ACVPro

Cryptographic Algorithm Test Harness

Document Version 1.4
Created: November 7, 2019

OVERVIEW

ACVPro is a complete solution for testing cryptographic modules using the new Automated Cryptographic Validation Test System (ACVTS). The ACVTS is the testing system created by the NIST Cryptographic Algorithm Validation Program (CAVP) to prove the correct operation of cryptographic modules.

The ACVTS replaces the Cryptographic Algorithm Validation System (CAVS) that has been used for many years. The CAVS system used a windows-based application, operated by validation laboratories, to produce test vectors and validated responses to those test vectors.

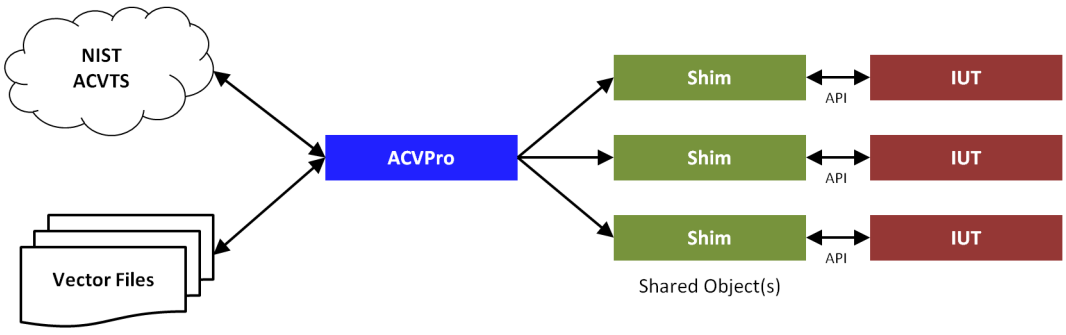
In contrast, the ACVTS relies on a cloud-based server to produce the test vectors, and to validate responses to those test vectors. The ACVTS utilizes the Automated Cryptographic Validation Protocol (ACVP), which describes the control flow that comprises a testing session, as well as the scope of data that passes between servers and clients using the protocol.

ARCHITECTURE

ACVPro consists of a Linux binary command line application and set of shared object libraries. Together these handle all command line options, ACVTS communication, and test data handling, as well as providing numerous utility functions for diagnostics and debugging. A "lite" version of the binary exists that is capable of handling vector files for processing, but cannot communicate with the NIST ACVTS servers.

Labs or Vendors provide a "shim", which is written to test a specific cryptographic module or Implementation Under Test (IUT). A shim consists of one or more functions that handle individual test cases for the algorithms supported by the module, as well as some supporting code. A shim is responsible for translating the data inputs into the data structures used by the IUT, using the IUT's API to execute the test, and finally translate the IUT output data to be returned. A shim is compiled into a shared object, and loaded by ACVPro at runtime. Shims have access to a wide variety of utility and helper functions provided by ACVPro's shared object libraries.

The following diagram covers the general usage model of ACVPro:



The ACVPro binary application is used to request vectors from the ACVP server, which are stored locally for processing. One or more shims exist to produce response files, which are also stored locally. Responses are submitted for validation and eventual certification. ACVPro can interface with any shim developed for use with ACVPro.

ACVPro is written in pure ANSI-C with minimal use of linked system libraries, to allow for compilation on a wide set of hardware and compiler platforms. ACVPro currently supports testing of the following algorithms:

- TDES/3DES (SP 800-38A)
- AES (SP 800-38A)
- AES-CCM (SP 800-38C)
- AES-GCM (SP 800-38D)
- AES-XTS (SP 800-38E)
- CMAC (SP 800-38B)
- GMAC (SP 800-38D)
- HMAC (FIPS 198-1)
- SHA-1, SHA-2 (FIPS 180-4)
- SHA-3 (FIPS 202)
- DRBG (SP 800-90A)
- RSA (FIPS 186-4)
- DSA (FIPS 186-4)
- ECDSA (FIPS 186-4)
- PBKDF (SP 800-108)
- ASKDF (SP 800-135)
- KAS (SP 800-56A)
- AES and TDES Key Wrap (SP 800-38F)
- Component Testing:
    - ECC-CDH (SP 800-56A Section 5.7.1.2)
    - ECDSA Signature Generation Component
    - RSA Signature Primitive (RSASP1)(PKCS#1 v2.1)
    - RSADP (SP 800-56B)

ACVPro was used to achieve certificate A:36 on the ACVTS production server, using Thingsoft's own "FOMRedux" OpenSSL product as the cryptographic module.

USE CASES

There are two primary use cases models that ACVPro is designed to address. The first use case is when the lab is responsible for the entire testing session using a shim that they control, either developed internally or provided by a vendor. The second use case is when the lab is responsible for generating and validating the vectors, but the vendor performs the actual testing using a shim of their own creation.

Both use cases follow the same 4 steps, with minor variation. These 4 steps encompass a complete testing session resulting in validation certificates being issued. If testing is completed on the ACVTS production server, the validation is posted to the ACVP website, and the certificate numbers are valid for use in a CMVP submission.

Step 1 – Request Vectors

To request vectors, the IUT's capabilities must be defined. These definitions include algorithms (e.g. AES, SHA), modes (e.g. CBC, CFB), value lengths (e.g. key, IV, message and plaintext sizes), and options (e.g. SHA byte-oriented, RSA Padding modes). These capabilities are saved in a JSON-formatted "capabilities" file. A capabilities file is developed (or a copy made) for each IUT and kept with the metadata for a project.

The capabilities JSON file is provided to ACVPro on the command line, and the NIST ACVTS server connection is established. The capabilities are transmitted to NIST, and the vector requests are returned. The vector requests are saved to a JSON-formatted file.

Step 2 – Process Vectors

To process vectors, a shim is required. The shim is specified on the command line, and is loaded and queried for the function handlers for each algorithm. The filenames for the requests and responses is also provided on the command line.

ACVPro will open the request file and process each individual test in sequence. For each test, the text data is translated into binary data and populated in a C structure. The structure is then handed off to a specific algorithm handler in the shim. When that function returns, the result of the cryptographic operation should be stored in the C structure provided to the shim function. ACVPro takes the results and buffers them for output. Upon completion of all test cases, the complete response file is written.

Step 3 – Validate Vectors

The completed response file is uploaded to the NIST ACVTS servers for online validation. The response file name is provided to ACVPro on the command line. The file contents are transmitted to NIST, and a polling process begins to wait for validation result responses. Once all responses are received, the overall Pass/Fail result is displayed. Additionally, the failed tests and reasons for failure can be output if desired.

Step 4 – Request Certificates (optional)

Assuming all vectors pass, the vendor can choose to request validation certificates on the passing results. This step is not required, for example if the session is just for testing or development.

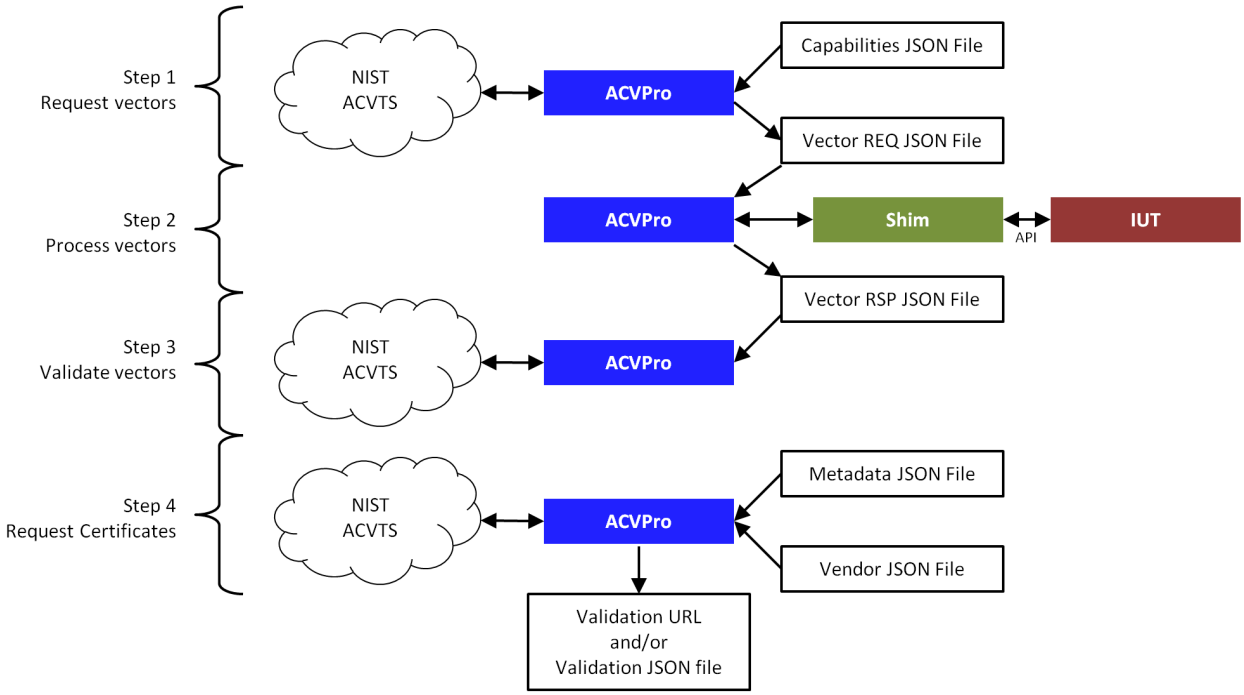
The submission for validation certificates requires detailed metadata to be transmitted to NIST. This metadata is contained in one or more JSON-formatted files, and includes most of the data found on the CAVS "vendor information" page, including:

- Vendor identification including address
- Vendor contacts (phone, email)
- Module information (name, software/hardware/firmware, version, etc.)
- Operational Environment for testing (hardware, OS, CPU, etc)

The metadata file(s) are specified on the ACVPro command line, and are transmitted to NIST. Upon completion of the NIST backend processing steps, a validation ID is returned and the validation is posted to the CAVP website. The Validation ID can be used to get validation information from the ACVTS servers, which can be output to a file for archival and reference.

CASE 1 – LAB TESTING

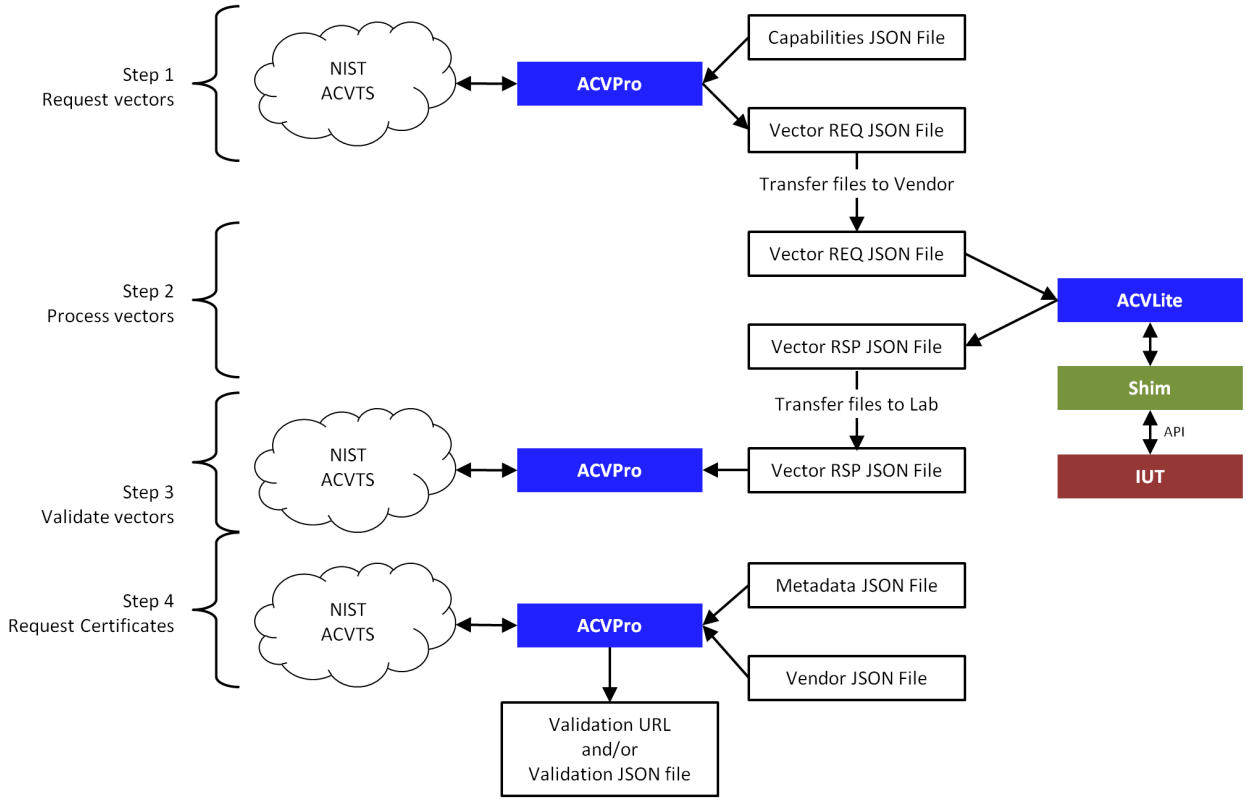
If the lab will perform the actual testing of the IUT, then the following use case applies:



ACVPro is used for each step, with intermediate files and metadata files generated and stored locally within the lab's IT infrastructure. A standard Linux host is used to perform all of the steps except 2. Step 2 takes place on the vendor's actual test hardware.

CASE 2 – VENDOR TESTING

If the vendor will be performing the testing themselves, then the following use case applies:



ACVPro is used to generate the request vectors, which are stored by the lab. The vector requests are transferred to the vendor for testing. The vendor uses ACVLite and their shim to generate the response files, which are transferred back to the lab. The lab uses ACVPro to transmit the responses to NIST and get the results. If the results pass, the vendor can use ACVPro to submit a request for validation certificates.

When a vendor is creating their own shim, they will need sample vectors to use for development. Under CAVS, the lab would also transfer the "FAX files" to the vendor so the authoritative answers would be known to the vendor. This would often be termed a "sample set" of vectors. The vendor would iteratively develop and test their shim until they achieved 100% passing results with the shim using the sample vectors. Once the work was completed, a new set of test vectors would be generated by the lab and transferred to the vendor (without FAX) for the actual testing.

Variations of this process are available with ACVTS. The lab or vendor can use OpenSSL or another library to create vector responses directly from the requests, which can then be directly compared to responses generated by the developmental shim. At some point in the future, with further enhancements by NIST, it may be possible to get the actual equivalent to FAX files directly from NIST at the time of generation. This is not currently allowed, but the protocol itself is under continual refinement and enhancement.

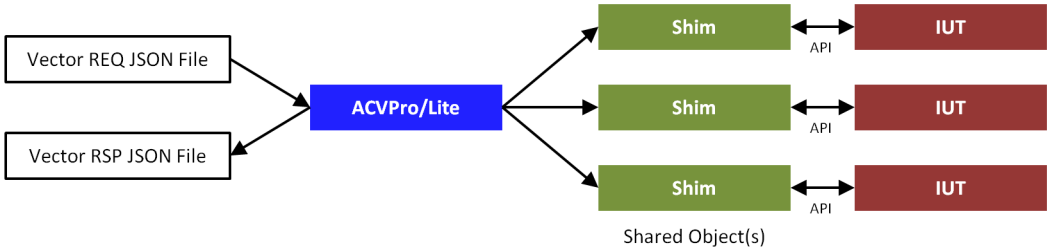
SHIM METHODOLOGIES

The interaction between the shim and IUT is a key feature of ACVPro. In the course of performing the algorithm testing of nearly 100 validation projects, we've learned a few tricks. Getting the tests into and out of the shim is one of ACVPro's specialties. We've developed several testing methodologies that are unique offerings of ACVPro in the ACVTS space.

Until this point, we have described the shim as a shared object made up of test functions and glue code. However, that collection of source code doesn't have to end up in a shared object. That same code can be compiled in a variety of ways to support these different methodologies. Below are the 3 primary testing methodologies.

STANDARD TESTING

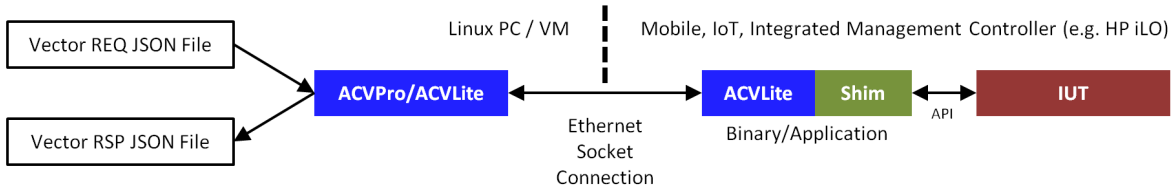
The most common methodology is where the testing binary, the shim, and the IUT all reside on the same hardware platform.



Here the shim is compiled as a shared object. The shim can be used by either ACVPro or ACVLite. The same ACVPro/ACVLite binary can be used with any number of shims interchangeably. For example, you can run a vector set through the OpenSSL shim to generate "known good" FAX vectors before running the same vectors against a custom IUT, with no recompilation or change in environment.

SOCKET TESTING

For modules with limited functionality or access, such as mobile phones, IoT devices, and embedded management systems (e.g. HP iLO), it may not be possible to physically store the vectors on the test hardware, or allocate enough memory to load the file for parsing. In these cases, ACVPro can be split into two halves, with the two sides communicating over an Ethernet link:

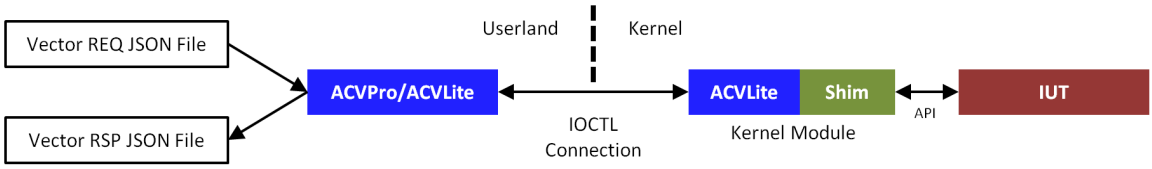


In this scenario, the shim code is compiled together into a single binary with a specially-prepared bundle of static object code containing parts of ACVLite. This single binary can be loaded on the remote device. When executed, it starts a socket listener.

On a separate Linux PC or Virtual Machine, ACVPro or ACVLite is executed. This binary is responsible for parsing the vector requests and isolating each single test in sequence. The single test case is passed over the socket, where it is handled by the binary running on the IUT. When the remote application responds with a result, that result is buffered for output to the vector response file.

KERNEL TESTING

For modules that reside in the Kernel, such as Linux Kernel Crypto, a userland binary with file access is paired to a kernel module with crypto library access.



In this scenario, the shim code is compiled together into a single binary with a specially-prepared bundle of static object code containing parts of ACVLite. Additionally, code must be provided to handle normal kernel module behaviors such as start/stop. Building a kernel module, and the specific code requirements thereof, is different for every operating system. ACVPro provides a lot of that work for you by abstracting away some of the details and allowing you to focus on the important part – interfacing with the IUT's API.

On the same PC, in userland, the ACVPro or ACVLite binary is executed. Each test in sequence is sent to the kernel via an IOCTL using simple read/write semantics. The ACVLite portion of the module receives the IOCTL, prepares the test, and calls the shim test function. Upon return from the shim, the ACVLite portion replies to the IOCTL sending the result bytes back to userland.

Kernel testing configurations require additional code and compilation steps. As such, they will generally require some support or collaboration with ThinqSoft to assist in the development and compilation of binaries and kernel modules.

CURRENT STATUS

ACVPro is fully functional for the algorithms listed above, using the standard use case model. It has been used to achieve passing results on the ACVTS Production Server. It is currently based on libacvp, an open source library created by Cisco. Due to the limitations of libacvp, the validation process currently requires many individual steps, and most of those steps require all data to be processed in a single monolithic submission.

ACVPro can be used to achieve production credentials for any NVLAP validation lab, satisfying the requirements imposed for the transition beginning January 1, 2020. As part of the initial license, Thinqsoft will mentor labs through the accreditation process for both the testing and production servers, and execute the initial testing to achieve accreditation.

The shim methodologies describing socket and kernel testing configurations are part of our roadmap for future development.

ROADMAP

Thinqsoft will continue to develop and improve ACVPro on a continuous basis. The roadmap is focused on improving the backend processing to achieve a more robust and simplified submission processes, and well as to improve overall usability. The end goal is to make the validation process as easy and familiar as the old CAVS, while maximizing the utility and flexibility that ACVP makes possible.

To achieve these goals, the following tasks outline the major improvement and features:

1) Remove external dependencies on curl and OpenSSL by migrating to different providers for TLS session management and crypto primitives
2) Replace libacvp with new custom-built application protocol handling. *The remaining items are not possible with libacvp.*
3) Create socket and IOCTL shim methodologies
4) Facilitate use of a single master “Project File” to track client metadata and progress
5) Allow for increased granularity in request (REQ), validation (RSP), and submission (metadata) streams:
    a. Request some vectors, validate some vectors, (steps 1-3)
    b. Repeat (a) as often as necessary,
    c. Submit metadata and reference the multiple session streams (step 4)
6) Create multiple binary options:
    a. Full binary with TLS (ACVPro) for use by labs for full-feature testing
    b. Slim binary with no TLS (ACVLite) for use by vendors to execute tests from files
    c. Static library for use in creating Linux kernel modules

ThinqSoft anticipates items 1-3 will be completed before ACVTS becomes mandatory on July 1, 2020. We currently have clients who require these capabilities within that timeframe. The remaining items, as well as additional requests received from customers, will be developed thereafter.

COMPETITIVE ANALYSIS

There are two competitors in the ACVTS space, "libacvp" by Cisco and "ACVPProxy/ACVPParser" by Atsec. Both of these products are Open Source Software (OSS) and their licenses allow for use in a commercial environment. For details of their licensing, see the individual websites.

Each of these solutions has limitations and complexities that stunt their flexibility with respect to the types of environments or modules with which they can interface. The following sections discuss each module and differentiation ThinqSoft has achieved. The information gathered and presented is based on our own internal research. We respect that our analysis may not be complete, and is certainly biased.

ThinqSoft doesn't hide the fact that at the time of product launch, ACVPro is based on libacvp. We have modified it, wrapped it, and refactored it as best we can to get the capabilities in our current product. We have used the current version to validated modules on both the NIST test and production environment. The product today is sufficient to meet the transition requirements set forth by NIST and to validate modules in the "standard" shim methodology discussed above.

In our roadmap, however, we note that several of the capabilities and shim methodologies that are important to us and our customers are impossible to achieve without extensive modification to libacvp. Consequently, we have elected to remove our dependency on libacvp, and the dependencies imposed by the use of libacvp (curl and openssl). We have identified a TLS and crypto provider that can be included in our source tree and can be compiled under our strict control. We discuss these issues in the section on libacvp below.

Please contact us directly if you would like to discuss the issue presented here in greater detail.

LIBACVP

Libacvp is created and maintained by Cisco. Cisco worked with the CAVP in the development of the ACV Protocol, and libacvp is a sample client provided to demonstrate how to interoperate with the NIST servers to perform testing. It includes a sample application using the library to perform testing with OpenSSL, either as the shared object libcrypto.so, or through the static linking of the FOM.

Architecture with respect to IUTs

Compilation of libacvp is straightforward. It requires either static or dynamic libraries for curl/libcurl and openssl (libssl and libcrypto). The result of compilation is the libacvp library and a binary of the sample application for testing OpenSSL. The libacvp library can then be used dynamically or statically linked into a project (the sample app can only use it dynamically). As libacvp is just a library, the sample application is responsible for CLI options, gathering and communicating IUT capabilities, managing process flow of the test session, responding to errors from libacvp, etc, in addition to interfacing with the IUT to perform tests.

To test IUTs other than OpenSSL, the sample application would have to be forked and modified, or rewritten from scratch. With all the responsibilities of the app stated above, the application becomes a "fat" binary, with lots of code repeated in each new fork. When dealing with dozens of modules over time, the challenges of maintaining bug fixes and features grows quickly.

ThinqSoft addresses this issue by separating the code that makes a usable application (ACVPro) from the code that interfaces with the IUT (shims). Replicating this separation requires a lot of effort.

Dependencies and Environments

One issue that became apparent in our initial uses of libacvp is the idea of shared objects in the environment. If the IUT is OpenSSL, then the shim is expecting runtime linking to libcrypto.so. However, libcurl, which is used by libacvp, is also expecting a runtime link to OpenSSL. Which OpenSSL does each of these link to? If it's a known-good OpenSSL to ensure reliable operation of curl, then you're not linking the IUT. If you link the IUT, then curl is *not* using a known-good library to perform the session and communications.

Static linking of the known-good OpenSSL solves this ambiguity, but then curl must also be statically linked to ensure it's using the static OpenSSL. With static linking of curl and openssl, there should be no dependencies on a shared OpenSSL, and the IUT can be specified using LD_LIBRARY_PATH or a similar means.

ThinqSoft has developed a build system to reliably generate static libraries of libcurl, libssl, libcrypto and libacvp. These 4 static objects are linked into the ACVPro binary and supporting shared libraries that comprise our solution. For use in a production environment, labs or vendors would also have to develop such a system of reliably generating these static or shared object dependencies, and would be required to perform this sequential build process for each new environment.

The reliance on these libraries, and their sometimes complicated build systems, can impose additional risk and complexity when working with unusual or customized environments. For example, the test environment may be closed (no build tools), and may not include libcurl as a system library. How, then, can this dependency be satisfied? Libcurl must be generated statically and linked into the libacvp shared object and application binary on a build machine and ported over. Now you've added the complexity of building libcurl and OpenSSL on a stand-in for the target environment.

Architectural Flexibility

The libacvp library/application is designed to solve a single problem – validate a module such as OpenSSL in as simple and straight-forward a way as possible. As such, many assumptions and simplifications have been baked into the code. Changing these built-in assumptions and behaviors is challenging, and would only serve to complicate their solution – it would add complexity to what is designed to be a simple process.

As a concrete example, when reporting IUT capabilities, you also provide a function pointer to the handler for each algorithm. That handler is called whenever a test is discovered for that algorithm name/class. There is no built-in capability to notify the start or end of testing of an individual algorithm (before the first test and after the last test).

Imagine an IUT that required some startup and shutdown code for RSA functions (to allocate and free some internal state). Further, due to limited system resources, you can't just make these internal state calls at the start and end of testing, because those resources are also needed for other tests. You desire to be notified so you can perform this setup only when RSA is about to be tested, and teardown as soon as this memory is no longer needed. There is no simple way to do this with libacvp. There are some workarounds that can be done, such as a singleton handler that monitors the test algorithm ID and maintains some internal state about the "previous ID". This is an onerous burden to implement at the application level, and is a feature that is much easier to safely and reliably perform at the point where the vectors are being read from file.

ThinqSoft solved this particular limitation, and ACVPro has this start/stop feature, but it required a lot extra code outside of libacvp to make it happen. There are dozens of other small architectural decision like this that limit the features we can offer; features that we offer in our CAVS product and our customers have relied on for more than 5 years.

Protocol completeness and session handling

With libacvp, capabilities are gathered in one large set, vectors are requested and downloaded in one large set, and results are uploaded in one large set. It's not possible to get some AES vectors and pass them, then grab SHA vectors and pass them, and finally get certificates for both by referencing two test session ID. During your request for certificates, you can only reference one single session ID, which must include all algorithms/test that are part of the module. This is not due to ACV Protocol limitations, but rather due to decisions made in the architecture of libacvp.

In the final step for validation, the metadata for the module is uploaded to NIST to be associated with your validation. What happens if you make a typo and want to correct it? The ACV Protocol has the capability to request an update to data previously submitted, but libacvp does not expose this capability. This means if you make a typo, you have to resubmit the entire metadata file. And under certain conditions, it requires a complete re-submission of the tests, from request through completion.

Summary

These types of low-level architecture decisions limit the overall flexibility in the types of features that can be built into any application based on libacvp. The environmental and library linking concerns create additional burdens and risks when building a repeatable and reliable line-of-business service offering. We have highlighted some of our largest concerns, but many others exist. These issues may or may not affect your intended usage.

ThinqSoft evaluated these risks and limitations to be too severe to consider libacvp for anything more advanced that a single-shot solution, or for use as we have as a temporary solution and proof-of-concept while development work on a robust solution continues.

ACVPPROXY / ACVPPARSER

ACVPProxy/ACVPParser is developed and maintained by Atsec, an NVLAP-approved lab in Austin, TX. It was initially announced to the public November 2018, but became prominent when Atsec became the first lab to validate on the ACVTS production environment in July of 2019. By this time, ThinqSoft was already well advanced in our use of libacvp, but we did evaluate the Atsec solution for a possible migration. In the end, there were no significant advantages over libacvp with respect to our feature sets and the needs of our customers.

Overall Architecture

The source code builds to two core components: ACVPProxy, a shared object that is responsible for all communications with the ACVTS servers, and ACVPParser, a binary that handles test sessions, file management, and the parsing and processing of test files. The code that interfaces with the IUT is placed in function stubs within the code base of ACVPParser, and is compiled into the binary.

This overall architecture is similar to libacvp, although the code contained in the two parts is factored differently. This imposes the same limitations as libacvp discusses above, where the IUT interface code is built into the binary. This architecture makes it difficult to build a line-of-business where you are testing a wide array of products from different manufacturers and on different environments, and are forced to maintain multiple copies of a large source code base which is identical across projects. ThinqSoft believes our architecture of robust, closed binaries combined with small, open, and focused shims is superior in real-world usage.

This architecture is not suitable to supporting the wide array of shim methodologies that ThinqSoft's solution can support. This has a material impact on the types of products this can validate in its current form, and creates a high burden on modifying it to enable testing on environments such as mobile devices and embedded systems.

Session Handling

The ACVPProxy/ACVPParser solution offers superior handling of session traffic and metadata. They offer the capability of storing relevant metadata and session information in an ongoing project file. There is also the capability of having this data stored in a database for additional flexibility and security, allowing for better backup/redundancy options, and even allowing for the possibility of data mining. The value of this feature is dependent on the sophistication of the subsystems surrounding it.

On the ACVPro roadmap, ThinqSoft plans to implement a "project file" for each testing project, that will aggregate and maintain all of the metadata for a project, along with maintaining an ongoing list of session and test IDs, and the state of each. This will greatly enhance our flexibility in handling session traffic, and will achieve some parity with ACVPProxy/ACVPParser in this regard. We do not plan to add the database backend capability, but we are happy to re-examine this capability if there is strong interest among our customers.

Summary

The architectural similarities to libacvp impose the same challenges with respect to interfacing with IUTs in a large, dynamic, and ongoing production environment. It is certainly a more robust solution that libacvp, and handles the management of projects better than libacvp.

The roadmap ThinqSoft has developed includes capabilities that are difficult or impossible to replicate with ACVPProxy/ACVPParser as it is currently constructed. In addition, we will reach parity with the metadata and session handling capabilities, to the degree we evaluate these to be useful to our clients (excepting the alternate storage options for the project metadata).

LICENSING

ACVPro is licensed for use on a yearly basis. Licensees will receive:

- Copies of the binaries and shared objects compiled for the x86_64 architecture
- Source code for a complete shim, capable of validating the OpenSSL FOM
- Source code and headers to produce additional shims
- Eight (8) hours of training, to be conducted at a time and place of mutual agreement, with reimbursement for travel expenses outside of DC Metro area

In addition, the licensees are entitled to:

- Receive mentoring and assistance in achieving fully credentialed access to the ACVTS testing and production environment to meet CAVP transition requirements
- Use the provided full binary and libraries to request and validate vector sets, and to submit results for certification
- Provide contracted vendors with the headers, documentation, and samples necessary to build shims for use with ACVPro
- Allow vendors to use the provided slim binary and shared libraries to product vector results, for completion of specific projects (vendors must cease use after completion of project). If desired for continuous use in a QA environment, specific vendor licenses are available directly from ThinqSoft
- Receive all updates, improvements, bug fixes, and other enhancements for the license period
- Make requests for features and pose novel use case scenarios
- Support in the discovery and remediation of bugs and defects

Additionally, all licensees will receive preferential pricing for:

- Creation of new binaries for additional architectures. We ship Linux x86_64 binaries, and will work with you to generate other platforms (e.g. iOS, Android, Solaris, HP-UX) on an as-needed basis. Cost recovery is only for actual time and materials. Assistance from the Customer with environmental details, and potential cross-compile details, is required.
- Engineering, development, and support for the creation of new shims. Reasonable support is always provided with the license for support of your development work. The definition of “reasonable” will be on a case-by-case basis, and always in consultation with you.
- Preferential pricing for development of custom shims on a work-for-hire basis
- Full service completion of ACVTS projects for your clients (outsourced testing)

At the end of the license period, a new license must be purchased or usage of ACVPro must be terminated and all binaries deleted.