Getting to know your Card: Reverse-Engineering the Smart-Card Application Protocol Data Unit

Andriana Gkaniatsou
University of Edinburgh, UK
a.e.gkaniatsou@sms.ed.ac.uk

Fiona McNeill
Heriot-Watt University, UK
f.mcneill@hw.ac.uk

Alan Bundy
University of Edinburgh, UK
a.bundy@ed.ac.uk

Graham Steel
Cryptosense, France
graham@cryptosense.com

Riccardo Focardi
Ca' Foscari University, Italy
focardi@unive.it

Claudio Bozzato
Ca' Foscari University, Italy
bozzato@dsi.unive.it

ABSTRACT

Smart-cards are considered to be one of the most secure, tamper-resistant, and trusted devices for implementing confidential operations, such as authentication, key management, encryption and decryption for financial, communication, security and data management purposes. The commonly used RSA PKCS#11 standard defines the Application Programming Interface for cryptographic devices such as smart-cards. Though there has been work on formally verifying the correctness of the implementation of PKCS#11 in the API level, little attention has been paid to the low-level cryptographic protocols that implement it.

We present REPROVE, the first automated system that reverse-engineers the low-level communication between a smart-card and a reader, deduces the card's functionality and translates PKCS#11 cryptographic functions into communication steps. REPROVE analyzes both standard-conforming and proprietary implementations, and does not require access to the card. To the best of our knowledge, REPROVE is the first system to address proprietary implementations and the only system that maps cryptographic functions to communication steps and on-card operations. We have evaluated REPROVE on five commercially available smart-cards and we show how essential functions to gain access to the card's private objects and perform cryptographic functions can be compromised through reverse-engineering traces of the low-level communication.

Keywords

Smart-card reverse-engineering, PKCS#11 low-level attacks, APDU formal modeling, APDU attacks.

1. INTRODUCTION

Smart-cards are ubiquitous and are universally considered to be secure, tamper-resistant, and trustworthy devices. They have been used to implement confidential operations such as user identification and authentication and sensitive data storage and processing. These operations involve a deliberately confidential communication between smart-cards and third-party systems. Such communication is prone to "man-in-the-middle" attacks thus rendering smart-cards vulnerable.

Sniffing the smart-card communication and consequently performing man-in-the-middle attacks has attracted a lot of attention and many tools have been proposed (*e.g.*, [7, 15]). Studies have exposed that such attacks reveal severe problems. For example, by blind-replaying a communication session one may distinguish different passports [6]. The way an attack is designed varies depending on the target *e.g.*, knowledge of the semantics of a communication session may suggest attacks like the previous one, PIN or authentication data sniffing, access to sensitive keys, execution of unauthorized operations or cloning the card. However, for an attack to be universally successful it has to deal with proprietary protocol implementations as well as with inter-industry ones, *an issue that previous studies do not address*.

Analyzing, attacking and fixing cryptographic standards used by smart-cards, such as PKCS#11, is an active area. As defined in PKCS#11 [18], cryptography is only one aspect of security and the token is only one component in a system; one must consider the environment the token operates in as well. Smart-cards supposedly offer a tamper-resistant environment for protecting sensitive data, but should also be designed so that this data remains secure. This is delegated to the communication protocols, under the assumption that these protocols are secure. Proprietary implementations create the illusion of security as they hide the card's code. A smart-card operates as a black-box: only access to the card's code may reveal the semantics of the communication protocol and its internal operations. We propose reverse-engineering the smart-card communication protocol, with respect to PKCS#11, to determine the security of that implementation. We present REPROVE, which stands for Reverse Engineering of PROtocols for VErification: an automated tool based on first-order logic, that infers the semantics of the communication, the on-card operations and their interconnection with the PKCS#11. REPROVE is implementation- and function-independent, as it deals with both inter-industry and proprietary implementations and does not require access to the card's code.

An alternative to REPROVE's automated reasoning is to manually reverse-engineer the trace. This is not straightforward and is far from a quick exercise. It requires access to the card's library and its internal calls, whereas REPROVE does not. If one tries to *guess* the meaning of the trace, without access to the card, then, given the combinatorial nature of the problem, one will need to test a considerably large number of combinations (*e.g.*, in some of the cards

ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA
© 2015 ACM. ISBN 978-1-4503-3682-6/15/12…$15.00
DOI: http://dx.doi.org/10.1145/2818000.2818020

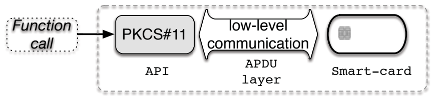

Figure 1: API and smart-card interaction when a PKCS#11 function is called.

we tested there are more than 540×86^8 possible combinations—see also Section 4) which will require a long time to decode. REPROVE does this in a matter of milliseconds.

PKCS#11 defines an Application Programming Interface (API) for smart-cards. Any API call (*i.e.*, calling a specific cryptographic function) initiates a low-level communication that manifests as a communication trace of API requests to the smart-card, as shown in Figure 1. REPROVE reverse-engineers the low-level implementation of the cryptographic protocol by automatically aligning the byte-wise decomposition of the communication trace to expected PKCS#11 calls for specific types of functionality. This process is helpful in multiple ways: (*a*) It provides the means to test smart-card implementations and discover their security vulnerabilities. (*b*) In the absence of detected vulnerabilities, it provides empirical evidence for the security of the implementation. (*c*) It can be used by the developers of smart-card technologies to test their implementations. (*d*) It can be used by the clients themselves, to test whether their card is vulnerable to attack and, therefore, fraud.

A security token, such as a smart-card, implements all the cryptographic operations internally. The token stores objects (*e.g.*, data and certificates) that can be accessed via session handles, and performs cryptographic functions. PKCS#11 is the most widely used cryptographic standard of functions like signing, encryption, decryption, *etc*. API-related attacks were first discovered in [14], followed by the exposure of the vulnerability to attacks of PKCS#11 [3, 8]. Formally analyzing security APIs and reasoning about attacks has been revisited [11, 16, 19, 20, 23] through approaches like model checking, theorem proving, customized decision procedures, or reverse-engineering for verification [3, 10, 12, 16, 22]. However, security analysis has mostly focused on the PKCS#11 itself. There has been less attention given to the implementations connected to the standard, such as the low-level communication between the on-card and the off-card applications, defined by the Application Data Protocol Unit (APDU).

The basic principles of the APDU, *e.g.*, the structure and the contents of the exchanged messages, the available inter-industry commands *etc*, are specified by the ISO 7816 standard. Precisely following the standard is not compulsory. Many smart-card manufacturers deviate from the standard under the assumption that a proprietary APDU implementation is more secure. REPROVE reverse-engineers the APDU implementation and deduces the card's functionalities, regardless of whether it is inter-industry, proprietary or a mixture of both.

A high-level description of REPROVE is shown in Figure 2. The card communicates with the reader and this communication generates a trace that we reverse-engineer. The analysis module accepts as parameters the trace and abstract models of the cryptographic protocols; and outputs how the card performs specific cryptographic functions. It models the low-level communication in first-order logic, and uses reasoning and inference over plug-in knowledge bases, which consist of APDU abstractions based on ISO 7816 and PKCS#11, to automatically reverse-engineer the model. Its algorithm parses a communication trace and uses these abstractions to draw conclusions about the semantics of the various elements of the trace, narrow-down their possible implementations and infer the card's actually executed operations.

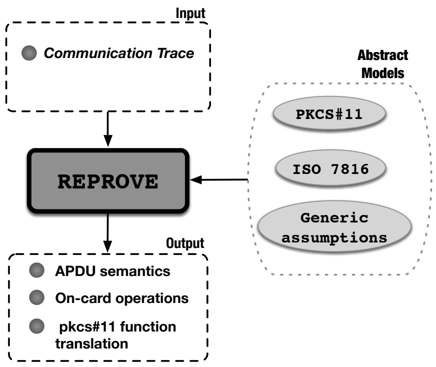

Figure 2: High-level overview of our technique.

To the best of our knowledge, this is the first work for modeling the APDU layer and formally reverse-engineering it by mapping the low-level communication to the on-card operations and to the PKCS#11 standard. The abstract models of the background knowledge do not hard-code the implementation. Instead, they offer a generic framework to automatically capture different implementations. Specific implementations are mapped to these abstractions by reasoning about the exact meaning of the input trace. Our novelty stems from not requiring access to the card's software and dealing with both inter-industry and proprietary implementations in a single setting.

Smart-card attacks If an attacker compromises the APDU level and the communication is not secure, she can have access to the card's sensitive information, *e.g.*, private keys, or data that should be encrypted but is not. [17], for instance, proposes a man-in-the-middle device to allow authentication without knowing the card's PIN by intercepting and modifying the communication between the card and terminal. A different approach is to bypass confidentiality by assuming access to the APDU buffer used to exchange data [1]. The SmartLogic tool [15] obtains full control over the smart-card communication channel for eavesdropping and man-in-the-middle attacks. These efforts motivate the need for a formal way of reverse-engineering the protocol and reasoning about smart-card security. A key assumption of all these approaches is that the implementation of the communication channel is known beforehand. However, a good number of smart-card vendors opt for a proprietary implementation, which renders the existing security analysis approaches inapplicable. A proprietary implementation of the communication, as witnessed by any security testing framework, looks like a random sequence of bits that needs to be deciphered to understand its semantics.

Reverse-engineering protocols Protocol reverse-engineering is a related area to our project, but works up to date do not satisfy the requirements of our project. For example, Polygot [4] automatically extracts protocol messages through binary analysis. Another similar approach is Prospex [9], which infers the protocol format and the corresponding state machines. Discoverer [13] reverse-engineers the protocol message formats. ReFormat [21] reverse-engineers encrypted messages, and [5] infers protocol state machines based on abstractions provided by the end users. All these projects either: (*i*) require software access, and/or (*ii*) assume known message semantics, and/or (*iii*) derive only the protocol message format without its semantics. What is central to our project is to make no assumptions about prior knowledge apart from what is publically available, *i.e.*, the inter-industry commands of ISO 7816, and map the communication to the card's internal operations.

Contributions and roadmap The main contributions of this paper are:

- Section 2 gives an overview of the PKCS#11 ISO/IEC 7816 standards. We show the discrepancies between the inter-industry and proprietary definitions of the commands covered by the standard, and how these discrepancies aggravate the problem of reverse-engineering communication traces.
- Section 3 presents the modeling of the APDU layer and its interconnection to PKCS#11. Due to space limitations we focus on sample implementations of the `C_logIn` function. REPROVE, however, is function independent as it is possible to plug in different models. Computing all potential proprietary implementations and testing them for correctness is practically infeasible, as it is a combinatorial problem. Instead, we produce a model that is based on decomposing the various functionalities of the API into finer-grained sub-functionalities and analyze how the commands of the standard can be used to implement these functionalities. We present the reverse-engineering algorithm to automatically analyze a trace of commands and group them according to their intended functionality as this has been captured by our model.
- Section 4 evaluates the accuracy of REPROVE, after reverse-engineering five commercially available smart-cards for nine cryptographic functions. Our results suggest that our methodology can be used to automatically reverse-engineer traces to detect security flaws for other PKCS#11 functions as well.
- Finally, Section 5 concludes the paper with a summary of our findings and with our future work directions.

2. BACKGROUND

2.1 RSA PKCS#11

Security APIs implement access to sensitive resources in a secure way. The design of such APIs is critical, as they have to ensure the secure creation, deletion, importing and exporting of a key from a device. Also, they are responsible for permitting the use of these keys for encryption, decryption, signing and authentication so that even if a device is exposed to malicious software the keys remain secure. The RSA PKCS#11 standard specifies an ANSI C API, called *Cryptoki*, for hardware devices that can perform cryptographic functions and store cryptographic-related and encrypted data. It aims to ‘sand-box’ an application and isolate it from the details of the underlying cryptographic device.

When an application connects to a security token it authenticates itself and initiates a session which is either public or private, defining the kind of objects the application can access and the types of operations that it can perform on them. Each session is assigned with a unique value by the Cryptoki, the session handle, to prevent a blind-replay of the same session: replaying the communication trace of the session and replicating its functionality thereby bypassing all the security mechanisms through repetition of the transmitted information. The application can then access the token’s objects *e.g.*, keys and certificates.

Objects have attributes which may be the value of the object or properties that define the allowed actions *e.g.*, `CKA_EXTRACTABLE` set to false means that the value of the object cannot be extracted from the token. PKCS#11 provides a set of functions for *e.g.*, key, token, session and object management, encryption, and decryption. When a function for a particular object is called, the token checks whether the attributes of that object allow the use of that object with respect to the called function.

The functions that we have successfully reverse-engineered are the following, according to the standard [18]:

`C_login` is called to log a user onto the token. A successful call can initiate a private session and provide user access to the token’s private objects. The function takes as inputs the session handle, the type of the user (user, or a privileged user termed a security officer), the location of the user’s PIN and the length of the PIN.

`C_generateKey` is called to generate a secret key or a set of domain parameters. It takes as inputs the session handle, the location of the generation mechanism, the location of the template for the new key’s attributes, the number of attributes in the template and the location of the handle of the new key.

`C_sign` signs data, with the signature being an appendix to the data. Its inputs are a session handle, the location of the data, the location of the signature and the length of the signature.

`C_findObjectsInit` is called to initiate a search for token and session objects that match an input template with attribute values to match. It takes as inputs the session handle, the location of the template and the number of attributes in the template.

`C_findObjects` is called after `C_findObjectsInit` and obtains the handles of the objects that match the given template. It takes as inputs the session handle, the maximum number of the returned handles, the location of the additional object handles and the location of the actual number of the returned handles.

`C_getAttributeValue` is called to obtain the value of one or more attributes of an object. It takes as inputs the session handle, the object’s handle, the location of a template with the attribute values to be obtained and the number of the template’s attributes.

`C_setAttributeValue` is called to modify the value of one or more attributes of an object. It takes as inputs the session handle, the objects’ handle, the location of the template with the attributes, the number of the attributes to change and the new values of the attributes.

`C_wrapKey` is called to encrypt a private or a secret key. It takes as inputs the session handle, the location of the wrapping mechanism, the handle of the wrapping key, the handle of the key to be wrapped, the location of the wrapped key and the length of the wrapped key.

`C_encrypt` is called to encrypt single part data. It takes as inputs the session handle, the data to be encrypted, the location of the encrypted data and the the length of the encrypted data.

`C_unwrapKey` is called to decrypt a wrapped key and creates a new private key or a secret key objects. It takes as inputs the session handle, the location of the unwrapping mechanism, the handle of the unwrapping key, the wrapped key, the length of the wrapped key, the location of the new key, the location of the template of the new key, the number of the attributes in the template and the location of the handle of the new key.

2.2 ISO/IEC 7816

ISO 7816 defines the contact smart-cards and comes into 15 parts each of them specifying different characteristics of the card. REPROVE is based on parts 4, 8 and 9 which specify the organisation of the card, security access, the commands for interchange, and the commands for security operations and card management. The communication consists of command-response pairs: a *command* is sent by the outside world to the card and a *response* is the card’s reply. A command consists of a compulsory 4-byte header, with the bytes named `Cla`, `Ins`, `P1` and `P2` and an optional body with fields `Lc`, `Data` and `Le`. The `Cla` field is the type of the command *i.e.*, inter-industry or proprietary. The `Ins` field indicates the specific command, *e.g.*, the `select_file` command. Fields `P1` and `P2` are the instruction parameters for the command, *e.g.*, the offset to write into the selected file. The `Lc` is the number of bytes of the `Data` field. The latter contains the data sent to the card. Finally, `Le` is the number of the expected (if any) response bytes. A response consists of an optional body, the response data, and a compulsory 2-byte trailer of bytes `SW1` and `SW2` encoding the expected status of the card after processing the command). A command can (*i*) send data to the card; (*ii*) expect data from the card; (*iii*) both send and expect data; or (*iv*) none of the above. The length of the response depends on the sent command. ISO 7816 specifies the inter-industry command class for the `Cla` field, the allowable values of the `Ins` field and the expected combinations of values for the `P1`, `P2` and `SW1`, `SW2` fields for all inter-industry commands/responses.

Type	Cla	Ins	P1	P2	Lc	Data	Le
inter-industry	00	84	00	00	00	00	08
proprietary	80	21	00	00	00	00	08

Table 1: Implementations of the get_challenge command.

An APDU implementation is defined according to ISO 7816 and can either be inter-industry, where the command codings are defined by the standards; or proprietary, where the developers define their own command codings. Table 1 presents an inter-industry implementation of the get_challenge command and a possible proprietary one. Each byte of the inter-industry command can be decoded, whereas the semantics of the proprietary command is unknown. The inter-industry implementation has its `Cla` field set to 00 as ISO 7816 defines, so, the remaining fields can be decoded. The proprietary one has an unknown `Cla` code, so, it is not possible to determine the semantics of the command using the ISO-based codings. REPROVE aims to infer such unknown semantics.

2.3 Threat model

Reverse-engineering the APDU layer exposes possible bad practices and vulnerabilities for both the APDU and the PKCS#11 implementation. For example, permitting the token to reveal sensitive data when it should not, an implementation that does not use protection mechanisms, *e.g.*, encryption, when transmitting sensitive data, or an implementation that performs cryptographic operations outside the token. In such cases the opportunity to steal sensitive information, *e.g.*, keys, is almost inevitable. Moreover, reverse-engineering the APDU layer provides the required knowledge to apply a wide range of attacks.

Attacker model The attacker model that we are considering is a non-legitimate user or a malicious software that control the communication layer to:

1. authenticate by compromising the `C_logIn` function and exploit their credentials to perform unauthorized operations and steal senstive data *e.g.*, keys, and/or
2. send a sequence of APDU commands that lead to sensitive information leakage by repeating the same operations initiated by the API calls during the execution of a cryptographic function.

Attacks Although performing such attacks is not part of this work, we have identified potential risks that can be addressed. Such attacks can be performed by using third party tools. Knowledge of the APDU semantics and the corresponding on-card operations may enable (*i*) manipulation of the communication to deceive the user, *e.g.*, let the user believe that a particular operation is executed while in reality a different one is taking place, (*ii*) sniffing sensitive data, (*iii*) repetition of a communication run, (*iv*) alteration of the transmitted data, (*v*) alteration of a communication run by injecting commands, (*vi*) bypassing security mechanisms, (*vii*) unauthorized access to the card's operations, or (*viii*) cloning of the card.

Additionally, having the know-how of the PKCS#11 implementation at the APDU layer may allow access to the standard's functions and the token's objects that library calls do not permit, or the application of already known PKCS#11 attacks *e.g.*, [8] by calling a function directly through the APDU layer. Also, it may allow an attacker (*i*) to compromise the `C_logIn` function to initiate a private session and gain access to the corresponding objects and operation, to steal the PIN, or even bypass that function, (*ii*) to blind-reply sessions with the token, (*iii*) to sniff sensitive data that may be transmitted during the execution of the function, (*iv*) to alter object attributes through `C_setAttributeValue`, or, (*v*) to identify the location of sensitive data.

The idea behind this work is that sufficient knowledge of the card's implementation and the APDU semantics may allow greater access than the API itself to specific PKCS#11 functions and sensitive objects, when the same access through library calls is restricted.

2.4 Reverse-engineering goals

Inferred model REPROVE takes as input an APDU trace and produces a model that describes the card's implementation of the communication protocol. Reverse-engineering addresses three different derivations of the protocol: the exchanged commands, the executed on-card operations during the communication and the interconnection with specific PKCS#11 functions, with each addressing different types of attacks. For example:

- Exchanged commands give insight into the semantics of the exchanged commands, may allow the identification of parts of transmitted data of interest to the attacker, or may gain knowledge of command semantics.
- On-card operations are mapped to a sequence of commands, so the attacker may have complete knowledge of the exact set of commands needed to execute unauthorized operations.
- Since each of the PKCS#11 function is recorded as sets of card operations, an attacker may be aware of which operations she needs to execute to perform already known PKCS#11 attacks.

Tested functions In our experiments we tested REPROVE in five commercially available smart-cards and checked for the following PKCS#11 violations: (*i*) the cryptographic function is not executed on-card, (*ii*) the cryptographic function does not respect the PKCS#11 specifications, (*iii*) misuse of session handles, (*iv*) sensitive data leakage, *i.e.*, when it should not be revealed and (*v*) lack of encryption when needed, *e.g.*, when sensitive data is transmitted.

The outputs of REPROVE can be useful to both the card suppliers and private users, in order to verify the security of logging into a particular card. REPROVE aids in understanding the security properties of the underlying implementation. Our technique is extensible and allows different formal models to be plugged-in, depending on the security properties to be checked.

3. METHODOLOGY

We have modeled the APDU reverse-engineering as an inference problem. REPROVE has a built-in inference engine that allows to plug-in different knowledge bases such as models, abstractions and specifications of the protocol. The background knowledge to our problem consists of abstract models, which need to be instantiated according to the input trace. These models are based on ISO 7816 and define: (*i*) the main properties, the restrictions and requirements of communication, (*ii*) possible implementations of the on-card operations, (*iii*) possible implementations of specific PKCS#11 functions. Such models do not hard-code the implementation of the card. They present abstractions of different functionalities that are then refined according to the input trace. The background knowledge is expressed in first-order logic as it is machine-readable and expressive enough to model the protocol's

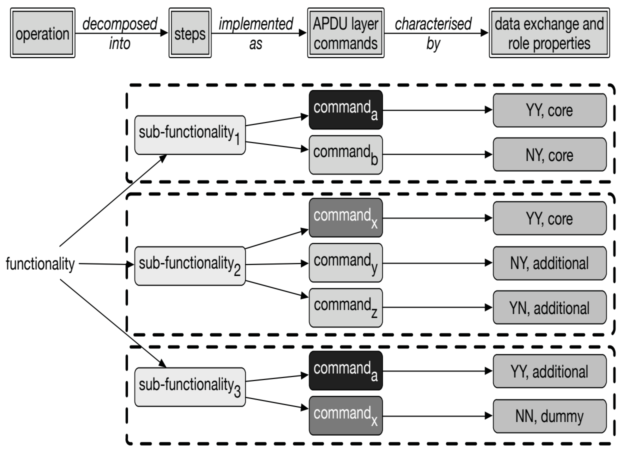

Figure 3: A single operation represents a specific functionality and it is modeled as a sequence of sub-functionalities. Each sub-functionality is further implemented as a sequence of commands. Commands are characterized by their data exchange properties and role within some particular sub-functionality.

rules. REPROVE's reverse-engineering algorithm constructs and refines the possible mappings while extracting abstract properties and functionalities.

More formally, REPROVE applies the transformation function $y(f(x))$ with $f : T^n \to I^n$ and $y : I^n \to O^m$, where T^n is an input trace of n commands, I^n is a set of n inter-industry commands and O^m is a set of m on-card operations.

3.1 Modeling the APDU layer

ISO 7816 defines different meanings for a command depending on particular fields of that command. REPROVE's background knowledge consists of all the meanings defined by the ISO and the corresponding preconditions: 49 individual commands with 122 different meanings in total. A sample of these commands are the following:

```
select                   get_data
read_binary              read_record
update_binary            erase_record
activate_file            put_data
get_response             perform_security_operation
append_record            create_file
append_file              get_challenge
verify                   activate_file
external_authenticate    mutual_authenticate
```

In Figure 3 we show a high-level description of our modeling approach. Each individual card operation (*functionality*) of the card is decomposed into a sequence of steps (*sub-functionalities*). Each step is then implemented as a sequence of APDU commands: proprietary, inter-industry, or a mix of them. The APDU commands are further characterized depending on their data exchange properties (shown, for example, as '`YY`' in the figure to indicate a command that both sends and receives data) and their role within the sub-functionality in question (core, additional, or dummy). The same command may have different data exchange properties and different roles depending the sub-functionality, *e.g.*, command$_a$ and command$_x$ in Figure 3.

APDU commands An APDU command is represented as a predicate $\mathsf{command}(Cla, Ins, P1, P2, Lc, D, Le)$ where the variables *Cla*, *Ins*, *P1*, *P2*, *Lc*, *D*, *Le* are instantiated according to the semantics of the command. A command is valid if it is: (*i*) an inter-industry command; (*ii*) a proprietary command that can be mapped[1] to an inter-industry command that does not occurre within the same trace. A command is categorized based on: (*i*) its data exchange properties; and (*ii*) the card operations.

Categorization according to data exchange properties Depending on the exchanged data, a command is assigned to one of the following categories:

(*i*) $\mathsf{command}_{nn}(Cla, Ins, P1, P2, Lc, D, Le)$: no data is sent, no data is expected,
(*ii*) $\mathsf{command}_{ny}(Cla, Ins, P1, P2, Lc, D, Le)$: no data is sent, data is expected,
(*iii*) $\mathsf{command}_{yy}(Cla, Ins, P1, P2, Lc, D, Le)$: data is sent, data is expected,
(*iv*) $\mathsf{command}_{yn}(Cla, Ins, P1, P2, Lc, D, Le)$: data is sent, no data is expected.

Variables *Lc*, *D* and *Le* define the category of a command. We defined rules that assign each command to the appropriate category. For example, if $Lc \neq 00$ and $D \neq 00$ then the command sends some data *D* with length *Lc* to the card. If *Le* is not `null`[2] then the response will be some data with length *Le*. The above is captured by the following rule:

$$\forall Cla, Ins, P1, P2, Lc, D, Le, ((\mathsf{command}(Cla, Ins, P1, P2, Lc, D, Le) \\ \wedge Lc = 00 \wedge D = 00 \wedge Le \neq \mathtt{null}) \\ \rightarrow (\mathsf{command}_{ny}(Cla, Ins, P1, P2, Lc, D, Le))$$

Categorization according to card operations The commands are further categorized depending on their role in a specific on-card operation, to:

(*i*) *Core*: the basic commands that perform the operation, *e.g.*, to create a new file `create_file` is a core command.
(*ii*) *Additional*: the commands that add extra properties to the operation, but they do not change its meaning; the same operation can be performed without them. For example, to create a file, `select` is an additional command as it merely adds information to the file creation (*e.g.*, selecting a path to create the file into) but the operation can be also performed without it.
(*iii*) *Dummy*: the commands that neither send nor expect any data. They usually just query, or check, the communication with the card. For example, a `verify` command when it does not send nor expect any data to/from the card. Such commands may occur any time during the communication and they do not affect the reverse-engineering output.

Command preconditions The preconditions of a command define: (*i*) the values of its parameters, (*ii*) the restrictions on the types of previously issued commands, (*iii*) different semantics for the same command, and (*iv*) the valid data types and file structures for that command. For instance, the common use of `read_binary` is to access the content of an elementary file (EF). Yet, if the value of parameter *P1* is between `128` and `160` then `read_binary` is used to select the EF file defined by the data field *D*. This precondition is modeled as:

$$\forall Cla, Ins, P1, P2, Lc, D, Le, ((\mathsf{command}(Cla, bo, P1, P2, Lc, D, Le) \\ \wedge P1 \in [128, 160]) \rightarrow (\mathtt{select}(\mathtt{file}, D) \wedge \mathtt{isa}(D, \mathtt{EF})))$$

Card operations We introduce a hierarchy of abstractions, the *functionality* models, which provide high-level views of different on-card operations, and the *sub-functionality* models which describe

[1]Under the condition that all preconditions are satisfiable.
[2]`Null` indicates absence of a field.

Sub-functionality	Core command set
selected	{select, read_binary}
read_data_sub	{get_data, read_binary, get_response, read_record}
data_updated	{update_binary} {update_record}
data_written	{write_binary, update_binary, write_record}

Table 2: Sample of sub-functionalities and the corresponding core commands.

the steps by which each operation is implemented. A valid (sub-)/functionality has: (*i*) all its preconditions satisfied by the commands[3] seen so far, or (*ii*) has a subset of its preconditions satisfied by the commands seen so far, but it is possible to satisfy the rest by the commands that will follow, *i.e.*, the (sub-)/functionality is partially satisfiable.

Sub-functionalities Sub-functionalities model the steps, in terms of the exchanged commands, that are needed to perform a card operation. The same sub-functionality may be performed in different ways, thus, it may have more than one model. REPROVE's background knowledge has 36 sub-functionality models. In Table 2, we give a sample of sub-functionalities and their corresponding core commands. For example, the authentication of the reader through the challenge-response protocol is expressed by the *external_authenticated*(*RD*, *D*) sub-functionality. The card issues a challenge *RD* and the reader authenticates itself by providing the corresponding response *D*. The following rule describes this:

$$\forall RD, Le, P1, P2, Lc, D, ((\mathsf{command}(00, 84, 0, 0, 0, 0, Le) \wedge \mathtt{response}(RD) \\ \wedge \mathsf{command}(00, 87, P1, P2, Lc, D, null) \\ \wedge P2 \in [128, 256]) \\ \rightarrow \mathtt{external\_authenticated}(RD, D))$$

which says that if the command $Ins = 84$ with a response of the card *RD*, is followed by the command $Ins = 87$ with its parameter P2 being between 128 and 256, then the reader has authenticated itself via a challenge-response external authentication. Furthermore, we categorize each sub-functionality as: (*i*) a *sensitive operation*: any process that we expect to deal with sensitive data, *e.g.*, the verification of a PIN; or (*ii*) a *non-sensitive operation*: any generic process over non-sensitive data, *e.g.*, the selection of a file.

Functionalities Functionalities model the on-card operations. As there are different implementation ways, each functionality consists of a set of possible core and additional sub-functionalities. For example, consider two cards Card_x and Card_y which both store data (*store_ data*). Card_x performs this operation through a *file_created* sub-functionality, while Card_y through a *data_written*. Table 3 presents a sample of the defined functionalities and their corresponding sub-functionality sets. The core sub-functionalities are extracted on the basis that at least one of them (but potentially more) are necessary for the implementation of the functionality. Additional sub-functionalities may appear in the implementation, but are not compulsory. REPROVE's background knowledge has 15 functionality models.

General rules We define rules to describe communication restrictions, card responses, file specifications and data types. For instance, the following rule requires that if some data *D* of length *Le* is expected, then the response should contain *D* and the corresponding length should be *Le*.

$$\forall Le, D(\mathsf{expected}(\mathtt{data}, Le, D) \rightarrow (\mathsf{response}(D) \wedge \mathsf{length}(D, Le)))$$

[3]Under the condition that the response is positive *i.e.*, 90 00.

Functionality	**Core** and **Additional sub-functionalities set**
store_ data	{*file_created*, *data_written*, *data_updated*} {*selected*, *read_data_sub*}
authenticated	{*challenge_sent*, *verified*, *external_authenticated*, *internal_authenticated*, *mutual_authenticated*} {*selected*, *read_data_sub*, *data_written*}

Table 3: Sample of functionalities and the corresponding sub-functionality sets.

RSA PKCS#11 models PKCS#11 models are expressed in terms of functionalities and represent our assumptions on how specific cryptographic functions might be implemented at the APDU level. These models aim to capture an abstraction of the expected on-card operations and they do not impose an implementation, but merely act as a flexible guide of the implemented functionality.

Each cryptographic function is modeled as a set of functionalities based on the PKCS#11 and the ISO 7816 specifications. For example, for the C_logIn function we expect one of the authentication operations to be a core one: a PIN/Pass-code verification or a challenge-response one. Also, an invocation of the *read_ data* functionality for authentication-related data is possible as an additional operation. Authentication is defined with respect to ISO 7816: (*i*) authentication with a PIN: the card compares received data from the outside world with internal data; (*ii*) authentication with a key: an entity to be authenticated has to prove the knowledge of a relevant key through the challenge-response procedure; (*iii*) data authentication: using internal data, secret or public, the card checks data received by the outside world. Another way is for the card to check secret internal data and compute a data element (cryptographic checksum or digital signature) and insert it to the data sent to the outside world; (*iv*) data encipherment: using secret internal data, the card enciphers a cryptogram received in a data field, or using internal data (secret or public) the card computes a cryptogram and inserts it in a data field, possibly together with other data.

3.2 Reverse-engineering algorithm

The algorithm consists of three steps, each addressing a different abstraction of the implementation: (i) the APDU semantics, (ii) the on-card operations that are executed during the communication, and (iii) the APDU implementation of a PKCS#11 function. Figure 4 shows how we restrict the search space during the three-step analysis: grey arrows indicate narrowing-down and black arrows indicate mapping; each path of a black tree is an individual mapping of the same APDU trace. The nodes appearing at the same depth represent different mappings of the same command; each path of a grey tree represents a sequence of operation steps (sub-/functionalities) and each path of a white tree represents a sequence of executed card operations (functionalities).

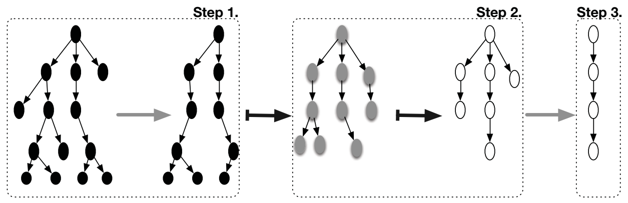

Figure 4: Reducing the search space.

Step 1: Semantics of the APDU trace. Given an input trace T^n of *n* commands, we generate a tree in which each path from root to leaf

$T_i^{n\prime}$ is a semantic mapping of the trace such that $T^n \mapsto T_i^{n\prime}$. As the exchange of the command-response pairs is sequential so is the analysis of the commands, which implies that the semantics of an unknown command heavily depend on the previous commands. Each unknown command is categorized and all corresponding mappings M are identified, which are then narrowed-down to a set P' based on precondition satisfiability. For each mapping $m \in P'$, the commands analyzed so far are grouped, and sets that fully or partially[4] satisfy any sub-functionality are considered valid. The outcome of this process is a set of valid[5] mappings M'' of each unknown command such that $M'' \subseteq P' \subseteq M$, and the set P which consists of different interpretations of T . More formally, Step 1 performs the transformation $f : f(T^n) = P^n$ where $\forall T_i^{n\prime} \in P^n : T^n \mapsto T_i^{n\prime}$.

Step 2: On-card operations. At this stage, given P^n from the previous step, the commands at each $T_i^{n\prime} \in P^n$ are grouped in all possible combinations. Each group is checked on whether there exist any sub-functionality(ies) that satisfy its preconditions. The outcome of this process is a set S^l of sub-functionalities such that $\forall {S^l}_k \in S^l \exists T_i^{n\prime} \in P^n : T_i^{n\prime} \mapsto {S^l}_k$. Then all sub-functionalities in S^l are grouped and the set of valid functionalities O^m is identified. The sub-functionalities that do not satisfy O^m are discarded along with the corresponding trace mappings. The overall step can be presented as a function y: $y(P^n) = O^m$ with $S^{l\prime} \mapsto O^m$, $S^{l\prime} \subseteq S^l$, and $P^{n\prime} \mapsto S^{l\prime}$, $P^{n\prime} \subseteq P^n$.

Step 3: APDU implementation of the PKCS#11 function. Here, the set of functionalities O^m from Step 2 is mapped to the background models of specific PKCS#11 functions, resulting in an interpretation of the communication in terms of the standard. The outcome is the APDU mapping to PKCS#11, the set of card operations that are executed during the communication $O^{m\prime} \subseteq O^m$, and the APDU trace $T_i^{n\prime} \in T^{n\prime}$ that satisfy them.

In each reverse-engineering step the low-level input (commands) evolves to abstract models (card operations). A schematic description of the transformations of the commands during the reverse-engineering process is presented in Figure 5. The trace itself goes through a sequence of transformations: from commands, to inter-industry mappings, to potential sub-functionalities, to groups of sub-functionalities into higher-level functionalities. If REPROVE is successful in providing a sequence of functionalities that describe a PKCS#11 function, then the trace is effectively reverse-engineered. This translates into a vulnerability for the card as it exposes its implementation.

Reverse-engineering algorithm The overall reverse-engineering process for a trace of commands is shown in Algorithm 1. The input to the algorithm is a list $\mathscr{T}$ of commands representing the communication trace, whereas the output is a list P of potential mappings of $\mathscr{T}$ (each mapping is a list itself) and a list O of card functionalities. The list P is initialized to [[]] which indicates that the first mapping is the empty one. Each command $c \in \mathscr{T}$ is then analyzed and depending on its value of Cla it is classified as proprietary or inter-industry. In the former case (lines 3 to 5) the values of its $\mathscr{L}_c$, $\mathscr{D}$, and $\mathscr{L}_e$ parameters are checked to categorize its data exchange properties and obtain a list $\mathscr{M}$ of potential mappings. From $\mathscr{M}$ we only keep the valid mappings (lines 5 to 6) and store them in P. The valid mappings are identified based on precondition and sub-functionality satisfiability (lines 6 to 9): for each potential mapping to an inter-industry command, we check that the preconditions of the inter-industry command are met by computing the union of the postconditions of all commands that precede it. If the preconditions of an inter-industry command are not met, the erroneous mapping is removed from M and the analysis continues to the next candidate mapping; else, we iterate over the analyzed trace so far, and look at the categorization of commands based on their role. Using this role, we group commands into different combinations that may form potential sub-functionalities. If such grouping exists, the mapping is stored in P. If c is an inter-industry command, there is only one such mapping n, so $\mathscr{M}$ is a singleton list. We search for satisfiable (sub-)/functionalities by this command and store the command in P (lines 13 to 17). At this point P consists of different mappings of the trace. Then, P is further narrowed-down based on the sub-functionality and functionality models (lines 18 to 25). For each different mapping of the trace, the commands are grouped into sub-functionalities which are then further grouped into higher-level functionalities that are added to O, all in the context of our models. If no such grouping is found for a candidate trace, the trace is removed from P. If a grouping is found, its constituents mappings are annotated accordingly to denote this. The final step of the algorithm is to further narrow-down P by matching the resulting functionalities in O with the PKCS#11 models. In the end, P will contain zero or more traces of candidate mappings. If P is empty, our reverse-engineering has failed to produce a mapping. If there is only one trace in P we say that the mapping is unique. If there are more than one candidate traces the reverse-engineering is successful, but we have only identified an abstraction of the correct mapping.

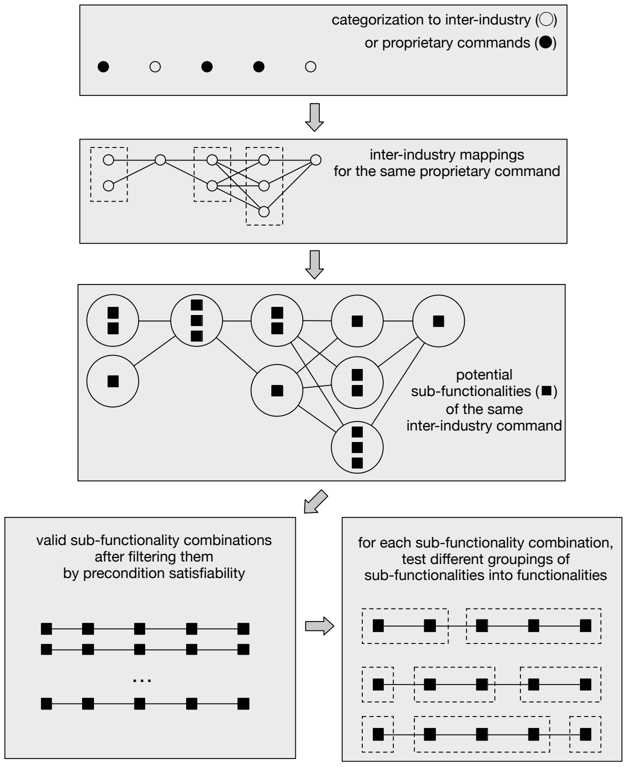
categorization to inter-industry (○)
or proprietary commands (●)
inter-industry mappings
for the same proprietary command
potential
sub-functionalities (■)
of the same
inter-industry command
valid sub-functionality combinations
after filtering them
by precondition satisfiability
for each sub-functionality combination,
test different groupings of
sub-functionalities into functionalities

Figure 5: The transformations of the APDU trace during the reverse-engineering process.

[4]Given a sub-functionality, there exists at least one core command that satisfies its preconditions.

[5]Valid here indicates that neither the ISO, nor any background model is violated.

Algorithm 1: The reverse-engineering process for a trace of commands

```
input  : List $\mathscr{T}$ of commands to be analyzed
output: Potential mappings and operation models $P$ for $\mathscr{T}$
$P = [[]]$;    $O = [[]]$;
foreach $c(Cla, Ins, P_1, P_2, \mathscr{L}_c, \mathscr{D}, \mathscr{L}_e) \in \mathscr{T}$ do
    if Ins indicates c is proprietary then
        use $\ell_c, d, \ell_e$ to extract data exchange properties $\delta$;
        $\mathscr{M}$ = list of APDU commands $c$ maps to based on $\delta$;
        foreach $m \in \mathscr{M}$ do
            $Z = \{z \mid (k \text{ precedes } m \text{ in } p) \wedge (z \in \text{postconditions}(s_k))\}$;
            if preconditions of m are not satisfied by Z then
                remove $m$ and move on to the next;
            foreach $p \in P$ do
                if a grouping of p to sub-functionalities can be found then
                    $s = p \oplus (c \mapsto m)$;    $P = P \oplus s$
    $n$ = inter-industry command $c$ maps to;    $\mathscr{M} = [n]$;
    annotate each command with its sub-functionality;
    annotate sub-functionalities with functionalities;
    $O = O \oplus$ functionalities;
    $s = p \oplus (c \mapsto n)$;    $P = P \oplus s$;
foreach $p \in P$ do
    foreach $(c \mapsto m) \in p$, potential sub-functionality of m do
        group sub-functionalities into functionalities;
        if no such grouping can be found then remove $p$ from $P$;
        else
            annotate each command with its sub-functionality;
            annotate command groups with functionalities;
            $O = O \oplus$ functionalities;
    foreach $f \in O$ do
        if $f \notin$ PKCS#11 models then remove $f$ from $O$;    remove $p$ from $P$ ;
return $P, O$;
```

4. EVALUATION

4.1 Experimental setting

We have evaluated REPROVE using five commercially available smart-cards. Each smart-card had its own API implementation, provided by the manufacturer, with none of them using an open-source implementation like opencryptoki. We were not able to test the same PKCS#11 functions for all cards. This is because in some cases the cryptographic function was executed library-side instead of token-side (*i.e.*, outside the card instead of on-card), which is violation of the standard as it allows for sensitive data, eg., keys, to be transmitted outside of the token.

Our purpose was to assess REPROVE along the following dimensions:

- Functional success: the system infers at least one model. If REPROVE is unable to infer a model then, there are two cases: (*i*) the system has failed, or (*ii*) the communication is encrypted. The latter case is not REPROVE's failure as it merely acts as a verification that the implementation is secure.
- Quality of the results: the output captures at least a high-level view of the implementation. REPROVE can produce more than one output models. We consider the following outcomes to be of high quality: (*i*) a unique model which matches exactly both with the low- and the high-level views of the implementation, *i.e.*, the exchanged commands and the on-card executed operations, and (*ii*) two or more models that exactly match the high-level view of the implementation, *i.e.*, on-card executed operations.

To address these aspects we used the standard precision and recall metrics, as defined by:

$$\text{precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$\text{recall} = \frac{\text{True Postives}}{\text{True Positives} + \text{False Negatives}}$$

where (*i*) True Positive: the outcome model suggests the correct on-card operations and the exact meaning of the APDU trace, (*ii*) False Positive: the outcome model suggests the correct on-card operations, and a partially correct meaning of the APDU trace (the APDU semantics does not exactly match with the actual implementation), (*iii*) False Negative: the outcome model suggest incorrect on-card operations and an incorrect meaning of the APDU trace.

For each smart-card we used the sniffed APDU trace as the input to REPROVE. The trace was produced when each PKCS#11 function was called. We were aware of the implementation of each smart-card from the beginning but we treated them as unknowns during the reverse-engineering. To evaluate the quality of the results we compared REPROVE's output with the actual implementation. Because of a non-disclosure agreement (NDA) we must refrain from naming the cards and revealing details of their implementation. More information about their implementations and a reproduction of our study can be obtained with appropriate permission through an NDA.

4.2 Results

Number of inferred models REPROVE performed well on all cards: it inferred at least one model for the exchanged commands, one model for the on-card operations and one model for the analyzed cryptographic function. In most cases the inferred model was unique and matched exactly with the actual implementation of the card. The results are presented in Table 4. For Card$_1$ and Card$_2$ in the case of `C_logIn` and for Card$_5$ in the case of `C_sign` REPROVE inferred the correct on-card operations but suggested two different implementation models. In all cases the correct model of the implementation existed within the suggested ones.

Security vulnerabilities suggested by the models We checked REPROVE's suggested models for *1.* security-vulnerabilities at the APDU layer, and *2.* violations of the PKCS#11 standard.

1. *APDU Level.* The first security vulnerability we checked was sensitive information leakage and identification of the location of the sensitive data. The functions that were not implemented in a secure way are the following:

i) `C_logIn`. In all cases we were able to identify the authentication data and the location that was stored. In two cases the authentication data was sent in plaintext, which consequently allowed man-in-the-middle attacks.

Moreover, the resulting models suggested the specific on-card operations that were executed during the authentication with the token. As the card uses the exact same authentication ways in each initiated session, such knowledge may enable a blind-replay attack where the attacker replays the exchanged data at specific points during communication; or, the attacker requests the same operations to be performed, in the hopes that these operations, albeit applied in a blind way, are enough to gain access to sensitive data.

ii) `C_wrapKey`. In one case the card returned the sensitive key in plaintext and the function was executed library-side.

iii) `C_generateKey`. In two cases the card returned the sensitive key in plaintext and the function was executed library-side.

Function		Precision	Recall
Card$_1$	`C_logIn`	0.5	1
	`C_wrapKey`	1	1
	`C_sign`	1	1
	`C_findObjects`	1	1
	`C_getAttributeValue`	1	1
	`C_generateKey`	1	1
	`C_getAttribute`	1	1
	`C_encrypt`	1	1
Card$_2$	`C_logIn`	0.5	1
	`C_sign`	1	1
	`C_findObjects`	1	1
	`C_generateKey`	1	1
	`C_setAttributeValue`	1	1
	`C_encrypt`	1	1
Card$_3$	`C_logIn`	1	1
	`C_sign`	1	1
	`C_findObjects`	1	1
	`C_getAttribute`	1	1
	`C_setAttribuyeValue`	1	1
Card$_4$	`C_logIn`	1	1
Card$_4$	`C_findObjects`	1	1
Card$_4$	`C_getAttributeValue`	1	1
Card$_4$	`C_sign`	1	1
Card$_5$	`C_logIn`	1	1
	`C_sign`	0.5	1
	`C_setAttributeValue`	1	1

Table 4: RSA PKCS#11 reverse-engineering evaluation results.

Function		Total B.CC	R.CC	R.SFC	R.FC	R.Model
Card$_1$	`C_logIn`	13932	24	11	3	2
	`C_wrapKey`	20	4	1	1	1
	`C_sign`	20	8	1	1	1
	`C_findObjects`	20	3	1	1	1
	`C_generateKey`	86	9	2	1	1
	`C_getAttribute`	400	6	1	1	1
	`C_encrypt`	200	4	1	1	1
Card$_2$	`C_logIn`	32000	12	4	2	2
	`C_sign`	20	24	1	1	1
	`C_findObjects`	400	3	1	1	1
	`C_generateKey`	$540x86^8$	512	69	8	1
	`C_setAttributeValue`	86	14	3	1	1
	`C_encrypt`	20	3	4	2	1
Card$_3$	`C_logIn 1`	1	1	1	1	1
	`C_sign`	1	1	1	1	1
	`C_findObjects`	1	1	1	1	1
	`C_getAttribute`	1	1	1	1	1
	`C_setAttribuyeValue`	1	1	1	1	1
Card$_4$	`C_logIn`	7396	65	39	21	1
	`C_findObjects`	7396	6	1	1	1
	`C_getAttributeValue`	54700816	3	1	1	1
	`C_sign`	86	1	1	1	1
Card$_5$	`C_logIn`	1	1	1	1	1
	`C_sign`	12322	53	7	4	2
	`C_setAttributeValue`	1	1	1	1	1

Table 5: Reduction in the number of alternative implementations during the analysis.

iv) `C_encrypt`. In one case the card returned the sensitive key in plaintext and the function was executed library-side.

2. *RSA PKCS#11.* We checked the resulting models for violations of the standard that may lead to security vulnerabilities. According to PKCS#11 each initiated session is uniquely identified by a freshly produced session handle. This handle will also be an input of each function that is called within that session. REPROVE's results showed that all tested cards violated this specification. Such departure from the standard allows blind-replaying a given session. Another problem is that trivial authentication methods were used. As the protocol is stateless, the same trivial authentication method is used before all operations over sensitive data. Therefore, by knowing how `C_login` is implemented one may employ this trivial authentication to gain access to unauthorized operations. Moreover, according to the PKCS#11 documentation *sensitive keys must not be revealed off the token in plaintext* which is another serious violation of the standard. Finally, one of the most significant findings is that in many cases the tested cryptographic functions took place library-side, instead of token-side as the intended use of smart-cards. Such misuse of the standard allows sensitive data to leave the token.

Narrowing-down the search space The reverse-engineering of proprietary APDUs is a combinatorial problem and the solution time grows exponentially with the size of the APDU trace. REPROVE uses search to advance towards the proof, and inference to block and exclude directions from the search. During the analysis, the search space is continuously restricted until the final model is produced. To demonstrate REPROVE's effectiveness on that matter, we have implemented a baseline algorithm that generates a search tree that consists of all possible mappings (including different meanings of each command) of the APDU trace, based on the category each command belongs to. Table 5 presents the command combinations produced by the baseline algorithm, termed *B.CC*. The terms *R.CC*, *R.SBC* and *R.FC* present REPROVE's total command, sub-functionality and functionality combinations respectively. *Model* is the number of final model(s) suggested by REPROVE for the specific cryptographic function. At each successive step the number of alternative implementations is progressively reduced. As Table 5 demonstrates there are cases that *B.CC* is 1. That happens when either most of the commands or all the commands of the trace are inter-industry, which suggests an 1-1 mapping. However, in some cases the search space is prohibitive, eg., in Card$_2$ for the `C_generateKey` function there are $540x86^8$ total command combinations. In such cases REPROVE narrows-down the combinations to a single mapping.

Discussion REPROVE inferred at least a high-level model of the actual implementation for all tested cards. In some cases the reverse-engineering outcome was more than one model, each one capturing the same on-card operations but differed at the implementation level. We do not consider this as a failure as REPROVE provided at least a high-level view of the implementation. However, this shows the necessity of incorporating feedback techniques to refine the reverse-engineering outcome. A straightforward technique is to send the analyzed commands to the card in order to check the validity of the results and discard suggestions that do not work.

5. CONCLUSIONS AND FUTURE WORK

We have presented REPROVE, a proof-of-concept system for automatically analyzing the low-level communication protocol of a smart-card by reasoning over a formal model of the ISO 7816 standard, regardless of the protocol's implementation. We have used REPROVE to successfully extract at least one model from each tested card and shown that, although analyzing proprietary implementations is a combinatorial problem, it is possible to leverage background knowledge to effectively reduce the search space. To the best of our knowledge, REPROVE is the first system that successfully reverse-engineers proprietary implementations. REPROVE's results can provide the necessary evidence to reason about the implementation of the protocol and discover possible security flaws. Obtaining such evidence is especially crucial, as bad implementations may lead to fraud and/or disputes between card issuer and client.

We have evaluated REPROVE by reverse-engineering the APDU implementations of five commercially available smart-cards. During these experiments we were surprised by our findings, which suggested many insecurities of the cards. All of the cards violate the specification of PKCS#11 that requires each session to be identified by a unique session handle. This specification aims at preventing blind-reply attacks and its violation automatically makes the token vulnerable. Also, the majority of the cards do not respect the security specifications of the standard by allowing sensitive information to leave the token or even performing the cryptographic functions library-side. Last but not least, we discovered implementations that allowed the cryptographic function to be executed outside the token. Regarding the implementation of the communication, in many cards sensitive data was treated as public: this part of the communication was not encrypted and the data was sent in plaintext. Such implementations are vulnerable to attacks and make the sectors that use them insecure. Detecting such violations manually is not trivial: it requires either knowledge of the semantics of the communication trace, access to the PKCS#11 library or/and to the card's code. REPROVE does not make any of these assumptions.

Reverse-engineering PKCS#11 based APIs and discovering vulnerabilities is not a new idea, *e.g.*, Tookan [3] reverse-engineers a card's API and discovers security flaws with respect to the standard. On another perspective, Caml Crush [2] acts as an attack filtering tool that sits between the PKCS#11 device and the calling application. Caml Crush considers attacks only at the API level and not at the low-level communication. Targeting the implementation of PKCS#11 at the low-level communication is a novel idea and suggests a new way of attacking the standard by bypassing the API and talking directly to the device, thereby avoiding API-level restrictions. Such attacks cannot be detected nor filtered by such tools, as they address strictly the API level. REPROVE addresses PKCS#11 attacks at the APDU layer. PKCS#11 defines specifications for secure implementations and applies to a broad range of cards. These specifications have to be addressed at the communication layer as well, *e.g.*, in session identification. REPROVE's analysis exposed several violations of the standard's specifications. Reaching these findings in the first place would not have been possible without reverse-engineering. We therefore believe our approach cuts across all layers of the PKCS#11 implementation and provides a blueprint that can be applied to other models and protocols as well.

6. REFERENCES

[1] G. Barbu, C. Giraud, and V. Guerin. Embedded eavesdropping on java card. In *SEC*, pages 37–48, 2012.

[2] R. Benadjila, T. Calderon, and M. Daubignard. Caml crush: A pkcs# 11 filtering proxy. In *Smart Card Research and Advanced Applications*, pages 173–192. Springer, 2014.

[3] 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, 2010.

[4] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 317–329, 2007.

[5] C. Y. Cho, D. Babi ć, E. C. R. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 426–439, 2010.

[6] T. Chothia and V. Smirnov. A traceability attack against e-passports. In *Financial Cryptography*, volume 6052 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2010.

[7] O. Choudary. The Smart Card Detective: a hand-held emv interceptor, University of Cambridge, Computer Laboratory, Darwin College, MPhil thesis, 2010.

[8] J. Clulow. On the Security of PKCS#11. In *IN Proceedings Of the 5TH International Workshop on Cryptographic Hardware and Embedded Systems*, CHES, pages 411–425, 2003.

[9] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 110–125, 2009.

[10] V. Cortier, G. Keighren, and G. Steel. Automatic analysis of the security of XOR-based key management schemes,. In O. Grumberg and M. Huth, editors, *TACAS 2007*, number 4424 in LNCS, pages 538–552, 2007.

[11] J. Courant and J.-F. Monin. Defending the Bank with a Proof assistant. In *In Proceedings of the 6th International Workshop on Issues in the Theory of Security*, pages 87–98, 2006.

[12] J. Courant and J.-F. Monin. Defending the bank with a proof assistant. In *WITS 2006*, 2006. In WITS proceedings.

[13] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 14:1–14:14, 2007.

[14] S. R. D. Longley. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1), 1992.

[15] G. de Koning Gans and J. de Ruiter. The SmartLogic Tool: Analysing and Testing Smart Card Protocols. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 864–871, 2012.

[16] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *CSF*, pages 331–344, 2008.

[17] S. J. Murdoch, S. Drimer, R. Anderson, and M. Bond. Chip and pin is broken. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 433–446. IEEE Computer Society, 2010.

[18] RSA Security INC. v2.20. PKCS#11: Cryptographic Token Interface Standard, 2004.

[19] G. Steel and A. Bundy. Deduction with xor constraints in security api modelling. In *In Proceedings of the 20th International Conference on Automated Deduction, volume 3632 of LNCS*, pages 322–336, 2005.

[20] E. Tsalapati. *Analysis of PKCS#11 using AVISPA tools*. Master thesis, University of Edinburgh, 2007.

[21] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 200–215, 2009.

[22] P. Youn, B. Adida, M. Bond, J. Clulow, J. Herzog, A. Lin, R. Rivest, and R. Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, 2005.

[23] P. Youn, B. Adida, M. Bond, J. Clulow, J. Herzog, A. Lin, R. L. Rivest, and R. Anderson. Robbing the bank with a theorem prover. Technical report, Univerisity of Cambridge, 2005.