Secure recharge of disposable RFID tickets*

Riccardo Focardi[1] and Flaminia L. Luccio[1]

DAIS, Università Ca' Foscari Venezia, Italy
{focardi,luccio}@dsi.unive.it

Abstract. We study the Mifare Ultralight cards in detail, and we present a new secure method for the recharge of these RFID disposable tickets that also extends to the case of multiple resources on a single device. We specify a formal but yet realistic semantics of these cards, and we also define a simple imperative language suitable to program secure APIs. In fact, the language is provided with a type-system enforcing security properties on resources stored in the card.

1 Introduction

In the last years, *Radio Frequency Identification* (RFID) systems have been widely employed in the transport payment systems of different countries. An example are the *Mifare Ultralight cards* (MU) [2], which are RFID cards, produced by NXP Semiconductors, used, e.g., in cheap disposable paper tickets, for the metro networks of Amsterdam, Rotterdam, Moscow and Venice [1, 6, 8]. Although these RFID cards are still widely used, they have been subject to many different attacks (see [6, 8, 9]). In particular, these cards are very simple devices with a small storages and a few security mechanisms. Interestingly, the attacks in the literature are not due to a flaw in such mechanisms but, instead on programming errors. In fact, MU cards do not provide any security API to the programmer, ensuring that the offered mechanisms are used in a correct way. The card API consists of two operations: *read* and *write*, and it is up to the programmer to execute them so that data are secured as desired.

One of the main motivation of this work is to provide a tool for developing secure APIs to MU cards. The imperative language we propose is simple, but still expressive, and can be also type-checked. It is suitable for programming the API layer between an application and the cards. A theorem proves that well-typed APIs enforce interesting security properties on the cards, such as the impossibility of reusing tickets. By understanding more deeply the security aspects of these devices, we have also found a method to extend the cards from the typical use-case that consists of a card containing a unique resource, and no recharging option. These cards use an irreversible counter, named OTP, to decrement available tickets that prevents any recharge, if used as suggested by the producer [2]. Here we propose a new way of using the OTP: we use the

* Work partially supported by the RAS Project "*TESLA: Techniques for Enforcing Security in Languages and Applications*".

standard read/write area of the card to store the actual (possible multiple) resource counters. The OTP is instead used only to track the generic event of consuming a resource, but it does not directly correspond to the number of resources charged on the card. A *Message Authentication Code* (MAC), ensures the integrity of the actual counters with respect to the unique card ID and the OTP, so to avoid the card cloning or the restoring some already used resources.

In summary, (*i*) we give a formal semantics of the MU cards which is extremely close to their real functioning (section 2); (*ii*) based on this, we propose a very simple but still expressive language for the construction of realistic APIs that interface with them (section 3). Moreover, we give some advices on how a secure application based on MU cards should be built. In our opinion, a key point is to separate the application from the APIs, and also to use a simple language for their representation. This is what we have done in this paper, and this has permitted us to easily check the security of this API-level; (*iii*) we propose a formal typing of this language that permits to prove that the APIs have the property that the resources stored on the card never exceeds that have been paid and not yet used, i.e., no attacker can steal or double-use resources (section 4).

Related work. There are many different attacks to the MU cards, many of which have been presented in [6] and that are here listed. Consider a user travelling with this card in a metro: at the check-in the card has to be validated at a ticket counter, then the user travels, finally, he validates again the card at a counter while checking-out. One vulnerability relies on the fact that, the only information changed at the check-out is stored in a user-writeable area. A user could thus save the transaction stored at the purchase, use the ride, check-out and overwrite the check-out transaction with the saved data. Since there is no on-line database that checks the card data, thus a user may be able to check-out an unlimited number of times during the time-frame offered by the check-in. The authors propose to solve this by adding a check-out counter that has to be compared with the check-in OTP value (which increases at each ride).

Another vulnerability shown in [6], is due to the operation order. Whenever a user travels, the system first checks if there are still rides left on the card, it increments their number, and then stores the check-in transaction on the card. By storing a backup of some particular pages of the card just before a trip, it is possible to restore them after the trip so to qualify a user to travel, up to the card expiration date, at the cost of a unique single ride. Another attack was presented in 2008 by Roel Verdult, in [9]. He implemented a *Ghost* device which was capable of intercepting sensible information (sent in plaintext between the reader and the MU card) and then emulate and clone MU cards.

The model presented in this paper is the first one that formalizes, via a linear type-and-effect system, the MU cards, and in particular the consumption and recharge of resources. Regarding formal models for APIs, we mention two works in a different setting: Steel [7] proposed the first formal analysis for the discovery of new attacks and possible patches to the APIs used for the *Personal Identification Number* (PIN) processing in the bank cash machine network. In [4], the authors presented a language-based environment for analysing the PIN

Byte number	0	1	2	3	Page
ID	ID0	ID1	ID2	Check1	0
ID	ID3	ID4	ID5	ID6	1
Check/Lock	Check2	Internal	Lock0	Lock1	2
OTP	OTP	OTP	OTP	OTP	3
Data	user memory	user memory	user memory	user memory	4
Data	user memory	user memory	user memory	user memory	5
Data	user memory	user memory	user memory	user memory	...
Data	user memory	user memory	user memory	user memory	15

Lock0	$lock_7$	$lock_6$	$lock_5$	$lock_4$	$lock_3$	$block_1$	$block_2$	$block_{OTP}$
Lock1	$lock_{15}$	$lock_{14}$	$lock_{13}$	$lock_{12}$	$lock_{11}$	$lock_{10}$	$lock_9$	$lock_8$

Table 1. The MU card Memory and Lock bits

processing API, formally modelled existing attacks, proposed some fixes and proved them correct via a type-system.

Type-and-effect systems have already been applied in the context of security, e.g., in [5], however these types are non-linear and they have been used for the authenticity of security protocols and not for the analysis of security APIs. The work that is most related to the present one is [3] where the authors propose a linear type system that combines affine logic, refinement types, and types for cryptography, in order to support authorization policies, which are used to determine whether a resource in a system should be accessed or not. Though there are similarities, the system is applied at the protocol level, moreover it does not address issues which are very relevant in our context, such as data integrity. As a future work it would be interesting to integrate the two approaches.

2 Modelling Mifare Ultralight cards

We briefly recall the technical characteristic of the Mifare Ultralight cards. For more details refer to [2, 6]. MU cards are low cost, contactless RFID smart cards compatible with the ISO/IEC 14443A standard, used as a single ride or short term use tickets. These cards do not need batteries as they are recharged by the electromagnetic field generated by a reader to which they may connect (with no encryption) in High Frequency, and with Operating frequency of 13.56 MHz, up to a distance of 100 mm. Different cards may simultaneously connect to the same reader and are distinguished by an anticollision mechanism that relies on a distinguished Unique Identifier (ID) assigned to each card, ensuring that no data corruption occurs during the transactions between each card and the reader.

The memory of these cards contains different type of data, some of which is only readable. Table 1 describes how this 16 pages for 4 bytes memory is organized: The card ID is composed of 7 bytes, here denoted by ID0, ID1, ID2, ID3, ID4, ID5, ID6, and is stored in the first two pages: bytes ID0, ID1, ID2 are in page 0 followed by a check byte Check1 (a bitwise XOR of ID0, ID1, ID2), bytes ID3, ID4, ID5, ID6 are in page 1. Another check byte Check2 is placed at the beginning of page 2, (a bitwise XOR of ID3, ID4, ID5, ID6) followed by an internal byte (whose use is unknown). This part of the memory is programmed by the IC manufacturer and is only readable.

The rest of the memory is in a read/write mode. Pages 4 to 15 are data pages. An exception to the read/write operation is provided by the last two bytes of page 2, which are called the *lock bytes*, and are denoted Lock0 and Lock1, and the four bytes of page 3, called OTP. The bits of these bytes are initially set to 0 and while they are changed to a 1 their value is 'frozen' and cannot be reverted to 0. As we will explain later in detail , this is implemented by executing a write operation as a bitwise OR between the actual value and the one to be written. If a 1 is already in one bit, the result of the OR will always be a 1.

Formally, let $lock_i$ denote the value of the lock bit for the i-th page, then $lock_i = 1$ corresponds to the locking of page i, i.e., page i can only be read. As shown in Table 1, Lock1 is composed of 8 bits, i.e., $lock_{15}, \dots, lock_8$ used to lock pages from 15 to 8. Lock0 is composed of 5 bits, i.e., $lock_7, \dots, lock_3$, used to lock pages from 7 to 3, and three block-locking bits $block_1$, $block_2$, $block_{OTP}$. These bits are used to freeze also the locking configuration of the 0 bits of pages 15-10, 9-4 and 3, respectively, i.e., preventing some pages from being locked. Page 3 is the so called *One Time Programmable* (OTP) area, which is typically used as a ticket counter. As we have previously said there are 4 bytes, set to 0 after production. Each bit can only be transformed to a 1. For example, 11111111 11111111 10000000 00000000 represents a ticket with 15 rides on. Each time the ticket is used the number of 1's in the OTP is increased.

The model. From the above description we can easily derive that MU cards $\mathcal{C}$ can be modelled as a mapping from a page index to a value. We consider the values $v ::= \mathsf{w} \mid MAC_k(v_1, \dots, v_n)$, i.e., 4-bytes long words w, and terms representing MACs under key k of values $v_1, \dots, v_n$. Keys k are picked from a special set $\mathcal{K}$, disjoint from values. Formally, $C \in \mathcal{C}$ is such that $C : i \mapsto v$ with $i \in [0, 15]$. Recall that the card ID is stored in page 0 and 1. Its uniqueness is ensured by requiring that $C', C'' \in \mathcal{C}$ implies $(C'(0) \neq C''(0)) \vee (C'(1) \neq C''(1))$. Notice that the presence of the check bytes (Check1 and Check2) is immaterial for uniqueness, as they are computed from the ID bytes.

As it is mentioned in previous section, MU cards provide a mechanism to solve collisions so that, even in the presence of multiple cards in front of the reader, the application correctly communicates with a single card, i.e., it does not mix commands directed to different cards. We can thus focus on the only operations allowed on MU cards: *read* and *write*. We model these operations in a memory-mapped fashion, assuming to have a memory with sixteen special locations used for I/O with cards. Formally, a memory $M : x \mapsto v$ is a mapping from a variable x to values v. The special variables $p_0, \dots, p_{15}$ are devoted to I/O with cards meaning that any read/write operation transfers information between those variables and the card pages.

The semantics of the I/O operations is given in Table 2 in terms of reductions between *configurations* $\langle \mathsf{c}, M, C \rangle$, representing the execution of command c on memory M and card C. We use ϵ to represent the 'consumption' of the command.

$$\begin{array}{lll}
\text{(read)} & \langle \mathsf{read}(i), M, C\rangle \rightarrow \langle \epsilon, M\{p_i \mapsto C(i)\}, C\rangle & \text{if } 0 \leq i \leq 15 \\
\text{(write)} & \langle \mathsf{write}(i), M, C\rangle \rightarrow \langle \epsilon, M, C\{i \mapsto \mathit{store}(M, C, i)\}\rangle & \text{if } (i = 2) \text{ or} \\
 & & (2 < i \leq 15 \text{ and } \mathit{lock}_i = 0)
\end{array}$$

$$\mathit{store}(M, C, i) = \begin{cases} C(i) \mid (M(i) \ \& \ \mathit{mask}_1 \ \& \ \mathit{mask}_2 \ \& \ \mathit{mask}_{OTP}) & \text{if } i = 2 \\ C(i) \mid M(i) & \text{if } i = 3 \\ M(i) & \text{otherwise} \end{cases}$$

$$\begin{array}{rll}
\mathit{mask}_1 = & 00000000\ 00000000\ 11111111\ 00000011 & \text{if } \mathit{block}_1 = 1 \\
\mathit{mask}_2 = & 00000000\ 00000000\ 00001111\ 11111100 & \text{if } \mathit{block}_2 = 1 \\
\mathit{mask}_{OTP} = & 00000000\ 00000000\ 11110111\ 11111111 & \text{if } \mathit{block}_{OTP} = 1 \\
 & & \\
= & 00000000\ 00000000\ 11111111\ 11111111 & \text{otherwise (for each mask)}
\end{array}$$

Table 2. Semantics of read/write operations on MU cards.

Our semantics is very close to the real behaviour of cards. The main difference is that our cards cannot fail. Failures can nevertheless be observed by the fact that the card is stuck, e.g., if we try to read out of the page index range.

A read operation $\langle \mathsf{read}(i), M, C\rangle \rightarrow \langle \epsilon, M\{p_i \mapsto C(i)\}, C\rangle$ copies the content of the device page $C(i)$ of card C into the relative memory variable p_i. Write operations $\langle \mathsf{write}(i), M, C\rangle \rightarrow \langle \epsilon, M, C\{i \mapsto \mathit{store}(M, C, i)\}\rangle$ are more involved since the way data are written depends on the specific page i. First, notice that pages 0 and 1 are read-only: writing on those pages is disallowed. For pages $2 \leq i \leq 15$, the write operation is formalized through the function $\mathit{store}(M, C, i)$ which picks the value $M(i)$ from the memory and returns the actual data to be stored in the card at page $C(i)$, provided (for pages $2 < i \leq 15$) that $\mathit{lock}_i = 0$, i.e, the page is not locked. For the lock bits in page 2 and the OTP in page 3, notice that the value is always bitwise or-ed (symbol '|') with the actual value on the card. This has the effect of making bits set to 1 sticky: once set to 1 they will never be set back to 0. For example, if we ask to write 00000000 00000000 00000000 00111111 to the OTP which contains value 00000000 00000000 00000000 11110000, the actual value written will be the bitwise or of the two, i.e., 00000000 00000000 00000000 11111111 (refer to case $i = 3$ of the store function). Symbolic MACs, when appearing in a bit-wise or operation are considered as value 0, i.e., they do no affect the result. This does not limit the attacker capabilities as he can choose to write whatever value he wants to the cards. Well-typed programs will never write MACs in these pages.

Finally, consider the attempt to execute a write on page $i = 2$. This write is also prevented by the block-locking bits (that are the last three bits in the third byte of page 2), which have thus to be checked. This is formalized through suitable bit-masks that are bitwise and-ed (symbol '&') with the value $M(i)$ to be stored. All masks contain the first two bytes set to 0 as the write operation should not change the Check2 and Internal values. Assume now block_1 is set, then the write over the locking bits of pages 10 to 15 will be prevented. This is captured by the corresponding mask mask_1 = 00000000 00000000 11111111 00000011. For example, if we try to lock pages from 3 to 15 by writing the

$$\langle \mathsf{skip}, M, C\rangle \rightarrow \langle \epsilon, M, C\rangle \qquad \langle \mathsf{a}, M, C\rangle \xrightarrow{\mathsf{a}_C} \langle \epsilon, M, C\rangle \qquad \frac{\langle \mathsf{c}_1, M, C\rangle \xrightarrow{\ell} \langle \epsilon, M', C'\rangle}{\langle \mathsf{c}_1;\mathsf{c}_2, M, C\rangle \xrightarrow{\ell} \langle \mathsf{c}_2, M', C'\rangle}$$

$$\frac{\langle \mathsf{c}_1, M, C\rangle \xrightarrow{\ell} \langle \mathsf{c}_1', M', C'\rangle}{\langle \mathsf{c}_1;\mathsf{c}_2, M, C\rangle \xrightarrow{\ell} \langle \mathsf{c}_1';\mathsf{c}_2, M', C'\rangle} \qquad \frac{e \downarrow_M v}{\langle x := e, M, C\rangle \xrightarrow{\hat{v}} \langle \epsilon, M\{x \mapsto v\}, C\rangle}$$

$$\frac{e \downarrow_M \mathit{true}}{\langle \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2, M, C\rangle \rightarrow \langle \mathsf{c}_1, M, C\rangle} \qquad \frac{e \downarrow_M \mathit{false}}{\langle \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2, M, C\rangle \rightarrow \langle \mathsf{c}_2, M, C\rangle}$$

Table 3. Semantics of the API-level language.

following four bytes 00000000 00000000 11111000 11111111 to page 2, this will be and-ed with the above mask giving 00000000 00000000 11111000 00000011, i.e., only pages from 3 to 9 will be locked (assuming that block_2 is not set).

Note that real read operations return 4 pages. This is useful for performance but it has no impact on security, we thus prefer to model a much simpler single page read, however, extending the semantics to 4 pages would be straightforward.

3 The API-level language

We define a simple imperative language for specifying the part of the application interacting with the cards. It would be desirable to have this part separated from the actual application, by providing a simple API for recharging and checking the tickets. In fact, this is the critical part of the application: any flaw in this code might lead to the possibility of cloning tickets or reversing ticket states, and consequently allowing malicious users to travel for free [6]. Note that the language we define is, on purpose, very simple so to allow formal reasoning but still allowing to program the relevant APIs needed from the higher-level application. Since security of the MU cards is completely in charge of the application, this 'layered' approach is, in our opinion, necessary to clearly separate the critical code interacting with the card from the rest of the application.

Our language builds on top of the card API: it admits read and write operations on cards, assignment of expressions e to variables and if-the-else branches. Formally $\mathsf{c} ::= \mathsf{read}(i) \mid \mathsf{write}(i) \mid \mathsf{skip} \mid \mathsf{a} \mid \mathsf{c}_1;\mathsf{c}_2 \mid x := e \mid \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2$ where annotations $\mathsf{a} ::= \mathit{produce}(\mathsf{R}) \mid \mathit{consume}(\mathsf{R})$ represent the production and consumption of a resource $\mathsf{R} \in \mathcal{R}$. They do not have any semantic import in the language apart from exhibiting a label, useful to define security properties. We do not specify the possible expressions in detail. We assume MACs under key $k \in \mathcal{K}$ can only be generated by the special expression $MAC_k(\ldots)$ and we write $\mathcal{K}(e)$ to note all MAC keys k syntactically occurring in e. The semantics of read/write operations is given in the previous section. The semantics of the remaining commands is largely standard and is given in table 3. Label $\hat{v}$ is defined as $MAC_k(\ldots)_C$ if e is $MAC_k(\ldots)$ and is empty otherwise.

The attacker model is tailored to the specific setting. We assume a worst-case scenario where the attacker has control of all the cards $C \in \mathcal{C}$ and has a snapshot of all previous card states. The only thing it does not possess is the key

K used for generating/checking MACs. The attacker can thus run whatever code he wants on the card, meaning that he can delete/copy/modify (writable) card pages, but he cannot forge new MACs under key K. Of course, read and write operations will respect card semantics. E.g, the OTP 1's can never be reverted to 0. We also assume the attacker can use/recharge the cards (even the one he has tampered with). This amounts to saying that he can run *trusted* API code containing the MAC key K. In real applications, this code is run inside some secure, protected hardware (think about the validating machines of a metro). Trusted code, which we will assume to be part of a set $\mathcal{T}$, can be thus arbitrarily run on the cards but it cannot be tampered with.

An attacker configuration is a pair $\langle M, \mathcal{C}\rangle$ consisting of a memory M and a set of cards $\mathcal{C}$. The attacker can execute untrusted code (as far as K and annotations does not appear in the code), or API code in $\mathcal{T}$ on each of the cards:

$$\text{(attackerAPI)}\ \frac{c \in \mathcal{T} \qquad \langle \mathsf{c}, M, C_i\rangle \xrightarrow{\gamma}{}^{*} \langle \mathsf{c}', M', C_i'\rangle}{\langle M, \{C_1, \dots, C_i, \dots, C_k\}\rangle \overset{\gamma}{\rightsquigarrow}_{\mathcal{T}} \langle M', \{C_1, \dots, C_i', \dots, C_k\}\rangle}$$

$$\text{(attackerUn)}\ \frac{K \notin \mathcal{K}(\mathsf{c}) \qquad \mathsf{a} \notin \mathsf{c} \qquad \langle \mathsf{c}, M, C_i\rangle \xrightarrow{\gamma}{}^{*} \langle \mathsf{c}', M', C_i'\rangle}{\langle M, \{C_1, \dots, C_i, \dots, C_k\}\rangle \overset{\gamma}{\rightsquigarrow}_{\mathcal{T}} \langle M', \{C_1, \dots, C_i', \dots, C_k\}\rangle}$$

We write $\langle M, \mathcal{C}\rangle \overset{\gamma}{\rightsquigarrow}{}^{*}_{\mathcal{T}} \langle M', \mathcal{C}'\rangle$ to note a, possibly empty, sequence of attacker executions $\langle M, \mathcal{C}\rangle \overset{\gamma_1}{\rightsquigarrow}_{\mathcal{T}} \langle M_1, \mathcal{C}_1\rangle \overset{\gamma_2}{\rightsquigarrow}_{\mathcal{T}} \dots \overset{\gamma_n}{\rightsquigarrow}_{\mathcal{T}} \langle M', \mathcal{C}'\rangle$ with $\gamma = \gamma_1\gamma_2 \dots \gamma_n$.

Example 1 (Double usage of a card ticket). We show a simple example of an API for consuming tickets and we present an attack to it. We assume that page 4 contains a counter of the ticket resource $\mathsf{R_T}$ on the card, pages 5 and 6 respectively contain the bus identifier and a timestamp, while page 7 is a message authentication code (MAC) of the card ID (p_0 and p_1), together with the lock bytes (p_2), the OTP (p_3), and the timestamp (p_6). We let $\mathsf{read}(i : j)$ and $\mathsf{write}(i : j)$, with $i < j$, respectively denote $\mathsf{read}(i); \mathsf{read}(i + 1); \dots ; \mathsf{read}(j)$ and $\mathsf{write}(i); \mathsf{write}(i + 1); \dots ; \mathsf{write}(j)$. We also write $consume(\mathsf{R_T})^n$ to denote n instances of $consume(\mathsf{R_T})$. The attacker first executes its own code by reading pages 4 to 7 in the card any by copying them in the card memory in a read/write area. Then, it executes an API program of $\mathcal{T}$. To check card integrity, the MAC is recomputed and checked on the value p_7 read from the card. Once the card is known to be valid, fields p_5 and p_6 are updated to store the bus ID and a timestamp, using two expressions that we do not specify in detail, and the MAC is recomputed. All the modified pages are written to the device. Finally, the n annotations

```
read(4 : 7);
$x_4 := p_4; x_5 := p_5; x_6 := p_6; x_7 := p_7;$
---------------------------------------------
read(0 : 7);
if ($MAC_K(p_0, p_1, p_2, p_3, p_4, p_6) = p_7$) then
    $p_4 := p_4 - n;$
    $p_5 := BUS\_ID();$
    $p_6 := TIMESTAMP();$
    $p_7 := MAC_K(p_0, p_1, p_2, p_3, p_4, p_6);$
    write(4 : 7)
    $consume(\mathsf{R_T})^n;$
else
    skip
---------------------------------------------
$p_4 := x_4; p_5 := x_5; p_6 := x_6; p_7 := x_7;$
write(4 : 7)
```

$\mathit{consume}(\mathsf{R_T})^n$ indicate that n resources $\mathsf{R_T}$ have been consumed from the card, and the calling application can make use of them, e.g., n travelers are using n tickets from the same cards. Then the attacker writes again in the memory the values stored at the beginning, with the original resources. To prevent this attack, it is necessary that the OTP (p_3) is incremented. The OTP, in fact, is irreversible.

As we have seen, the attacker aims at obtaining more resources than the ones produced. In order to count such resources we consider the events $\mathit{produce}(\mathsf{R})$ and $\mathit{consume}(\mathsf{R})$, mentioned above, that are exhibited as labels $\mathit{produce}(\mathsf{R})_C$ and $\mathit{consume}(\mathsf{R})_C$ of the semantic reduction $\rightarrow$, with C representing the status of the card on which the API program is running. Intuitively, we count the number of $\mathit{produce}(\mathsf{R})$ and we subtract the number of $\mathit{consume}(\mathsf{R})$ to obtain the residual instances of resource R: In the following, we write $\mathsf{Id}(C)$ to note the pair $(C(0), C(1))$ and $\mathsf{Otp}(C)$ to note $C(3)$. We also write $\gamma\downarrow_C$ to note the subsequence of γ only containing labels in set $\{\ell_{C'} \mid \mathsf{Id}(C') = \mathsf{Id}(C)\}$, i.e., relative to card C.

Definition 1. *Let γ be a sequence of labels. Then, we define $\mathit{count}(C, \gamma, \mathsf{R}) = |\{\mathit{produce}(\mathsf{R})_{C'} \in \gamma\downarrow_C\}| - |\{\mathit{consume}(\mathsf{R})_{C'} \in \gamma\downarrow_C\}|$.*

4 Type-based analysis

In order to statically type-check APIs we need to know how card data are organized and in particular where crucial data such as the counter and the MAC are stored, and which pages the MAC actually authenticates. In real applications this is defined once for all, and is used for all MU cards.

We consider types $\tau ::= \mathsf{Data} \mid \bullet \mid \mathsf{R} \mid \mathsf{Mac}[\mathsf{i_0}, \ldots, \mathsf{i_m}] \mid \mathsf{Id} \mid \mathsf{Lock} \mid \mathsf{Otp}$. Intuitively, type Data is for generic data, $\bullet$ is a special type for variables waiting to be checked through a MAC and synchronized with the actual values on the card, R is a resource counter and ranges in the set of resources $\mathcal{R}$, $\mathsf{Mac}[\mathsf{i_0}, \ldots, \mathsf{i_m}]$ is for MACs that take as input the values of pages $i_1, \ldots i_m$, Id is for the 2 pages containing the unique identifier, Otp and Lock are for the OTP and lock bytes.

In order to track linear production and consumption of resources or events such as the increment of the OTP we use effects $\mathsf{e} ::= \mathsf{R} \mid \mathsf{iOtp} \mid \mathsf{W_0} \mid \ldots \mid \mathsf{W_{15}}$. In particular, R represents the effect of producing a resource R, effect iOtp the increment of the OTP, and effect W_i requires page i-th to be written and it is used to track the change of variable p_i that has to be written back to the card.

A typing environment Γ is an unordered list of effects e and of bindings $x : \tau$ and $i : \tau$ among variables and their types, and page indexes and their types, respectively. Bindings must be unique, that is we can never have $x : \tau$ and $x : \tau'$ simultaneously in $\mathit{\Gamma}$. Effects, instead, may appear in multiple instances. When this holds we say that the environment is well-formed, written $\Gamma \vdash \diamond$. Mixing types and effects into the same environment is useful in order to simplify the typing rule notation. We will write $\Gamma(z)$ to note the (unique) type of variable/index z in Γ, and $\Gamma\{x : \tau'\}$ to change the binding of x to the new type τ' in Γ. Finally, we will write $\mathit{eff}(\Gamma)$ to denote the unordered list of all the effects in Γ.

Types of card page indexes are subject to the following constraints.

Definition 2 (card page types). *Let $\Gamma \vdash \diamond$. We say that Γ is a valid card page typing environment, written $\Gamma \vdash_C \diamond$, if the following properties hold:*

1. $\Gamma(i) \neq \bullet$ *for* $i = 0, \ldots, 15$ *and* $\Gamma(i) \neq \mathsf{Id}, \mathsf{Lock}, \mathsf{Otp}$ *for* $i = 4, \ldots, 15$
2. $\Gamma(i) = \mathsf{R}$ *implies* $\nexists k \neq i$ *such that* $\Gamma(k) = \Gamma(i)$
3. $\Gamma(i) = \begin{cases} \mathsf{Id} & \textit{if } i = 0, 1 \\ \mathsf{Lock} & \textit{if } i = 2 \\ \mathsf{Otp} & \textit{if } i = 3 \end{cases}$
4. *Let* $i_1, \ldots, i_k$ *be the set of indexes* $\{i \mid \Gamma(i) \in \mathcal{R}\}$ *ordered from the smallest to the biggest. Then,* $\Gamma(i) = \mathsf{Mac}[\ldots]$ *implies* $\Gamma(i) = \mathsf{Mac}[0, 1, 2, 3, \mathsf{i}_1, \ldots, \mathsf{i}_k, \ldots]$

Intuitively, (1) forbids the use special type $\bullet$ for card pages and ensures that only the first 4 pages can be typed $\mathsf{Id}, \mathsf{Lock}, \mathsf{Otp}$; (2) states that each $\mathsf{R} \in \mathcal{R}$ can be given to a unique page; (3) as expected, types $\mathsf{Id}, \mathsf{Lock}, \mathsf{Otp}$ are given to the first four pages; (4) any MAC is required to at least contain the card ID, the lock bytes and the OTP, and any resource counter $\mathsf{R} \in \mathcal{R}$ on the card. Without loss of generality, we assume that these special values appear in any MAC in the very same positions. This will ease the treatment of MACs at runtime as there will be no ambiguity about the position of relevant values.

We write $\mathcal{R}(\Gamma)$ to note the set of all the resources on the card, i.e., the ones that have a counter in Γ. Formally $\mathcal{R}(\Gamma) = \{\mathsf{R} \in \mathcal{R} \mid \exists i.\Gamma(i) = \mathsf{R}\}$.

Example 2 (Typing card pages). Consider again the simple card structure presented in Example 1. Recall that page 4 containes a counter of the ticket resource $\mathsf{R_T}$ on the card, pages 5 and 6 contain data such as the bus identifier and a timestamp, while page 7 is for a message authentication code (MAC) of the card ID (pages 0 and 1), together with the lock bytes (page 2), the OTP (page 3), the resource counter, and the timestamp (pages 4 and 6). Pages from 8 to 15 are not used (we do not report them but they can safely be given type Data). We can thus give the types reported on the right to the card pages. Finally, since there is only one resource on this card we have $\mathcal{R}(\Gamma) = \{\mathsf{R_T}\}$.

i	$\Gamma(i)$
0	Id
1	Id
2	Lock
3	Otp
4	$\mathsf{R_T}$
5	Data
6	Data
7	$\mathsf{Mac}[0, 1, 2, 3, 4, 6]$

When reading values whose integrity needs to be checked by recomputing a MAC we want to avoid that these values are changed before their integrity has been actually verified. To force the program to do so, we temporarily give type $\bullet$ to the variables containing such values. After the MAC has been checked we can safely give the actual types. To formalize this step we use a transformation of Γ, noted $\hat{\Gamma}$, that gives type $\bullet$ to all the pages that are arguments of at least one MAC and are not themselves MACs. Formally:

$$\hat{\Gamma}(i) = \begin{cases} \bullet & \text{if } \Gamma(i) \neq \mathsf{Mac}[\ldots] \text{ and } \exists j, z.\Gamma(j) = \mathsf{Mac}[\mathsf{i}_0, \ldots, \mathsf{i}_m] \wedge \mathsf{i}_z = i \\ \Gamma(i) & \text{otherwise} \end{cases}$$

(empty) $\vdash \Gamma\ \llbracket \epsilon \rrbracket\ \Gamma$ (skip) $\vdash \Gamma\ \llbracket \mathsf{skip} \rrbracket\ \Gamma$

(assign) $$\frac{\Gamma(x) = \mathsf{Data} \qquad x \neq p_i \qquad K \notin \mathcal{K}(e)}{\vdash \Gamma\ \llbracket x := e \rrbracket\ \Gamma}$$ (c-assign) $$\frac{\Gamma(p_i) = \mathsf{Data} \qquad K \notin \mathcal{K}(e)}{\vdash \Gamma\ \llbracket p_i := e \rrbracket\ \Gamma + \mathsf{W}_i}$$

(inc-res) $$\frac{\Gamma(p_i) = \mathsf{R}}{\vdash \Gamma, \mathsf{R}^n\ \llbracket p_i := p_i + n \rrbracket\ \Gamma + \mathsf{W}_i}$$ (dec-res) $$\frac{\Gamma(p_i) = \mathsf{R}}{\vdash \Gamma\ \llbracket p_i := p_i - n \rrbracket\ \Gamma, \mathsf{R}^n + \mathsf{W}_i}$$

(inc-otp) $$\frac{\Gamma(p_3) = \mathsf{Otp}}{\vdash \Gamma\ \llbracket p_3 := incOTP(p_3) \rrbracket\ \Gamma + \mathsf{iOtp} + \mathsf{W}_3}$$

(create-mac) $$\frac{\Gamma(p_i) = \mathsf{Mac}[\mathsf{i}_0, \ldots, \mathsf{i}_\mathsf{m}] \qquad \forall \mathsf{j} = 1, \ldots, m. \Gamma(p_{\mathsf{i}_\mathsf{j}}) = \Gamma(\mathsf{i}_\mathsf{j}) \qquad \mathsf{W}_3 \notin \Gamma}{\vdash \Gamma\ \llbracket p_i := MAC_K(p_{\mathsf{i}_0}, \ldots, p_{\mathsf{i}_\mathsf{m}}) \rrbracket\ \Gamma + \mathsf{W}_i}$$

(seq) $$\frac{\vdash \Gamma\ \llbracket \mathsf{c}_1 \rrbracket\ \Gamma' \qquad \vdash \Gamma'\ \llbracket \mathsf{c}_2 \rrbracket\ \Gamma''}{\vdash \Gamma\ \llbracket \mathsf{c}_1; \mathsf{c}_2 \rrbracket\ \Gamma''}$$ (if-then-else) $$\frac{\vdash \Gamma\ \llbracket \mathsf{c}_1 \rrbracket\ \Gamma' \qquad \vdash \Gamma\ \llbracket \mathsf{c}_2 \rrbracket\ \Gamma'}{\vdash \Gamma\ \llbracket \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 \rrbracket\ \Gamma'}$$

(MAC-check) $$\frac{\begin{array}{c}\Gamma(p_i) = \mathsf{Mac}[\mathsf{i}_0, \ldots, \mathsf{i}_\mathsf{m}] \qquad \Gamma(p_{\mathsf{i}_0}), \ldots, \Gamma(p_{\mathsf{i}_\mathsf{m}}) = \bullet \\ \vdash \Gamma\{p_{\mathsf{i}_0} : \Gamma(\mathsf{i}_0), \ldots, p_{\mathsf{i}_\mathsf{m}} : \Gamma(\mathsf{i}_\mathsf{m})\}\ \llbracket \mathsf{c}_1 \rrbracket\ \Gamma' \qquad \vdash \Gamma\ \llbracket \mathsf{c}_2 \rrbracket\ \Gamma'\end{array}}{\vdash \Gamma\ \llbracket \mathsf{if}\ MAC_K(p_{\mathsf{i}_1}, \ldots, p_{\mathsf{i}_\mathsf{m}}) = p_i\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 \rrbracket\ \Gamma'}$$

(produce) $\vdash \Gamma\ \llbracket produce(\mathsf{R}) \rrbracket\ \Gamma, \mathsf{R}$ (consume) $$\frac{\mathsf{iOtp} \in \Gamma \qquad \mathsf{W}_3 \notin \Gamma}{\vdash \Gamma, \mathsf{R}\ \llbracket consume(\mathsf{R}) \rrbracket\ \Gamma}$$

(read) $$\frac{\Gamma, p_i : \hat{\Gamma}(i) \vdash \diamond}{\vdash \Gamma\ \llbracket \mathsf{read}(i) \rrbracket\ \Gamma, p_i : \hat{\Gamma}(i)}$$ (write) $$\frac{\Gamma(p_i) = \Gamma(i) \qquad i \neq 0, 1}{\vdash \Gamma\ \llbracket \mathsf{write}(i) \rrbracket\ \Gamma - \mathsf{W}_i}$$

(dec-trust) $$\frac{\vdash \Gamma\ \llbracket \mathsf{c} \rrbracket\ \Gamma' \qquad \Gamma'(p_i) = \Gamma(i) \qquad \mathsf{W}_i \notin \Gamma'}{\vdash \Gamma\ \llbracket \mathsf{c} \rrbracket\ \Gamma'\{p_i : \hat{\Gamma}(i)\}}$$ (iOtp) $$\frac{\vdash \Gamma\ \llbracket \mathsf{c} \rrbracket\ \Gamma'}{\vdash \Gamma\ \llbracket \mathsf{c} \rrbracket\ \Gamma' - \mathsf{iOtp}}$$

Table 4. Type and effect system for secure APIs.

Example 3 (Transformation $\hat{\Gamma}$). If we compute $\hat{\Gamma}$ of Example 2 we have that all the types of pages 0,1,2,3,4,6 are transformed into $\bullet$ as they are MAC arguments.

Operations on the OTP. In order to deal with the OTP we need a function to increment its value against a given one, that we call $incOTP : Bytes^4 \to Bytes^4$. Recall that the OTP is 32 bits long and is such that once a 1 is written it cannot be restored to 0. Thus, the way the OTP is incremented is by changing to 1 a 0, which gives, at most, 33 different values. *incOTP* takes the OTP word and returns a new value with one of the 0s changed into a 1. The property that we want is that either $incOTP(v) > v$ or $incOTP(v)$ is undefined (e.g., when the OTP has reached the maximum possible value). An example of efficient implementation is a left shift and a bitwise or with 1, i.e., $(v << 1) \mid 1$, if $v < 2^{31}$ and undefined otherwise.

The type and effect system. Typing rules are reported in Table 4. Judgements have the form $\vdash \Gamma\ \llbracket \mathsf{c} \rrbracket\ \Gamma'$ meaning that program c typechecks under trusted key K and typing environment Γ and transforms it into Γ'. As we will see, this is useful to add/remove effects and to update types, for examples after a MAC check. We add/remove effects in two ways: we use the list notation Γ, e to note that e is appended to Γ, meaning that multiple instances of e can be in the resulting environment. We write $\Gamma + \mathsf{e}$, instead, to add e in Γ only if it is not already there and $\Gamma - \mathsf{e}$ to remove it is already in Γ. The former notation is used for linear effects R while the latter for effects related to events such as iOtp and W_i, that are added/removed once.

Intuitively, rules (ϵ) and (skip) state that the empty command ϵ and skip always typecheck with no modification of Γ. Rule (assign) admits any assignment of any expression to variables of type Data different from the card variables p_i, as soon as the expression does not compute any MAC under the trusted key K. For these variables, rule (c-assign) and the following ones, also track that they need to be written back to the card, via the effect W_i. More interestingly, rule (inc-res) regulates an increment of n resources of type R: this consumes n instances of R from the environment. Dually, rule (dec-res) adds n instances of effect R. Intuitively, effect R compensates for the increments and decrements of variables of type R. Rule (inc-otp) increments the OTP using the ad-hoc *incOTP* function presented above and adds the corresponding effect iOtp.

Rule (create-mac) is for computing new MACs and requires that the arguments are exactly the variables p_i specified by the MAC type and that they are typed accordingly to the card page type $\Gamma(i)$, meaning that they have been authenticated by a MAC check. Rule (seq) makes sure that sequential commands are typed using the environment produced by the previous command, while (if-then-else) typechecks both branches under the same environment Γ and produces the same environment Γ'.

Rule (MAC-check) regulates MAC checks: a MAC can be recomputed by value of type $\bullet$. This type is temporarily given to the value that needs to be checked with a MAC when they are read from the card (see rule (read) below), and freezes any update of such values until the MAC is actually checked. When this happens, the command in the if branch c_1 can be typed using the actual types of checked pages, i.e., $\Gamma(\mathsf{i}_1), \ldots, \Gamma(\mathsf{i}_\mathsf{m})$ which are updated in Γ and Γ'. We also require that modified variables have all been written back to the cards by checking the absence of the corresponding W_i effects. Notice that this makes the resulting environment different from the one used to type the MAC check and the else branch. (dec-trust) below will fix this anomaly.

Rules (produce) and (consume) respectively produce and consume one resource R by adding/removing the relative effect R from Γ. Additionally, rule (consume) requires that the OTP has been incremented (presence of iOtp) and written back to the card (absence of W_3). This is crucial to guarantee that resources cannot be consumed twice. Rule (read) add the type of the read variable using $\hat{\Gamma}$, which transforms into $\bullet$ all the types of arguments of MACs. Rule (write) removes, if present, the non-linear effect W_i requiring page writing.

Rule (dec-trust) updates the resulting environment Γ' by changing the type of variables p_i from $\Gamma(i)$ to $\hat{\Gamma}(i)$. This is safe if the value stored in p_i is not different from the value $C(i)$ on the card. Recall, in fact, that $\hat{\Gamma}(i)$ might be $\bullet$, meaning that the variable is 'frozen' and its value corresponds to the one on the card.

Finally, (iOtp) removes effect iOtp from Γ'. These two last rules are used when exiting a MAC-check branch, to bring types and effects to the original status.

Example 4 (A type-checkable travelling API). We first show that the (insecure) code given in Example 1 does not typecheck as it performs resource consumption without incrementing the OTP. As we have seen, this allows for resuming the state the ticket had before the trip and to reuse it an unbound number of times (similarly to what happened to real systems [6]). We let Γ be the typing environment defined in Example 2, and consider $\Gamma'(p_i) = \Gamma(i)$ and $\Gamma''(p_i) = \hat{\Gamma}(i)$:

i	Γ
0	Id
1	Id
2	Lock
3	Otp
4	$\mathsf{R_T}$
5	Data
6	Data
7	$\mathsf{Mac}[0,1,2,3,4,6]$

	Γ'	Γ''
p_0	Id	$\bullet$
p_1	Id	$\bullet$
p_2	Lock	$\bullet$
p_3	Otp	$\bullet$
p_4	$\mathsf{R_T}$	$\bullet$
p_5	Data	Data
p_6	Data	$\bullet$
p_7	$\mathsf{Mac}[0,1,2,3,4,6]$	$\mathsf{Mac}[0,1,2,3,4,6]$

Notice that $\Gamma''\{p_0 : \Gamma(0), p_1 : \Gamma(1), p_2 : \Gamma(2), p_3 : \Gamma(3), p_4 : \Gamma(4), p_6 : \Gamma(6)\} = \Gamma'$. For the sake of readability we abbreviate $\mathsf{W}_i, \mathsf{W}_j$ with $\mathsf{W}_{i,j}$.

$\vdash \Gamma$	$[\![\mathsf{read}(0:7);$	$]\!]\Gamma, \Gamma''$
$\vdash \Gamma, \Gamma''$	$[\![\mathsf{if}\ (MAC_K(p_0,p_1,p_2,p_3,p_4,p_6) = p_7)$	$]\!]$
$\vdash \Gamma, \Gamma'$	$[\![\quad p_4 := p_4 - n;$	$]\!]\Gamma, \Gamma', \mathsf{R}_\mathsf{T}^n, \mathsf{W}_4$
$\vdash \Gamma, \Gamma', \mathsf{R}_\mathsf{T}^n, \mathsf{W}_4$	$[\![\quad p_5 := BUS\_ID();$	$]\!]\Gamma, \Gamma', \mathsf{R}_\mathsf{T}^n, \mathsf{W}_{4,5}$
$\vdash \Gamma, \Gamma', \mathsf{R}_\mathsf{T}^n, \mathsf{W}_{4,5}$	$[\![\quad p_6 := TIMESTAMP();$	$]\!]\Gamma, \Gamma', \mathsf{R}_\mathsf{T}^n, \mathsf{W}_{4,5,6}$
$\vdash \Gamma, \Gamma', \mathsf{R}_\mathsf{T}^n, \mathsf{W}_{4,5,6}$	$[\![\quad p_7 := MAC_K(p_0,p_1,p_2,p_3,p_4,p_6);$	$]\!]\Gamma, \Gamma', \mathsf{R}_\mathsf{T}^n, \mathsf{W}_{4,5,6,7}$
$\vdash \Gamma, \Gamma', \mathsf{R}_\mathsf{T}^n, \mathsf{W}_{4,5,6,7}$	$[\![\quad \mathsf{write}(4:7)$	$]\!]\Gamma, \Gamma', \mathsf{R}_\mathsf{T}^n$
$\vdash \Gamma, \Gamma', \mathsf{R}_\mathsf{T}^n$	$[\![\quad consume(\mathsf{R_T})^n;$	$]\!]???$

Here the type-check fails as rule $(\mathsf{consume})$ requires effect iOtp in the environment, i.e., that the OTP has been incremented. To get a type-checkable API we just need to increment the OTP and write it back to the card, as follows:

$\vdash \Gamma$	$[\![\mathsf{read}(0:7);$	$]\!]\Gamma, \Gamma''$
$\vdash \Gamma, \Gamma''$	$[\![\mathsf{if}\ (MAC_K(p_0,p_1,p_2,p_3,p_4,p_6) = p_7)$	$]\!]$
$\vdash \Gamma, \Gamma'$	$[\![\quad p_3 := incOTP(p_3);$	$]\!]\Gamma, \Gamma', \mathsf{iOtp}, \mathsf{W}_3$
$\vdash \Gamma, \Gamma', \mathsf{iOtp}, \mathsf{W}_3$	$[\![\quad \mathsf{write}(3)$	$]\!]\Gamma, \Gamma', \mathsf{iOtp}$
$\vdash \Gamma, \Gamma', \mathsf{iOtp}$	$[\![\quad p_4 := p_4 - n;$	$]\!]\Gamma, \Gamma', \mathsf{R}^n, \mathsf{W}_4$
$\vdash \Gamma, \Gamma', \mathsf{R}^n, \mathsf{iOtp}, \mathsf{W}_4$	$[\![\quad p_5 := BUS\_ID();$	$]\!]\Gamma, \Gamma', \mathsf{R}^n, \mathsf{iOtp}, \mathsf{W}_{4,5}$
$\vdash \Gamma, \Gamma', \mathsf{R}^n, \mathsf{iOtp}, \mathsf{W}_{4,5}$	$[\![\quad p_6 := TIMESTAMP();$	$]\!]\Gamma, \Gamma', \mathsf{R}^n, \mathsf{iOtp}, \mathsf{W}_{4,5,6}$
$\vdash \Gamma, \Gamma', \mathsf{R}^n, \mathsf{iOtp}, \mathsf{W}_{4,5,6}$	$[\![\quad p_7 := MAC_K(p_0,p_1,p_2,p_3,p_4,p_6);$	$]\!]\Gamma, \Gamma', \mathsf{R}^n, \mathsf{iOtp}, \mathsf{W}_{4,5,6,7}$
$\vdash \Gamma, \Gamma', \mathsf{R}^n, \mathsf{iOtp}, \mathsf{W}_{4,5,6,7}$	$[\![\quad \mathsf{write}(4:7)$	$]\!]\Gamma, \Gamma', \mathsf{R}^n, \mathsf{iOtp}$
$\vdash \Gamma, \Gamma', \mathsf{R}^n, \mathsf{iOtp}$	$[\![\quad consume(\mathsf{R})^n;$	$]\!]\Gamma, \Gamma''$
$\vdash$	$[\![\mathsf{else}$	$]\!]$
$\vdash \Gamma, \Gamma''$	$[\![\quad \mathsf{skip}$	$]\!]\Gamma, \Gamma''$

Notice that $consume(\mathsf{R_T})$ is typed by n applications of rule $(\mathsf{consume})$, then by six applications of $(\mathsf{dec\text{-}trust})$ to bring Γ' back to Γ'' and by one final application of (iOtp) to remove iOtp. The fact $\mathsf{R_T}$ does not appear in the final environment guarantees that the resources $\mathsf{R_T}$ have a 0 final balance. In fact, n such resources have been removed from the card and n *consume*$(\mathsf{R_T})$ have been performed, allowing the calling application to use such resources. The attack of Example 1 is no more effective: since the OTP is incremented and irreversible, when the attacker copies back the initial values, the MAC would result to be invalid.

4.1 Security of well-typed APIs

We now prove that well-typed programs guarantee interesting security properties. Intuitively, for such programs: (*i*) Whenever resources are consumed from a card, the OTP is always incremented; this is fundamental to avoid that cards are restored to states with more resources than the expected ones (Theorem 1); (*ii*) the number of resources on a card, during its lifetime, never exceeds the difference between the resources produced and the ones consumed; for example if we charge 3 tickets and we consume 2, we can at most have 1 ticket on the card. This is crucial to guarantee that resources are never double-used (Theorem 2); (*iii*) resources can never be consumed by a card with no valid MACs; this is crucial to avoid the attacker can freely forge cards without knowing the MAC key K (Theorem 3). Notice that for lack of space proofs are omitted.

Valid cards. Our starting point is the concept of valid cards. A card is valid if it contains at least a valid MAC. Recall that resource counters are required to appear as argument of all MACs on a card together with the ID, the lock bytes and the OTP (item 4 of Definition 2). Thus, if a card has at least one valid MAC we can safely recover the number of different resources on the card. Moreover, the valid MAC cannot be copied on a different device as the different ID would break the MAC check. This is the standard security mechanism to avoid cloning RFID devices. We formalize the notion of valid cards as follows:

Definition 3 (Valid cards). *Let* $\Gamma \vdash_{\mathcal{C}} \diamond$. *A card* $C \in \mathcal{C}$ *is* Γ*-valid, written* $\Gamma \vdash C$*, if* $\exists i.\Gamma(i) = \mathsf{Mac}[\mathsf{i_0}, \ldots, \mathsf{i_m}]$ *and* $MAC_K(C(\mathsf{i_1}), \ldots, C(\mathsf{i_m})) = C(i)$.

Given that a valid card ensures the integrity of every resource counter, for valid cards we can safely lookup the number of resources. This is done by looking up the maximum number of resources validated by MACs.

Definition 4 (MAC and card resources). *Let* $\Gamma \vdash_{\mathcal{C}} \diamond$ *such that* $\Gamma(i) = \mathsf{Mac}[\mathsf{i_0}, \ldots, \mathsf{i_m}]$ *and* $\Gamma(\mathsf{i_j}) = \mathsf{R}$. *The number of resources* R *in* $MAC_K(v_0, \ldots, v_m)$*, denoted with* $\mathsf{R}(MAC_K(v_0, \ldots, v_m))_\Gamma$*, is* $v_{\mathsf{i_j}}$. *Let* $\Gamma \vdash C$ *thanks to MACs* $m_1, \ldots,$ m_z. *The number of resources* R *on* C *is* $\mathsf{R}(C)_\Gamma = max\{\mathsf{R}(m_1)_\Gamma, \ldots, \mathsf{R}(m_z)_\Gamma\}$.

Security of well-typed programs. First theorem proves that whenever resources are consumed by a well-typed API program, the OTP has been incremented.

Theorem 1 (Consuming affects OTP). *Let* $\Gamma \vdash_{\mathcal{C}} \diamond$ *and* $\vdash \Gamma\ [\![\mathsf{c}]\!]\ \Gamma''$ *with* *eff*$(\Gamma) = \emptyset$. *Then,* $\langle \mathsf{c}, M, C\rangle \xrightarrow{\gamma}{}^{*} \langle \mathsf{c}', M', C'\rangle \overset{consume(\mathsf{R})_{C'}}{\longrightarrow} \langle \mathsf{c}'', M'', C''\rangle$ *implies* $\mathsf{Otp}(C') > \mathsf{Otp}(C)$.

Based on this theorem and on a subject reduction result that we omit for lack of space, we can prove our main technical result: any valid card that the attacker can obtain will never contain a number of resources exceeding the expected ones, i.e., the one recharged and not yet used.

From now on, we consider initial attacker configurations $\langle M_0, \mathcal{C}_0 \rangle$ such that M_0 does not contain any MAC under K, i.e., the attacker does not initially know any secure MAC; moreover, all cards C_0 in $\mathcal{C}_0$ are such that $\mathsf{R}(C_0) = 0$, for any resource R, and $\mathsf{Otp}(C_0) = 0$, i.e., they contain 0 resources and have an empty OTP. We write $\Gamma \vdash \mathcal{T}$ to note that $\Gamma \vdash_{\mathcal{C}} \diamond$ and $\mathsf{c} \in \mathcal{T}$ implies $\vdash \Gamma \llbracket \mathsf{c} \rrbracket \Gamma''$ with $\mathit{eff}(\Gamma) = \emptyset$ and $\Gamma(x), \Gamma(p_i) \not\leq$.

Theorem 2 (Security of valid cards). *Let* $\Gamma \vdash \mathcal{T}$ *and* $\langle M_0, \mathcal{C}_0 \rangle \overset{\gamma}{\rightsquigarrow}{}^{*}_{\mathcal{T}} \langle M, \mathcal{C} \rangle$ *and* $C \in \mathcal{C}$ *such that* $\Gamma \vdash C$. *Then,* $\mathsf{R}(C)_\Gamma \leq \mathit{count}(C, \gamma, \mathsf{R})$.

The above theorem shows that the attacker can never obtain a valid card with more resources than the expected ones. Note that, if a card forged by the attacker is invalid, then the type system guarantees that resources will never be consumed from it by well-typed APIs, that is, an invalid card is useless.

Theorem 3 (Resistance against invalid cards). *Let* $\Gamma \vdash \mathcal{T}$ *and* $\langle M_0, \mathcal{C}_0 \rangle \overset{\gamma}{\rightsquigarrow}{}^{*}_{\mathcal{T}} \langle M, \mathcal{C} \rangle \overset{\gamma'}{\rightsquigarrow}_{\mathcal{T}} \langle M', \mathcal{C} \rangle$ *and* $C \in \mathcal{C}$ *such that* $\Gamma \nvdash C$. *Then,* $\nexists C'$ *such that* $\mathsf{Id}(C') = \mathsf{Id}(C)$ *and* $\mathit{consume}(\mathsf{R})_{C'} \in \gamma'$.

5 Conclusion

We have modelled and analysed Mifare Ultralight cards giving a type-and-effect system that can be used to develop and check an API which is resistant to double-usage of resources and card forging.

Note that there are other interesting properties that one would expect from an electronic ticket, e.g., cards of legitimate users should not be corrupted. This is, in fact, impossible to prevent since the attacker can overwrite card pages, including the OTP, and make the MAC check fail. This problem can be mitigated using centralized information about purchases and usages of tickets on a specific MU card: a user could be then refunded in case her card is corrupted.

In the introduction we have mentioned the possibility of creating devices that emulate MU cards with arbitrary IDs. This of course would allow an attacker to clone cards and to arbitrarily reuse tickets. In fact, security of MU cards is based on the assumption that the ID is unique and the OTP cannot be reversed. However, note that cards might be inspected or observed at gates and such devices are far from being physically identical to real cards. In order for this attack to be effective, it would be necessary to produce fake, emulating cards that looks like the real ones and this seems to be not cost-effective at the moment.

Regarding the length of the MACs, note that being 32-bits long are, in general, insecure. It is important to notice, however, that to forge a MAC it would be necessary to interact with a legitimate validation or recharging machine for a long time without being noticed. In fact, transactions are rather slow taking at least about 7ms to solve collisions and perform one read operation [2]. Trying 2^{31} MACs would then take $\approx$174 days. A valid MAC should be forged for any single trip, which makes this attack inconvenient. Finally, it is of course possible to use longer MACs at the price of consuming more memory from the cards.

The OTP mechanism is such that at most 32 ticket rides can be loaded. This could seem a limitation even in the recharging solution we have proposed. However, there are some issues to consider. First a 32 rides ticket could be used to store, e.g., 3 types of tickets (metro, bus and boat), each of which could, e.g., be recharged up to 10 single rides. MU cards are very cheap, however, given that the number of users can be millions, and that each ticket costs to the seller a few cent of Euros, 32 recharges in replace of 32 single ride tickets are already a big saving. It is also reasonable to assume the production of new cheap cards with a bigger memory and thus OTP, on which the same solution would apply.

As a future work, we intend to investigate on how our analysis can be generalized to other disposable cards from different producers. It would be desirable to have a generic semantics and type-system that is 'configurable' by specifying which security mechanisms are offered by the analysed card. We also intend to implement our type-and-effect system on a simple fragment of a real programming language in order to try to type-check real APIs.

Acknowledgements We are grateful to the anonymous reviewers for their helpful comments and suggestions.

References

1. Moscow metro: the worlds first major transport system to operate fully contactless with nxps mifare technology. press statement, 2009. Available at `http://www.nxp.com/news/content/file_1518.html`.
2. Mifare ultralight contactless single-ticket IC, 2010. Product data sheet. Rev. 3.8 22 Dec. 2010, 028638, `www.nxp.com/documents/data_sheet/MF0ICU1.pdf`.
3. M. Bugliesi, S. Calzavara, F. Eigner, and M. Maffei. Resource-aware Authorization Policies for Statically Typed Cryptographic Protocols. In *In Proc. of 24th IEEE Symposium on Computer Security Foundations*, 2011.
4. M. Centenaro, R. Focardi, F. Luccio, and G. Steel. Type-based analysis of PIN processing APIs. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS'09)*, pages 53–68. Springer, LNCS 5789, 2009.
5. A. Gordon and A. Jeffrey. A Type and Effect Analysis of Security Protocols. In *Proc. of the 8th Int. Symp. on Static Analysis*, page 432. Springer LNCS 2126, 2001.
6. P. Siekerman and M. van der Schee. Security evaluation of the disposable ov-chipkaart v1.7, 2007. Research Project, University of Amsterdam, `http://staff.science.uva.nl/~delaat/sne-2006-2007/p41/Report.pdf`.
7. G. Steel. Formal Analysis of PIN Block Attacks. *Theoretical Computer Science*, 367(1-2):257–270, November 2006.
8. A. Tanenbaum. Dutch public transit card broken, 2008. Available at `http://www.cs.vu.nl/~ast/ov-chip-card/`.
9. R. Verdult. Proof of concept, cloning the ov-chip card, 2008. Technical report, Radboud University Nijmegen, `http://www.cs.ru.nl/~flaviog/OV-Chip.pdf`.