This is the extended version of a paper originally appearing in the proceedings of CARDIS 2014. The final publication is available at Springer via http://dx.doi.org/DOI (to be updated).

Caml Crush: a PKCS#11 Filtering Proxy

Ryad Benadjila, Thomas Calderon, and Marion Daubignard

ANSSI, France
{ryad.benadjila, thomas.calderon, marion.daubignard}@ssi.gouv.fr

Abstract. PKCS#11 is a very popular cryptographic API: it is the standard used by many Hardware Security Modules, smartcards and software cryptographic tokens. Several attacks have been uncovered against PKCS#11 at different levels: intrinsic logical flaws, cryptographic vulnerabilities or severe compliance issues. Since affected hardware remains widespread in computer infrastructures, we propose a user-centric and pragmatic approach for secure usage of vulnerable devices. We introduce *Caml Crush*, a PKCS#11 filtering proxy. Our solution allows to dynamically protect PKCS#11 cryptographic tokens from state of the art attacks. This is the first approach that is immediately applicable to commercially available products. We provide a fully functional open source implementation with an extensible filter engine effectively shielding critical resources. This yields additional advantages to using *Caml Crush* that go beyond classical PKCS#11 weakness mitigations.

Keywords: PKCS#11, filter, proxy, OCaml, software

Introduction

The ever increasing needs for confidentiality and privacy of information advocates for a pervasive use of cryptography. However, the security provided by cryptography itself completely relies on the confidentiality and integrity of some (quite small) pieces of data, e.g., secret keys. Therefore, sound management of this sensitive data proves to be as critical in ensuring any amount of security as the use of cryptography itself. In practise, cryptographic material is accessed and operated on through an Application Programming Interface (API). Protection and handling of sensitive objects thus fall back on *security APIs*, which enable external applications to perform cryptographic operations.

Normalization efforts have yielded the RSA PKCS#11 standard, which nowadays appears as the *de facto* standard adopted by the industry [19]. Therefore, much effort should be devoted to the provision of solutions allowing for safe and sound implementations of the PKCS#11 security API. In this article we present *Caml Crush*, a secure architecture meant to protect vulnerable PKCS#11 *middlewares*. As an additional software layer sitting between applications and the original PKCS#11 middleware, *Caml Crush* acts as a mandatory checkpoint controlling the flow of operations. The result is a PKCS#11 filtering proxy which can enforce dynamic protection of cryptographic resources through the use of an extensible filtering engine.

Though software tokens do exist, it is rather classical to depend on hardware assisted solutions, such as smartcards and Hardware Security Modules (HSMs). Having put to test numerous platforms exposing the PKCS#11 interface, it has come to our attention that the available implementations, be it open-source or commercial solutions, often do not meet average requirements in terms of standard compliance, robustness, let alone security properties. As many end-users of HSMs are not granted the ability to modify (or even access) the source code of their interface, tending to these diverse weaknesses whilst preserving standard compliance commends for a global approach. Additionally, we aim to provide users with means to dynamically customize such APIs according to self-imposed restrictions or needs for vulnerability patches.

Possible improvements of the exposed API. There are many ways in which one can strenghthen existing security APIs. The first and rather obvious step to take is to *enforce more acute conformity* to the PKCS#11 standard. Elementary as it seems, it really forms an inescapable axis of improvement, as there exist deployed tokens dutifully answering direct requests to output sensitive values, oblivious to the fact that the standard does explicitly prohibit it (see, e.g., [12]). That being said, security requirements stated in the PKCS#11 specification cannot be reached by solely implementing the standard to the letter. Indeed, the quite generic API described in the document bears inherent flaws which enable logical key-revealing attacks, such as the notorious wrap-and-decrypt attack. References depicting such attacks include [12,15,17], and we refer the reader to appendix A for more details. It is worth mentioning that Bortolozzo et. al. introduce in [12] a tool, Tookan, allowing for automatic API analysis and attack search. A second relevant amelioration of tokens consists in *patching PKCS#11 logical defects* while remaining as close as possible to the standard. In the meantime, it seems welcome to *address possible cryptographic attacks* such as padding oracle existence.

Fixing the PKCS#11 standard. Obviously, two main alternatives can be chosen to get a secure API: either try and fix the ubiquitous standard, or start over from scratch and design a whole new API. This latter possibility has been explored by Cortier and Steel in [16], and by Cachin and Chandran in [13], who propose a server-centric approach. As mentioned earlier, the need we address is to allow for a secure use of already available – and even possibly deployed – tokens. This calls for the choice of the first and more pragmatic alternative.

In [12], the authors exhibit a succesfully fixed PKCS#11 middleware: the software token named CryptokiX [2], whose security has been verified using the Tookan tool. CryptokiX is the work that bears the more similarities to our approach, in the sense that it successfully patches a number of the PKCS#11 standard flaws. There is no way to ensure that vendors provide customers with a patched version of their software. Hence, we believe that CryptokiX might not be a viable alternative for customers using HSMs as they operate the cryptographic resource with a proprietary and binary-only middleware. This objection put aside, this work proves their patches realistic, and we reuse them in our work. Though it is clear that no piece of software can replace a secure API embedded in the hardware itself, we advocate a best-of-both-worlds approach in which users can suit the trade-off between security, performance and confidence in the token native implementation to their needs and constraints.

Our contributions. In this paper, we propose an additional middleware and a software stack running a filtering proxy service between client applications using cryptography and PKCS#11 compatible security devices. The idea is to exclusively expose to regular users - or potential adversaries - the API as made available by the proxy, rather than letting them interact with the commercially available middleware. We show that *Caml Crush* provides the means to effectively augment the security properties of the resulting solution. Obviously, these security guarantees rely on the assumption that adversaries cannot bypass the proxy - which we find to be relevant, according to several examples of deployment scenarios presented in the paper.

We emphasize that *Caml Crush* allows to adequately patch problems in PKCS#11 implementations, but not to search for them. Indeed, our architecture includes a filtering engine able to hook API function calls to either simply block them or filter them based on a run-time policy. Our proxy can feature any tailored filtering functionality throughout the client connection's lifetime. In particular, it can be configured to enforce some or all of the aforementioned hardening measures on top of any PKCS#11 interface.

Below is a non-exhaustive list of noticeable functionalities:

- as detailed in section 1, every feature offered by CryptokiX is implemented in the filter module included in *Caml Crush*. In particular, patches to all known logical attacks are readily available in our implementation.
- the PKCS#11 standard allows to tag cryptographic objects using labels or identifiers. *Caml Crush* twists this feature to filter objects and thus restrict their visibility. It finds an immediate application in virtualized environments or resource sharing scenarios.
- our implementation and design choices ensure great portability and interoperability even on platforms with different operating systems and endianness.
- we provide solutions to some more attacks (such as coding flaws and buffer overflows vulnerabilities) by blocking, altering, or detecting and disabling repeated calls to a function.

We have validated our solution using both known attack implementations of our own and the more exhaustive trials performed by the Tookan tool. Eventually, we underline the practical relevance of our work on two accounts. Firstly, the filter engine possible configurations allow for flexible filtering policies. The complete source code of our implementation is made publicly available [1]. Moreover, the project was architectured with modularity in mind: it features user-defined extensions through plug-ins. Secondly, the performance cost measured in concrete deployment scenarios turns out to be reasonable.

Outline. Section 1 introduces PKCS#11 key concepts, briefly describes shortcomings of the API and details our motivations. Then, section 2 depicts the proxy architecture while justifying our design choices. Section 3 focuses on the filtering engine description, its extensible rule system and other benefits of using *Caml Crush*. Section 4 discusses deployment scenarios to secure various classes of devices, while section 5 is both a security and performance evaluation.

1 Motivations of the Work

1.1 An Introduction to PKCS#11

PKCS are a set of standards developed to allow interoperability and compatibility between vendor devices and implementations. The PKCS#11 standard specifies a cryptographic API. This allows the cryptographic resource vendors to expose common interfaces so that application developers can implement portable code, while hiding low-level implementation details. A common way of exposing the API is through OS shared libraries. The interface defined in the standard is called *Cryptoki*, as are sometimes called the corresponding libraries.

To abstract away from the cryptographic resource, PKCS#11 defines a logical view of the devices: the **tokens**. In order to interact with the token, an application has to open a **session** in which **objects** are manipulated. The objects can be keys, data or certificates and are used as input of cryptographic **mechanisms** defined by the standard. The objects can differ in their lifetime and visibility:

- Non-volatile objects are called **token objects**. They are accessible from all client applications. This is not the case for **session objects**, which are not meant to be shared between applications. Session objects are destroyed once the session ends.
- The visibility of objects is fundamental within the PKCS#11 API. An application for which no authentication has been carried out is only allowed to handle **public objects**. On the other hand, users are required to authenticate themselves before being able to use **private objects**. Once a session is opened with a resource, users traditionally achieve authentication by providing a PIN.

The purpose of a cryptographic resource is not generally limited to providing support for cryptographic operations. Most of the time, the token is meant to enforce security measures w.r.t. the objects it stores. Namely, the main feature expected from tamper-resistant devices is that even legitimate users logging in on the token are not able to **clone** it using the API. Thus, one of the key concepts behind PKCS#11 is to enable the use of cryptographic mechanisms without passing sensitive values in plaintext as arguments. The API uses **handles** to refer to objects, they are local to an application and bound to a session.

PKCS#11 objects can be exported from or injected into a token. This allows to save and restore keys (useful in case of broken or obsolete devices), but also to share keys over public channels between tokens. PKCS#11 objects are defined by a set of **attributes** which may vary depending on the object nature: symmetric secret keys have their value as an attribute, while asymmetric private keys have their modulus and exponents as attributes. Some attributes are common to all the storage objects though: examples are the `private` attribute and the `token` attribute characterizing the nature of the object (session vs. token objects as introduced previously).

Since the confidentiality of secret objects must be preserved, only their encrypted values are to be given to the user. PKCS#11 offers specific functions to export and import objects: `C_WrapKey` for **wrapping** and `C_UnwrapKey` for **unwrapping**. The result of a wrapping operation is an encrypted key value with a key that is inside the token, so that only the ciphertext is exported. In turn, keys used to protect other objects must be carefully managed. The PKCS#11 standard defines a few specific attributes to capture properties of keys allowing to monitor their use. Briefly, the `sensitive` attribute, when set to `TRUE`, is meant to prevent the user from fetching the value of the object, while an `extractable` attribute with value `FALSE` should prevent the user from exporting the object through a wrapping operation. Moreover, keys with attributes amongst `encrypt`, `decrypt`, `sign`, `verify`, `wrap` and `unwrap` can be used for the corresponding operation.

1.2 Attacker Model and Usual Shortcomings Exhibited by PKCS#11 Middlewares

The idea behind the definition of a standard such as PKCS#11 is to abstract away the implementation details of the underlying cryptographic resource. This allows a client application to use indifferently a wide range of resources, no matter how these latter carry out cryptographic operations. All these resources are some combination of software and hardware, and need a piece of software to export the PKCS#11 API. This latter is usually refered to as a PKCS#11 middleware. In the case of a Hardware Security Module, this middleware might be partly hosted inside the token, whereas for smartcards, it is a library to be loaded by the operating system.

The issues addressed by *Caml Crush* mainly fall into two categories:

- **defects** in the way middlewares implement the PKCS#11 API, leading to unexpected behaviors that can break applications expecting standardized answers.
- **purposeful attacks** that consist in any interaction with the PKCS#11 middleware resulting in the leak of sensitive information (such as the values of sensitive keys), or in the tampering with the middleware itself (through classical buffer overflow attacks for instance).

In a nutshell, our attacker model encompasses applications or users (be it legitimate or not) forging **any sequence of API calls leading to a successful leak of sensitive information or API defect**. Our definition of the success criterion of an attacker might seem quite unusual: whereas a successful attack typically results in sensitive information disclosure, we also take into account less obvious threats. Let us also emphasize that the attackers that we consider remain at the PKCS#11 API level: this implies that they only interact with the resource through the PKCS#11 middleware and never gain a direct lower level access to the token. We discuss this attacker model in 1.3 and give valid use cases in 4. In this model, issues related to PKCS#11 can be classified in one of the three elements detailed hereafter.

PKCS#11 Compliance Defects The PKCS#11 standard comprehends a broad set of features without providing a reference implementation, compliance is therefore hard to achieve. Most tokens only implement part of the specification. Even then, quite trivial inconsistencies have been found. Serious mishandling of the attributes of keys probably feature amongst the most critical disagreements with the standard requirements. Indeed, they very concretely lead to the output in plaintext of the value of secret keys. Such **behaviors** are explicitly **not compliant with the specification**.

PKCS#11 API-level Attacks Even strict compliance with the standard is not enough. **Logical attacks** that only exploit flaws in the API design itself confirm it. The most famous example is perhaps the so-called wrap-and-decrypt attack. It exploits the possibility to use keys for more than one type of operation, in order to extract sensitive keys from the token. Other attacks exploit use of **obsolete cryptographic schemes** (e.g., DES) and of combinations of mechanisms yielding **padding oracle attacks**. We encourage the interested reader to consult more detailed accounts of these flaws and possible patches in appendices A and B, and obviously in the seminal references [12,15,17].

Classic Vulnerabilities Middlewares are also prone to the **generic** pitfalls yielding **vulnerabilities** that an adversary can exploit in any piece of code. These oversights include absence of checking for errors, presence of buffer overflows or null-pointer dereferences. Consequences range from the pure and simple crash of the middleware to the redirection of the control flow of the programs or execution of arbitrary code. The quite large size and relatively low-level at which the PKCS#11 standard is specified make the resulting token implementations rather subject to exhibit such weaknesses.

1.3 Our Motivations for Providing a Filtering Proxy

Limitations of State of the Art Solutions The ideal way to detect and prevent the attacks described in 1.2 would be to provide **fixed token implementations and middlewares**. Related work in [12] provides two tools that – only partially – cover this issue:

- Tookan that allows automated search for attacks on PKCS#11 tokens.
- CryptokiX that is a reference implementation of a fixed software token. It is a fork of openCryptoki [5], a famous PKCS#11 software implementation of the standard. CryptokiX implements patches that turn out to be sufficient to fix the API against logical and cryptographic attacks.

However, these tools suffer from two practical limitations. Firstly, they only take into consideration a subset of the attacks described in 1.2 (namely PKCS#11 API-level attacks). Thus, compliance defects as well as classic vulnerabilities are not covered. Secondly, they can be of interest to token vendors, but are of limited interest to token users in the field. Users are able to check with Tookan whether their token is vulnerable to certain classes of attacks. Unfortunately, without the vendor support nothing can be done, and the user still ends up using his token despite its possible vulnerabilities.

As a consequence, the matter of fixing commercially available tokens is not addressed by the related work. We envision two possible scenarios regarding this issue. One can hope that vendors successfully repair existing vulnerable tokens and integrate the countermeasures in their future designs. In our experience, it takes a long time to achieve such a goal. We rather believe that vulnerable tokens are not to completely disappear anytime soon. Many PKCS#11 devices, e.g., smartcards, cannot be updated easily, if they are updatable at all. Furthermore, vendors will probably not maintain obsolete PKCS#11 devices, even if some are still being used. Finally, when some vendors provide a patch for their tokens, it is very likely that only the most recent platforms benefit from them. Deprecated operating systems interfacing with the token will not be able to get updates.

Using *Caml Crush* to Dynamically Protect Vulnerable Tokens All the previous limitations motivated us to design a suitable solution for users who want to protect potentially vulnerable tokens, but are deprived of patches. With *Caml Crush*, we aim at dynamically detecting and applying mitigations against attacks on PKCS#11 requests **before they reach the token**. In order to achieve this, we have implemented a solution that sits between the original middleware and the PKCS#11 applications, namely a **PKCS#11 proxy**. Alternatives include developing a replacement middleware, but low-level interfaces with devices are often proprietary. Therefore, we opted for a lightweight and more portable solution. This induces some limitations, though, discussed in 5.3.

Not only does our design implement the state of the art patches inherited from [12], but it also comes with supplementary features. *Caml Crush* adds to tokens a detection and protection layer against adversaries who can forge PKCS#11 requests that exploit vulnerabilities on tokens that are known to be vulnerable (e.g., to a buffer overflow on a PKCS#11 function argument). We stress out that the hardware device remains in charge of secure key storage and cryptographic operations.

We recall that we make one working hypothesis about the attacker capabilities though: no adversary can bypass the PKCS#11 proxy and directly communicate with the resource (see the attacker model discussed in 1.2). This is obviously not a limitation in cases where the cryptographic resource is a – part of – a dedicated machine on a managed network. This approach is easily applied to network HSMs and more thoroughly discussed in 4.

2 Architecture

Using a proxy is an efficient approach in order to protect cryptographic resources and vulnerable PKCS#11 middlewares. There exist some projects implementing PKCS#11 proxies among which GNOME Keyring [3] and pkcs11-proxy [6]. However, these give priority to performance, usability or ergonomic concerns. As it does not really address our need for a robust security solution, we have chosen to propose a completely new architecture. In this section, we motivate our design choices and present the components of *Caml Crush*.

2.1 Design Choices

Critical pieces of the software use the OCaml programming language: it offers a static type system, a type-infering compiler and relieves the programmer from memory management issues. The functional programming paradigm is well-suited to express filtering rules.

The communication layer plays an essential role in a proxy architecture. *Caml Crush* uses standard *Sun RPC* [8] Remote Procedure Call and its XDR [9] data serialization format. This ensures greater portability as most operating systems have a native implementation of this standard. *Caml Crush* can operate over *Unix domain* or TCP sockets and the link can be secured using TLS mutual authentication. The list of acceptable TLS cipher suites is tunable on the server side.

In order to end up with code of higher quality, we generalize the use of automatic code generation. Indeed, this practice allows the programmer to rely on the code of the tool, which is generally smaller and well tested-out. It is very likely that it also reduces the introduction of vulnerabilities in the resulting code, e.g., stemming from bad memory management and human errors.

In *Cryptoki*, each application has a context which amounts to a list of handles and session states (read-only, user logged, etc). As stated in the PKCS#11 standard “an application consists of a single address space and all the threads of control running in it”, meaning that an application is mapped to a single process. Therefore the standard relies on the logical separation of processes to isolate multiple PKCS#11 contexts. This is handled by all operating systems supporting virtual memory. In our opinion using a *multi-threaded* architecture for the proxy is in contradiction with the standard and bound to create unforeseen issues. This partially explains why thread-based projects such as GNOME-Keyring or pkcs11-proxy [3,6] were not reused. For this reason, *Caml Crush* is a *multi-process* architecture handling client connections through *fork-exec*. Each process is tied to a client and runs its own instance of the filter engine, with its own object and session handles stored in its memory space.

2.2 Components

One of the design goals of *Caml Crush* is modularity. Having the possibility to replace portions of code while minimizing the impact is essential. This is why *Caml Crush* is split in several sub-components. Figure 1 illustrates this architecture.

OCaml PKCS#11 Binding ① PKCS#11 middlewares are shared libraries, *DLL* for Windows and *.so* files for Linux. A client application that needs to perform cryptographic operations uses operating system features (*LoadLibrary* or *dlopen*) to load the middleware and then calls PKCS#11 functions. Unfortunately, OCaml does not natively support loading a C shared library. However, calling C foreign functions is allowed.

The binding is the low-level part of *Caml Crush* which is used to load the middleware and forward calls to the cryptographic resource. The code of this component is mostly generated with the help of CamlIDL [14]. This tool can generate the necessary stubbing code to interface OCaml with C. CamlIDL works with an IDL file whose syntax is derived from C and enhanced to add type information. This greatly simplified our work as the conversion code and memory allocation are handled automatically.

The resulting stubbing functions point to corresponding symbols that call the PKCS#11 functions of the real middleware. These were manually written and mainly act as a pass-through.

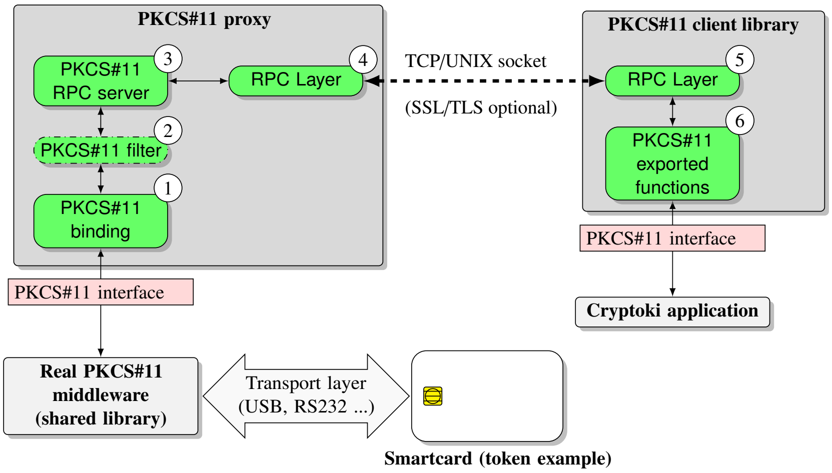


Fig. 1: *Caml Crush* architecture overview

PKCS#11 Filter ② The filtering engine is thoroughly detailed in section 3 but we point out that this component relies on the OCaml PKCS#11 binding ① to communicate with the real middleware.

PKCS#11 Proxy ③ ④ The proxy server is a critical component of this architecture. Because it is facing potentially hostile clients it has to be robust and secure.

We based our proxy service on the **Ocamlnet** library. More specifically the **Netplex** subclass was used to implement our PKCS#11 RPC listening service ③ . The great benefit of using Ocamlnet is its support for the *Sun RPC* standard. As for the binding described earlier we can use a description file to produce the code in charge of data serialization on the transport layer ④ . A file with the XDR syntax is used to define the available RPC functions and the various structures. Both the client and server take advantage of this.

Our study of concurrent access to PKCS#11 resources explained in 2.1 drove us to conclude that using one process per client connection was the only solution worth considering. With this approach it is not as easy for a hostile client to successfully use the handles of another application.

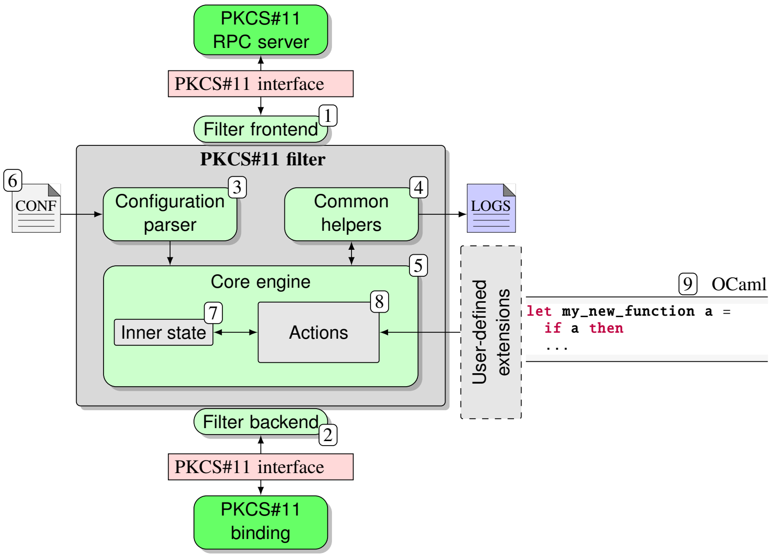
Best security practices recommend dropping all unnecessary privileges for system daemons. Since OCaml does not provide the necessary APIs to accomplish this task to harden the server process we provide a custom primitive. After its initialization, we instruct **Netplex** to call a function that performs capabilities dropping and privilege reduction from our C bindings. Further hardening can be achieved depending on the sandboxing features available on the operating system running the *Caml Crush* daemon.

PKCS#11 Client Library ⑤ ⑥ The final component is the PKCS#11 shared library that substitutes to the original middleware. Client applications load it to perform cryptographic operations. The main task of the client library is to set up a communication channel with the server, export PKCS#11 symbols ⑥ to the calling application and relay function calls to the proxy server with serialized arguments. The transport layer code ⑤ is generated from the XDR file in the same way as for the proxy. Some sanity checks are performed within the library to prevent invalid requests from reaching the proxy server. However, we want to stress that the client library **plays no role** in the security of this architecture (i.e. an attacker controlling the library does not reduce the overall security).

3 PKCS#11 Filtering Engine

3.1 Architecture of the Filter

Overview The engine is divided into several components detailed in Figure 2. Firstly, it is isolated from the PKCS#11 proxy by a frontend $\boxed{1}$ and from the OCaml PKCS#11 binding by a backend $\boxed{2}$. Secondly, it includes a configuration parser $\boxed{3}$, to process set-up data provided by the administrator. Helpers $\boxed{4}$ are also used for common tasks such as logging. Eventually, the filter core engine $\boxed{5}$ performs the filtering actions within PKCS#11 calls, helped by requests to the backend. In this section, we start by describing the filter engine, before explaining how it can be used to fix implementations of the PKCS#11 standard.



```
let my_new_function a =
  if a then
  ...
```

Fig. 2: *Caml Crush* filtering engine overview

Core Engine $\boxed{5}$ The configuration parser takes as input a configuration $\boxed{6}$ (defined by the administrator) and uses it to build a static filtering policy. This policy is expressed as a mapping from PKCS#11 function names to a sequence of operations performed $\boxed{8}$ each time the given function is called. The most basic example of operation consists in simply forwarding the call to the backend, getting the matching output and forwarding it back to the frontend. A filter instance is loaded when an application opens a connection with the server, a new process is forked on the proxy side. It is unloaded when the connection is closed. The *multi-process* model grants *Caml Crush* the ability to load and isolate multiple PKCS#11 middlewares. The filter configuration allows to apply fine-grained filtering policies depending on the target middleware.

Actions $\boxed{8}$ **and User Extensions** $\boxed{9}$ We have conceived our filter with modularity in mind. The engine is architectured to allow precise tuning of the filtering policy and user-specific extensions. This modularity requirement accounts for our introduction of an intermediate abstraction layer, built on the notion of *filtering actions* $\boxed{8}$. Two alternatives are available to users to adapt the filter to their needs:

- a few completely predefined configurations $\boxed{6}$ are proposed, which we believe are relevant in concrete use-cases. They comprise all of PKCS#11 patches as well as function blocking and *label*/*id* filtering (details about these features follow).
- it is possible for the user to extend the built-in actions through plug-ins written in OCaml. Since each PKCS#11 function is hooked inside the filter, it can be configured to call any other user-defined function implemented in the plug-ins. Thus, adapting the filter essentially boils down to implementing new functions suiting one's needs and editing the configuration file accordingly.

3.2 Filtering Features Involving Standard PKCS#11 Mitigations

Mitigations against Logical Attacks Logical attacks described in 1.2 are mainly due to exposing wrap and unwrap functions. A radical way to partially fix the API would be to **completely remove them**. However, such an approach is too drastic as some use cases require the wrap/unwrap mechanisms, e.g., when they are needed to share keys between tokens over public channels. To address this, Fröschle *et al.* have proposed patches in [18], then extended in [12], to fix PKCS#11 against logical attacks without removing the wrapping functionality. They put forward two sets of patches, that each presents their own advantages and drawbacks. Details about the patches can be found in appendix B. In our proxy design, these fixes are naturally implemented as **filtering actions**. The checks are **dynamically enforced** at runtime each time a PKCS#11 request is sent to the middleware. *Caml Crush* provides the same security level as CryptokiX against logical attacks.

Mitigations against Cryptographic Attacks The only way to efficiently prevent the usage of **obsolete ciphers and mechanisms** is to globally prohibit their usage in the token. Our filter engine allows to mimic the absence from a token of weak mechanisms – e.g., substandard cipher suites or poor key derivation schemes. Indeed, all the cryptographic functions (`C_EncryptInit` and so forth) called with these mechanisms can be blocked, as well as the creation of keys supporting them. Since we do not want to impact client applications, we also amend the behavior of functions that list mechanisms supported by the token such as `C_GetMechanismList`. Such a technique can be applied to counter **padding oracle** attacks: mechanisms as PKCS#1 v1.5 and `CBC_PAD` can indeed be considered as “weak mechanisms”. As padding oracles exploit the unwrapping functionality, the utter removal of unwrapping operations appears as a solution. As mentioned earlier, this is unrealistic in some use cases. A better alternative is provided by the *wrapping format* patch (see appendix B for details). This latter precludes the decryption of malformed ciphertexts, from which the information leakage useful to padding oracle attacks comes from.

3.3 Object and Structure Filtering

Resource Sharing and Label/Id Filtering Although client applications can have different criticality levels, they most likely share the same cryptographic resource. This can lead to involuntary information leaks: as PKCS#11 defines a single user mode of operation, once authenticated to the token, any private token object is usable by an application.

PKCS#11 allows applications to search for objects matching certain attributes. One can fetch a handle to a specific object using its `label` or `identifier` attribute. As a result, we propose to use both attributes in the filter engine to restrict the set of token objects with which an application can operate. It can be done in a completely transparent way. For instance, by prefixing or suffixing labels used by applications with keywords related to their criticality level. Then, calls to PKCS#11 functions with which objects can be accessed, read or modified are adapted by the filter to simulate a token containing only the objects of a given criticality level. A concrete use case of this feature is given in section 4.2.

Key Usage Segregation As we have already discussed, many PKCS#11 flaws result from some keys being allowed multiple usages or roles. Even subtle ways of disrespecting the key separation principle yield confusions at the API level resulting in successful attacks. The fixes presented in [18,12] mainly focus on `wrap/unwrap` and `encrypt/decrypt` segregation. We however highlight the fact that one might also want to push this logic further with the `sign/verify` attributes. For example, a PKI (Public Key Infrastructure) application only needs to sign and verify data with the asymmetric keys. Disabling other uses of these keys seems relevant. All these patches have been easily integrated to the filtering rules we provide.

Token Information Filtering PKCS#11 describes a set of structures that characterize a token. For instance, the `CK_TOKEN_INFO` structure contains information such as a serial number, a manufacturer ID and so on. The filtering proxy can be used to transparently modify such information: for instance, a PIN length policy can be set up by changing the `ulMinPinLen` and `ulMaxPinLen` fields. A policy on the characters set as well as protection against dictionary attacks can also be enforced when the PIN is set by the user. It is readily enabled by the hooking of PKCS#11 functions `C_InitToken` and `C_SetPIN` performed in the filter engine, to allow returning an error if the PIN does not respect the policy.

3.4 Blocking PKCS#11 Functions and Mechanisms

Function blocking offers a simple way to deactivate unused or dangerous features of PKCS#11. For example, one can express a filtering policy to block administration operations. Clients connecting to this instance cannot use PKCS#11 administration functions but can normally use the token.

Though rather elementary, disabling functions can prove effective to prevent security breaches often left unadressed by the common PKCS#11 patches. Indeed, provided that the user is authenticated, he can freely create and modify objects on the token. This in turn potentially enables him to tamper with the device in order to force known values as keys. Blocking object creation and modification offers a way to impede the aforementioned attacks, thus addressing the issue of hostile users, while object management can still be performed on a dedicated trusted filter instance.

Finally, as pointed out before, mechanisms filtering can also be of interest, be it to completely block unwanted mechanisms, or, according to more fine-grained scenarios, to filter out some combination of operations.

3.5 Improving PKCS#11 Compliance

Fixing a Non-conforming Implementation The filtering proxy can prove useful to enhance PKCS#11 compliance. A real example that we encountered using a smartcard middleware affects calls to the function `C_GetAttributeValue`. The standard allows to fetch multiple object attributes in a single call. Listing 1.1 illustrates such a call, three boolean values should be retrieved. The issue arises because the middleware does not have support for the second attribute, namely `CKA_ALWAYS_SENSITIVE`. The correct way to handle this is for the library to change the *ulValueLen* field of the unknown attribute to hold the value -1. The other requested attributes are supposed to be populated with their values. However, our smartcard stops processing the `C_GetAttributeValue` call when it encounters an unsupported attribute, leaving our third attribute (`CKA_WRAP`) to a default value. This behavior is not compliant with the standard and provokes undefined behavior in client applications. Fortunately, the filtering engine is well-suited to correct such behaviors. We have written a dedicated filter action to fix the issue for this PKCS#11 middleware. The rule hooks the `C_GetAttributeValue` function and seamlessly performs sequential calls to the middleware and returns an aggregated output to the client application. We emphasize that the custom rule is only triggered for the misbehaving middleware thanks to *Caml Crush* versatility.

```
  ...
  CK_ATTRIBUTE stemplate[] = {
    {CKA_SENSITIVE, &sensitive_set, sizeof(CK_BBOOL)},
    {CKA_ALWAYS_SENSITIVE, &a_sensitive_set, sizeof(CK_BBOOL)},
    {CKA_WRAP, &wrap_set, sizeof(CK_BBOOL)},
  };

  ret = C_GetAttributeValue(session, obj_handle,
          stemplate, sizeof(stemplate)/sizeof(CK_ATTRIBUTE));
  ...
  if(wrap_set){
    ...
  }
}
```

Listing 1.1: `C_GetAttributeValue` conformance issue

Fixing a Broken Implementation Another self-explanatory example is discussed in [12]: some tokens answer with sensitive key values to calls placed to `C_GetAttributeValue`, whereas this is explicitly forbidden in the specification. The current implementation of the filter ensures that sensitive data is never output in response to such direct queries: **before** forwarding these calls for key values to the backend, we check whether the sensitive attribute of the object is set to `TRUE`.

3.6 Security Breaches Beyond PKCS#11 Flaws

Fixing Generic Coding Errors Since the filter sits between the client application and the PKCS#11 middleware, one can detect, filter and alter any known bad request or behaviour of malicious applications. Thus, **prevention of vulnerability exploitation**, or more generally mending design flaws in middlewares, puts the proxy to good use. Let us illustrate these words with a realistic example of an error that we found in an existing middleware, in the PKCS#11 `C_SetPIN` function call, as presented on listing 1.2.

```
CK_RV C_SetPIN(CK_SESSION_HANDLE hSession, CK_UTF8CHAR_PTR pOldPin, CK_ULONG ulOldLen, CK_UTF8CHAR_PTR
     pNewPin, CK_ULONG ulNewLen){
  ...
  /* Compare stored PIN with old PIN  */
  if(memcmp(StoredPin, pOldPin, ulOldLen) == 0){
    /* If test is ok, store the new PIN */
    *StoredPinLen = ulNewLen;
    memcpy(StoredPin, pNewPin, ulOldLen);
    return CKR_OK;
  }
  /* Provided old PIN is incorrect */
  return CKR_PIN_INCORRECT;
}
```

Listing 1.2: `C_SetPIN` coding error example

As we can see, the newly stored PIN is either truncated or extended to the old PIN length; either way it is rendered erroneous by a call to `C_SetPIN`. The inherent risk is to block the underlying token, the user having no clue which PIN is actually set. Even though it is not possible to truly patch this error without modifying the code or the binary of the middleware, the filtering proxy can help avoiding such a pitfall. The filtering actions associated to the `C_SetPIN` function can consist in checking that the old and new PIN share the same length before forwarding the call to the middleware. In case lengths do not coincide, the proxy returns the error `CKR_PIN_LEN_RANGE` and the PIN is not modified. The client application can later fetch the correct length it needs using another PKCS#11 function and call `C_SetPIN` again. Although a constant PIN length is forced, the entered PIN and the stored one are consistent.

Preventing Denial of Service PKCS#11 defines a calling convention described in [19, p. 101] for functions returning variable-length output data. In some cases, the affected functions are supposed to handle either null or valid pointers. During our development we observed that some middlewares end up dereferencing null pointers. These vulnerabilities are easily prevented by implementing a filter action that performs input sanitizing.

Another example we encountered is that using a cryptographic function with a malformed input (a non-standard mechanism) we could freeze a token, leading to the unavailability of the cryptographic resource. Again, this behavior was corrected using a custom filter action, the malformed input is not sent to the device and a PKCS#11 compliant error is returned to the client application.

We advocate that a large set of such coding errors and vulnerabilities can similarly be corrected by stopping or modifying malformed requests before they reach the middleware.

4 Deployment Scenarios

Security guarantees provided by *Caml Crush* rest upon the assumption that going through the proxy is mandatory. Yet it is potentially still possible to connect to the cryptographic resource directly. For instance, an attacker could try to load the vendor middleware or use the transport layer to directly communicate with the device. Though such attacks are realistic, we advocate that for any type of token, complementary security measures can mitigate this issue. This section discusses secure deployment strategies for *Caml Crush*.

4.1 HSMs in Corporate Networks

Network HSMs provide a convenient way to perform cryptographic operations and securely store keys in a corporate environment. They are frequently used as backends for PKI solutions, timestamping servers and document or code signing applications. Traditionally, these devices can be considered as black boxes, accessed using the interfaces provided by the vendor (usually PKCS#11). In this context of use, *Caml Crush* is to be installed on a dedicated server with at least two network cards. The first card shall be directly connected to the network HSM, thus shielding the device from any other hosts, while the second network card shall be connected to the corporate network. Since the HSM is only linked to the proxy, client applications are forced to access the cryptographic resource through our filtering proxy using the *Caml Crush* client library. Clearly, meticulous users can apply complementary hardening measures to further reduce the attack surface of the server hosting *Caml Crush*.

In rare cases, HSM vendors allow non-proprietary code to run on their platform. These particular devices offer a way to tightly couple *Caml Crush* with the cryptographic device without needing additional hardware. We also point out that OEM vendors who integrate standalone HSMs (such as PCI devices) can benefit from *Caml Crush* when it is accessed using PKCS#11. As they may face the same issues as customers when provided with binary-only middlewares, they shall integrate *Caml Crush* within their designs.

4.2 Virtualized Environment

Caml Crush can be used within virtualized operating systems in order to securely use a cryptographic resource. Figure 3 illustrates such a deployment scenario. In this example, the PKCS#11 device is only exposed to the trusted hypervisor, virtual machines wishing to use the resource can only do so using the *Caml Crush* client library. This architecture also leverages *Caml Crush* resource sharing capabilities using a filtering policy dedicated to each virtual machine. Here, the policy for Virtual Machine 1 restricts PKCS#11 applications to use objects with a label in the set `A` (resp. `B` for VM 2). Therefore, the filtering engine transparently compels virtualized environments to use objects matching their respective policy.

While this scenario uses the hypervisor isolation features, more lightweight isolation alternatives exist for standalone desktops using USB smartcards. The Linux operating system can be enhanced with Mandatory Access Control (MAC) support such as SELinux [7] or Grsecurity role-based access control [4]. Building on discretionary access control and MAC enforces a security policy restricting PKCS#11 and low-level smartcard access to *Caml Crush* instances.

4.3 Mobile and Embedded Platforms

Given the fact that vendors provide binary-only PKCS#11 middlewares, compatibility is generally limited to mainstream operating systems and microarchitectures. In our opinion, running an unconventional CPU platform (such as MIPS or ARM to a lesser extent) should not stand in the way of the use of hardware-assisted cryptography. Having chosen standardised communication protocols ensures great portability of our code. Our initial implementation was

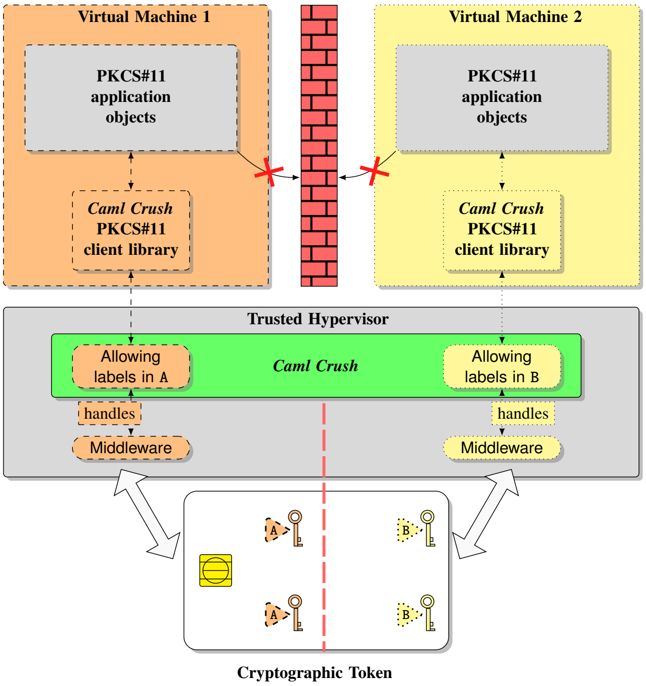


Fig. 3: *Caml Crush* used for resource sharing in a virtualized environment

Linux specific but it is worth mentioning that porting to Mac OS X and FreeBSD required little efforts. Windows support is limited to the client library, running the server code through Cygwin is a work in progress. A native Windows port for the server is not excluded but requires significant development. We stress that *Caml Crush* is fully capable of handling clients with a foreign endianness. As a result we have successfully validated interoperability scenarios using our PKCS#11 client library on ARM, MIPS and PowerPC architectures. A corporate environment can benefit from the variety of systems supported, from embedded to mobile devices or legacy systems, in order to access remote PKCS#11 resources through the use of *Caml Crush*.

5 Evaluation

5.1 Security Evaluation

We ensure that the filtering engine performs as expected, i.e. protects vulnerable devices. To do so, we use two complementary approaches. First, we implemented classic PKCS#11 attacks to manually verify the efficiency of our filtering rules. There exist several combinations of PKCS#11 API flaws that can lead to leak sensitive information. Since manual verification can only go so far, the Tookan tool is used to try finding attack paths.

On a Linux computer, we installed openCryptoki, a software HSM. We also compiled and installed *Caml Crush* on this machine and configured it to use the default filtering rules. These latter enforce the needed properties described in appendix B to secure the PKCS#11 API (*conflicting* and *sticky* attributes, *wrapping format*). The table 1 summarizes the results obtained both using our manual approach and using Tookan. Unsurprisingly, the unprotected device remains vulnerable. However, once instructed to use the *Caml Crush* client library, our filtering engine works as expected since neither Tookan nor our manual tools are able to identify or perform attacks. The completeness result obtained by the authors of Tookan allows to deduce that the filter efficiently prevents all attacks that can be carried out in the model underlying their tool.

5.2 Performance Evaluation

In this section we present the various test cases that we used to quantify the performance impact of our solution. The experiments were conducted on three different platforms, a PCI HSM, a network HSM and a USB smartcard. For each cryptographic device, our benchmark is run three times. First, the raw performance is computed using various cryptographic operations. Second, we run the same benchmarks using *Caml Crush* with the filtering engine disabled to measure the architectural cost. Finally, we enable all of our filtering rules to add up the remaining cost of *Caml Crush*. The table 2 summarizes the types of operations we used during our performance testing, as well as the number of such operations performed on each type of device. We point out that the card is a USB smartcard

	Prevented Attacks					
	Logical Attacks				**Compliance Defects**	**Cryptographic Attacks**
	Wrap/ Decrypt	Unwrap/ Encrypt	Sticky	Create object	Get sensitive	Padding Oracles
openCryptoki	✗	✗	✗	✗	✓	✗
openCryptoki with *Caml Crush*	✓	✓	✓	✓	✓	✓

Table 1: Security evaluation of *Caml Crush*.

using an open source middleware and has fewer capabilities compared to HSMs. We iterated the type of operation depending on the device performance. HSMs and network HSMs are fast devices capable of handling multiple requests at the same time. Therefore, we also ran benchmarks simulating multiple client applications performing the described operations (about ten clients running various operations).

	Token types			
	PCI HSM and NetHSM		**USB Smartcards**	
	Operation type	Number†	Operation type	Number†
`key-gen`	AES-128 Generate keys	10^4	✗	✗
`rand-dgst`	random/SHA-1 Generate random then hash it	10^4	random/SHA-1 Generate random then hash it	10^3
`rsa`	RSA-2048 encrypt/decrypt sign/verify	10^4	RSA-2048 sign/verify	10^3
`aes`	AES-128 encrypt/decrypt	10^5	✗	✗

✗ The token does not support the operation types.

†Number of operations performed on the token to measure performance.

Table 2: Performance evaluation benchmark of *Caml Crush*.

PCI HSM The left side of figure 4 illustrates the performance impact of *Caml Crush* using a sequential client application. The most significant performance drop affects the `aes` operations. These are fast operations and adding *Caml Crush* on top of such local devices reduces throughput. The `key-gen` and `rand-dgst` operations respectively have a 25% and 50% performance penalty. On the other hand, `rsa` tests are time-consuming operations and the impact is negligible. The right side of the figure clearly demonstrates that when the resource is accessed using multiple applications at the same time, the impact of *Caml Crush* is low.

Network HSM We now focus on the evaluation on a network HSM, the results are shown on figure 5. The observation is similar to the PCI-HSM, using a single sequential client, *Caml Crush* has roughly the same performance impact. We recall that the filter engine fetches attributes from the device when processing PKCS#11 calls (using `C_GetAttributeValue`). Those supplementary calls account for a large portion of the throughput drop. Again, we emphasize that *Caml Crush* cost is reduced when the cryptographic resource is under heavier load from multiple clients.

Smartcard The performance impact of *Caml Crush* related to the smartcard at our disposal is illustrated on figure 6. Smartcards are rather slow devices and perform through the USB bus. Given this, we observe a 20% drop on `rsa` tests and less than 10% on the `rand-dgst` operations.

We used various benchmarks to quantify the performance cost of our solution. Assembling a software layer on top of another one obviously consumes some resources. In our case, the RPC layer accounts for a substantial part of the performance penalty. Furthermore, the supplementary calls needed by the filtering logic add an overhead that is device-specific. Nevertheless, we state that the performance trade-off remains acceptable.

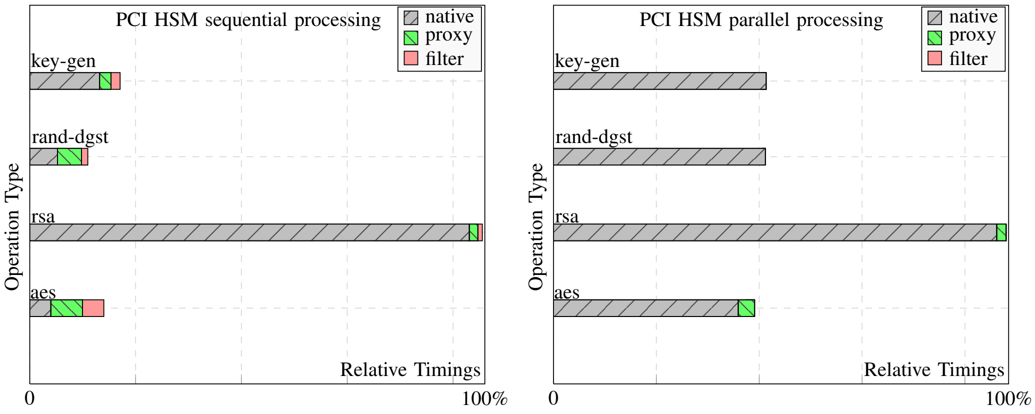


Fig. 4: PCI HSM performance
Relative timings are used, the operation taking maximum time is at 100%

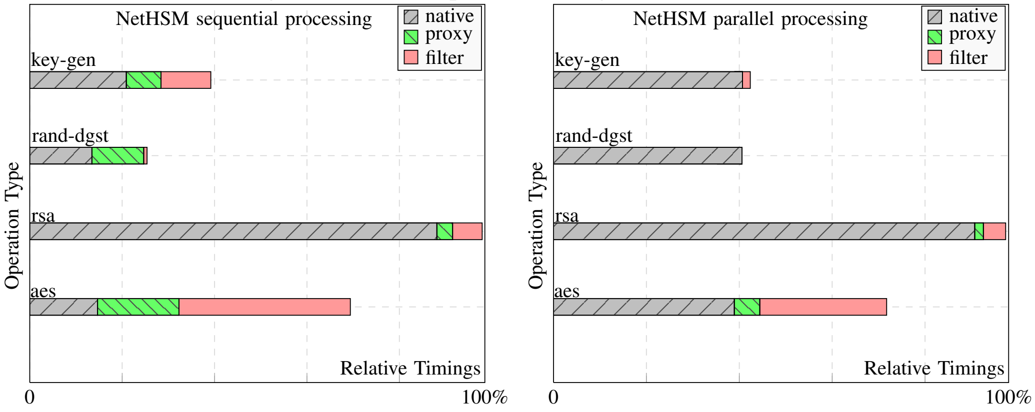


Fig. 5: NetHSM performance
Relative timings are used, the operation taking maximum time is at 100%

5.3 Filter Limitations and Future Work

Currently, the filtering engine lacks the ability to adapt filtering actions based on the state between different client connections. We described in 2.1 that each client's connection is isolated in separate processes. *Caml Crush* would need to use Inter-Process Communication (IPC) mechanisms in order to exchange state-related messages. It could prove useful in some filtering scenarios but would require development of synchronization primitives and significantly increase the code complexity. Such feature would probably have further impact on the overall performance.

Another limitation is that the current filter plug-ins use the OCaml marshalling module that lacks type safety: this means that extra care must be taken by users when writing code as filter extensions. Errors in the plug-in code could indeed evade the compile-time checks, and might allow an attacker to tamper with the memory of the server instance (process) dealing with the client. The implications of such memory tampering of OCaml native structures is not clear, but it would at least provide the attacker with a denial of service capability on the instance. Albeit, the attacker would not be able to attack other clients instances thanks to the fork-exec model (provided that appropriate operating system level protections and sandboxing features are used).

Furthermore, writing plug-ins requires expertise in OCaml. We are currently working toward the removal of marshalling functions. We profit from this step in the filter development to rethink the way filter actions are encoded. We plan on introducing an intermediate domain-specific language using more generic and fine-grained atomic actions. This would allow advanced users to use this intermediate language to specify filter actions. Such an abstraction is meant to relieve users from dealing with the complexity of OCaml and adherence to our design choices in the filter backend.

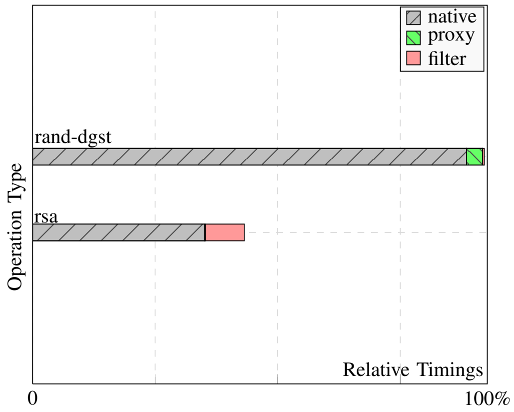


Fig. 6: Smartcard performance
Relative timings are used, the operation taking maximum time is at 100%

Conclusion

We are able to dynamically address security issues of the PKCS#11 API. Related work has paved the way to resolve these issues with a reference PKCS#11 software implementation. However, applying such countermeasures is left to the vendors of cryptographic devices. This is insufficient as commercially available and already deployed devices remain vulnerable. *Caml Crush* offers an alternative to protect cryptographic resources from state of the art attacks. Substituting the original middleware with our proxy and filtering PKCS#11 function calls is a pragmatic and effective approach. Moreover, the filter engine is conceived to be modular: it is possible for to customize and extend the filter with plug-ins written in OCaml.

Caml Crush filtering engine is versatile enough to enable complementary features such as function blocking, improved PKCS#11 compliance and secure resource sharing comes. We are confident that these functionalities find immediate application for users of compliant cryptographic devices.

Future efforts could focus on relaxing our dependency on middleware implementation of the PKCS#11 standard. To do so, a complete PKCS#11 stack with its own management of objects and sessions needs to be developed. This extension also requires to enhance current attribute parsing and handle their storage. Finally, the filtering engine could also benefit from IPC mechanisms to enrich current rules.

References

1. `Caml Crush`. `https://github.com/ANSSI-FR/caml-crush/`.
2. CryptokiX. `http://secgroup.dais.unive.it/projects/security-apis/cryptokix/`.
3. GNOME Keyring. `http://live.gnome.org/GnomeKeyring`, .
4. `grsecurity`. `http://http://grsecurity.net/`.
5. openCryptoki. `http://sourceforge.net/projects/opencryptoki/`.
6. pkcs11-proxy. `http://floss.commonit.com/pkcs11-proxy.html`, .
7. `SELinux`. `http://selinuxproject.org/`.
8. Sun RPC RFC 1057. `http://www.ietf.org/rfc/rfc1057.txt`, 1988.
9. XDR RFC 4506. `http://tools.ietf.org/html/rfc4506`, 2006.
10. R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay. Efficient padding oracle attacks on cryptographic hardware. In *Advances in Cryptology - CRYPTO 2012*, pages 608–625, 2012.
11. D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1. In *Advances in Cryptology - CRYPTO '98*, pages 1–12, 1998.
12. M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *ACM Conference on Computer and Communications Security*, pages 260–269. ACM Press, Oct. 2010.
13. C. Cachin and N. Chandran. A secure cryptographic token interface. In *CSF 2009*, pages 141–153. IEEE Computer Society, 2009.
14. CamlIDL. CamlIDL project page. `http://caml.inria.fr/pub/old_caml_site/camlidl/`, 2004.
15. J. Clulow. On the security of pkcs#11. In *CHES 2003*, pages 411–425, 2003.
16. V. Cortier and G. Steel. A generic security api for symmetric key management on cryptographic devices. In *ESORICS*, pages 605–620, 2009.
17. S. Delaune, S. Kremer, and G. Steel. Formal security analysis of pkcs#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, 2010.
18. S. B. Fröschle and G. Steel. Analysing pkcs#11 key management apis with unbounded fresh data. In *ARSPA-WITS 2009*, pages 92–106, 2009.

19. RSA Security Inc. PKCS#11 v2.20: Cryptographic Token Interface Standard, 2004.
20. S. Vaudenay. Security flaws induced by cbc padding - applications to ssl, ipsec, wtls ... In *Advances in Cryptology - EUROCRYPT 2002*, pages 534–546, 2002.

Notations. Please mind the following naming convention for the appendices:

- `wr` (resp. `un`, `de`, `en`, `se`, `ex`, `ve`, `si`) stands for the `wrap` (resp. `unwrap`, `decrypt`, `encrypt`, `sensitive`, `extractable`, `verify`, `sign`) attribute set to `TRUE`. Similarly, ~~`wr`~~ and so on represent the same attributes set to `FALSE`. For example, the key $k^{\{\cancel{\mathtt{wr}},\mathtt{se}\}}$ is `sensitive` but cannot wrap other keys.
- If we consider two keys k_1 and k_2, $\{k_1\}_{k_2}$ denotes k_1 wrapped with k_2.
- $||$ is the concatenation operation.

Appendix A: Taxonomy of the Attacks on PKCS#11

Two attack strategies can be used to reveal sensitive keys. Firstly, an attacker can inject a corrupted key in the token with **controlled attributes** by submitting a random blob: no verification is imposed by the standard. This is called *key conjuring*.

In addition, the PKCS#11 standard does not enforce rules regarding the possible key usage: for example, the same key can be used for both wrapping and decryption. This *key separation violation* is also the basis of the *wrap-and-decrypt* class of attack introduced by Clulow in [15]. Let us consider two keys $k_1^{\{\mathtt{wr},\mathtt{de}\}}$ and $k_2^{\{\mathtt{ex},\mathtt{se}\}}$ instantiated in a token, so that k_2 should not be revealed since it is sensitive. However, it can be exported outside the token wrapped with other keys since it is extractable. The attacker first asks for $\{k_2\}_{k_1}$. Then, he asks for a decryption of this blob with k_1: this is possible because k_1 has the `decrypt` attribute. Hence, the value of k_2 is revealed, which is a clear violation of the confidentiality supposedly imposed by the `sensitive` attribute.

It can be argued that patching this vulnerability is easy: the token vendor must enforce the fact that `wrap` and `decrypt` attributes are incompatible when creating new keys on the token. Some vendors have implemented this fix since the publication of [15]. However, refinements of the previous attack exist (many of them have been revealed in [17] and applied to real devices with Tookan in [12]). Some of these refinements can be combined with *key conjuring* attacks. Consider for instance two keys $k_1^{\{\mathtt{un}\}}$ and $k_2^{\{\mathtt{ex},\mathtt{se}\}}$. The attacker requests the token to unwrap a random blob twice with k_1: the first unwrapping yields a key $k_3^{\{\mathtt{wr}\}}$ with the `wrap` attribute set to `TRUE`, and the second one produces a key $k_4^{\{\mathtt{de}\}}$ with the `decrypt` attribute set to `TRUE`. The value of the sensitive key k_2 is finally revealed through the classic *wrap-and-decrypt* attack by using k_3 to wrap it and k_4 to decrypt it (since k_3 and k_4 are two instances of the same key value). It is worth noting that similar *unwrap and encrypt* attacks also exist.

Padding Oracle Attacks We assume the presence in the token of a key $k^{\{\mathtt{un},\cancel{\mathtt{de}}\}}$ which can be used for unwrapping but not for decryption. The purpose of these attacks is to obtain from a token enough information to decrypt a ciphertext encrypted with a key of the same value as k. Such a ciphertext can come from an encryption performed by the same token or some other means. These attacks exploit the possible side channels induced by the verification of some padding mechanisms: the token returns an error whenever an unwrap operation cannot be completed as the resulting blob contains a malformed padding, thus becoming an oracle. By forging many requests, the attacker is able to guess the plaintext associated to a given ciphertext. These attacks have first been conducted against RSA used in combination with the PKCS#1 v1.5 padding by Bleichenbacher in [11] and improved in [10]. It is to be noted that even if the ciphertext was produced using another padding such as OAEP, the attacker can still guess the plaintext provided the token supports PKCS#1 v1.5. Finally, similar attacks have been mounted against the CBC padding by Vaudenay in [20].

Weak Cryptographic Mechanisms Among the list of mechanisms supported by *Cryptoki*, there are obsolete or broken ciphers. For instance, a token can instantiate DES keys, provided that this mechanism is advertised as supported by the device. Furthermore, PKCS#11 allows vendors to include proprietary – and potentially unreviewed – mechanisms through “vendor defined” flags.

Reduced Key Space Attacks PKCS#11 allows to derive keys from other keys using weak mechanisms [15]. For example, `CKM_EXTRACT_KEY_FROM_KEY` allows to create a secret key from a subset of the original key bits. Hence, it is possible to ask for a key k_2 to be derived from k_1. From then, the attacker can use a reduced exhaustive search to guess k_2, consequently reducing the key space for k_1. Furthermore, if the cipher used with k_2 is weak, cryptanalysis could further reduce this key space. Similarly, `xor`-based derivation mechanisms pave the way for related key attacks.

Appendix B: Fixing the PKCS#11 API

Conflicting Attributes When combined, some attributes lead to sensitive key extraction. This is the case for the `wrap` (resp. `unwrap`) and `decrypt` (resp. `encrypt`) attributes. The idea of the patch is to prevent these attributes to be set to `TRUE` at the same time.

Sticky Attributes The previous patch is not sufficient to fix PKCS#11 against all logical attacks. A key $k^{\{\texttt{wr},\cancel{\texttt{de}}\}}$ does not violate the conflicting attribute rule, and an attacker can change its attributes to $k^{\{\cancel{\texttt{wr}},\texttt{de}\}}$ without violating the rule again. Yet, $k^{\{\cancel{\texttt{wr}},\texttt{de}\}}$ and $k^{\{\texttt{wr},\cancel{\texttt{de}}\}}$ allow the attacker to perform a variant of the *wrap-and-decrypt* attack. Sticky attributes are introduced in [18,12]: once set to `TRUE` (or unset), the attribute cannot be unset (or set). This prevents the transition from $k^{\{\texttt{wr},\cancel{\texttt{de}}\}}$ to $k^{\{\cancel{\texttt{wr}},\texttt{de}\}}$ since unsetting `wr` will be detected.

Wrapping Format The previous two patches are still incomplete to ensure that the API is immune to logical attacks. As pointed out in appendix A, *key conjuring* based attacks are possible because almost no check – excepted for padding or parity – is performed when unwrapping a key. Furthermore, the user has control over the attributes of the imported key. A solution to this issue is to use a *wrapping format* where the attributes are bound to a key value during a wrap operation.

Conflicting and *sticky attributes* comply with the PKCS#11 standard: a vendor can implement them without breaking compatibility. This is **not** the case for the *wrapping format* patch: a token implementing this patch loses interoperability with other PKCS#11 tokens. Hence, in addition to the previous three patches, Bortolozzo *et al.* introduced a new standalone patch in [12] that can be seen as an alternative.

Secure Templates In order to ensure that no logical attacks are possible, only a controlled and safe set of critical attributes are allowed and are bound to a key when it is instantiated in a token (at key generation, key unwrapping, as well as key creation). Tying a key to its usage enforces key separation in the token, at the expense of imposing that the critical attributes become read-only. Though lacking flexibility, the main advantage of this patch lies in its full compatibility with the PKCS#11 standard.

To conclude, fixing the API requires PKCS#11 token designers to either choose using the {*conflicting/sticky attributes, wrapping format*} set of patches, or the standalone {*secure template*} patch with no possible attribute modification.