Executive Summary

PKCS#11, also known as Cryptoki, is a widely adopted C-language API and interoperability standard for communicating with cryptographic libraries. While comprehensive and prescriptive, the standard is also extremely complex. The base specification alone describes approximately 50 functions through over 100 pages of documentation. Add-on specifications provide additional functionality, but also impose additional complexity. As the standard evolves through collaboration and expansion, new areas of imprecision and ambiguity are introduced which make it difficult for vendors to implement libraries that are 100% functionally accurate and compliant with the specification. Users of cryptographic libraries are familiar with the industry reality that PKCS#11 libraries will not always interoperate flawlessly with each other. In the best case, these divergences result in development delays as build issues and functional deficiencies are root-caused and remedied. In the worst case, customers may face production outages or data corruption due to incorrect assumptions or imperfect testing.

To address this problem at its core, Galois has developed an API specification and testing framework that captures the complex behaviors of Cryptoki in a mathematical specification language. This specification, in turn, is used to automatically synthesize a test suite that enforces compliance with (a mathematical model of) the PKCS#11 standard. The approach, known as model-based testing, was used to generate a corpus of PKCS#11 compliance tests that provide complete test coverage over the formal model.

While model-generated tests are designed to enforce compliance with the PKCS#11 standard, they have also proven effective at keeping bugs out of production. Error handling scenarios defined in the standard often represent corner cases which, if not properly handled by implementation code, will cause program crashes or other serious defects. Because the aim of model-based testing is to generate test cases for all the behaviors in the standard, errors of this variety are naturally uncovered with compliance tests. As a result, assuring PKCS#11 libraries with model-generated compliance tests leads to client applications that are portable, more robust and have fewer security vulnerabilities.

The Complexities of Cryptoki

We begin by outlining what the Cryptoki API is, and why it is difficult to model and test. This lays the foundation through which we highlight the kinds of bugs we hope to reduce and/or eliminate with model-based testing. It will also provide a backdrop for describing the details of our approach.

Each cryptographic operation supported by the Cryptoki API is defined in terms of a set of functions necessary for performing the operation from beginning to end. For example, Cryptoki provides the following four functions which together enable encrypting data: **C_EncryptInit**, **C_Encrypt**, **C_EncryptUpdate** and **C_EncryptFinal**. For each function, the standard gives a description of how it interacts statefully with other functions in the operation group. For example, the stateful behavior for C_EncryptInit, as written in version 2.40 of the standard, is given below.

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.

While the basic usage pattern appears intuitive on the surface, failures in function invocations often result in surprising state transitions. For example, in successive calls to C_EncryptInit, if the first call succeeds, then the second call should return the CKR_OPERATION_ACTIVE error and leave the operation state unchanged (as opposed to resetting the operation with new parameters). This behavior is not explicitly specified, but rather, must be inferred by examining the return values listed for a function. The standard provides a generic description of each return value, but typically does not describe them in the context of particular functions. The return values for C_EncryptInit, as listed in version 2.40 of the standard, are below.

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.

Return codes are intended to provide a behavioral contract for each function, with each value having an associated set of constraints over the operation state and function arguments. The conditions of this contract are often not made explicit when API functions are described in the standard. Furthermore, the list of behaviors is not guaranteed to be exhaustive, as highlighted by the following excerpt from the base specification:

Because of the complexity of the Cryptoki specification, it is recommended that Cryptoki applications attempt to give some leeway when interpreting Cryptoki functions' return values. We have attempted to specify the behavior of Cryptoki functions as completely as was feasible; nevertheless, there are presumably some gaps.

Indeed, one of the parameters to C_EncryptInit is a pointer to an encryption mechanism data type. If this pointer is NULL, a correct behavior for C_EncryptInit is to return the CKR_ARGUMENTS_BAD error. Observe, however, that this return code is not enumerated in the list above. Adding an additional layer of complexity, the standard also defines a partial order over the set of all return codes. This order dictates what return code applies when the conditions for multiple errors are met in a single function invocation. For example, the standard requires that a call to C_EncryptInit with both an invalid session handle and an invalid mechanism should return the error CKR_SESSION_HANDLE_INVALID instead of CKR_MECHANISM_INVALID.

Non-Compliance Errors in Cryptoki

There are many ways in which Cryptoki library implementations can deviate from the standard. This section gives a non-exhaustive list of these error categories, focusing on the types of errors that can be reduced and/or eliminated through model-based testing.

State-machine errors: Recall from the previous section that the behavior of a function depends on the state of the system, which is (implicitly) defined by the standard. A state-machine error occurs when a function call produces the wrong behavior because the API implementation's definition of state does not match the one specified in the standard. For example, a Cryptoki implementation that returns the CKR_OK error code for two consecutive (valid) calls to C_EncryptInit has a state machine error. The correct behavior is that the latter call to C_EncryptInit returns CKR_OPERATION_ACTIVE, as the first call updates the operation's state. Providing standards-compliant state-machine implementations simplifies code development, testing and root-causing of run-time issues.

Memory safety errors: Although this class of errors is not specific to Cryptoki, there are many scenarios where invalid function arguments have defined behaviors (return codes) in the standard which, if not properly handled, will result in memory errors. For example, supplying a NULL pointer to the pMechanism argument of C_EncryptInit should result in the return code CKR_ARGUMENTS_BAD. However, if the library implementation does not explicitly check if the pointer is NULL, the application may crash due to a memory error. Identifying memory safety violations protects applications from potential security-critical memory corruption or data leakage problems.

Invalid return code errors: This class of error occurs when a function call produces a return code that is not applicable in the given context. For example, a valid call to C_EncryptInit should not return CKR_DATA_INVALID, as it does not process any data. Providing standards-compliant error codes simplifies code development and enables predictable, consistent interoperability with third party software and hardware components.

Return code precedence errors: It is often the case that multiple return codes are possible for a given API call. A precedence error occurs when an implementation returns a value having lower priority than one of the other applicable return codes. For example, recall the scenario from the previous section where C_EncryptInit is invoked with an invalid session handle and an invalid mechanism. Because CKR_SESSION_HANDLE has a higher precedence than CKR_MECHANISM_INVALID, an implementation should be producing the latter return code. Providing standards-compliant error codes simplifies code development and enables predictable, consistent interoperability with third party software and hardware components.

Formalizing the Cryptoki API

To enable model-based testing of an API, we encode the behavior of the API into a formal model. The model expresses, in an unambiguous mathematical manner, how the API should behave in every situation. We can then compare the modeled behavior with the real behavior of the implementation in the concrete test cases we synthesize. To detect the non-compliance

errors detailed in the previous sections, our model must formalize both the stateful and functional behaviors of the API.

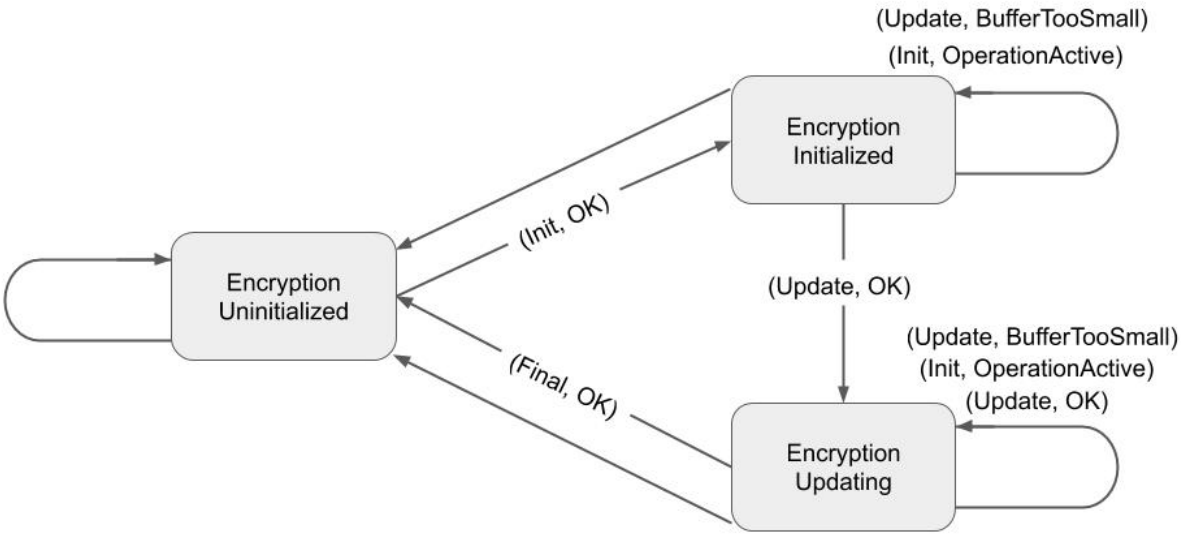
Stateful behavior: It is important to observe that sequences of function calls alone are not sufficient to describe the stateful behavior of an operation. Rather, state transitions are defined by both the function that is called and the return code it produces. For example, consider the following sequence of function and return code pairs

(C_EncryptInit, CKR_OK), (C_Encrypt, CKR_OK), (C_EncryptInit, CKR_OK)

which describe an interaction with a Cryptoki library in which sequential calls to C_EncryptInit, C_Encrypt, C_EncryptInit all produce the CKR_OK return code. This represents a valid sequence of stateful operations. On the other hand, consider the following sequence:

(C_EncryptInit, CKR_OK), (C_Encrypt, CKR_BUFFER_TOO_SMALL), (C_EncryptInit, CKR_OK).

While the order of function calls is the same, the stateful behavior is not compliant with the standard. A call to C_Encrypt that results in the CKR_BUFFER_TOO_SMALL return code doesn't terminate the active operation and thus cannot be followed by a call to C_EncryptInit that returns CKR_OK. We will refer to a sequence of function and return code pairs as a *path*. A path is said to be valid if it is permitted by the stateful behavior described in the standard. We formalize the set of all valid paths in Cryptoki as a state machine where transitions are function and return code pairs and states abstract the set of configurations for a given set of operations. For example, the state machine for the incremental encryption operation is given below.



There are three function calls permitted by this state machine C_EncryptInit (Init), C_EncryptUpdate (Update) and C_EncryptFinal (Final). Transitions (edges) in the state machine are labeled by pairs consisting of a function name and return code. Edges without labels represent the fallback transition that occurs when none of the labeled transitions apply. Observe that this state machine assumes proper initialization of the library and creation of a session, which themselves are stateful operations. The full state machine for Cryptoki is the composition of the state machines for every operation type.

Functional behaviors: Describing how functions behave is one of the most fundamental aspects of Cryptoki, and as such, formalizing the descriptions in the standard is essential to the model-based testing approach. Our formalization is accomplished by specifying pre and post conditions for functions, which are boolean combinations of predicates over the system state and function input / output variables. Each pair of pre and post conditions for a function describes a contract the function must obey. If the constraints specified by the preconditions are true prior to executing the function, then the constraints over the postcondition must hold when the function's execution completes. For example, consider the C_OpenSession function that accepts, among other arguments, a "flags" parameter of type CK_FLAGS and "Notify" parameter of type CK_NOTIFY. A (partial) specification of the pre and post conditions describing the normal (intended) behavior of this function are given below.

---

requires: validSessionFlags(flags),
equals(Notify, NULL_PTR),

ensures: equals(rv, CKR_OK)

Informally, this contract states, if a call to C_OpenSession is such that the flags argument is valid and the Notify parameter is NULL, then the return value should be CKR_OK. To make this statement more precise, each predicate must be equipped with a formal meaning. We employ Cryptol[1] for this purpose, which is a domain specific language for cryptography. By way of example, the following Cryptol excerpt gives the semantics for the validSessionFlags predicate.

```
type CK_ULONG = [32]
type CK_FLAGS = CK_ULONG

(CKF_RW_SESSION : CK_FLAGS) = 0x00000002
(CKF_SERIAL_SESSION : CK_FLAGS) = 0x00000004

validSessionFlags : CK_FLAGS -> Bit
validSessionFlags flags =
   (flags == (CKF_SERIAL_SESSION || CKF_RW_SESSION)) \/
   (flags == CKF_SERIAL_SESSION)
```

---

[1] https://cryptol.net/

The types CK_ULONG and CK_FLAGS are defined as bit-vectors of length 32 and both CKF_RW_SESSION and CKF_SERIAL_SESSION are specification derived values of type CK_FLAGS. The validSessionFlags predicate accepts a flags parameter and returns true (i.e. the parameter is valid) if flags is equal to CKF_SERIAL_SESSION or equal to the bitwise-or of CKF_SERIAL_SESSION and CKF_RW_SESSION.

To the best of our knowledge, this is the first work that formally captures the function contracts specified in the PKCS#11 standard. While these formal descriptions alone are a valuable asset to developers, our toolchain also leverages them to automatically synthesize test code. Recall that transitions in stateful Cryptoki operations are defined by both the function that is called and the return value it produces. Further recall that the goal of model-based testing is to automatically generate a program that explores the paths through such a state machine. To enable this, we need a mechanism for deriving test inputs to functions that produce a desired behavior (return code). Fortunately, Cryptol is equipped with an SMT solver that allows our synthesis toolchain to automatically translate constraints over function arguments to concrete test cases. In particular, given a set of predicates over the parameters of a function, Cryptol can generate an assignment of values to the parameters such that the predicates evaluate to true under this assignment. When the function is invoked with these concrete values in a program, it should produce the behavior (return code) specified in the model. In other words, we can automatically generate test code that triggers a particular transition in our state model for an operation. Combining this with graph exploration techniques, we can generate sequences of concrete function calls that give path coverage over a stateful operation.

Results

Utilizing the formal specification and inference techniques described in the preceding section, we have developed an automated test synthesis engine capable of producing test suites designed to ensure PKCS#11 implementations conform to formal models. Our models formalize both the behavior of individual function calls (state transitions) as well as stateful behavior that combines multiple functions calls to perform cryptographic operations. The tests we synthesize provide total coverage over the transitions in the state machine models. For example, our model captures 34 conditions under which the C_EncryptInit function generates the CKR_MECHANISM_INVALID error. And each of these represents (as least) one synthesized test case. In addition to testing individual transitions, we also synthesize tests that cover paths, or sequences of transitions, in the state machine models. When testing paths, we start from the initial state of an operation and iteratively generate the set of all valid paths up to a user specified depth. Each path whose length is less than or equal to the configured depth is packaged into a single test case. Path based tests improve test coverage by triggering transitions from different states and are necessary in detecting the state-machine non-compliance errors described earlier.

The tests generated by our framework provide total coverage over state machine transitions and depth-bounded coverage over state machine paths. Because our function and state machine models formalize the Cryptoki standards document, we provide test coverage over the standard up to the precision of our models. The generated tests have been used to test PKCS#11

implementations, for example, as described in this report. The list of Cryptoki operations and functions we have modeled and tested is given below.

- Encryption (C_EncryptInit, C_Encrypt, C_EncryptUpdate, C_EncryptFinal)
- Decryption (C_DecryptInit, C_Decrypt, C_DecryptUpdate, C_DecryptFinal)
- Digest (C_DigestInit, C_Digest, C_DigestUpdate, C_DigestFinal)
- Sign (C_SignInit, C_Sign, C_SignUpdate, C_SignFinal)
- Verify (C_VerifyInit, C_Verify, C_VerifyUpdate, C_VerifyFinal)
- Sign Recover (C_SignRecoverInit, C_SignRecover)
- Verify Recover (C_VerifyRecoverInit, C_VerifyRecover)
- Session Management (C_OpenSession, C_CloseSession, C_CloseAllSessions)
- User Management (C_Login, C_Logout)
- Library Management (C_Initialize, C_Finalize)
- Key Generation (C_GenerateKey, C_GenerateKeyPair)
- Key Management (C_WrapKey, C_UnwrapKey)
- Object Management (C_CreateObject, C_DestroyObject, C_CopyObject, C_FindObjectsInit, C_FindObjects, C_FindObjectsFinal, C_GetAttributeValue)
- Slot and Session Management (C_GetSlotList, C_GetSlotInfo, C_GetTokenInfo, C_GetMechanismList, C_GetMechanismInfo, C_GetFunctionList, C_GetInfo, C_GetSessionInfo, C_GetOperationState)

In addition, the test framework provides test cases that ensure the following properties hold for the attributes associated with cryptographic keys.

1. Attribute values specified by key creation or import templates are correctly configured in the generated keys.
2. The proper default attribute values are assumed when they are not specified by the key creation or import templates.
3. Keys can only be created or imported from templates when all of the required attributes (as defined by the key type) are supplied.
4. Keys cannot be created or imported from templates when the template contains an invalid attribute type.
5. Keys cannot be created or imported when templates contain conflicting values for attributes.

Limitations

Because our approach uses black-box testing to enforce compliance, it is not possible to guarantee implementations that pass our test suite are completely free of compliance related errors. For each requirement in the specification, we use a representative set of example configurations and values to test the requirement. For example, when testing a transition in an encryption operation, our test suite picks sample values for the plaintext and cryptographic key. If a library has a compliance error that is specific to individual data elements, such as keys, our test suite is not likely to uncover those kinds of errors. This is an inherent limitation of black-box testing approaches that do not assume any knowledge of the system under test. On the other hand, most compliance errors are not data specific, and the black-box approach allows our tests to be run against any PKCS#11 library. Additionally, our test framework does not cover

behaviors that cannot be reliably reproduced by client applications, such as network outages and hardware/token failures.

Future Work

The PKCS#11 test suite we have developed provides test coverage over a significant portion of the standard. However, there are some areas where we cannot claim our test cases are exhaustive. These areas, which we leave to future work, are described below.

- The test suite does not yet cover all functions. For example, dual-function cryptographic functions (C_DigestEncryptUpdate and C_DecryptDigestUpdate) are not covered by this work.
- The test suite supports a fixed set of mechanisms for each operation type. As a result, some mechanism-specific compliance flaws may not be detected.
- The test suite is designed to enforce conformance with the standard, and not to detect security flaws that arise from ill-formed inputs. Although the compliance tests do occasionally detect security critical bugs, the test suite is not designed to eliminate this category of errors. Fuzzing is a good complimentary technique that can be used to detect such security flaws.
- The test suite focuses on the API and only provides limited testing of the underlying cryptographic operations. It should not be used to validate the implementations of basic cryptographic primitives.