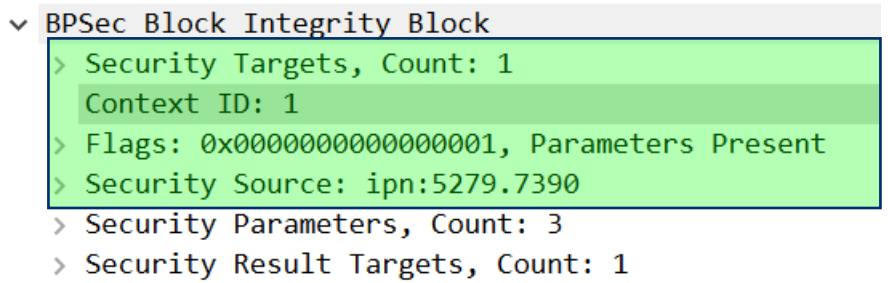
- Assumptions:
  - All data is sourced from the BPA (see BPA Data Interface slides)
  - The Bundle Context can index and cache block data for faster lookup
- Bundle block functions (for a bundle context handle):
  1. Access to all of the BPA Data Interface functions (see other slide)
  2. Get list of block numbers of a specific type
  3. Source a single security operation (will be aggregated into new blocks as needed)
  4. Accept a single security operation (when all targets in a block are accepted it is removed)

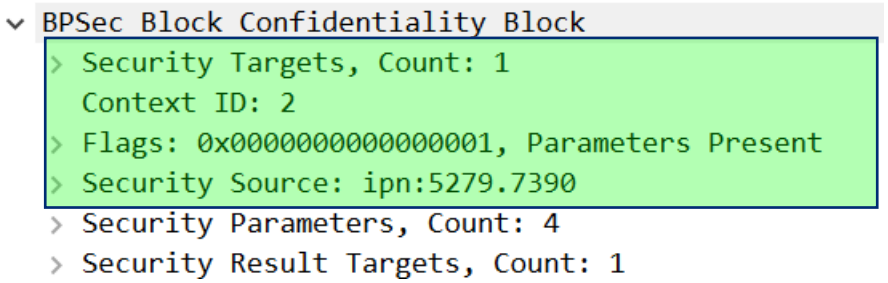
# Streaming access to BTSD

- BSL defines a struct for maintaining read or write state
  - Part of the state is an opaque pointer back to some BPA-specific context
- Block access function populate these structs to initialize a read or write
- Separate functions allow BSL to read or write a fragment
- Secondary capabilities:
  - Although the BSL interfaces are not interruptible/resumable the streaming access does allow a BPA to block a “large” BTSD access depending upon the fragment size used by the BSL
  - The fragment size chosen by the access is likely related to the crypto API, matching the block size for ciphertext codec or sequential HMAC/signature (on the order of 100-200 bytes)

# ASB Access by Policy Providers and Security Contexts

- A single BIB or BCB is composed of multiple fields with different purpose and scope
- The general structure is called an Abstract Security Block (ASB)
- Top fields identify the targets of the security block, Security Context associated with the operations, and Source of the block
- Later fields are Security-Context-specific to give “parameters” for the whole block and “results” associated with each target
- The BSL will iterate and decode the first three fields, the BSL will delegate decoding Security Source EID to the BPA, and the BSL will iterate over and index (but not store) parameters and results



# Notional Security Context Interfaces

- Assumptions:
  - This interface is tailored to the existing required security contexts
  - The BSL may pre-cache some bundle information based on expected policy
- Registration functions (provided by BSL):
  1. Register a Security Context Descriptor globally by a unique Context ID number
  2. Lookup a Security Context Descriptor from a Context ID
- Bundle data functions (provided by BSL): see Bundle Context slide
- Security Context functions (for each descriptor):
  1. Process a single security operation with its context-specific options

# Notional Crypto Function Interface

- Assumptions:
  - This interface is tailored to the existing required security contexts
- Cryptographic functions:
  1. Generate random byte string for BCB IV use
  2. Process a target byte string and AAD byte string for HMAC
  3. Encrypt a plaintext byte string and AAD byte string for AES-GCM, giving a resulting ciphertext
  4. Decrypt a ciphertext byte string and AAD byte string for AES-GCM, giving a resulting plaintext

# Example Security Block Processing Workflow

1. BPA initializes first library context
   A. Optionally, BPA copies library context for later work thread use
2. BPA takes first bundle context and initializes its indices based on actual block content
   A. Part of this initialization involves decoding and indexing the contents of each BIB and BCB within the bundle
3. BPA calls into BSL library context to inspect the bundle context
   A. BSL iterates over all Policy Providers to inspect the bundle context
      I. Each Policy provider accesses block data via the bundle context
      II. Logic within each Policy Provider decides what, if any, actions are needed
      III. If needed, an action is allocated and added to the resulting action set
   B. BSL validates the resulting action set and the security operations of each action
   C. BSL returns the validated action set to the BPA
4. The BPA optionally subdivides the action set and allocates to work threads for processing
   A. Each action is self-contained plain-old data (POD) and copyable between threads
5. BPA calls into BSL library context to execute one action
   A. For each security operation in the action the BSL calls into the associated Security Context to execute the operation
      I. The Security Context decodes and normalizes all options for the security operation
      II. For verifier or acceptor roles, the Security Context uses the bundle context to extract existing parameters and results
      III. The Security Context calls into crypto API, and a specific linked library, for crypto functions
      IV. For source role, the Security Context uses the bundle context to construct new parameters and results
      V. The result of the security operation is returned to the BSL
6. BPA calls into the BSL library context to finalize the results of executed security operations
   A. BSL iterates over operations and calls into the originating Policy Provider to perform any finalizing activities
      I. Failure to verify or accept BIBs or BCBs can result in dropped blocks
      II. Also possible to indicate re-processing of security blocks after inspecting ADU contents
7. BPA de-initializes the bundle context
8. BPA de-initializes all library contexts

# Example API Fragment #1: Library Context

Simple example of forward-declared opaque structure and memory pool functions

```c
/// Forward declaration for this type
typedef struct bsl_ctx bsl_ctx_t;

/** Initialize resources for a library context.
 *
 * @param[in,out] lib The library context.
 * @return Zero if successful.
 */
int bsl_ctx_init(bsl_ctx_t *lib);

/** Initialize a library context as a copy from an existing context.
 *
 * @param[in,out] lib The library context.
 * @param src The existing context to copy from.
 * @return Zero if successful.
 */
int bsl_ctx_init2(bsl_ctx_t *lib, const bsl_ctx_t *src);

/** Release resources from a library context.
 *
 * @param lib The library context.
 * @return Zero if successful.
 */
int bsl_ctx_deinit(bsl_ctx_t *lib);
```

```c
/** Allocate a list of bsl_pp_action_t from the memory pool.
 * Each of the structs is initialized to zeros.
 *
 * @param lib The library context.
 * @param[in,out] list The list to append to.
 * @param count Total number of options desired.
 * @return Zero if successful.
 */
int bsl_pp_action_alloc(bsl_ctx_t *lib, bsl_pp_action_set_t list, size_t count);

/** Free a list of bsl_pp_action_t back to the memory pool.
 *
 * @param lib The library context.
 * @param list The list containing options to free.
 */
void bsl_pp_action_free(bsl_ctx_t *lib, bsl_pp_action_set_t list);
```

# Example API Fragment #2: Bundle Context

More complex, data-focused frontend interfaces

```c
/** Forward declaration for the Bundle Context.
 * This struct is used as the context for all single-bundle data accesses.
 */
typedef struct bsl_bundle_ctx bsl_bundle_ctx_t;

/** Initialize resources for a bundle context.
 * @param[in,out] bundle The context struct.
 * @param parent The library context to work under.
 * @return Zero if successful.
 */
int bsl_bundle_ctx_init(bsl_bundle_ctx_t *bundle, bsl_ctx_t *parent);

/** Release resources from a bundle context.
 * @param[in,out] bundle The context struct.
 * @return Zero if successful.
 */
int bsl_bundle_ctx_deinit(bsl_bundle_ctx_t *bundle);
```

```c
/** Get block metadata from a bundle.
 *
 * @param bundle The bundle to query.
 * @param blk_num The block number to look up.
 * @param[out] blk_type Pointer to the type code of the block, if found.
 * @param[out] flags Pointer to the flags of the block, if found.
 * @param[out] crc_type Pointer to the CRC type of the block, if found.
 * @param[out] btsd_size Pointer to the size of the BTSD for the block, if fo
 * @return Zero if successful.
 */
int bsl_bundle_ctx_get_meta(const bsl_bundle_ctx_t *bundle, uint64_t blk_num,
                            uint64_t *crc_type, uint64_t *btsd_size);

/** Initialize a new BTSD reader.
 *
 * @param bundle The bundle to query.
 * @param blk_num The block number to look up.
 * @param[out] reader The initialized reader object, which must have
 * bsl_seq_read_deinit() called on it after using it.
 * @return Zero if successful.
 */
int bsl_bundle_ctx_btsd_read(bsl_bundle_ctx_t *bundle, uint64_t blk_num, bsl_
```

JOHNS HOPKINS
APPLIED PHYSICS LABORATORY

# Documentation

# Project Documentation Areas

- The BSL documentation will take a few different forms for different purposes
- BSL source repository Wiki
  - AMMOS Task Description
  - Dynamic Release Plan
  - Ticket Workflow
- BSL source repository content
  - Contents of README.md, CONTRIBUTING.md, SECURITY.md, LICENSE.txt
  - Contents of actual C header files with Doxygen markup
  - Contents of Doxygen-read markdown files
  - Generated documentation in *Pages*
- BSL source repository metadata
  - Pull Requests and per-branch job status
  - Static analysis results
- BSL-docs repository
  - Copies of approved AMMOS design documentation
  - Source for User Guide and Product Guide
  - Generated User Guide and Product Guide documents in *Pages*
- BSL project
  - Views into tickets and pull requests across all repositories

# Example API Documentation

Documentation and visualization extracted from source files

## Typedefs

typedef struct **bsl_ctx** **bsl_ctx_t**
Forward declaration for this type.

## Functions

int **bsl_ctx_init** (**bsl_ctx_t** *lib)
Initialize resources for a library context. More...

int **bsl_ctx_init2** (**bsl_ctx_t** *lib, const **bsl_ctx_t** *src)
Initialize a library context as a copy from an existing context. More...

int **bsl_ctx_deinit** (**bsl_ctx_t** *lib)
Release resources from a library context. More...

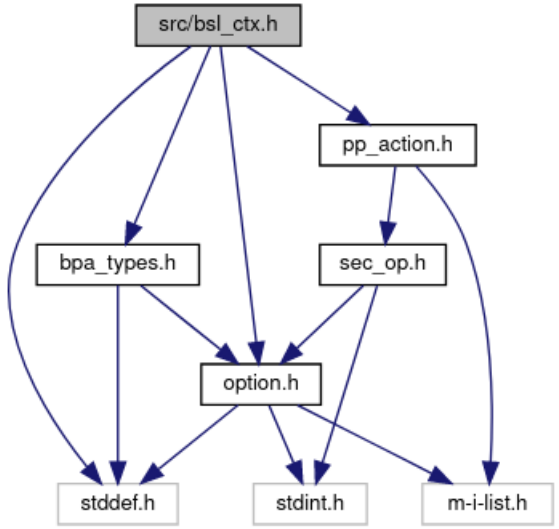
**bsl_eid_h** **bsl_get_secsrc** (**bsl_ctx_t** *lib)
Get the local EID used when this node is a security source. More...

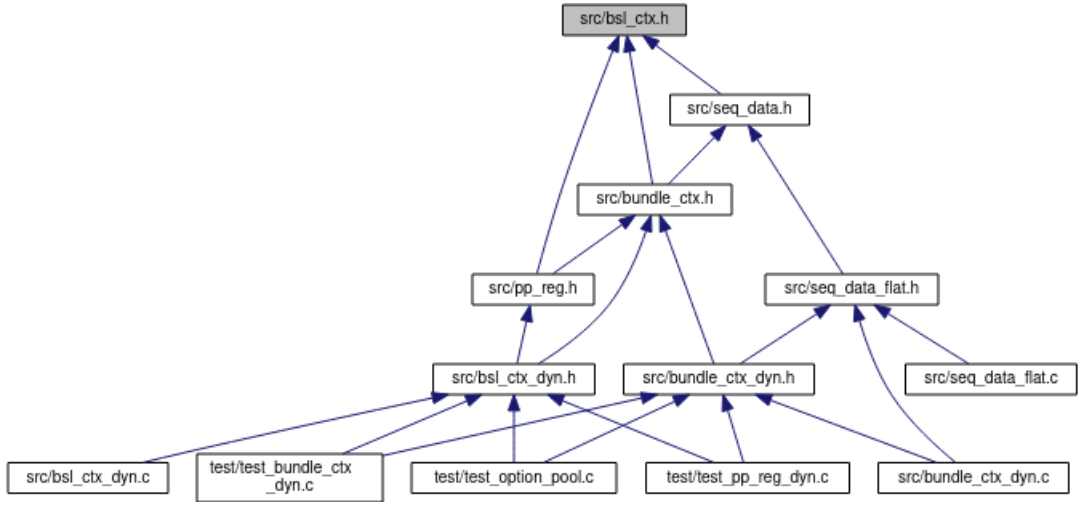
int **bsl_option_alloc** (**bsl_ctx_t** *lib, bsl_option_list_t list, size_t count)
Allocate a list of bsl_option_t from the memory pool. More...

void **bsl_option_free** (**bsl_ctx_t** *lib, bsl_option_list_t list)
Free a list of bsl_option_t back to the memory pool. More...

int **bsl_pp_action_alloc** (**bsl_ctx_t** *lib, bsl_pp_action_set_t list, size_t count)
Allocate a list of bsl_pp_action_t from the memory pool. More...

void **bsl_pp_action_free** (**bsl_ctx_t** *lib, bsl_pp_action_set_t list)
Free a list of bsl_pp_action_t back to the memory pool. More...



JOHNS HOPKINS
APPLIED PHYSICS LABORATORY

# Testing

# Unit and Integration Testing

- Unit Tests:
  - Library sources and unit tests are in parallel trees `src` and `test` respectively
  - Mock interfaces for testing provided for BPA and Host interfaces
  - Unit test assertions and logic provided by Unity C library
  - Test sequencing provided by CTest tool
  - Test coverage accounting and reporting provided by `gcovr` library and utilities
- Integration testing is deferred to integrations with specific BPAs and target operating systems and architectures
  - There are external projects which intend to integrate and test with BPAs/OSes and provide feedback

# Continuous Testing

- Within the software repository (initially private) there will be CI jobs for:
  - Static analysis of source
  - Building for Ubuntu/x86, collecting warnings
  - Unit testing on Ubuntu, collecting failures
  - Building Doxygen documentation, collecting warnings
- Within the docs repository there will be CI jobs to build HTML and PDF documents for:
  - Product Guide
  - User Guide
- These jobs will be used as automated feedback to inform Pull Requests and other source reviews


JOHNS HOPKINS

APPLIED PHYSICS LABORATORY