Semantics-Driven Testing of the PKCS11 API

Matthew Bauer, Mike Dodds

Joint work with Charisee Chiw, Joey Dodds and Stephen Magill

galois

Galois Consultancy

DARPA

Homeland Security

Amazon

NATIONAL SECURITY AGENCY UNITED STATES OF AMERICA

U.S. AIR FORCE

NAVY

NIST

U.S. DEPARTMENT OF ENERGY

LG

U.S.ARMY

NASA

MARINES

Founded in 1999 • Based in Portland, OR • 100 Employees

Galois / AWS Collaboration

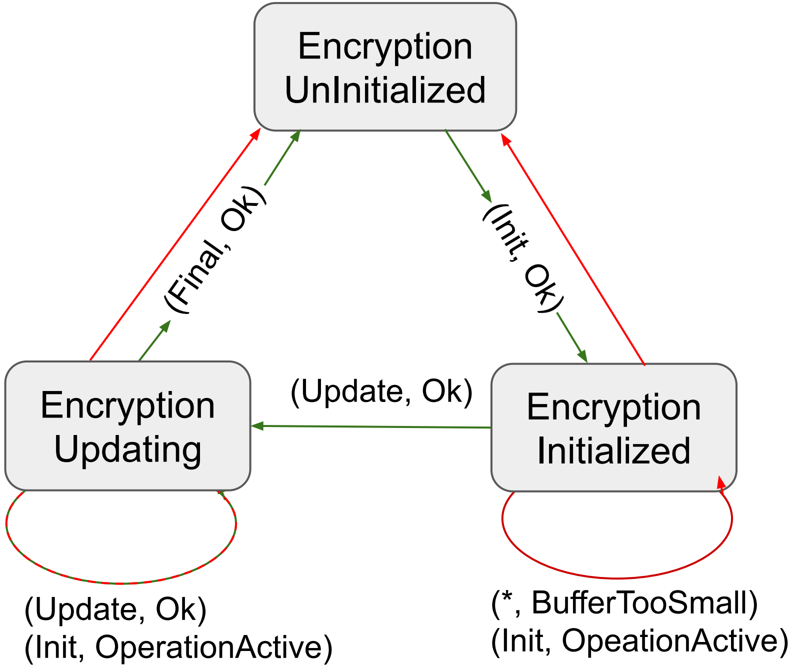
- Collaboration: Key Security related projects in partnership with AWS
- Approach: continuous formal methods
    - Tight integration with engineering processes
    - Integration into CI / CD pipeline
    - High levels of automation
    - Low cost of maintenance
    - Actionable bug reports
- Ongoing formal methods success story: we're helping bringing high assurance software to AWS end users *(ie. everyone)*
- NB: lots of other AWS formal methods work - see other talks at HCSS!

Galois / AWS Projects

- Cryptographic Algorithm verification
    - SIKE / BIKE - Post-quantum algorithms
    - HMAC/DRBG - TLS core algorithms
- Cryptographic Protocol verification
    - s2n - Amazon TLS handshake protocol
- Cryptographic APIs testing
    - **PKCS11 - Public-Key Cryptography Standards**

This Project: Assuring API Implementations

- Each API defines expected sets of behaviours - an API is pretty much a specification.
- Library implementations should match the spec i.e.
  - Not crash
  - Return expected values
- Often not the case!



Target: The PKCS11 API

- A platform independent API standard for interacting with cryptographic tokens such as hardware security modules (HSMs) and smart cards

- Functionality:
    - Store cryptographic tokens on devices
    - Generate cryptographic keys and random numbers
    - Encrypt, decrypt, hash, sign and verify data
    - Wrap and unwrap keys

The PKCS11 API - Keys

- Keys hold cryptographic data and properties, which include:
  - Key type (e.g. AES)
  - Key class (e.g. private key, public key, secret key)
  - Storage characteristics (e.g. does it persist on the device)
  - Supported operation types (e.g. encryption, decryption, etc...)
  - User defined labels

The PKCS11 API - Mechanisms

- Each cryptographic operation is parameterized by a mechanism that describes the underlying algorithms used in the cryptographic operation

- Example: AES-CBC for encryption and decryption describes:
  - The algorithm (AES) and mode (CBC)
  - Parameters to the algorithm (such as an initialization vector)

The PKCS11 API is Complicated

- ~350 pages of specification (~150 base spec, ~200 key/mechanism spec)
- ~50 function specifications
- ~45 cryptographic algorithms
- ~90 error codes

→ size makes it challenging to formally verify code.

Instead: Model-based Test Synthesis

- Write a strict formal model of the API - values and transitions
- Synthesize a test set by exploring the model
- Use formal techniques to ensure a high level of coverage
- Test the implementation library, add tests to CI

→ Achieve a high level of API confidence.

PKCS11 Testing in Detail

PKCS11 API Function Descriptions

¨ **C_EncryptInit**

```
CK_DEFINE_FUNCTION(CK_RV, C_EncryptInit)(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hKey
);
```

C_EncryptInit initializes an encryption operation. *hSession* is the session's handle; *pMechanism* points to the encryption mechanism; *hKey* is the handle of the encryption key.

The **CKA_ENCRYPT** attribute of the encryption key, which indicates whether the key supports encryption, MUST be CK_TRUE.

After calling **C_EncryptInit**, the application can either call **C_Encrypt** to encrypt data in a single part; or call **C_EncryptUpdate** zero or more times, followed by **C_EncryptFinal**, to encrypt data in multiple parts. The encryption operation is active until the application uses a call to **C_Encrypt** or **C_EncryptFinal** *to actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the application MUST call **C_EncryptInit** again.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_EncryptFinal**.

```
CK_DEFINE_FUNCTION(CK_RV, C_EncryptInit)(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hKey
);
```

Function name and return type

Argument types and order

PKCS11 API Function Descriptions

¨ **C_EncryptInit**

```
CK_DEFINE_FUNCTION(CK_RV, C_EncryptInit)(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hKey
);
```

C_EncryptInit initializes an encryption operation. *hSession* is the session's handle; *pMechanism* points to the encryption mechanism; *hKey* is the handle of the encryption key.

The **CKA_ENCRYPT** attribute of the encryption key, which indicates whether the key supports encryption, MUST be CK_TRUE.

After calling **C_EncryptInit**, the application can either call **C_Encrypt** to encrypt data in a single part; or call **C_EncryptUpdate** zero or more times, followed by **C_EncryptFinal**, to encrypt data in multiple parts. The encryption operation is active until the application uses a call to **C_Encrypt** or **C_EncryptFinal** *to actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the application MUST call **C_EncryptInit** again.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_EncryptFinal**.

Informal Description: C_EncryptInit initializes an encryption operation. *hSession* is the session's handle; *pMechanism* points to the encryption mechanism; *hKey* is the handle of the encryption key

PKCS11 API Function Descriptions

¨ **C_EncryptInit**

```
CK_DEFINE_FUNCTION(CK_RV, C_EncryptInit)(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hKey
);
```

C_EncryptInit initializes an encryption operation. *hSession* is the session's handle; *pMechanism* points to the encryption mechanism; *hKey* is the handle of the encryption key.

The **CKA_ENCRYPT** attribute of the encryption key, which indicates whether the key supports encryption, MUST be CK_TRUE.

After calling **C_EncryptInit**, the application can either call **C_Encrypt** to encrypt data in a single part; or call **C_EncryptUpdate** zero or more times, followed by **C_EncryptFinal**, to encrypt data in multiple parts. The encryption operation is active until the application uses a call to **C_Encrypt** or **C_EncryptFinal** *to actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the application MUST call **C_EncryptInit** again.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_EncryptFinal**.

Stateful Behavior: After calling C_EncryptInit, the application can either call C_Encrypt to encrypt data in a single part; or call C_EncryptUpdate zero or more times, followed by C_EncryptFinal, to encrypt data in multiple parts ....

PKCS11 API Function Descriptions

¨ **C_EncryptInit**

```
CK_DEFINE_FUNCTION(CK_RV, C_EncryptInit)(
  CK_SESSION_HANDLE hSession,
  CK_MECHANISM_PTR pMechanism,
  CK_OBJECT_HANDLE hKey
);
```

C_EncryptInit initializes an encryption operation. *hSession* is the session's handle; *pMechanism* points to the encryption mechanism; *hKey* is the handle of the encryption key.

The **CKA_ENCRYPT** attribute of the encryption key, which indicates whether the key supports encryption, MUST be CK_TRUE.

After calling **C_EncryptInit**, the application can either call **C_Encrypt** to encrypt data in a single part; or call **C_EncryptUpdate** zero or more times, followed by **C_EncryptFinal**, to encrypt data in multiple parts. The encryption operation is active until the application uses a call to **C_Encrypt** or **C_EncryptFinal** *to actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the application MUST call **C_EncryptInit** again.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example: see **C_EncryptFinal**.

Error Handling: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED … (20 Returns in total)

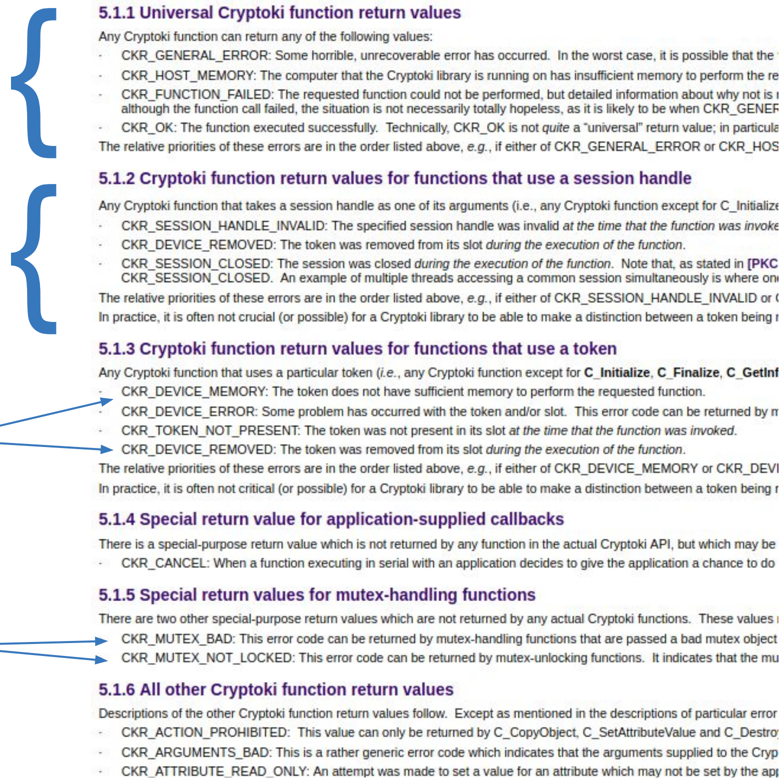
PKCS11 API Return Code Descriptions

- Returns codes are organized by section, where a section's order in the document defines a precedence on the return codes it contains
- Each section also defines a order on the return codes within in.
    - In some sections, this is a total order according to order of appearance
    - In some sections, this is a partial order where all returns are unordered unless explicitly stated

<

<

??



5.1.1 Universal Cryptoki function return values

Any Cryptoki function can return any of the following values:

- CKR_GENERAL_ERROR: Some horrible, unrecoverable error has occurred. In the worst case, it is possible that the
- CKR_HOST_MEMORY: The computer that the Cryptoki library is running on has insufficient memory to perform the re
- CKR_FUNCTION_FAILED: The requested function could not be performed, but detailed information about why not is although the function call failed, the situation is not necessarily totally hopeless, as it is likely to be when CKR_GENEF
- CKR_OK: The function executed successfully. Technically, CKR_OK is not *quite* a "universal" return value; in particula

The relative priorities of these errors are in the order listed above, *e.g.*, if either of CKR_GENERAL_ERROR or CKR_HOS

5.1.2 Cryptoki function return values for functions that use a session handle

Any Cryptoki function that takes a session handle as one of its arguments (i.e., any Cryptoki function except for C_Initialize

- CKR_SESSION_HANDLE_INVALID: The specified session handle was invalid *at the time that the function was invoke*
- CKR_DEVICE_REMOVED: The token was removed from its slot *during the execution of the function.*
- CKR_SESSION_CLOSED: The session was closed *during the execution of the function.* Note that, as stated in [PKC CKR_SESSION_CLOSED. An example of multiple threads accessing a common session simultaneously is where one

The relative priorities of these errors are in the order listed above, *e.g.*, if either of CKR_SESSION_HANDLE_INVALID or

In practice, it is often not crucial (or possible) for a Cryptoki library to be able to make a distinction between a token being

5.1.3 Cryptoki function return values for functions that use a token

Any Cryptoki function that uses a particular token (*i.e.*, any Cryptoki function except for **C_Initialize**, **C_Finalize**, **C_GetInf**

- CKR_DEVICE_MEMORY: The token does not have sufficient memory to perform the requested function.
- CKR_DEVICE_ERROR: Some problem has occurred with the token and/or slot. This error code can be returned by n
- CKR_TOKEN_NOT_PRESENT: The token was not present in its slot *at the time that the function was invoked.*
- CKR_DEVICE_REMOVED: The token was removed from its slot *during the execution of the function.*

The relative priorities of these errors are in the order listed above, *e.g.*, if either of CKR_DEVICE_MEMORY or CKR_DEVI

In practice, it is often not critical (or possible) for a Cryptoki library to be able to make a distinction between a token being

5.1.4 Special return value for application-supplied callbacks

There is a special-purpose return value which is not returned by any function in the actual Cryptoki API, but which may be

- CKR_CANCEL: When a function executing in serial with an application decides to give the application a chance to do

5.1.5 Special return values for mutex-handling functions

There are two other special-purpose return values which are not returned by any actual Cryptoki functions. These values

- CKR_MUTEX_BAD: This error code can be returned by mutex-handling functions that are passed a bad mutex object
- CKR_MUTEX_NOT_LOCKED: This error code can be returned by mutex-unlocking functions. It indicates that the mu

5.1.6 All other Cryptoki function return values

Descriptions of the other Cryptoki function return values follow. Except as mentioned in the descriptions of particular error

- CKR_ACTION_PROHIBITED: This value can only be returned by C_CopyObject, C_SetAttributeValue and C_Destro
- CKR_ARGUMENTS_BAD: This is a rather generic error code which indicates that the arguments supplied to the Cryp
- CKR_ATTRIBUTE_READ_ONLY: An attempt was made to set a value for an attribute which may not be set by the app

PKCS11 API Pitfalls

- Behavior is underspecified
  - Possible return codes may not be listed
  - Extra return codes may be listed
  - Not all return codes are described
- Return code precedences are not concisely and uniformly described
- The description of stateful behavior is imprecise and scattered across the document

→ Need a formal description of the API!

Formally Modeling the API

- Cryptographic algorithms
- Error conditions associated with each function
- Stateful behavior that defines how functions interact

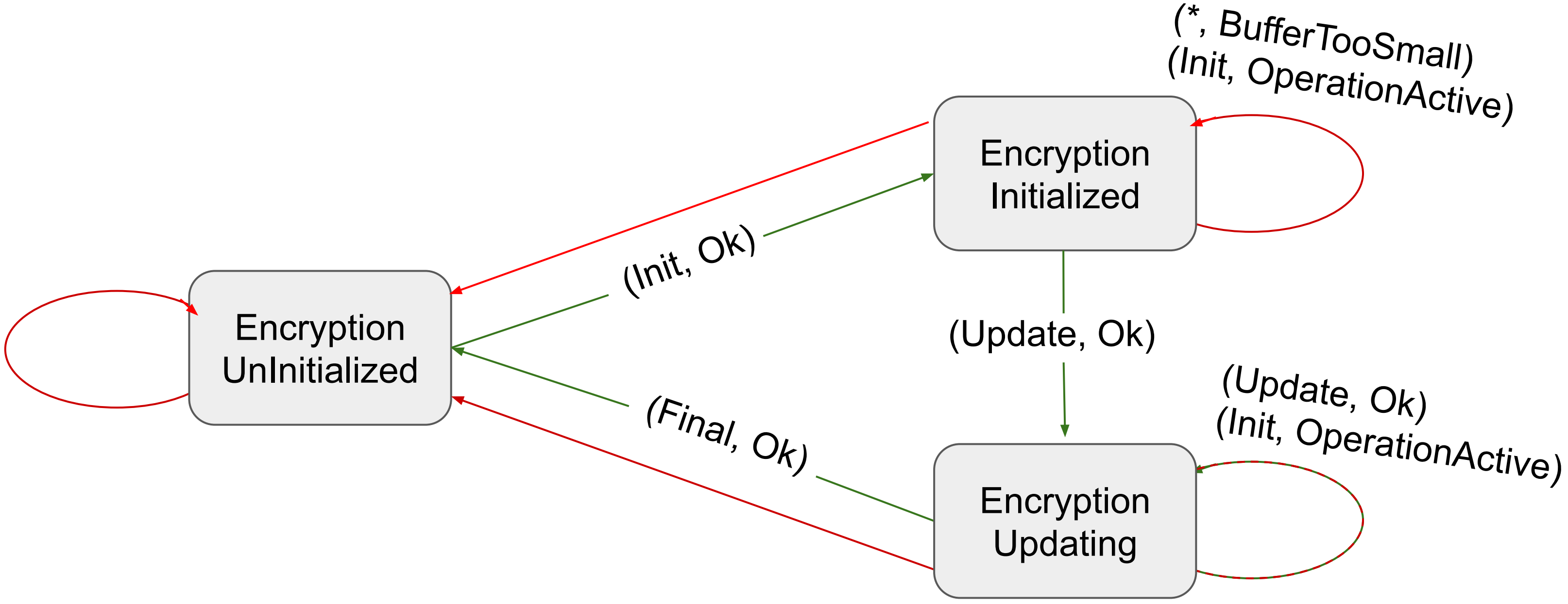
Formal Cryptographic Specifications

- Specified using Cryptol, a domain specific language for cryptography
- Cryptol specifications are executable programs that closely resemble their mathematical definitions
- We have specified the following algorithms
    - AES
    - Triple DES
    - ECDSA
    - RSA
    - SHA

Formal Return Code Specifications

- What error conditions are possible and how are they triggered?
    - We describe errors as constraints over function arguments and the (model of) the system state
    - Return code precedence complicates constraints, all conditions of higher priority errors must not be true
    - SMT solvers are used to synthesize the necessary system state and function inputs that generate a particular error

Formal State Models



Transitions = (Function Name, Return Code)

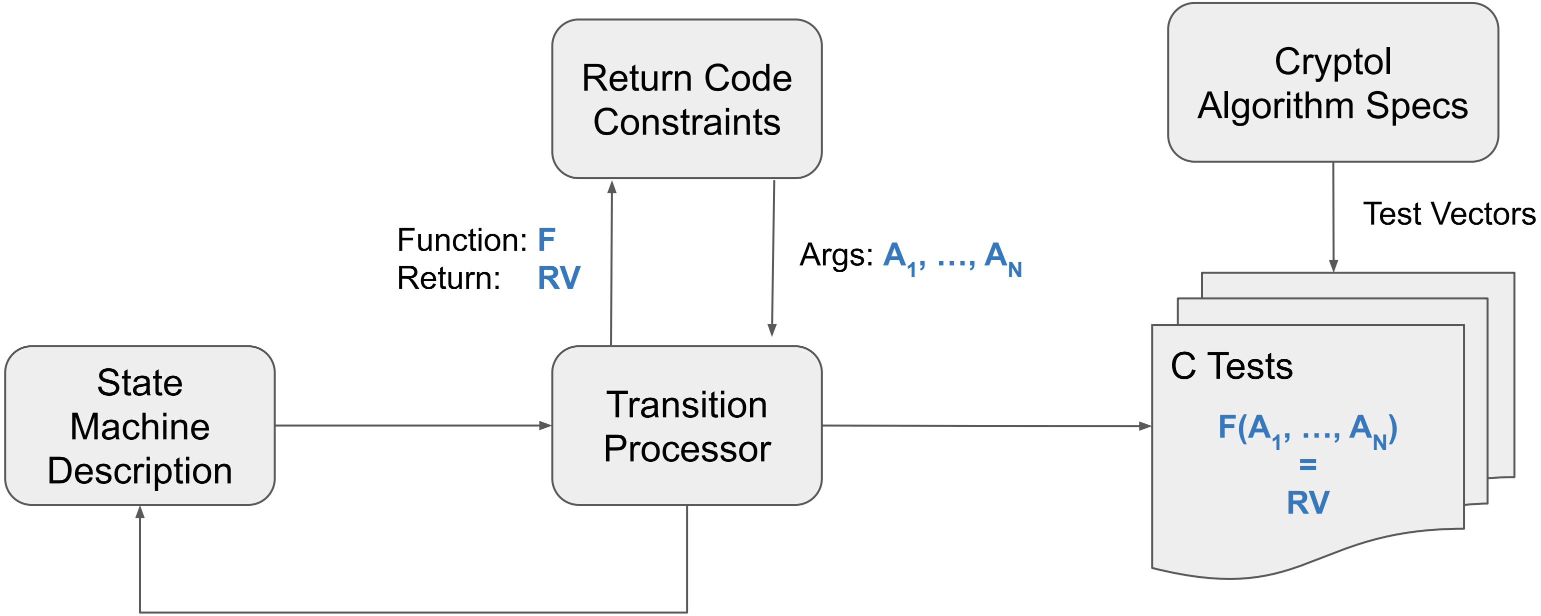
State Model Encoding in Haskell

```
incrementalStateTransition UnInit      C_EncryptInit   CKR_OK                 = Initialized
incrementalStateTransition UnInit      _               _                      = UnInit

incrementalStateTransition Initialized C_EncryptUpdate CKR_BUFFER_TOO_SMALL   = Initialized
incrementalStateTransition Initialized C_EncryptInit   CKR_OPERATION_ACTIVE   = Initialized
incrementalStateTransition Initialized C_EncryptUpdate CKR_OK                 = Updating
incrementalStateTransition Initialized _               _                      = UnInit

incrementalStateTransition Updating    C_EncryptUpdate CKR_OK                 = Updating
incrementalStateTransition Updating    C_EncryptInit   CKR_OPERATION_ACTIVE   = Updating
incrementalStateTransition Updating    C_EncryptFinal  CKR_OK                 = UnInit
incrementalStateTransition Updating    _               _                      = UnInit
```

System Architecture



Test Generation

- Generated tests that exercised every software triggered return code for every stateful operation
- Explored every path through the finite state machines as well as all meaningful compositions of different paths
- Over 1,500 test cases in total

Encryption - 442 tests

Decryption - 438 tests

Digest - 182 tests

Sign - 128 tests

Verify - 150 tests

Sign Recover - 30 tests

Verify Recover - 30 tests

Session Management - 42 tests

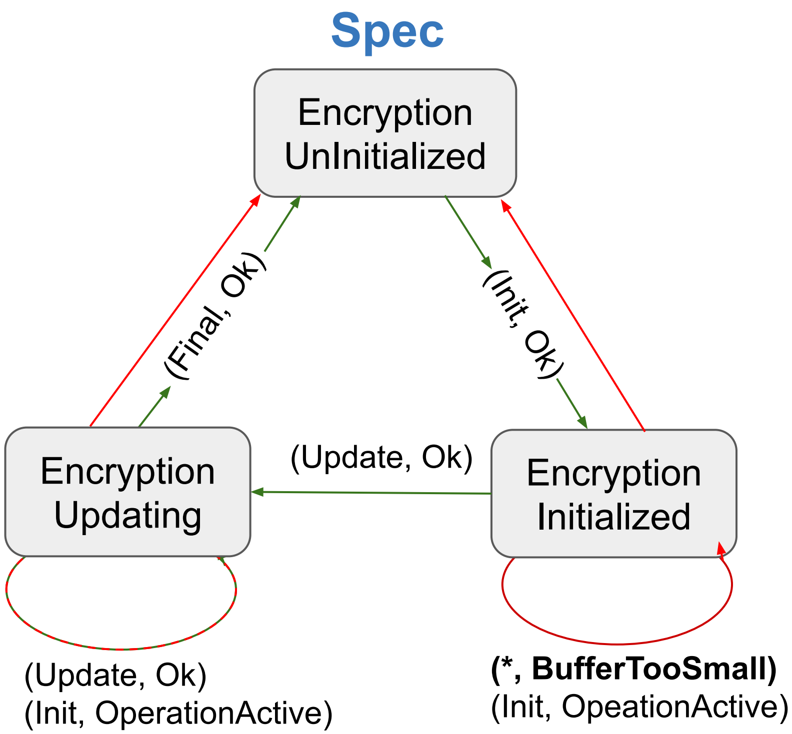
Test Results

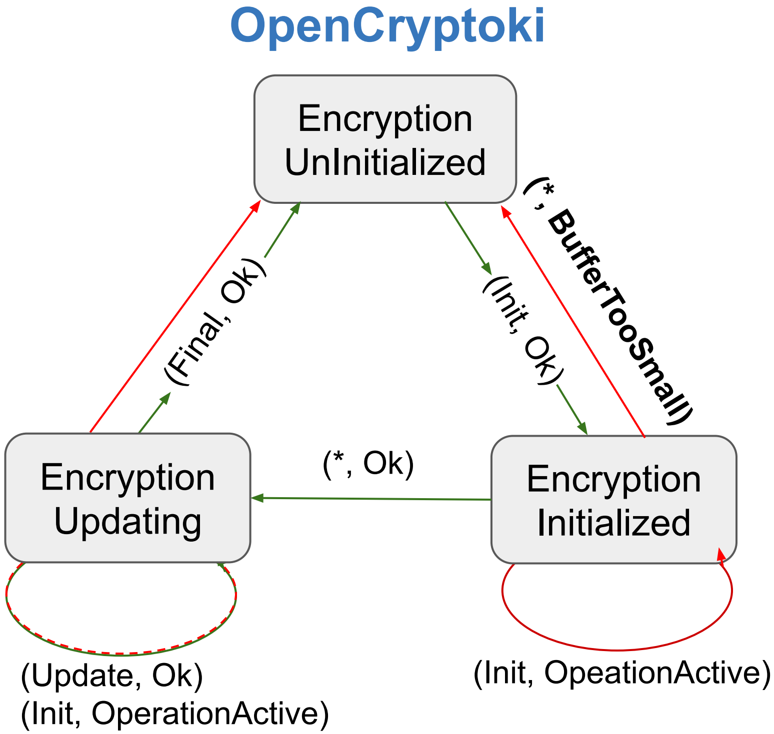
- Tested against **OpenCryptoki** and **pre-release Amazon CloudHSM**
- Bugs fixed before production:
  - Library segmentation faults
  - Invalid state machines
  - Improper return codes
  - Missing null pointer handling
  - Lossy object copies

CloudHSM

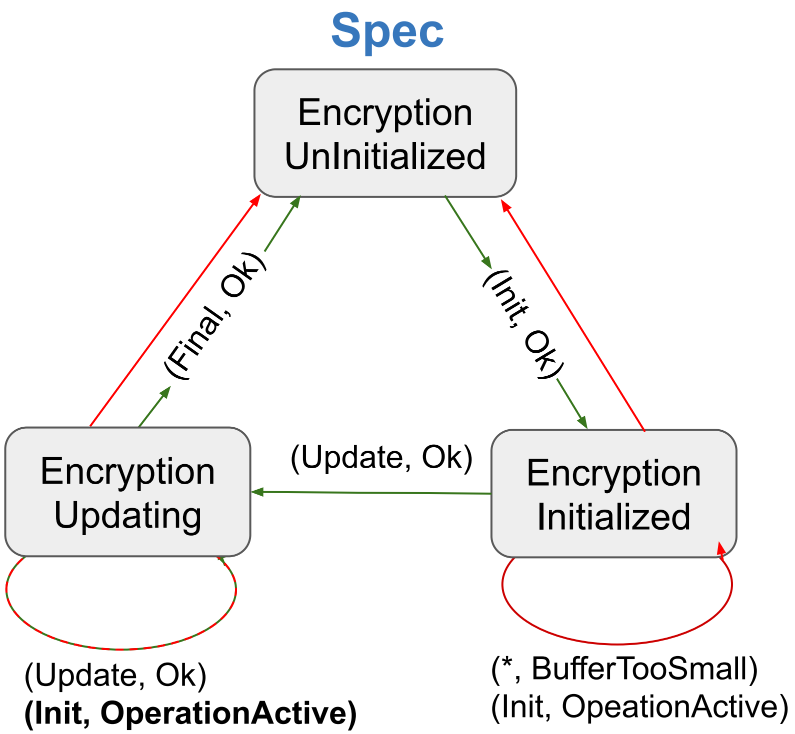
7	Library Segfaults
2	State machine errors
20+	Improper return codes

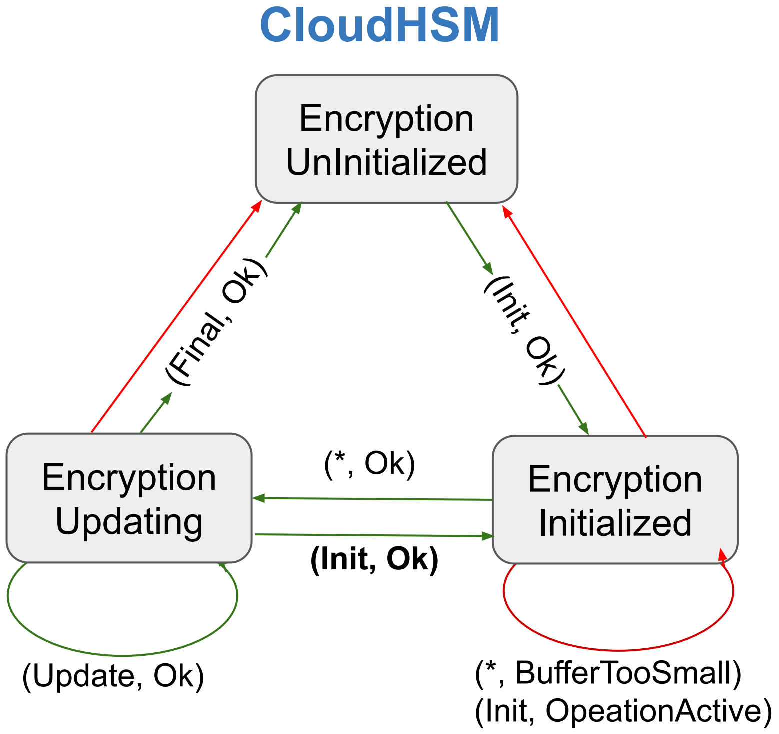
Example State Machine Error





Example State Machine Error





Keys to Deployment Success

- Categorization of failures by type and severity
- Test case reproducibility
- Speed
- Metrics
- Configurability

Amazon Deployment

- The test suite is deployed in Amazon's CI/CD pipeline.
  - Test suite and specification compliance is continuously maintained
  - Searchable metrics are published with each run
  - Detailed logs capturing all object instantiations, function calls and return values are captured
  - Over 10,000 tests execute in less than 30 minutes

Thank you!

|galois|