Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

OpenVPN TLS-Crypt-V2 Key Wrapping with Hardware Security Modules

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von:	Emily Ehlert
geboren am:	
geboren in:	
Gutachter*innen:	Prof. Dr. Jens-Peter Redlich
	Prof. Dr. Florian Tschorsch
eingereicht am:	....

HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

OpenVPN TLS-Crypt-V2 Key Wrapping with Hardware Security Modules

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von:	Emily Ehlert
geboren am:	10.05.2000
geboren in:	Königs Wusterhausen
Gutachter*innen:	Prof. Dr. Jens-Peter Redlich
	Prof. Dr. Florian Tschorsch
eingereicht am:	....

Abstract

The control channel of OpenVPN can be protected using pre-shared keys distributed to clients using the tls-crypt-v2 mechanism. A wrapped client key is send to the server when establishing a tunnel. If the server wrapping key is compromised, all client keys would need to be renewed. This bachelor thesis explores methods of implementing the functionality of tls-crypt-v2 using Hardware Security Modules, making the server key difficult to extract. For this purpose, the Java Card technology, YubiKey cryptographic tokens, and the PKCS#11 interface are analyzed, and example implementations are showcased. The technologies are integrated with OpenVPN using its plugin capability, which requires an extension to it to support the mechanism. A key-derivation scheme is used for the YubiKey tokens and the available SmartCard HSM PKCS#11 token to imitate the functionality of tls-crypt-v2 since neither has support for the required cryptographic operations. The results show, that while hardware security modules can be used to handle tls-crypt-v2, providing improved security, there are some drawbacks. Even the fastest tested device, the YubiKey 5C NFC, able to authenticate a client key within 27 ms, is significantly slower than the 0.01 ms required by plain OpenVPN, leading to potential delays and a substantially increased Denial-of-Service attack surface. Faster hardware solutions exist, but at a price of several hundred to thousands of euros, they are not a viable option for small-scale usage.

Contents

1. Introduction

For security reasons, in many organizations, important documents and digital resources are only available on their intranet. To allow the usage of these resources outside the network, Virtual Private Networks (VPNs) have become a key tool. Since the start of the COVID-19 pandemic, this trend has accelerated dramatically and the use of VPNs has become common even among traditionally low-tech businesses like the classical trades. Since they allow direct access to an organization's internal network, the security and integrity of the VPNs are of the utmost importance.

One of the most popular VPN solutions is OpenVPN, which creates an encrypted IP or Ethernet tunnel connecting a remote host or site to a company's network. An OpenVPN connection consists of a data and control channel, with the second one being used to set up the TLS session and other connection management purposes.

OpenVPN offers multiple methods of protecting the control channel. One is tls-crypt-v2 which offers confidentiality, integrity, and authenticity, through AES-256-CTR and HMAC-SHA-256. Each client receives an individual key. OpenVPN uses key wrapping for this mechanism, allowing the server to discard the client keys while not connected. Wrapping uses a server key to encrypt and add an authentication code to each client key. When a client wants to establish a connection, it must include the wrapped client key. The server then can unwrap it, i.e., decrypt and verify, to obtain the plain key. This procedure, however, leaves the server as a single point of failure. If a malicious actor were to acquire the server key, they would be able to decrypt all wrapped keys rendering the control channel protection ineffective, and requiring to renew all keys. If the server key were to be kept on a Hardware Security Module (HSM), an attacker could not or only with significant costs extract it, even if they had (physical) access to the token.

The objective of this bachelor thesis is to examine different hardware methods of handling tls-crypt-v2, create an implementation for each, and test their performance. To achieve this, the ability of OpenVPN to extend its capabilities through the use of plugins is utilized. Since OpenVPN, as of version 2.6, does not offer a way for a plugin to perform the tls-crypt-v2 operations, a new plugin hook within OpenVPN has to be implemented, allowing the server to hand over wrapping operations to a plugin. The extensions will either receive a wrapped or plain client key and unwrap or wrap it, returning the result to OpenVPN.

2. OpenVPN

2.1. Basics

OpenVPN is an open-source VPN solution utilized for creating point-to-point or site-to-site network connections. A VPN is a software for enabling a client to route their traffic through a secure, i.e., encrypted and authenticated, connection (tunnel) to a server. The main applications are either to remotely connect to a private network, like the intranet of a company, or for privacy reasons, by concealing the traffic from the internet service provider and the IP address from the server. These properties help circumvent censorship.

The OpenVPN software can run in a server or client configuration. For cryptography, OpenVPN predominantly uses OpenSSL but also supports Mbed TLS, previously called PolarSSL [1]. OpenVPN tunnels can be encrypted and authenticated using any symmetric cipher and MAC function provided by the crypto backend [2, 3]. Keys for encryption are either pre-shared (PSK) or negotiated using the Transport Layer Security (TLS) protocol [4]. The exchange allows the use of public key infrastructure and certificates. Authenticating a client using a username and password is also supported. The thesis will assume the use of TLS. The connection between the client and server can be via UDP or TCP. Both control and data channel packages are sent over the same connection. The TLS session on the control channel requires "a reliable, in-order data stream"[5]. For UDP, the VPN provides its own reliability layer for control packages on top of transport protocol [6]. After setting up the network interface, OpenVPN can drop its privileges by setting the effective user ID of the calling process to "nobody".

To initialize a VPN session, the client sends either a `P_CONTROL_HARD_RESET_CLIENT_V2` or if the client wants to use the tls-crypt-v2 control channel protection `P_CONTROL_HARD_RESET_CLIENT_V3` package [6][7]. Version 2 and 3 employ the same procedure to derive the session keys. Version 1 utilized a slightly different method, but is not in use anymore [8]. When the server receives the first packet, if desired, it handles control channel protection and establishes a TLS session. To derive the MAC and symmetric cipher keys, random key material is generated on both sides and exchanged via the control channel. The keys are used unidirectional, meaning each side has its own keys for authenticating and encrypting messages they send. After the keys are established, traffic data can be sent through the data channel.

2.2. Control Channel Protection

By default, the control channel is only protected by TLS. An attacker can read the initial handshake messages before a secure session is established, allowing them to see which certificates were used. Besides privacy risks, an adversary can access the TLS stack, exposing a large attack surface for potential vulnerabilities and Denial Of Service (DOS) attacks. Streun *et al.* [9] discovered, that an attacker may perform a forced key renegotiation attack by inserting a `P_CONTROL_HARD_RESET_CLIENT_V2` message with a spoofed source IP address of a current connection. The server responds to the feigned key renegotiation. The real client receiving the response believes the server wants to renegotiate keys and answers accordingly. This process leaves both sides with improper handshakes, resulting in an unusable connection. The client cannot recover from the attack by itself, even when restarting. The only solution is restarting the server application.

By authenticating the control channel, an attacker is unable to modify the TLS stack or insert genuine-looking packages into a connection, mitigating the forced key renegotiation attack. By further encrypting the control channel, an adversary can not see which certificates were used for the TLS session. Symmetric encryption also provides some post-quantum security for the whole OpenVPN connection, including the data channel. Even though the cryptographic primitives used for encrypting and authenticating the tls-crypt-v2 client keys, AES-256-CTR and HMAC-SHA-256, can be weakened with a quantum computer, they are not efficiently breakable. Thus, the quantum insecure, asymmetric TLS key negotiation is protected.

OpenVPN has three options for securing the control channel [4]:

- tls-auth — The "HMAC firewall" provides authentication for the control channel. Clients belonging to a server cluster receive the same key during setup through a secure channel. Messages sent via the control channel require a valid message authentication code. It is generated using the HMAC construct and utilizes by default the SHA1 message digest, but other digests can be specified via the auth option [10]. The HMAC firewall has been part of OpenVPN (almost) since the beginning. The earliest source code available on the internet, OpenVPN 1.0.2 released on 28th March 2002, already contains the tls-auth option [11]
- tls-crypt — Adds encryption in addition to authentication. Like with tls-auth, each client in a server cluster receives the same keys during setup, used for a symmetric cipher and HMAC. As digest SHA-256 and as cipher AES-256-CTR is employed [12]. This option was introduced in version 2.4 [13]
- tls-crypt-v2 — Improves on tls-crypt by moving from global keys to client individual keys for cipher and authentication. The option uses the same cryptographic primitives as tls-crypt. It was introduced in version 2.5 [14]

OpenVPN's control channel protection uses fixed crypto functions to minimize overhead, removing the need for a handshake, and determine as fast as possible if a package is legitimate or not. To help DOS protection, invalid packages get silently discarded. However, this DOS protection does not seem to make any significant impact. According to Streun *et al.* [9], these options even harm DOS resilience compared to an unprotected control channel. With only 100 Mb/s of generic initiation package traffic, a single OpenVPN instance can be completely denied from processing any legitimate traffic.

2.3. TLS Crypt v2

The tls-crypt-v2 mechanism is described in the [tls-crypt-v2.txt] document and implemented in the [tls_crypt.c] source file. An overview of the procedure is shown in fig. 1. The option is used to encrypt and authenticate control channel packages with AES-256-CTR and HMAC-SHA-256 using a client-specific pre-shared key. To accomplish this, tls-crypt-v2 performs three different tasks.

First, the server needs its own set of keys to wrap the client keys K_c. Since the server does not save any K_c permanently, a client sends their keys to the server in the initial message of a session. To accomplish this securely, OpenVPN uses a technique called wrapping. At its core, it consists of encrypting a sensitive key, in this case, the client key using a different key and adding an authentication code to the cipher text. The wrapped data can then be stored and transferred in any way desired. Only the key used for wrapping must be stored safely. This strategy can be useful when a cryptographic device has limited space. The keys are stored off-device and imported when operations need to be performed. With tls-crypt-v2, the server key is the wrapping key. To generate

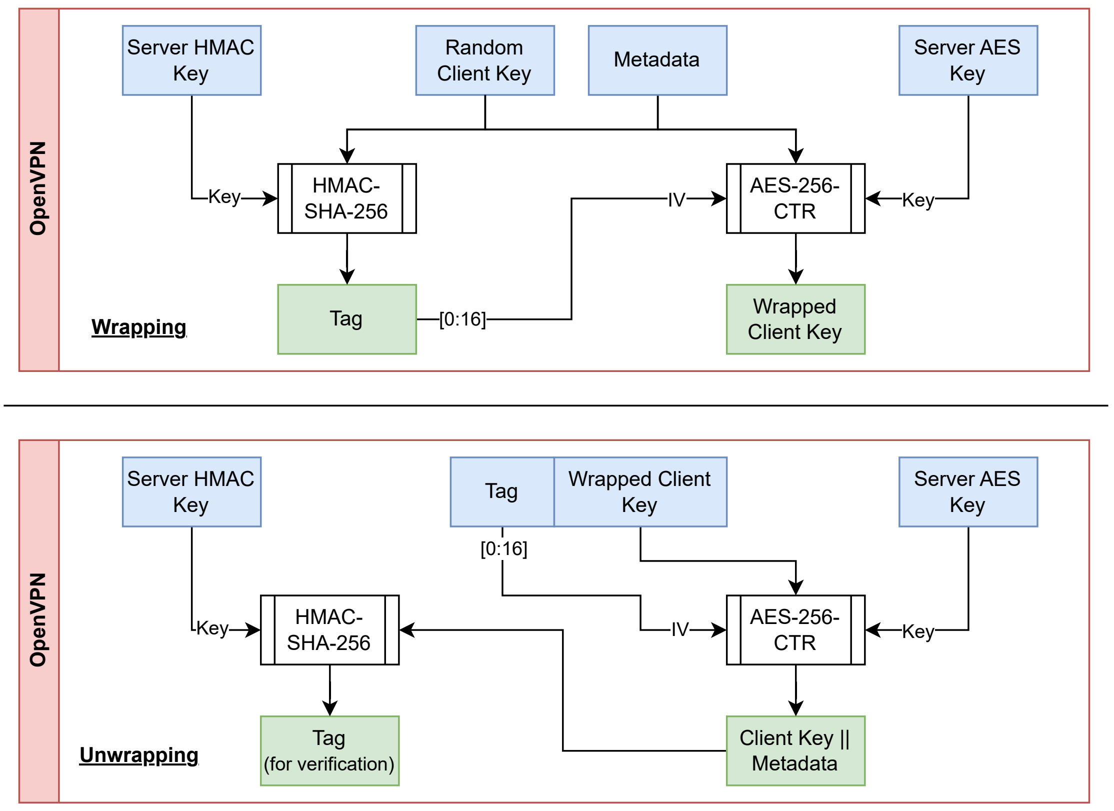


Figure 1: Overview of TLS Crypt V2 as specified in [tls-crypt-v2.txt]. Length component omitted for improved clarity.

any client keys, the server first needs to create its key file. This generation consists of creating two 512-bit keys and encoding them using the Privacy-Enhanced Mail (PEM) file format. The first 256 bits of the first key are later utilized as AES-256-CTR key K_e and the first 256 bits of the second key are used as HMAC-SHA-256 key K_a. The keys are each 512 bits long because OpenVPN's key structures can contain a maximum of 512 bits of key material. This circumstance also helps future-proofing the mechanism. Should more key material be needed, no new keys have to be distributed. All OpenVPN server instances in a VPN group must use the same server key.

After server key creation, client keys K_c can be generated. A K_c is 2048-bit long, containing four keys. Each side of the connection has two 512-bit long keys for encrypting and authenticating packages. Besides the plain K_c, the server also creates the wrapped key WK_c. During key generation, metadata may be added to K_c before wrapping it. OpenVPN's key wrapping is based on Rogaway & Shrimpton [16]. $|$ is used as symbol to indicate the concatenation of two bit sequences.

$$\begin{aligned}
&len = \text{byte length of } WK_\text{c} \\
&T = \text{HMAC-SHA-256}(K_\text{a}, len|K_\text{c}|metadata) \\
&IV = \text{128 most significant bits of } T \\
&WK_\text{c} = T|\,\text{AES-256-CTR}(K_\text{e}, IV, K_\text{c}|metadata)|len
\end{aligned}$$

K_c and WK_c are then concatenated and PEM encoded. The client uses the plaintext key for itself and sends WK_c to the server when establishing a tunnel.

To protect against replay attacks, the server sends a challenge in its first reply. To prevent resource exhaustion attacks, it does not save the K_c. The client must resend WK_c in its response to the challenge. The server thus unwraps the client key twice. All further details are contained in [tls-crypt-v2.txt].

With the current implementation of tls-crypt-v2, the server is the single point of failure. Thus, if it would get compromised and the attacker had access to K_e and K_c, all client keys would need to be replaced since the attacker could unwrap all WK_c, rendering the control channel protection ineffective.

2.4. Plugins

Since Version 2.0, OpenVPN has support for plugins [17]. They extend OpenVPN's capabilities without the need to modify its source code. Plugins are pre-compiled, dynamically-linkable libraries implementing the plugin interface defined in [openvpn-plugin.h.in]. When an extension is loaded during the start-up sequence of the program, it registers itself with OpenVPN to intercept specific callbacks (plugin hooks). A hook is a specific event occurring during the execution, like connecting a client. Currently, as of version 2.6, OpenVPN supports 15 different callbacks. See appendix A. Plugins can receive and return data to OpenVPN. Generally, there are two categories: passive and active callbacks. Plugins intercepting a passive callback should not return anything, while active plugins are expected to. The return data and status of an active plugin are used to influence the execution of the main program. For example, the `OPENVPN_PLUGIN_AUTH_USER_PASS_VERIFY` callback, authenticates a client using a username and password. If the plugin returns with a negative return/exit code, the connection attempt is rejected. This can be used to verify a user against an LDAP server.

Since its initial release, more callbacks and features have been added to the plugin interface. Presently the interface is version 3 and the Plugin V3 StructVer is 5. StructVer refers to the version of the structs the plugin receives when loaded or called during a plugin hook. With the current iteration, plugins have access to five callback function pointers. These functions are provided and implemented by OpenVPN, but they can be executed from within a plugin.

- `plugin_log()` and `plugin_vlog()` — These functions improve upon the standard `printf()` by adding more information to the output like plugin name and allowing to set a verbosity level, only printing a message if the verbosity level is high enough.

- `plugin_secure_memzero()` — Function used to securely erase memory. It was developed by Yang *et al.* [19] to be specifically not removed by a compiler during optimization.
- `plugin_base64_encode()` and `plugin_base64_decode()` — Used for encoding data from and to its base64 representation.

The plugin interface defines required and optional functions to be implemented by a library. For version 3 plugins, the following three are required: `openvpn_plugin_open_v3()`, `openvpn_plugin_func_v3()` and `openvpn_plugin_close_v1()`. From the optional functions, only the `openvpn_plugin_select_initialization_point_v1()` to choose the plugin's load point will be used in this project. This ability is important when the extension has to be set up before OpenVPN potentially drops its privileges.

`openvpn_plugin_open_v3()`

OpenVPN calls this function during the start-up sequence when the program loads the requested plugins. The extension uses it to declare which callbacks it wants to intercept. `openvpn_plugin_open_v3()` usually performs setup work like creating a log file or connecting with an LDAP server. The plugin context is a plugin-defined struct containing runtime-persistent information. The program provides this data to the plugin during every callback. If the context is needed, it must be allocated here and added to the return structure of the function. The arguments provided to it contain all parameters set when specifying the plugin in the OpenVPN configuration file or command line options, for example, a file path of a log file. The `struct openvpn_plugin_args_open_in` also contains pointers to the callback functions, which should be saved within the plugin. Depending on the initialization point set, the plugin can add more options to the OpenVPN configuration. For more information concerning arguments available to `openvpn_plugin_open_v3()`, see [openvpn-plugin.h.in].

`openvpn_plugin_func_v3()`

When OpenVPN intercepts a callback and the plugin has indicated support for it, this function is called. Depending on the callback type, the plugin receives various information from OpenVPN. It always receives a pointer to its plugin context or `NULL` if not set, the type of callback intercepted, and a pointer to the environmental variable set in OpenVPN, e.g., the verbosity level. Depending on the plugin hook, further information like a client-specific context or the currently used X.509 certificate may be provided. For other data, the `char** argv` pointer is available, which is similar to the `argv` pointer received by the `main()` function of a C program. Since this is simply a list of strings, data must either be encoded to exclude `0x00` bytes or contain some form of length specification.

Depending on the plugin hook, the return code implementation differs. With a passive callback, a negative response is considered an exception. For active ones, it is regarded as regular behavior. A negative return code for the verification of a client with a username and password is a rejection of the connection attempt. Besides the return code, a plugin can also send back data to OpenVPN using a so-called string list, essentially a list of key-value pairs containing strings. This capability is vital to return data like an unwrapped client key.

`openvpn_plugin_close_v1()`

When OpenVPN unloads the plugin, it calls this function. It cleans up the plugin (context), like closing the file handle for a log file. The function returns nothing and only receives the plugin context as an argument.

3. Objective

The aim of this bachelor thesis to answer the question if HSMs can be a viable solution in improving the security of the tls-crypt-v2 mechanism, as described in section 2.3. It is desirable, but not a requirement, that an implementation behaves the same as the original specification describes the mechanism. With the additional ability to import an existing server key into the HSM, working setups could easily migrate without the need to reissue any client keys. To achieve this goal, I want to use the plugin function of OpenVPN to extend the server's capabilities. This approach requires the least amount of modification to the OpenVPN source code, which makes it easier to validate and integrate the changes into OpenVPN. Some modifications are necessary since the application does not provide a plugin hook for handling tls-crypt-v2 operations. Through the use of plugins, it is also easier to implement new HSMs in the future.

First, I will define the requirements an HSM needs to fulfill to be used for this application and then research some devices and standards which do. Afterward, I will implement the plugin hook and sample plugins for every capable HSM previously selected. Finally, I want to test the performance of each plugin and compare it to the current implementation within OpenVPN.

4. Hardware Security Modules for Key Wrapping

4.1. Requirements

The token should preferably support AES and HMAC. However, the functionality of tls-crypt-v2 can also be implemented using any deterministic quantum-resistant key derivation scheme. This method allows a root server key to be stored on the HSM, with client-specific server keys derived from it. The resulting keys then can be used in the plugin to perform the wrapping without compromising the root key. Quantum resistance means that a quantum computer can not efficiently, i.e., in polynomial time, break the crypto primitive. Secure key storage must be available. Using key derivation would differ from the tls-crypt-v2 specifications and make drop-in replacement impossible. Further details are discussed in section 5.3. Support for AES-256, preferably the Counter (CTR) or Electronic Code Book (ECB) mode, and HMAC-SHA-256 would thus be optimal. If HMAC generation is unavailable, support for internal key hashing would be sufficient. Only performing both the cipher and authentication on the device allows the implementation to follow the specifications.

The HSM must not require interaction for key unwrapping, besides potential start-up authentication. Being readily available, inexpensive, and yet performant would be beneficial. Device libraries should be written in C or C++, but if not, a bridge for communicating between the library and the plugin would be required, complicating the implementation.

4.2. Smart Cards with Java Card

Overview

Java Card is a software technology used on smart cards to provide a Java-like development experience and broad compatibility across different smart cards. It contains a limited subset of the Java programming language, only including what is necessary for a smart card. These severe restrictions are due to the low processing power and small memory available on such a device. A program for a Java Card is called an applet and is compiled with the Java Card SDK. The app can then be uploaded onto a smart card running Java Card. From now on, JCard refers to smart cards supporting Java Card. A card can hold multiple applets concurrently, each selectable via the card management applet. Java Card makes the applet compatible with smart cards from different vendors. To achieve this, a Java Card Virtual Machine (JCVM) for executing byte code runs on the smart card [20]. On top of the JCVM are the Java Card Framework and the Application Programming Interface (API) for the applet to interact with, as well as potential vendor-specific extensions. Combined, it is the Java Card Runtime Environment (JCRE). The JCRE makes applets portable and provides a higher-level programming interface for easier development.

The most recent release of the technology, version 3.2, was published in January 2023 [21]. However, not every JCard supports every version. Furthermore, even though Java Card unifies the programming experience, not all specified cryptographic functions are available on a certain smart card. The JCard available for this thesis (NXP JCOP3 J3H145 [22]) does not support HMAC or AES-CTR, but AES-ECB and SHA-256. Even though a JCard is generally slower compared to more tightly integrated HSMs, they provide the advantage of being freely programmable, allowing the full implementation of tls-crypt-v2 on the card. JCards can be had for less than 40 Euros. They are also available with a USB interface. It is important to verify that the specific card supports AES-256 and SHA-256. Unfortunately, few shops sell (open) JCards, and of those that do, most of them are business-to-business only [23].

Programming with Java Card

Developing a Java Card applet requires the Java Card Development Kit (JCDK). It is essential to verify that the smart card supports the used JCDK. An applet compiled with an incompatible JCDK may not install on the JCard. Programming with Java Card is challenging since most of the features known from regular Java are unavailable.

Furthermore, even though the official documentation is decent, barely any guides with best practices on the topic of programming with Java Card are available. The programming language only supports bytes, shorts, and optionally ints as variable types. There are no strings or floating point numbers provided. One-dimensional arrays are supported. Besides the limited type support, Java Card also has no garbage collector [20].

There are three types of memory found on a JCard [24]:

- Read Only Memory (ROM) — This memory contains the smart card operating system, the JCRE, and pre-installed applets. It can only be modified during manufacturing. Writing data to it is called masking.
- Non-Volatile Memory (NVM) — This is persistent memory, which is retained even when disconnected from the card reader. It holds other applets and is also used to store class variables and by default objects. NVM is slow to write to and read from.
- Random Access Memory (RAM) — This memory is significantly faster than NVM, but also smaller. It is cleared when the card loses power. Since smart cards do not have their own power source, data saved in the RAM is lost, when the card is disconnected from the reader.

The lifetime of an applet is split into two phases. The first being "install" and the second "process". During installation, while the JCard is inserted into a card reader, the applet is loaded onto the smart card and performs setup tasks like allocating a large chunk of transient (RAM) memory and initializing cryptographic functions. The large chunk of memory is later reused for storing data when processing a command. It is faster to allocate memory once during installation than whenever a new instruction is processed since memory allocation is time-consuming. Should a requested cryptographic function be unavailable, the installation process will fail. Afterward, the applet is selectable with the card manager. Almost every command sent to the smart card, while an applet is selected, is forwarded to it. The applet then processes the command and sends the return data to the host. After completion, the applet is inactive until the next command arrives [24].

Deploying a Java Card Applet

After writing the applet, the source code is compiled into Java class files using the usual `javac` command. This process is the same as with a regular Java development workflow. The only difference is that the debug option must be added to the compilation for the converter[25]. The `converter` command is included in the JCDK and converts the class files into a CAP file. The final applet is then loaded onto the smart card. A build tool automates the process of compiling and converting the source code. For this project, ant-javacard is used, which provides a Java Card build task for the Java ant build system [26]. The upload to the smart card can also be automated using the tool.

A typical JCard contains a pre-installed root applet. This card management applet differs from a normal one by having more permissions, including the ability to install and select other applets. Virtually every Java Card root applet available is compatible with

Attribute	**Description**	**Length**	**Extended Length**
CLA	Class identifier	1 Byte	1 Byte
INS	Instruction identifier	1 Byte	1 Byte
P1, P2	Parameters	1 Byte	1 Byte
Lc	Length of data segment	1 Byte (optional)	3 Bytes (optional)
Data	Additional data send to applet	variable (optional, up to 255 Bytes)	variable (optional, up to 16 MB)
LE	Length of expected data returned by smart card	1 Byte (optional)	3 Byte (optional)

Table 1: Description of the fields of a regular APDU

the Global Platform (GP) specifications. GP defines the deployment and management procedures for applets on smart cards. By using a common standard, it is easier to communicate with different cards. To manage applets on a GP-compatible smart card, a tool such as Global Platform Pro (GPP) [27] is used. The root applet requires a key to modify the smart card, but for development purposes, it is set to the default one (`0x40..0x4F`). For actual deployment, this must be changed. All communication with a JCard happens via commands, called Application Protocol Data Units (APDU). APDUs are byte sequences sent to a smart card and used to communicate with the root and other applets. They are defined in *ISO/IEC 7816-4:2020* [28]. The basic structure of an APDU is `CLA | INS | P1 | P2 | Lc | Data | Le` , further described in table 1. A regular APDU supports up to 255 Bytes of additional data. Since it is sometimes required to send more data and splitting it into multiple chunks would be very inconvenient, extended APDUs were introduced. The Lc and Le fields of extended APDUs are three bytes instead of one, allowing transmissions of up to 16 MB of data if a card can store or process that much. APDUs come in four variations, depending on whether additional data needs to be sent and whether return data is expected. They are as follows:

1. `CLA | INS | P1 | P2`
   APDU for sending no data and expecting no return data

2. `CLA | INS | P1 | P2 | Le`
   APDU for sending no data and expecting return data

3. `CLA | INS | P1 | P2 | Lc | Data`
   APDU for sending data and expecting no return data

4. `CLA | INS | P1 | P2 | Lc | Data | Le`
   APDU for sending data and expecting return data

Global Platform Pro can send custom APDUs to an applet, which is useful for some basic testing and debugging. During the deployment of a smart card, communication is handled with a library. To transmit commands to a card the Personal Computer / Smart Card (PC/SC) specification is used. PC/SC defines the integration of smart cards with computers. Tools like GPP also utilize it in the background to communicate with a smart card. This thesis project employs the C library pcsclite, which acts as a "[m]iddleware to access a smart card using the SCard API (PC/SC)" [29]. Drivers for communication with the smart card reader are also required, but will not be further discussed.

4.3. YubiKey

YubiKeys are USB security tokens mainly used for (online) authentication providing functions like WebAuthn (the core of Fast Identity Online (FIDO)) and One Time Passwords (OTP) [30]. However, they can also be used as a smart card providing asymmetric cryptography, including elliptic curves. YubiKeys are manufactured by Yubico Inc. The most recent token is the Series 5 [31], which currently costs 50 €. It is one of the most popular consumer cryptographic tokens. Unfortunately, the YubiKey 5 does not expose the internal AES cipher, meaning tls-crypt-v2 can not be implemented as the specifications demand. Besides asymmetric cryptography, YubiKey 5 supports two challenge-response methods. One of them, HMAC-SHA1, can be repurposed as a key derivation function. The root server key is stored securely on the token and a client-specific server key is derived with some client information used as input for the challenge-response procedure. The plugin then performs the cryptographic functions. Since SHA1 is not based upon a mathematically complex problem but rather the scrambling of data, the current best approach to breaking it would be using the Grover algorithm with a quantum computer to reduce the effective security parameter (the key) from 160 to 80 Bits. HMAC-SHA1 can therefore be considered quantum-safe and utilized as a key derivation function. The National Institute of Standards and Technology (NIST) continues to approve the HMAC-SHA1 as secure in its latest recommendation for key management [32]. The token only supports a key length of 20 Bytes [33], which is less than the possible maximum of 64 Bytes, the internal block size of SHA1. Depending on the configuration of the plugin, this limitation effectively reduces the security parameter from two independent 32 Byte keys to one or two 20 Byte ones. Since YubiKeys are one of the most common and easily accessible cryptographic tokens for consumers, it is nevertheless a good candidate to be used as an HSM for tls-crypt-v2 for demonstration purposes.

Interacting with a YubiKey from a C/C++ program is challenging because there is no dedicated YubiKey C library as a standalone project available. However, the yubikey-personalization package for managing and interacting with the token is mostly written in C and contains a C library ykcore. The library is utilized as the backend to the various tools contained in the package and communicates with the token using the libusb library. The ykcore library can be extracted from the Git repository and integrated into other projects thanks to its BSD 2-Clause License. ykcore can also be found in other projects supporting YubiKeys, such as KeepassXC [34].

4.4. PKCS#11

The Public-Key Cryptography Standard 11 (PKCS#11) is a cryptographic standard specifying an API to interact with cryptographic tokens, such as performing cryptographic operations or managing keys. The most recent version 3.0 was released in June 2020 [35]. The standard contains five parts, including the base and current mechanism specification. The Organization for the Advancement of Structured Information Standards (OASIS), responsible for PKCS#11, also provides generic C Header files. Any manufacturer wanting to create a PKCS#11 library for their tokens can implement the provided interface.

The specification contains, among others, mechanisms for asymmetric and symmetric ciphers, as well as digests and HMAC calculations. All currently supported mechanisms are defined in *PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0* [36]. For symmetric cryptography, AES in different modes, including the Counter mode, is available as well as HMAC-SHA-256. Similar to Java Card, specified operations do not have to be implemented, making development more difficult. The `C_GetMechanismList()` function lists all available mechanisms for a specific device. Manufacturers have neglected symmetric cryptography since the tokens are most commonly used for authentication and signing, resulting in a lack of support. There appears to exist no affordable HSM with support for HMAC-SHA-256 and AES-256. The negligence of symmetric ciphers is also mirrored in the support for it in PKCS#11 helper libraries, providing a higher-level interface to interact with tokens.

The SoftHSM project provides a PKCS#11 library, which implements many of the current mechanisms in software [37]. The program does not provide any security benefit compared to the direct use of OpenSSL but helps to implement a program using the interface. SoftHSM supports both AES-256-CTR and HMAC-SHA-256.

4.5. SmartCard HSM

The SmartCard HSM is a Hardware Security Module in the form of a smart card developed by CardContact Systems GmbH [38]. The HSM is also available as a USB token and MicroSD card. According to the manufacturer's website, due to a lack of processing power, it only supports AES for key derivation, but not for encryption. HMAC support is not listed as a feature. OpenSC, a middleware providing a PKCS#11 library to developers for various cryptographic tokens, supports the token. Unfortunately, the integration is limited to only asymmetric cryptography. OpenSC does not provide access to AES. For further information regarding supported mechanisms, see appendix B. Besides the OpenSC integration, CardContact released its own lightweight PKCS#11 library sc-hsm-embedded intended for embedded systems. The library supports, apart from PC/SC, also the smaller Card Terminal API. It has a reduced set of asymmetric functionality. However, since version 2.1 it does support AES-CBC [39]. This feature was added with the support of SmartCard HSM 4, letting one assume that previous revisions of the card may not have exposed the AES capabilities. The product page may have not been updated to reflect the change. For all available mechanisms, see appendix B. The token does not support the import of secret keys using PKCS#11. The token's PKCS#11 implementation also does not support the import of secret keys.

Another product, which uses a SmartCard HSM chip at its core, is the NitroKey HSM 2. The NitroKey uses a chip with firmware version 3.5. The product page explicitly advertises the AES-CBC support [40]. HMAC generation is unsupported. Thus, some key derivation is required to implement the functionality of tls-crypt-v2, making it not that much different from a YubiKey. NitroKey claims support for internal and external hashing, however, neither OpenSC nor sc-hsm-embedded offer hashing on the token. The `C_DigestKey()` function described in the PKCS#11 specifications used for adding a stored key to a hash calculation is not available. No HMAC calculations can be implemented on the hardware. Only a single retailer, cardomatic.de [41], based in Germany, sells the SmartCard HSM. It costs around 40 Euros, similar to the YubiKey. A card reader is required, adding additional cost. At approximately 70 Euros, the USB token version is significantly more expensive. The NitroKey HSM 2 currently costs 99 Euros [40].

The advantages of the SmartCard HSM compared to the YubiKey are the support for symmetric key storage and ciphers. AES can be performed in hardware. Compared to the one or two 160 Bit keys stored in the YubiKey, the SmartCard HSM can hold two full 256-Bit keys. The speed of the current SmartCard HSM (hardware version: 24.13 and firmware version: 4.0) is decent. Encrypting a block of data takes around 14 ms, with each additional block around 1.2 ms. This is considerably faster than the NitroKey HSM 2, which needs 93 ms to encrypt the first block and 17 ms for further blocks. The SmartCard HSM only supports AES-CBC, meaning to implement AES-CTR, each block needs to be encrypted individually. The high static cost of the operation can lead to overall suboptimal performance. The performance tests were performed with sc-hsm-embedded version 2.12.

5. Implementation

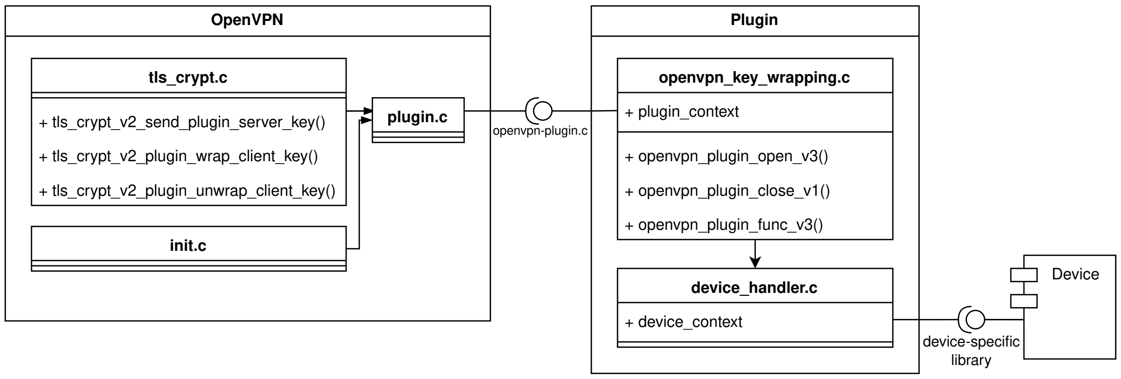

OpenVPN
tls_crypt.c
+ tls_crypt_v2_send_plugin_server_key()
+ tls_crypt_v2_plugin_wrap_client_key()
+ tls_crypt_v2_plugin_unwrap_client_key()
init.c
plugin.c
openvpn-plugin.c
Plugin
openvpn_key_wrapping.c
+ plugin_context
+ openvpn_plugin_open_v3()
+ openvpn_plugin_close_v1()
+ openvpn_plugin_func_v3()
device_handler.c
+ device_context
device-specific library
Device

Figure 2: Overview of general implementation.

The source code for the modified OpenVPN version and the plugins can be downloaded from the Git repository [42]. The project also contains a testing environment that allows experimenting with the various plugins. No formal security analysis has been performed with the OpenVPN patch or the individual plugins. It should not be used in a production environment but for research and testing.

Figure 2 is an implementation schematic of a generic tls-crypt-v2 plugin. The [init.c] file of OpenVPN issues the opening (loading) and closing (unloading) function calls for all plugins. [tls_crypt.c] handles wrapping, unwrapping, and server key import. Each plugin mainly consists of two parts. The `openvpn_key_wrapping.c` file is responsible for OpenVPN and the high-level execution of operations. The other, `device_handler.c`, communicates with the HSM using a device-specific library, executing the low(er)-level tasks.

5.1. Plugin Hook

As explained in section 2.3, the tls-crypt-v2 mechanism consists of three operations: server and client key generation, as well as unwrapping. The modifications only concern the server. Clients do not need to use the modified OpenVPN version. Before implementing the callback, the plugin interface in [18] has to be adapted by adding a new compile constant to the already existing ones (see listing 3) identifying the plugin hook. The change enables plugins to declare that they want to handle the wrapping mechanism.

```
#define OPENVPN_PLUGIN_CLIENT_KEY_WRAPPING      16
```

Implementing a plugin hook is poorly documented. Therefore, the thesis discusses the OpenVPN patch in greater detail than the other parts of the project.

Server Key Generation

A new tls-crypt-v2 server key is created with `$ openvpn --genkey tls-crypt-v2-server [key-file]`. The key file is optional and defaults to stdout. Key generation is separate from normal VPN operation and is handled in `do_genkey()` in [init.c]. The function then proceeds to call `tls_crypt_v2_write_server_key_file()`, which itself hands over the key generation to either OpenSSL or Mbed TLS. When a plugin is available, OpenVPN generates a random key and sends it to the extension. The key can be additionally written to a file and saved in a safe location for backup. The backup key can only work if the HSM supports secret key import.

First, the `do_genkey()` function needs to decide whether a plugin is available to handle the tls-crypt-v2 operation or not. Then the new `tls_crypt_v2_send_plugin_server_key()` is called. This function creates a key structure containing a random cipher and HMAC key. OpenVPN hands the key to the plugin via a command-line style list of type `char** argv`. The key is encoded into its base64 representation to eliminate `NULL` bytes. A preceding length advertisement of the raw key bytes could also have be used. Data exchange with the plugin is possible with the `struct argv` and `struct plugin_return` data types provided by [plugin.h]. OpenVPN has functions to encode base64 data.

For the key import, the plugin returns no data. Only the `struct argv` must therefor be populated using `argv_printf()`. The function is based on `printf()` from the C standard library. Like the original, it first generates a string based on the format and variables, but then splits the resulting text by white space into an `char **argv`. To signal the plugin what task needs to be performed, the first argument provided, is the type of work as string, either: “import”, “wrap” or “unwrap”. For key import, the second and third arguments are the AES and HMAC keys. The plugin call also requires the `struct plugin_list`, containing a list of all available plugins, and the `struct env_set` with key-value pairs of environmental variables set in OpenVPN.

After the plugin returns, depending on whether a file name is specified, the server key is written to a file. The code to call the plugin is in listing 1.

Listing 1: Excerpt from `tls_crypt_v2_send_plugin_server_key()` in [tls_crypt.c]. Error handling was removed for clarity.

```
openvpn_base64_encode(server_key.cipher,
                      server_kt.cipher_length,
                      &b64_aes_key);
openvpn_base64_encode(server_key.hmac,
                      server_kt.hmac_length,
                      &b64_hmac_key);
argv_printf(&av,
            "%s %s %s",
            "import",
            b64_aes_key,
            b64_hmac_key);

plugin_call(plugins,
            OPENVPN_PLUGIN_CLIENT_KEY_WRAPPING,
            &av,
            NULL,
            es);
```

Client Key Unwrapping

The client key unwrapping is the core of the modification and is executed whenever a client wants to establish a tunnel with the server or the server tests a newly generated client key. OpenVPN currently implements it in `tls_crypt_v2_unwrap_client_key()` in [tls_crypt.c]. The original unwrap implementation does not always have access to the required `struct plugin_list`, making modifications in a parent function in [mudp.c] necessary. The fix includes setting pointers to the plugin and environmental lists in `struct tls_options`. This structure is then available to the `tls_crypt_v2_extract_client_key()` function. Which can hand them to the unwrapping function. For code details, see appendix C.

Plugin unwrapping is in `tls_crypt_v2_plugin_unwrap_client_key()`. The plugin call is similar to the key generation procedure described in section 5.1. However, some return data, the unwrapped client key and metadata, are expected. The `struct plugin_return` contains a fixed length array of `struct openvpn_plugin_string_list`, which is a linked list containing key-value pairs. Since a plugin call can execute multiple plugins, each one gets a dedicated string list. The arguments provided to the plugin include the "unwrap" instruction and the base64 encoded wrapped client key. The `tls_crypt_v2_plugin_unwrap_client_key()` function uses a custom data structure, the `struct buffer`, with an internal pointer to the last read data. The code for preparing the plugin call is in listing 6.

Listing 2: Excerpt from `tls_crypt_v2_plugin_unwrap_client_key()` in [tls_crypt.c] containing the handling of return data from a plugin

```c
// Handle return
uint8_t plaintext_data[TLS_CRYPT_V2_MAX_WKC_LEN];
// [...]
struct plugin_return unwrapped_return;
plugin_return_get_column(&pr,
                         &unwrapped_return,
                         "wrapping result");
// [...]

for (int i = 0; i < unwrapped_return.n; ++i)
{
    if (unwrapped_return.list[i] &&
        unwrapped_return.list[i]->value)
    {
        char *b64_return = unwrapped_return.list[i]->value;
        int b64_len = (int) strlen(b64_return);
        int unwrapped_len = openvpn_base64_decode(b64_return,
                                                  plaintext_data,
                                                  b64_len);
        int expected_key_len = sizeof(client_key->keys);
        // [...]
        memcpy(&client_key->keys,
               plaintext_data,
               expected_key_len);
        if (!buf_write(metadata,
                       plaintext_data + expected_key_len,
                       unwrapped_len - expected_key_len))
        {
            CRYPT_ERROR("metadata too large for supplied
               buffer");
        }
        ret = true;
        break;
    }
}
```

After the plugin returns and the response is negative, the connection attempt is rejected. With the current implementation, OpenVPN does not know the reason for the failure. It could be a hardware problem like a disconnect of the token or a client-side problem, i.e., a wrong WK_c. The reason is only documented in the logs, making server-side error detection difficult. A possible solution could be adding a failure code to the reply.

On a successful plugin execution, the return data needs to be managed. To find a value with a specific key in the return, OpenVPN provides the function, `plugin_return_get_column()`. The returned list has the same number of elements as the input. If a `struct openvpn_plugin_string_list` did not contain the specified key, the list pointer is set to NULL. Thus OpenVPN still needs to iterate through the whole list until it finds a non-null element, which then contains the unwrapped client key plus metadata. This procedure seems rather complicated but is done so the return data of all the plugins called, can be stored in the same structure. After the proper return is located and decoded, the fixed length key at the beginning of the data block is copied into the client key structure with the remaining part moved into the metadata buffer. The code for handling the return is in listing 2.

Client Key Generation

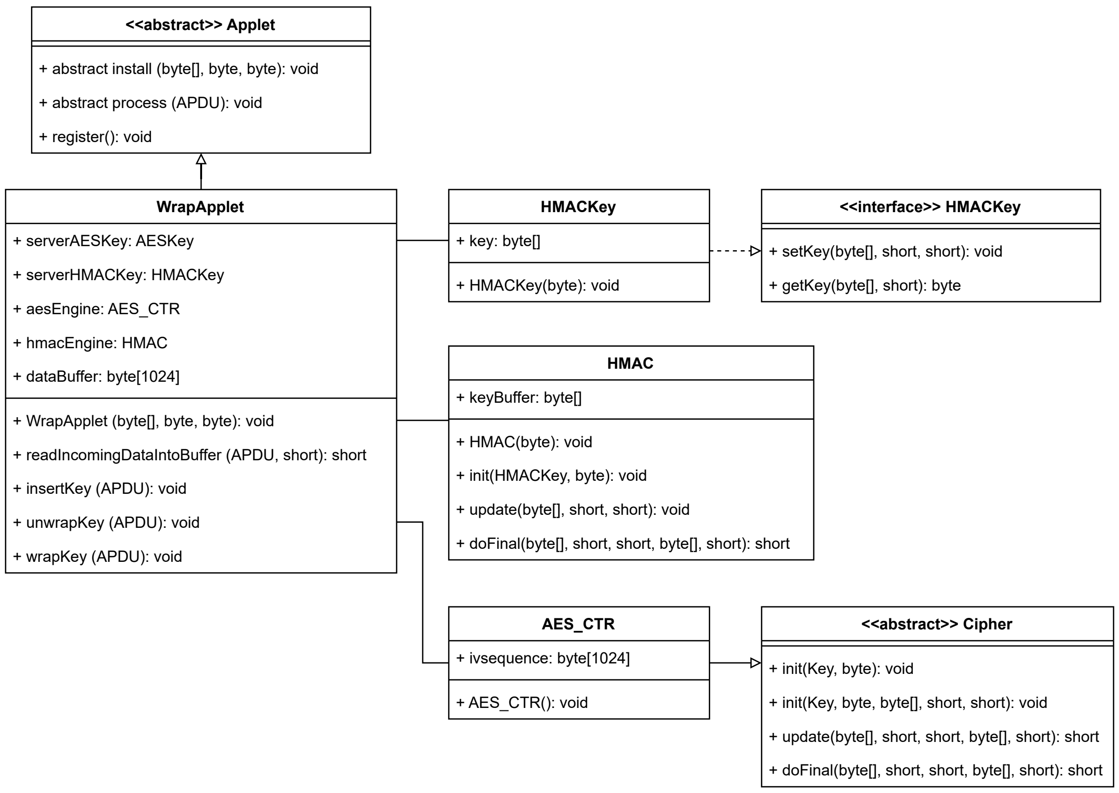
To add another client, a new tls-crypt-v2 client key has to be generated for it with `$ openvpn --tls-crypt-v2 <server_key> --genkey tls-crypt-v2-client [key-file]`. Creating a client key involves generating random data, K_c, and wrapping it using the server key. For this, the server key has to be loaded. However, if a plugin is available, this requirement is dropped since the plugin is responsible for the server key management. The same is applied to the unwrapping operation. The whole process, i.e., generating keys, adding metadata, loading the server key, wrapping, and preparing the output is managed by `tls_crypt_v2_write_client_key_file()`. The plugin list is inaccessible from the function, requiring a signature change. Inside, only key loading and wrapping have to be modified to accommodate the plugin's capabilities. OpenVPN hands over the wrapping to the plugin if a plugin is available and no server tls-crypt-v2 file is set. The program performs the plugin-based wrapping in `tls_crypt_v2_plugin_wrap_client_key()`. Since the operation is similar to unwrapping, only containing slight variations concerning error handling and input/output management, details are omitted.

5.2. Java Card

The integration of smart cards with Java Card consists of two parts. One is the applet on the smart card itself. The other, the plugin, acts as a bridge between OpenVPN and the card.

5.2.1. Applet

For this thesis, the NXP JCOP3 J3H145 is available. It uses the Java Card 3.0.4 platform. The device does not support AES-256-CTR or HMAC. However, AES-256-ECB and SHA-256 are available, thus the Counter mode and HMAC calculation can be efficiently implemented on the card. As a build tool, the project works with ant and ant-javacard [26]. To install and test the applet on the JCard, Global Platform Pro [27] is used. For better organization, the project is split into four parts as shown in fig. 3.



<<abstract>> Applet
+ abstract install (byte[], byte, byte): void
+ abstract process (APDU): void
+ register(): void
WrapApplet
+ serverAESKey: AESKey
+ serverHMACKey: HMACKey
+ aesEngine: AES_CTR
+ hmacEngine: HMAC
+ dataBuffer: byte[1024]
+ WrapApplet (byte[], byte, byte): void
+ readIncomingDataIntoBuffer (APDU, short): short
+ insertKey (APDU): void
+ unwrapKey (APDU): void
+ wrapKey (APDU): void
HMACKey
+ key: byte[]
+ HMACKey(byte): void
<<interface>> HMACKey
+ setKey(byte[], short, short): void
+ getKey(byte[], short): byte
HMAC
+ keyBuffer: byte[]
+ HMAC(byte): void
+ init(HMACKey, byte): void
+ update(byte[], short, short): void
+ doFinal(byte[], short, short, byte[], short): short
AES_CTR
+ ivsequence: byte[1024]
+ AES_CTR(): void
<<abstract>> Cipher
+ init(Key, byte): void
+ init(Key, byte, byte[], short, short): void
+ update(byte[], short, short, byte[], short): short
+ doFinal(byte[], short, short, byte[], short): short

Figure 3: Overview of the Java Card Applet. Some methods were omitted for clarity.

- `WrapApplet` — The main class for processing and executing all APDUs received.
- `AES_CTR` — An implementation of the Counter mode using AES-ECB. The class supports 128, 192, and 256-bit keys.
- `HMAC` — An implementation of HMAC. It can theoretically work with any digest. However, it is currently only implemented for SHA1 and SHA-256.
- `HMACKey` — A simple class implementing the HMACKey interface.

`WrapApplet` `WrapApplet` is the applet's main class. It extends the abstract `Applet` class, which provides two abstract methods, **`public static void`** `install()` and **`public void`** `process()`. Every Java Card applet must implement these.

The `install()` method is called when the applet is installed onto the card. `WrapApplet` instantiates all objects later needed, including an AES and HMAC key, as well as an `HMAC` and `AES_CTR` object. Another task is the allocation of a large chunk, one kibibyte, of RAM to later store the APDU data. The maximum size of WK_c is 1024 Bytes [12]. Finally, it calls `register()`, provided by the `Applet` class, to register itself with the Java Card Runtime Environment, enabling a user to select it. Example APDUs are at appendix C.

The JCRE calls the `process()` method of the applet when it is selected, and the runtime has received an APDU. After verifying the APDU for basic validity, depending on the instruction byte, the program executes a different method. It receives an `APDU` object as an argument, containing all data associated with the received command. The object furthermore provides methods to handle the transmission. Wrapped client keys are more than 255 Bytes, so extended APDUs as described in section 4.2 are used. Since the data buffer of the `APDU` has a limited capacity, the applet has to repeatedly copy the current data in the `APDU` buffer to the pre-allocated RAM array and then receive the next chunk until no data is left.

The server key import requires two separate APDUs each containing a single key. `P1` determines the key type. After reading the data from the APDU, either the `HMAC` or `AES` key object is set to the received key.

Here the thesis will focus on the unwrapping. After the WK_c is received, offsets for the tag, key, and len in the RAM buffer are calculated. The method then decrypts the key using the `AES_CTR` engine. With the decrypted key available, the tag can be computed using the `HMAC` object. Afterward, the tag is checked, and the resulting K_c is sent back. Should the check fail, an exception is thrown. Transmitting return data is accomplished with the *APDU* object. The direction needs to be set to outgoing, which allows sending data back to the host. Depending on the amount of transmitted data, different methods are available.

`HMACKey` `HMACKey` implements the key object close to the specifications at Oracle [46]. A `HMACKey` depends on the digest used. For example, an HMAC-SHA1 key has a different maximum size compared to an HMAC-SHA-256 key. The internal block size determines the upper limit. Whenever a key larger than the allowed limit is set, the `HMACKey` hashes it with the digest specified at the object initialization. However, it should not happen with the keys of tls-crypt-v2, since they are 32 Bytes long and within bounds.

`HMAC` `HMAC` performs HMAC calculations as defined in [47]. Java Card does not support HMAC. The class follows the design principle of the `MessageDigest` class with a constructor, `init()`, `update()`, and `doFinal()` methods. During its construction, the `HMAC` object creates a new digest object. Like the `HMACKey`, it supports SHA1 and SHA-256. The `init()` method sets the key and already adds the inner padded key to the message digest engine. The `update()` method adds more data to the inner digest calculation. The `doFinal()` method first finalizes the inner hash and then directly performs the outer hash using the already prepared outer padded key. After the operation is finished, the `HMAC` object must be reinitialized with the AES key for the next cycle. The code could also be modified to remember the inner and outer keys, saving some CPU time, but increasing the memory footprint.

`AES_CTR` `AES_CTR` implements the Counter mode as described by Dworkin [48] for AES by utilizing the available ECB mode. It is an extension of the `javacardx.crypto.Cipher` class. `AES_CTR` only supports one-shot operations. tls-crypt-v2 does not need the `update()`

method, and omitting it simplifies the implementation. The `init()` method sets the key for the AES-ECB instance. The `doFinal()` method is the core of this class, performing the encryption. The Counter mode has the convenient property that the procedure for the decryption is identical to the encryption. The in-place capabilities of the used AES-ECB cipher object can be kept. With the current implementation, the nonce (number used once) for every block is calculated and concatenated into a single array. The list can then be encrypted in a single operation. By avoiding block-by-block encryption, performance is drastically improved. The disadvantage of this approach is that a second same-sized block of memory as the data has to be available. This requirement can be problematic on a memory-constrained device such as a smart card. However, the card available for this project has enough memory for 1024 Bytes of WK_{c} and 1024 Bytes for the nonce sequence.

5.2.2. Plugin

The Smart Card Key Wrapping plugin is the bridge between OpenVPN and the JCard. The project is split into two files. `openvpn_smartcard_key_wrapping.c` is responsible for communicating, i.e., receiving and sending data to OpenVPN, as well as performing high-level tasks like key unwrapping. `smart_card_handler.c` handles the smart card with APDUs utilizing the pcsclite library [29]. The project is written in C and built with CMake. The project repository contains a small helper app, ImportKey, for importing an existing key file onto the JCard for a seamless transition.

`openvpn_smartcard_key_wrapping.c`

During `openvpn_plugin_open_v3()`, a connection with the JCard is established. The card connection context can later be used for communicating with the smart card. Furthermore, the applet is selected during plugin load and only unselected at unloading. If an error occurs while the plugin initializes, the OpenVPN start-up is aborted.

OpenVPN calls the `openvpn_plugin_func_v3()` during the runtime of the VPN. Depending on the type of action desired, the function then either calls `smartcard_import_key()` to transmit the keys to the JCard or `smartcard_process_key()` for wrapping and unwrapping. The code for the two operations is nearly identical and therefore consolidated into a single function. To (un)wrap a key, the function first creates a new APDU, including the operation-dependent return length. The plugin then sends the APDU to the smart card and checks the result. The response is valid if the status bytes are `0x90` and `0x00`. If the check passes, the return data is base64 encoded and copied into the return struct. Otherwise, if the library has returned no result because it enCountered a problem, the plugin tries to reconnect with the card and resend the APDU.

The `smartcard_import_key()` function transmits the two server keys in separate APDUs to the JCard. The plugin expects no return data. The applet does not require a passcode to overwrite the keys, meaning a malicious actor with access to the card could delete the server key. Therefore, a backup of the server key should be stored in a secure place. Protection using a PIN to prevent tampering remains future work.

`smart_card_handler.c`

The file handles the interaction with the smart card. `connectToCard()` establishes a connection with the JCard. To interact with a card reader and smart card, the plugin creates a pcsclite card context. This `SCARDCONTEXT` is stored in a sub-structure, the `struct CardConnectionContext`, of the plugin context. It is used to retrieve a list of available readers. The plugin assumes only one reader is connected, so the first entry in the list is taken to connect with the card already inserted into the reader. The resulting `SCARDHANDLE` and connection protocol, either T0 or T1, are saved to the `struct CardConnectionContext`.

The `selectApp()` and `unselectApp()` functions use APDUs defined by [49] with a hard-coded app ID specified in the applet project to select and unselect the program on the smart card. `sendAPDU()` is a simple function, only sending an APDU to the smart card. The APDU and response buffers are provided to the function.

5.3. YubiKey

The YubiKey plugin uses YubiKey's HMAC-SHA1 capabilities to derive client-specific server keys. The device does not provide or expose any AES or HMAC-SHA-256 functionality. Similar to the JCard plugin, the project is split into two parts: `openvpn_yubikey_key_management.c` and `yubikey_handler.c`. The first is responsible for implementing the different procedures including the cipher and authentication as well as data exchange with OpenVPN and the latter handles the communication with the YubiKey. For cryptographic operations, OpenSSL is used. To communicate with the YubiKey, the plugin utilizes the ykcore library and some other source files contained in the YubiKey Personalization project [50].

The idea of this implementation, as described in section 4.3 and fig. 4, is to have a root server key on the device and use client-specific information to derive server keys, which can then be exported from the YubiKey and do not compromise other server keys. A server breach could only compromise the server keys of clients which connect during the attack window. Thus, no more harm is done compared to other solutions, since an attacker will always have access to actively used client keys. Two keys for encryption and authentication have to be derived. The client-server AES key Ke_{c} is generated with the complete 32-byte-long tag of WK_{c}, T and the root server key Ks. The tag is either calculable or supplied before the encryption is performed. If the client were to supply a wrong tag, a wrong key would be derived, ultimately leading to a failed tag check. To derive the client-server HMAC key Ka_{c} the first 32 bytes of K_{c} and the Ks are used. The client key is either available during wrapping or can be first decrypted during the unwrapping of a WK_{c}.

$$\begin{aligned} Ke_{\mathrm{c}} &= \text{HMAC-SHA1}(K_{\mathrm{s}}, T) \\ Ka_{\mathrm{c}} &= \text{HMAC-SHA1}(K_{\mathrm{s}}, K_{\mathrm{c}}[0..32]) \end{aligned}$$

Two key derivations are required because, with a single derivation, either the tag or K_{c} would need to be always available during the start of both key wrapping and unwrapping. However, this is not the case since the two operations are performed in reverse.

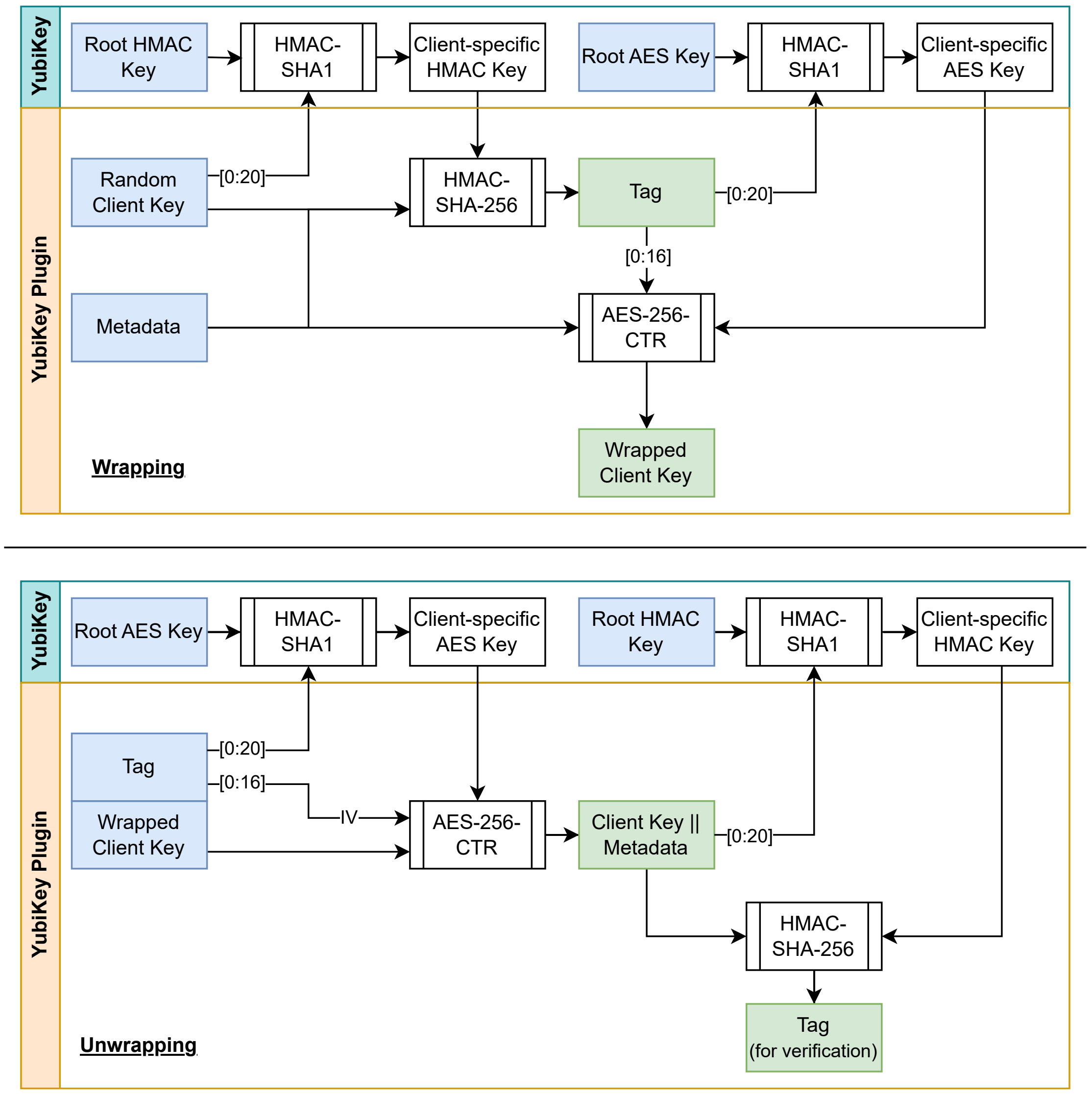


Figure 4: Overview of YubiKey TLS Crypt V2 implementation. The length component is omitted for improved clarity.

Instead of deriving Ka_{c} and then using it for HMAC-SHA-256, the output of the HMAC-SHA1 challenge could be used as the tag ($T = \text{HMAC-SHA-256}\,1(K_{\mathrm{s}}, K_{\mathrm{c}}[0..32])$), since applying the HMAC-SHA-256 does not significantly increase the security parameter of the implementation. The cipher can not so easily be replaced with a pure key derivation approach, because the metadata within the wrapped key should be encrypted. For improved security, both key slots on the YubiKey could theoretically be chained together to derive either a client-specific HMAC or AES key. To get two different purpose keys, applying individual strings like "Authentication" and "Encryption" with an XOR to the key derivation input could be used.

`openvpn_yubikey_key_management.c`

`openvpn_plugin_open_v3()` During load, the plugin establishes a connection with the YubiKey. The extension allows some customization through two arguments. Specifically, the slot and its access code. A YubiKey has two slots, with the second one being usually used for challenge-response procedures. In the configuration file, either slot one or two can be chosen, with an option to use both slots at the same time for separate root keys for cipher and authentication operations. The slots can be protected with an access code. The protection requires a passcode for modifying the configuration. With an access code, an adversary can not delete the root server key. If no backup was made, a loss of the root key would require the reissuing of all client keys. The code is only needed for generating the root key and should not be included in the regular OpenVPN configuration. The plugin indicates support for `OPENVPN_PLUGIN_CLIENT_KEY_WRAPPING` but also `OPENVPN_PLUGIN_UP`.

The ykcore library used by the plugin utilizes libusb in the background to communicate with the YubiKey. This library requires elevated privileges or access to the YubiKey device node via an udev rule. See [51]. The second option is advisable to follow the principle of least privilege. OpenVPN is usually launched as a root process to initialize its network interface. This circumstance allows libusb to work. However, OpenVPN can "down root", i.e., drop its privileges, after setting up the network. The plugin would lose access to the YubiKey device. This loss of permissions is a common problem for plugins and can be solved by spawning a background process before down root, allowing the plugin to retain the ability to talk with the token. The plugin performs a privilege separation during `OPENVPN_PLUGIN_UP`. The callback is reached early during OpenVPN start-up while OpenVPN still has root access. It is not reached during key generation. For these kinds of tasks, OpenVPN is usually started as a user process, which typically has access to all USB devices. The plugin also verifies that OpenVPN was compiled with OpenSSL since it will need the library as the crypto backend.

`OPENVPN_PLUGIN_UP` When OpenVPN reaches this callback, the plugin sets up the background process to retain elevated permissions. For bidirectional communication, it creates two pipes. After forking the main OpenVPN process, the parent stores the pipe file descriptors in the persistent plugin context and waits for a response from the background process. OpenVPN can become a daemon, which requires all child processes to follow along. The child, after being spawned, checks if that is desired and, if so, daemonizes. Afterward, it connects with the YubiKey. Then the background process writes a single success byte into its sending pipe. After reading the byte, the plugin knows the background process is ready and returns control to OpenVPN. The child now waits for a slot and fixed-length challenge from its receiving pipe. The use of fixed-length messages simplifies the exchange. If the background process should fail its `read()` command, it assumes the plugin is about to close, and exits.

Key Import The server key generation does not use the background process and has to first connect with the YubiKey. The same applies to the wrapping for client key generation. Both operations do not trigger the `OPENVPN_PLUGIN_UP` callback, which means no background process is available. `yubikey_handler.c` handles the import.

Key (Un)Wrapping After the usual base64 decoding and some verification, the library performs the tag and cipher operations. Both functions `calculate_cipher()` and `calculate_tag()` consist of two elements. In the first part, the client-specific key, Ke_{c} or Ka_{c}, is derived with the challenge-response procedure of the YubiKey using some client information as described in section 5.3. The second part consists of the cipher or authentication function with OpenSSL.

The entry function for the challenge is `do_challenge_response()`. Depending on the availability of the background process, an approach is chosen. For a local challenge, the `challenge_response()` function in `yubikey_handler.c` is called with the challenge text as an argument. For a background challenge, the plugin uses the pipe to first send the slot and then the fixed-length text, so the tag, or the first part of the client key, as raw bytes to the background process. The challenge needs to be 32 Bytes long since the background process expects this amount of data. After receiving the challenge, it calls `challenge_response()`. If the function should fail, the plugin makes a reconnection attempt with the YubiKey. It should be able to recover from the YubiKey being removed and reinserted. After receiving the fixed-length response from the YubiKey, the process sends it back to the parent process, which can then use the key to perform the crypto operation.

For AES-256-CTR, the EVP API of OpenSSL is used. The challenge response is 20 Bytes long and needs to be expanded to a 32-Byte-long AES key. The plugin pads the key with zero bytes. Adding password-hashing would improve resilience against an attacker with only knowledge of a client key, but HMAC-SHA1 and AES can be considered secure enough to justify not spending the additional compute cost. For authentication, OpenSSL's `HMAC()` function is used. The `CRYPTO_ECHECK()` macro is utilized for error handling. If a check does not pass, an error message is printed and the program jumps to a label for the cleanup, such as securely deleting the client key.

`yubikey_handler.c`

The actual challenge-response procedure in the `challenge_response()` function is straightforward and mostly handled by the ykcore library. This file is not concerned with the background process mechanisms and just provides its functionality to `openvpn_yubikey_key_management.c`. Connecting with a YubiKey, and creating a new context is managed by ykcore. Only setting up a new slot is more involved. The extension imports the server key in `import_server_key()`. For this, a new YubiKey configuration is created and then written to the token. The configuration contains the slot and key as well as if desired an access code. With ykpersonalize, the command would be `$ ykpersonalize [-c<curr_acc_code>] -<slot> -ochal-resp -ochal-hmac -a<server_key>`. For the server key, the plugin uses the first 20 Bytes of the AES key generated by OpenVPN. If both slots are used, the AES and HMAC key is used. When an access code should be used, it has to be already set for the slot. This can be done with `ykpersonalize <-1|-2> -oaccess=[new_acc_code]`. To remove the current slot configuration, including the access code, the following command can be used `$ ykpersonalize -c<curr_acc_code> <-1|-2> -z`.

5.4. PKCS#11

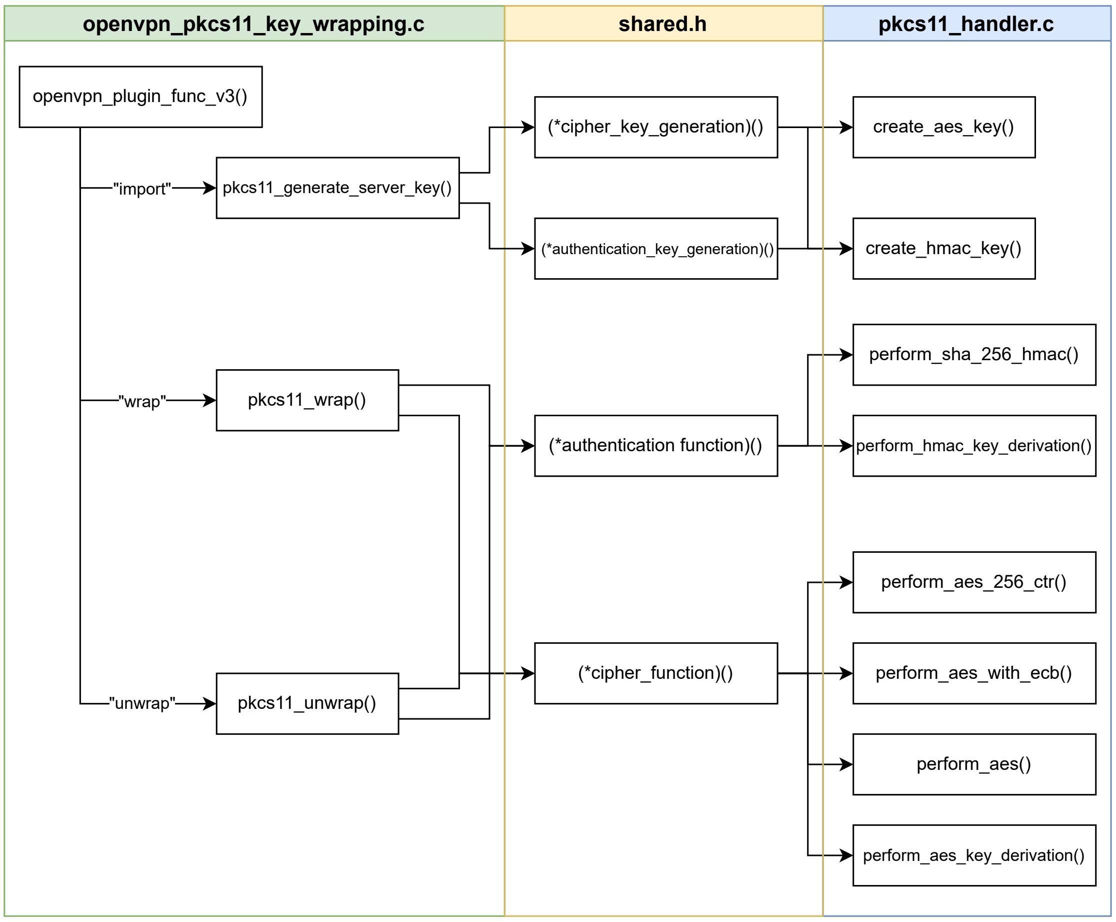


Figure 5: Overview of PKCS#11 plugin. Details are omitted for improved clarity. Execution is top to bottom. If multiple edges exit a node, only one path is followed.

The PKCS#11 plugin implements tls-crypt-v2 with tokens supporting PKCS#11. As discussed in section 4.4, the standard gives no guarantees about available mechanisms. This circumstance makes implementing a universal PKCS#11 plugin difficult. One could implement tls-crypt-v2 using only AES-CTR and SHA-256-HMAC, but in reality, almost no token supports these mechanisms, making the plugin useless. For example, the SmartCard-HSM only supports AES-CBC and no HMAC generation. For the implementation, multiple mechanisms defined by the specification were considered which can be used to implement the functionality of tls-crypt-v2. For encryption, the following are suitable.

- AES-256-CTR — The token supports `CKM_AES_CTR` with 256-bit keys, allowing the plugin to offload the cipher operation to the token completely.
- AES-256 — The token supports a different AES encryption mode with 256-bit keys. The available mechanism can then be used to implement AES-256-CTR with the token utilized for the actual cipher operation.
- (H)MAC or Digest with DigestKey — If AES is not available, a key derivation scheme, similar to the one used for the YubiKey, can be utilized.

For authentication, there are three different variations available.

- HMAC-SHA-256 — The token supports `SHA-256-HMAC`. Tag calculation can be solely handled by the device.
- SHA-256 with DigestKey — This token supports the SHA-256 digest and the `DigestKey()` functionality allowing internal keys to be hashed. This would enable the implementation of HMAC-SHA-256. However, it is uncommon and will therefore not be implemented.
- AES-256 or other (H)MAC — if the methods above are not available, either a symmetric cipher or a different MAC could be used for a client-server key derivation scheme like HMAC-SHA1 or even AES-CMAC. For encryption, 256-bit keys are desired for future-proofing.

More mechanisms can be repurposed, but are out of the scope of this thesis. Considering the options, the implementation is already complex enough. Similar to the other plugins, the project is split into two parts, `openvpn_pkcs11_key_wrapping.c`, and `pkcs11_handler.c`, with a third part containing some shared functions, structs, and variables. A broad overview of the project is given in fig. 5

Plugin Load During the opening of the plugin, a connection with the PKCS#11 Token is established and its capabilities determined. With PKCS#11, an application does not communicate with the token directly but through a PKCS#11 library, which can be either provided by the manufacturer like sc-hsm-embedded [39] for the SmartCard-HSM or developed independently with support for various tokens. An example is OpenSC [52]. The library is dynamically loaded during the start-up phase. The standard specifies `C_GetFunctionList()` to get a list of pointers to all functions. The plugin saves the list in its context. To perform a key operation, it is required to log in to the HSM with a user pin, which can be set in the OpenVPN configuration plugin arguments. This approach leaves the PIN vulnerable. An attacker would be able to read the PIN from the configuration file and use it to delete the keys saved on the token. To prevent this, the key could be set to non-destroyable and non-modifiable, but this would disable a legitimate user from deleting the object. Only a token reset could delete the key. Therefore, a different option was chosen where the user PIN is queried via the command line when the plugin is loaded, removing the need to store the PIN in the configuration file. Another approach for getting the PIN could be implemented using an environmental variable containing the secret. PKCS#11 does not seem to have the capability of protecting an object from modification using a passcode while still enabling read access without it.

The last step of the initialization queries the capabilities of the token. Depending on them, the extension selects a cipher and authentication function or rejects the device. It saves a pointer to the chosen ones in the plugin context as `int (*cipher_function)()` and `int (*authentication_function)()`. The capabilities are printed to stdout. With this technique, the cipher and authentication operation is transparent and abstracts the used mechanism away from the wrapping functions. The same approach is also utilized for key generation since, depending on whether a key derivation is used, a different type of key has to be created. The plugin does not implement privilege separation, which could be problematic for some PKCS#11 libraries.

Server Key Generation For server key generation, the plugin either loads the provided server keys onto the card or generates them on-device. Not all PKCS#11 tokens, including the SmartCard HSM [53], support the import of secret keys. This circumstance is problematic because the key returned by OpenVPN to the user is not the one on the token. If the token becomes inaccessible, a new server key and therefore new client keys would need to be issued. It is highly advisable to not use any tokens not supporting secret key import. The token identifies keys by a label, and old keys are deleted before new ones are generated.

Key (Un)Wrapping The main functions for key wrapping and unwrapping are minimalistic and similar to the YubiKey plugin. Both are provided with pointers to a cipher and authentication function. The actual operations depend on the capabilities of the token and are performed in `pkcs11_handler.c`. For encryption, the following four functions are available:

- `perform_aes_256_ctr()` — The function uses the `AES-CTR` PKCS#11 mechanism to perform the AES CTR encryption
- `perform_aes()` — This function encrypts the nonce for every block of data separately using an available AES mode. As previously discussed in the thesis, block-by-block encryption is suboptimal for performance, because the static cost associated with the operation can stack up when separately encrypting 32 or more blocks of data.
- `perform_aes_with_ecb()` — The function is similar, but an improvement to `perform_aes()`. The performance of the AES function can be significantly improved when the ECB mode is available. ECB allows the plugin to encrypt all nonces of the Counter Mode with one operation, incurring the static cost only once and not multiple times.

- `perform_aes_key_derivation()` — This function utilizes the HMAC-SHA-256 capabilities of the token for derivation of Ke_{c}. The extension then uses the key to perform AES-256-CTR in the plugin with OpenSSL. The function could be extended with support for other (H)MACs.

For authentication, only two functions are available. The first `perform_sha_256_hmac()` uses the `SHA-256-HMAC` mechanism of PKCS#11 to calculate the tag. The other `perform_hmac_key_derivation()` calculates the HMAC in OpenSSL with Ka_{c} as key. Ka_{c} is derived with AES and a symmetric root key stored on the token.

6. Performance

6.1. Methods

The analysis of the plugins includes the duration of an unwrap procedure in total and the cipher and authentication operation on the token. It is expected that the time for authentication plus encryption should be close to the overall duration, with the plugin only making up a small part. The performance of the server and client key generation will not be measured, because these operations are rarely executed and not time critical. For benchmarking, a small C++ project is used which loads a plugin and then executes it. For the YubiKey plugin, it will also trigger a `PLUGIN_UP` callback allowing the plugin to create a background process. The benchmarking tool provides a custom `plugin_vlog()` function and includes the `base64.c` file from OpenVPN for encoding and decoding the keys. To measure the time within the plugin for cipher and authentication, the POSIX function `clock_gettime()` with `CLOCK_MONOTONIC` is utilized. This clock is counting the time from an unspecified point in the past. For Linux, this is from the time of boot. The clock is guaranteed to never go backward and is not affected by jumps in the system time [54]. The clock source on the benchmarking computer used for `CLOCK_MONOTONIC` has a frequency of 3998.839 MHz and thus a period of 0.2501 ns. To measure the plugin execution duration within the benchmarking tool, CPP's `std::chrono::steady_clock` is employed. The tool is non-destructive, and a base64 encoded WK_{c} has to be set. A test run consists of dynamically loading the plugin library, calling the `openvpn_plugin_open_v3()` function and for YubiKey `openvpn_plugin_func_v3`() with `PLUGIN_UP` as callback type. If the set-up is successful, the pre-set WK_{c} is unwrapped 100 times, with a one-second sleep in between each execution to try to prevent cache speed-ups on the host computer. During each execution, the plugin measures the cipher and authentication duration and reports it to the benchmarking tool with `plugin_vlog()`.

The Java Card Plugin is tested with two JCards the NXP JCOP3 J3H145 and the NXP JCOP4 J3R180 [55]. The first was available during the whole thesis project, while the JCOP4 was only on-hand during the final stages of the writing process. The cards are connected to the reader wirelessly via ISO 14433. The JCOP3 is tested with both the memory-optimized solution, where the nonce is encrypted block-by-block, and with the current speed-optimized solution, which requires twice the memory, but encrypts the nonce sequence with a single encryption operation. The JCOP4 card is tested with the current Counter mode implementation and the available inbuilt AES-CTR implementation. While Java Card has introduced support for time measurements in version 3.1, the JCards available do not support this release. To measure the time for cipher and authentication operations on the smart cards, the benchmark was run three times: once without any cryptographic functions performed, then with only the AES engine or HMAC engine enabled. The base measurement enabled the determination of the approximate times for AES and HMAC.

The SmartCard-HSM (SCH) will be measured using the PKCS#11 plugin with the available AES-CBC and HMAC key derivation. Since the performance expected performance is suboptimal, another method was added for informative purposes. The new method utilized the AES on the card, not for encryption, but for the key derivation of both a Ke_c and Ka_c and performs the cryptographic operations in software. Two YubiKeys, the NEO and the 5C NFC, will be measured. The SoftHSM is not included, because it would not be used in an actual deployment, since it does not provide any additional security compared to just using the regular tls-crypt-v2 provided by OpenVPN.

6.2. Results

The results are plotted in fig. 6. Values have been rounded in this section. None of the tokens performed particularly well compared to the existing implementation in OpenVPN, taking from around 2'700 to 37'000 times as long. Even with the fastest device, the YubiKey 5C NFC, performing an unwrapping within 27 ms, only 37 operations can be performed per second or 2200 per minute allowing for 1100 connections per minute. The YubiKey NEO performs notably worse, taking three times as long as the 5C, with 81 ms on average to unwrap a key.

One of the slowest tokens is the JCOP3 JCard. When the applet encrypts each nonce separately, an operation takes 373 ms. Even with the single encryption optimization, the token still needs 201 ms. The JCOP3 JCard can thus only authenticate 150 clients per minute. Depending on the number of clients and the time-locality of their logins, this could already lead to longer connection times. However, for a small to medium-sized institution, this may work. For example, the HU-Berlin only has an average daily peak of 150 OpenVPN and 600 total sessions. The newer JCOP4 JCard performs notably better. With the custom AES-CTR implementation, unwrapping takes 93 ms, while the inbuilt AES Counter mode completes an operation within an average of 58 ms. The actual cipher only takes 8 ms, leaving the supporting code around it and the authentication (21 ms) with 29 ms as the largest contributor to the overall runtime. The "Other" category is the total time minus cipher and authentication. The Java Card has a significantly higher "Other" time since the measurements for the cryptographic operation are taken on the card instead of from within the plugin. With other plugins, the cryptographic device is a black box, thus the cipher and authentication times include the whole call to the HSM.

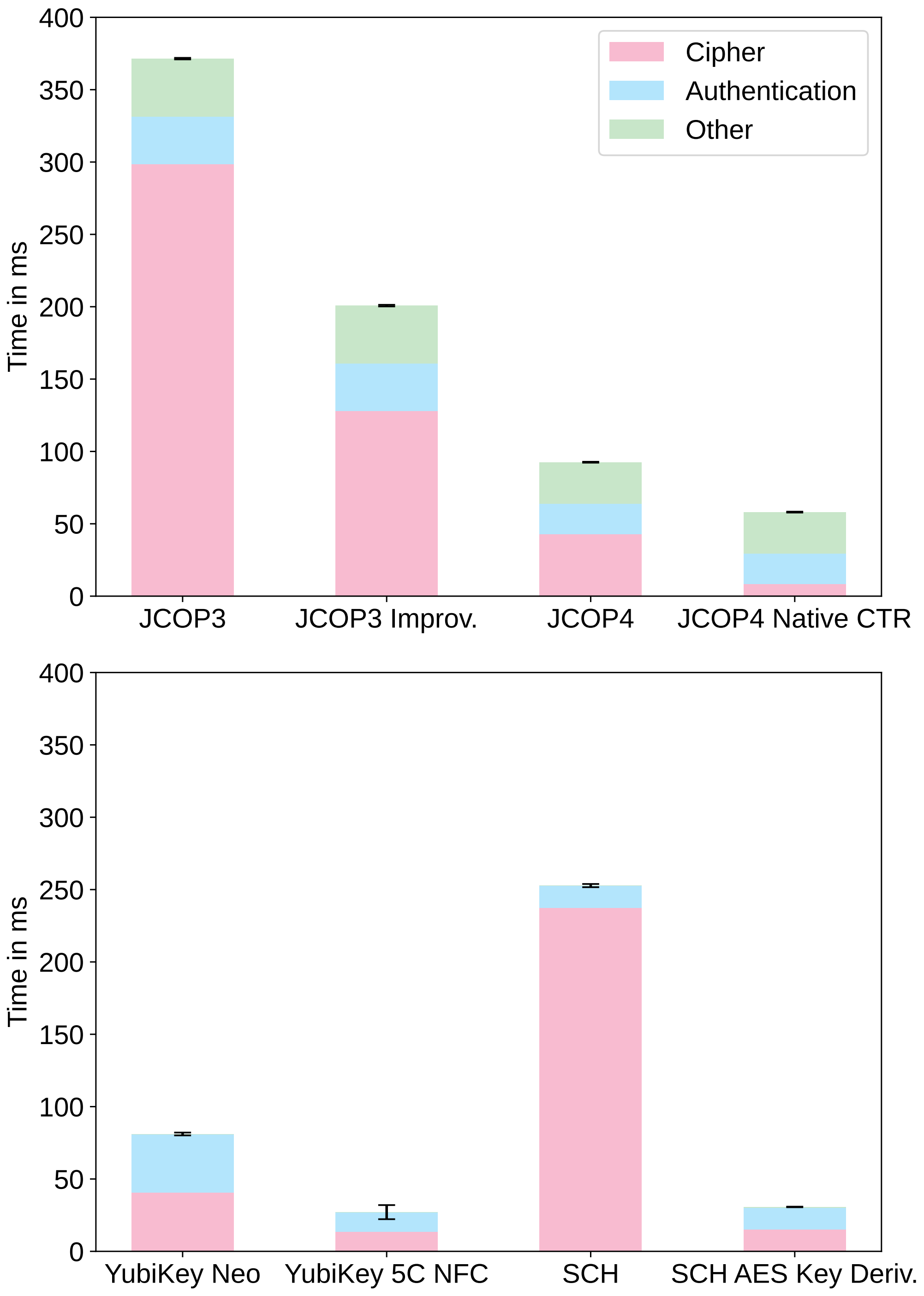


Figure 6: Mean unwrapping duration for each plugin, including 90% confidence interval. The YubiKeys have a reduced security parameter of 160 Bit compared to the regular 256 Bit. Both the YubiKeys and SCH do not conform to the tls-crypt-v2 specifications and can therefore not be used as drop-in replacements for existing setups. Data source at table 2

The SmartCard HSM has a decent encryption performance with a 12 ms static cost plus 1.2 ms to encrypt a block of data: $1.2x + 12.2$. Unfortunately, since the token only supports the CBC mode, the WK_c needs to be decrypted block-by-block, leading to 254 ms to unwrap a client key. If the SmartCard HSM supported AES-ECB, the decryption could be implemented with one operation taking around 21 ms, if the performance were to be comparable with the CBC mode. This fast speed is displayed when both the HMAC and AES server keys are derived using AES. With this mode, unwrapping only takes 30 ms. While the Nitrokey HSM 2 was benchmarked, it is not included in the plot. The token's symmetric capabilities are significantly worse with high static and high per-block encryption costs, resulting in a mean duration of 1.55 seconds for an operation.

6.3. Discussion

It is important to highlight, that even though the YubiKey 5C NFC is the fastest, it does provide a smaller security parameter of only 160 Bits and is thus not directly comparable to the JCards. The same applies to the YubiKey Neo. Both the YubiKey and the SCH are not compatible with a current OpenVPN setup, since they do not conform to the specifications of tls-crypt-v2. The results seem to indicate that the best HSM for implementing tls-crypt-v2 would be a performant JCard. Thanks to its ability to import existing server keys and compliance with the tls-crypt-v2 specification, it allows it to fit seamlessly into an existing set-up. However, a PKCS#11 token with AES-CTR or AES-ECB, HMAC-SHA-256 and support for secret key import is also a viable option, which even has the advantage of being easier to set up.

The results paint a bleak picture for any DOS protection provided by OpenVPN. Even though the regular OpenVPN implementation does not provide any additional DOS protection compared to an unprotected control channel, using a hardware token to handle tls-crypt-v2 reduces the DOS resilience to (almost) nothing. The plugins create a large attack surface for adversaries to shut down an OpenVPN instance. Since OpenVPN is a single-threaded application, data traffic can not be processed while it authenticates a client. This problem is mitigated with the latest OpenVPN release. Version 2.6 introduced the Data Channel Offload (DCO), which is a kernel module employed for handling data channel packages [56]. With DCO, only control channel traffic has to be processed by the user space OpenVPN application. Thus, a DOS attack should have a reduced impact against already connected clients, only affecting new connections.

To improve performance, parallelization may help. An implementation could consist of a dispatcher with a cache. When a new unwrap request is received by it, the cache is checked. If a miss occurs, the request is added to the work queue and the plugin response is deferred. OpenVPN can continue handling other requests and check back after a second, while multiple HSMs asynchronously work through the queue. This approach helps to reduce the cost of the double unwrap. It could be implemented using a meta plugin, which forks a new plugin instance for each HSM.

Otherwise, performance on a smart card or affordable HSM is simply limited. Much faster, business-grade solutions exist for smart cards and HSMs, such as the YubiKey HSM 2, able to create an HMAC-SHA-256 within 4 ms [57] or the Thales Luna PCIe

HSM, which performs up to 20’000 AES-GCM transactions per second [58]. However, at a price of several hundred to thousands of Euros, they are not a viable option for small-scale or private usage but could be interesting for large organizations.

7. Conclusion

This bachelor thesis explored ways of implementing the functionality of OpenVPN’s control channel protection tls-crypt-v2 with different hardware tokens. To add the capability of using hardware tokens for tls-crypt-v2, OpenVPN’s plugin mechanism was extended to support the mechanism.

For hardware tokens, three different approaches were showcased. The first employed the Java Card technology. A custom applet was created to implement tls-crypt-v2 on a JCard, enabling it to be used as a drop-in replacement for current setups. Testing showed large performance varieties between different JCards highlighting the importance of choosing an appropriately performant card for one’s use case.

While YubiKey’s wide availability, fast speed, and key protection make it a promising device, the lack of AES and the alternative key derivation scheme, means the implementation is incompatible with regular tls-crypt-v2. Furthermore, the reduced security parameter of 160-bit discourages its use.

The PKCS#11 plugin combines the ease of setup of the YubiKey with the wide range of supported tokens. The plugin enables a variety of mechanisms to perform tls-crypt-v2. Problematic is that not all tokens support key import, including the PKCS#11 libraries of the SmartCard HSM. If the key has to be generated on-device, backups are limited or impossible.

The biggest shortcoming of using an HSM to improve the security of the control channel protection of OpenVPN is the drastically reduced performance and thus increase in DOS attack surface for adversaries for connection establishment. Due to the single-threaded nature of the VPN, lengthy unwrapping can impact both the control and data traffic of all clients. This problem can be somewhat remedied by using the new Data Channel Offload introduced in OpenVPN 2.6.

Overall, HSMs can be considered a viable solution in improving the security of the tls-crypt-v2 mechanism as shown in this paper, but it is important to be aware of the reduced performance and the cost of a powerful enough and feature-rich HSM.

A. Plugin Interface Specification

Available Plugin Hooks

Listing 3: Available callbacks as of OpenVPN release/2.6

```
#define OPENVPN_PLUGIN_UP                        0
#define OPENVPN_PLUGIN_DOWN                      1
#define OPENVPN_PLUGIN_ROUTE_UP                  2
#define OPENVPN_PLUGIN_IPCHANGE                  3
#define OPENVPN_PLUGIN_TLS_VERIFY                4
#define OPENVPN_PLUGIN_AUTH_USER_PASS_VERIFY     5
#define OPENVPN_PLUGIN_CLIENT_CONNECT            6
#define OPENVPN_PLUGIN_CLIENT_DISCONNECT         7
#define OPENVPN_PLUGIN_LEARN_ADDRESS             8
#define OPENVPN_PLUGIN_CLIENT_CONNECT_V2         9
#define OPENVPN_PLUGIN_TLS_FINAL                10
/*#define OPENVPN_PLUGIN_ENABLE_PF                11  *REMOVED
   FEATURE* */
#define OPENVPN_PLUGIN_ROUTE_PREDOWN            12
#define OPENVPN_PLUGIN_CLIENT_CONNECT_DEFER     13
#define OPENVPN_PLUGIN_CLIENT_CONNECT_DEFER_V2  14
#define OPENVPN_PLUGIN_CLIENT_CRRESPONSE        15
#define OPENVPN_PLUGIN_N                        16
```

B. Supported Mechanisms

SmartCard HSM with OpenSC

Available Mechanisms with OpenSC (version 0.23.0-1) and a SmartCard HSM (hardware version: 24.13 and firmware version: 4.0)

```
[~] $ pkcs11-tool --module /usr/lib/pkcs11/opensc-pkcs11.so -M
Using slot 0 with a present token (0x0)
Supported mechanisms:
  SHA-1, digest
  SHA224, digest
  SHA256, digest
  SHA384, digest
  SHA512, digest
  MD5, digest
  RIPEMD160, digest
  GOSTR3411, digest
  ECDSA, keySize={192,521}, hw, sign, verify, EC F_P, EC
     parameters, EC OID, EC uncompressed
  ECDSA-SHA384, keySize={192,521}, sign, verify
  ECDSA-SHA512, keySize={192,521}, sign, verify
  ECDSA-SHA1, keySize={192,521}, hw, sign, verify, EC F_P, EC
     parameters, EC OID, EC uncompressed
  ECDSA-SHA224, keySize={192,521}, hw, sign, verify, EC F_P, EC
     parameters, EC OID, EC uncompressed
  ECDSA-SHA256, keySize={192,521}, hw, sign, verify, EC F_P, EC
     parameters, EC OID, EC uncompressed
  ECDH1-COFACTOR-DERIVE, keySize={192,521}, hw, derive, EC F_P,
     EC parameters, EC OID, EC uncompressed
  ECDH1-DERIVE, keySize={192,521}, hw, derive, EC F_P, EC
     parameters, EC OID, EC uncompressed
  ECDSA-KEY-PAIR-GEN, keySize={192,521}, hw, generate_key_pair,
     EC F_P, EC parameters, EC OID, EC uncompressed
  RSA-X-509, keySize={1024,4096}, hw, decrypt, sign, verify
  RSA-PKCS, keySize={1024,4096}, hw, decrypt, sign, verify
  SHA1-RSA-PKCS, keySize={1024,4096}, sign, verify
  SHA224-RSA-PKCS, keySize={1024,4096}, sign, verify
  SHA256-RSA-PKCS, keySize={1024,4096}, sign, verify
  SHA384-RSA-PKCS, keySize={1024,4096}, sign, verify
  SHA512-RSA-PKCS, keySize={1024,4096}, sign, verify
  MD5-RSA-PKCS, keySize={1024,4096}, sign, verify
  RIPEMD160-RSA-PKCS, keySize={1024,4096}, sign, verify
  RSA-PKCS-PSS, keySize={1024,4096}, hw, sign, verify
  SHA1-RSA-PKCS-PSS, keySize={1024,4096}, sign, verify
  SHA224-RSA-PKCS-PSS, keySize={1024,4096}, sign, verify
  SHA256-RSA-PKCS-PSS, keySize={1024,4096}, sign, verify
  SHA384-RSA-PKCS-PSS, keySize={1024,4096}, sign, verify
  SHA512-RSA-PKCS-PSS, keySize={1024,4096}, sign, verify
  RSA-PKCS-OAEP, keySize={1024,4096}, hw, decrypt
  RSA-PKCS-KEY-PAIR-GEN, keySize={1024,4096}, generate_key_pair
```

SmartCardHSM with sc-hsm-embedded

Available Mechanism with sc-hsm-embedded (version V2.12) and a SmartCard HSM (hardware version: 24.13 and firmware version: 4.0)

```
[sc-hsm-embedded] $ pkcs11-tool --module
   ./src/pkcs11/.libs/libsc-hsm-pkcs11.so  -M
Using slot 0 with a present token (0x1)
Supported mechanisms:
  RSA-X-509, keySize={1024,4096}, hw, encrypt, decrypt, sign,
     verify
  RSA-PKCS, keySize={1024,4096}, hw, encrypt, decrypt, sign,
     verify
  RSA-PKCS-PSS, keySize={1024,4096}, hw, sign, verify
  SHA1-RSA-PKCS, keySize={1024,4096}, hw, sign, verify
  SHA256-RSA-PKCS, keySize={1024,4096}, hw, sign, verify
  SHA1-RSA-PKCS-PSS, keySize={1024,4096}, hw, sign, verify
  SHA256-RSA-PKCS-PSS, keySize={1024,4096}, hw, sign, verify
  ECDSA, keySize={192,521}, hw, sign, verify
  ECDSA-SHA1, keySize={192,521}, hw, sign, verify
  AES-CBC, keySize={128,256}, hw, encrypt, decrypt
  AES-CMAC, keySize={128,256}, hw, sign
  RSA-PKCS-OAEP, keySize={1024,4096}, hw, encrypt, decrypt
  SHA-1, digest
  SHA224, digest
  SHA256, digest
  SHA384, digest
  SHA512, digest
  ECDSA-KEY-PAIR-GEN, keySize={192,521}, hw, generate_key_pair
  RSA-PKCS-KEY-PAIR-GEN, keySize={1024,4096}, hw,
     generate_key_pair
  AES-KEY-GEN, keySize={128,256}, hw, generate
  mechtype-0x80000001, keySize={1024,4096}, hw, sign, verify
  mechtype-0x80000003, keySize={1024,4096}, hw, sign, verify
  mechtype-0x80000010, keySize={192,521}, hw, sign, verify
  mechtype-0x80000011, keySize={192,521}, hw, sign, verify
```

C. Code Snippets

Plugin Hook Code

- I added a dummy `struct tls_options` data structure to `do_pre_decrypt_check()`, which is the last function having access to the plugin list. This function then calls `tls_pre_decrypt_lite()`. I had to change `tls_pre_decrypt_lite()` from calling `read_control_auth()` with a NULL pointer to a pointer to the dummy `struct tls_options` structure.

- `read_control_auth()` then calls `tls_crypt_v2_extract_client_key()` if the client indicates it wants to use –tls-crypt-v2.

Listing 4: `mudp.c`

```
static bool
do_pre_decrypt_check(struct multi_context *m,
                     struct tls_pre_decrypt_state *state,
                     struct mroute_addr addr)
    // ...
+   struct tls_options dummy_temp_opts = { 0 };
+   dummy_temp_opts.plugins = m->top.plugins;
+   dummy_temp_opts.es = m->top.es;
    verdict = tls_pre_decrypt_lite(tas, state, &m->top.c2.from,
-                                  &m->top.c2.buf);
+                                  &m->top.c2.buf,
+                                  &dummy_temp_opts);
    // ...
```

Listing 5: `ssl.c`

```
bool
tls_pre_decrypt_lite(const struct tls_auth_standalone *tas,
                     const struct link_socket_actual *from,
-                    const struct buffer *buf)
+                    const struct buffer *buf,
+                    struct tls_options *opts)
    // ...
   bool status = read_control_auth(&state->newbuf,
                                   &state->tls_wrap_tmp,
-                                  from, NULL);
+                                  from, opts);
    // ...
```

Listing 6: Excerpt from tls_crypt_v2_plugin_unwrap_client_key() in [12] containing the preperation for the plugin call. Slightly modified for better code formatting

```
static bool
tls_crypt_v2_plugin_unwrap_client_key(struct key2 *client_key,
    struct buffer *metadata,
                                                           struct buffer
                                                              wrapped_client_key,
                                                           const struct plugin_list
                                                              *plugins, struct
                                                              env_set *es)
{
    // [...]
    // Prepare unwrapped key for plugin
    struct argv av = argv_new();
    char *b64_key;
    int plug_ret;
    ASSERT(openvpn_base64_encode(BPTR(&wrapped_client_key),
                                 BLEN(&wrapped_client_key),
                                 &b64_key)
           >= 0);
    ASSERT(argv_printf(&av, "%s %s", "unwrap", b64_key)
           == true);
    free(b64_key);

    // Prepare response structure
    struct plugin_return pr;
    plugin_return_init(&pr);

    // Call the plugin
    plug_ret = plugin_call(plugins,
                           OPENVPN_PLUGIN_CLIENT_KEY_WRAPPING,
                           &av,
                           &pr,
                           es);
```

Applet Code

```
// CLA = 0xA0
// Key Import (INS = 0x01)
// AES Key (param1=0x01)
A0 01 01 00 20 <AES key hex encoded>
// HMAC Key (param1=0x02)
A0 01 02 00 20 <HMAC key hex encoded>

// Unwrap APDU (INS = 0x02)
// The length and return length need to be adjusted for the
   individual WKc, these are default values
A0 02 00 00 00012B <WKc hex encoded> 0109

// Wrap APDU (INS = 0x03)
A0 03 00 00 000109 <Kc + metadata hex encoded> 012B
```

Listing 7: `private short readIncomingDataIntoBuffer(APDU apdu, short bufferOffset)`

```
private short readIncomingDataIntoBuffer(APDU apdu,
                                         short bufferOffset) {
    byte[] apduBuffer = apdu.getBuffer();
    short offsetCData = apdu.getOffsetCdata();
    short bytesLeft = apdu.getIncomingLength();
    short readCount = apdu.setIncomingAndReceive();
    short bufferPosition = bufferOffset;
    while(bytesLeft > 0) {
        Util.arrayCopyNonAtomic(apduBuffer,
                                offsetCData,
                                dataBuffer,
                                bufferPosition,
                                readCount);
        bytesLeft -= readCount;
        bufferPosition += readCount;
        readCount = apdu.receiveBytes(offsetCData);
    }

    return (short) (bufferPosition - bufferOffset);
}
```

Listing 8: `private void unwrap_key(APDU apdu)`

```
private void unwrap_key(APDU apdu) {
    short bytesReceived = readIncomingDataIntoBuffer(apdu,
                                                     (short) 0);

    short ciphertextLength = (short) (bytesReceived -
       TAG_LENGTH - LEN_LENGTH);

    short tagOff = 0;
    short wrapOff = (short) (tagOff + TAG_LENGTH);
    short lenOff = (short) (wrapOff + ciphertextLength);

    short wrappedKeyLength = Util.makeShort(dataBuffer[lenOff],
       dataBuffer[(short)(lenOff+1)]);

    if (wrappedKeyLength != bytesReceived) {
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    }

    aesEngine.init(serverAESKey,
                   Cipher.MODE_DECRYPT,
                   dataBuffer,
                   tagOff,
                   AES_CTR.IV_SIZE);
    short bytesProcessed = aesEngine.doFinal(dataBuffer,
                                             wrapOff,
                                             ciphertextLength,
                                             dataBuffer,
                                             wrapOff);
    if(bytesProcessed != ciphertextLength) {
        throw new
           CryptoException(CryptoException.ILLEGAL_VALUE);
    }

    hmacEngine.init(serverHMACKey, MessageDigest.ALG_SHA_256);
    hmacEngine.update(dataBuffer, lenOff, LEN_LENGTH);
    hmacEngine.doFinal(dataBuffer, wrapOff, bytesProcessed,
       apdu.getBuffer(), apdu.getOffsetCdata());

    if (Util.arrayCompare(dataBuffer,
                          tagOff,
                          apdu.getBuffer(),
                          apdu.getOffsetCdata(),
                          TAG_LENGTH)
        != 0) {
        ISOException.throwIt(ISO7816.SW_DATA_INVALID);
    }
    short le = apdu.setOutgoing();

    if (le != ciphertextLength) {
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    }
    apdu.setOutgoingLength(ciphertextLength);

    apdu.sendBytesLong(dataBuffer, wrapOff, ciphertextLength);
}
```

D. Result Data

Device	Mean	Cipher	Auth	Other	90% CI	SD
JCOP3	373.22	298.67	32.92	39.87	0.53	0.17
JCOP3 Improv.	201.04	128.01	32.88	39.88	0.56	0.18
JCOP4	92.69	42.90	20.97	28.67	0.31	0.09
JCOP4 Native CTR	58.33	8.44	20.94	28.67	0.31	0.09
YubiKey NEO	80.62	40.51	40.39	0.20	0.99	0.31
YubiKey 5C NFC	27.01	13.57	13.32	0.20	4.87	1.53
SCH	253.69	237.42	15.20	0.14	1.11	0.35
SCH AES Key Deriv.	30.50	15.12	15.22	0.34	0.20	0.14

Table 2: Results of benchmark for unwrapping. Mean, Cipher, Auth and Other in ms.

Bibliography

Software

2. [SW exc.] OpenVPN Inc, “crypto_openssl.c”, from *OpenVPN* version release/2.6, 2023. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.6.0/src/openvpn/crypto_openssl.c.
3. [SW exc.] OpenVPN Inc, “crypto_mbedtls.c”, from *OpenVPN* version release/2.6, 2023. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.6.0/src/openvpn/crypto_mbedtls.c.
7. [SW exc.] OpenVPN Inc, “tls_crypt.h”, from *OpenVPN* version release/2.6, 2023. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.6.0/src/openvpn/tls_crypt.h.
8. [SW exc.] OpenVPN Inc, “ssl.c”, from *OpenVPN* version release/2.6, 2023. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.6.0/src/openvpn/ssl.c.
10. [SW exc.] OpenVPN Inc, “options.c”, from *OpenVPN* version release/2.6, 2023. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.6.0/src/openvpn/options.c.
11. [SW Rel.] OpenVPN Inc, *OpenVPN* version release/1.0.2, 2002. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn2-historical-cvs/tree/9c59b6126949ca1946dd3fb526bdf55df4b8ed0f.
12. [SW exc.] OpenVPN Inc, “tls_crypt.c”, from *OpenVPN* version release/2.6, 2023. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.6.0/src/openvpn/tls_crypt.c.
13. [SW exc.] OpenVPN Inc, “ChangeLog”, from *OpenVPN* version release/2.4, 2016. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.4.0/ChangeLog.
14. [SW exc.] OpenVPN Inc, “ChangeLog”, from *OpenVPN* version release/2.5, 2020. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.5.0/ChangeLog.
15. [SW exc.] OpenVPN Inc, “tls-crypt-v2.txt”, from *OpenVPN* version release/2.6, 2023. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.6.0/doc/tls-crypt-v2.txt.
17. [SW exc.] OpenVPN Inc, “ChangeLog”, from *OpenVPN* version release/2.0, 2005. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn2-historical-cvs/tree/openvpn-2-0/ChangeLog.
18. [SW exc.] OpenVPN Inc, “openvpn-plugin.h.in”, from *OpenVPN* version release/2.6, 2023. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.6.0/include/openvpn-plugin.h.in.

26. [SW] Paljak, M., *ant-javacard* 2015. URL: https://github.com/martinpaljak/ant-javacard.
27. [SW] Paljak, M., *GlobalPlatformPro* URL: https://javacard.pro/globalplatform/.
29. [SW] David Corcoran, L. R., *PCSC lite project* URL: https://pcsclite.apdu.fr/.
34. [SW Rel.] KeePassXC team, *KeePassXC* version release/2.7.4, 2022. URL: https://github.com/keepassxreboot/keepassxc/tree/2.7.4.
37. [SW] OpenDNSSEC, *SoftHSM* version 2. URL: https://github.com/opendnssec/SoftHSMv2.
39. [SW] CardContact Systems GmbH, *sc-hsm-embedded* URL: https://github.com/CardContact/sc-hsm-embedded.
42. [SW] Ehlert, E., *OpenVPN TLS Crypt V2 Plugins* 2022. URL: https://gitlab.informatik.hu-berlin.de/ehlertto/openvpn-tls-crypt-v2-plugins/-/tree/811dbc295ef7622afee001ee6d3285553e82bfe1.
43. [SW exc.] OpenVPN Inc, "init.c", from *OpenVPN* version release/2.6, 2023. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.6.0/src/openvpn/init.c.
44. [SW exc.] OpenVPN Inc, "plugin.h", from *OpenVPN* version release/2.6, 2023. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.6.0/src/openvpn/plugin.h.
45. [SW exc.] OpenVPN Inc, "mudp.c", from *OpenVPN* version release/2.6, 2023. OpenVPN Inc. URL: https://github.com/OpenVPN/openvpn/tree/v2.6.0/src/openvpn/mudp.c.
50. [SW], *Yubikey Personalization* Yubico Inc. URL: https://github.com/Yubico/yubikey-personalization.
52. [SW] OpenSC contributors, *OpenSC* URL: https://github.com/OpenSC/OpenSC.
53. [SW exc.] CardContact Systems GmbH, "token-sc-hsm.c", from *sc-hsm-embedded* version release/2.12, 2022. URL: https://github.com/CardContact/sc-hsm-embedded/tree/V2.12/src/pkcs11/token-sc-hsm.c.

References

1. *Overview of OpenVPN — OpenVPN Community Wiki* OpenVPN. https://community.openvpn.net/openvpn/wiki/OverviewOfOpenvpn, archived at https://web.archive.org/web/20220707113857/https://community.openvpn.net/openvpn/wiki/OverviewOfOpenvpn on July 7, 2022.
4. Yonan, J. *Reference manual for OpenVPN 2.6* (OpenVPN Inc). https://openvpn.net/community-resources/reference-manual-for-openvpn-2-6/ (Feb. 6, 2023).

5. Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.3* RFC 8446 (RFC Editor, Aug. 2018). https://www.rfc-editor.org/rfc/rfc8446.txt.

6. *Security Overview — OpenVPN Community Wiki* OpenVPN. https://community.openvpn.net/openvpn/wiki/SecurityOverview, archived at https://web.archive.org/web/20220710033757/https://community.openvpn.net/openvpn/wiki/SecurityOverview on July 10, 2022.

9. Streun, F., Wanner, J. & Perrig, A. *Evaluating Susceptibility of VPN Implementations to DoS Attacks Using Adversarial Testing* in *Proceedings 2022 Network and Distributed System Security Symposium* (Internet Society, 2022). https://doi.org/10.14722%2Fndss.2022.24043.

16. Rogaway, P. & Shrimpton, T. *A Provable-Security Treatment of the Key-Wrap Problem* in *Advances in Cryptology - EUROCRYPT 2006* (ed Vaudenay, S.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006), 373–390. ISBN: 978-3-540-34547-3.

19. Yang, Z., Johannesmeyer, B., Olesen, A. T., Lerner, S. & Levchenko, K. *Dead store elimination (still) considered harmful* in *26th USENIX Security Symposium (USENIX Security 17)* (2017), 1025–1040.

20. Ortiz, C. E. *An Introduction to Java Card Technology - Part 1* https://www.oracle.com/java/technologies/java-card/javacard1.html, archived at https://web.archive.org/web/20220814102941/https://www.oracle.com/java/technologies/java-card/javacard1.html on Aug. 14, 2022.

21. Ponsini, N. *Announcing Java Card 3.2 Release* https://blogs.oracle.com/java/post/announcing-java-card-32-release.

22. *NXP JCOP3 J3H145 Java Card 3.0.4 Dual-Interface* CardLogix. https://www.cardlogix.com/product/nxp-jcop3-j3h145-java-card-3-0-4-dual-interface/, archived at https://web.archive.org/web/20230205095611/https://www.cardlogix.com/product/nxp-jcop3-j3h145-java-card-3-0-4-dual-interface/ on Feb. 5, 2023.

23. Paljak, M. *JavaCard Buyer's Guide of 2018* https://github.com/martinpaljak/GlobalPlatformPro/tree/v21.12.31/docs/JavaCardBuyersGuide (May 1, 2023).

24. Chen, Z. & Giorgio, R. D. *Understanding Java Card 2.0* InfoWorld. https://www.infoworld.com/article/2076617/understanding-java-card-2-0.html (Apr. 24, 2023), archived at https://web.archive.org/web/20210501210354/https://www.infoworld.com/article/2076617/understanding-java-card-2-0.html.

25. Ort, E. *Developing a Java Card Applet* https://www.oracle.com/java/technologies/java-card/developing-javacard-applet.html, archived at https://web.archive.org/web/20220209093147/https://www.oracle.com/java/technologies/java-card/developing-javacard-applet.html on Feb. 9, 2022.

28. *Identification cards — Integrated circuit cards — Part 4: Organization, security and commands for interchange* en. Standard ISO/IEC 7816-4:2020 (International Organization for Standardization, May 2022). https://www.iso.org/standard/77180.html.

30. YubiCo. *YubiKey 5 NFC Product Page* https://www.yubico.com/de/product/yubikey-5-nfc/ (May 1, 2023).

31. *Yubico Homepage* Yubico Inc. https://www.yubico.com.

32. Barker, E. *Recommendation for key management* tech. rep. (2020). https://doi.org/10.6028/nist.sp.800-57pt1r5.

33. *.NET YubiKey SDK: User's Manual. Challenge-response* Yubico Inc. https://docs.yubico.com/yesdk/users-manual/application-otp/challenge-response.html.

35. *PKCS #11 Cryptographic Token Interface Base Specification Version 3.0* en. Standard (OASIS, June 2020). https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/os/pkcs11-base-v3.0-os.html.

36. *PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0* en. Standard (OASIS, June 2020). https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/os/pkcs11-curr-v3.0-os.html.

38. *SmartCard-HSM home page* https://www.smartcard-hsm.com/ (Jan. 17, 2023).

40. Nitrokey GmbH. *Nitrokey HSM 2* https://shop.nitrokey.com/shop/product/nkhs2-nitrokey-hsm-2-7 (Feb. 9, 2023).

41. *SmartCard-HSM home page. Where to Buy* https://www.smartcard-hsm.com/buy.html (Jan. 17, 2023).

46. Oracle. *Java Card API. Interface HMACKey* version 3.0.5. https://docs.oracle.com/javacard/3.0.5/api/javacard/security/KeyBuilder.html (Jan. 18, 2023).

47. H. Krawczyk M. Bellare, R. C. *HMAC: Keyed-Hashing for Message Authentication* RFC 2104 (RFC Editor, Feb. 1997). https://www.rfc-editor.org/rfc/rfc2104.txt.

48. Dworkin, M. J. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques* tech. rep. (2001). https://doi.org/10.6028/nist.sp.800-38a.

49. GlobalPlatform Technology. *Card Specification* tech. rep. Version 2.3.1 (Mar. 2018). https://globalplatform.org/specs-library/card-specification-v2-3-1/.

51. libusb. *libusb Frequently Asked Questions* https://github.com/libusb/libusb/wiki/FAQ (Jan. 19, 2023).

54. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7. clock_getres, clock_gettime, clock_settime - clock and timer functions. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008),* 1–3951 (2018).

55. *NXP JCOP3 J3H145 Java Card 3.0.4 Dual-Interface* CardLogix. https://www.cardlogix.com/product/nxp-jcop-4-java-card-3-0-5-classic/, archived at https://web.archive.org/web/20230205095457/https://www.cardlogix.com/product/nxp-jcop-4-java-card-3-0-5-classic/ on Feb. 5, 2023.

56. *OpenVPN Data Channel Offload (aka OVPN-DCO)* OpenVPN. https://community.openvpn.net/openvpn/wiki/DataChannelOffload, archived at https://web.archive.org/web/20220809131809/https://community.openvpn.net/openvpn/wiki/DataChannelOffload on Aug. 9, 2022.

57. YubiCo. *YubiHSM 2 Product Page* https://www.yubico.com/products/hardware-security-module/ (Mar. 8, 2023).

58. Thales. *Luna-PCIe-HSM Product Page* https://cpl.thalesgroup.com/de/encryption/hardware-security-modules/pcie-hsms (Mar. 8, 2023).

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den October 2, 2023 ....