DOCUMENTATION

The Implementation of a Trustworthy Voting System in MINIX 3

August 2010

Author:
Victor van der Veen

vrije Universiteit *amsterdam*

Contents

1 Introduction

This document describes the implementation of a trustworthy voting system in MINIX 3. The Computer Systems group of the Vrije Universiteit Amsterdam initiated an "Individual Programming Assignment", which target was to provide a prototype voting system implementation based on the design of Paul and Tanenbaum [1]. This documentation and the source trunk are the final results of this assignment.

As outlined in [1], a significant part of the voting system consists of the ability to attest the machine. The implementation below does not concern attestation. Due to the amount of work it would take to implement this, attestation was set up as a different project.

This document has the following structure. Section 2 outlines which hardware and software setup was used for development. The contents of the source trunk is explained in Section 3. Section 4 describes the new CD-R driver. Finally, Section 5 enumerates notes about what should be done in the future to gain a fully operational voting machine.

2 Setup & Installation

To implement attestation in MINIX 3, the voting machine requires a Trusted Platform Module (TPM) chip that supports TPM specification 1.2

For development, we used the latest MINIX version available, version 3.1.5, when the project started. During development, newer versions were released. Except for the CD-R driver, the voting system source should compile perfectly under these releases.

To compile the source, it is necessary to install the following packages. Note that to avoid dependency problems, the order of installation is important.

1. `glib-1.2.10` (37)
2. `binutils-2.16.1` (13)
3. `make-3.80` (66)
4. `gcc-libs` (33)
5. `gcc-4.1.1` (32)
6. `openssl-0.9.8a` (83)
7. `subversion-1.4.0` (111)

After installing the above packages, it is necessary to add the gcc directory (`/usr/gnu/bin`) to your path in `~/.ashrc`. Downloading and compiling the voting system source code is now possible:

1. `svn checkout svn://svn.few.vu.nl/home/vvn200/voting/`
2. `cd voting/current/minix/`
3. `sh update.sh`
4. `cd /usr/src/tools/`

5. `make install`
6. (reboot)
7. `cd voting/current/`
8. `make all`

Steps 2 to 6 add CD-R support to MINIX. Since this simply copies and altered MINIX 3.1.5 device driver to the MINIX source trunk, followed by recompling the system, this only works when MINIX 3.1.5 is used. If one wants to add CD-R support to newer MINIX releases, it is necessary to check whether files in `voting/current/minix` need to be updated.

Disable CD-R and LP Support CD-R and LP (print) support may be disabled by modifying the Makefiles. More information about how this is achieved, can be found in the corresponding README files.

3 SVN Trunk

Now that we have added CD write support and compiled the SVN source in Section 2, this section together with Section 4 contains the actual source code documentation. This documentation along with the source comments should give a good overview of how the system is implemented.

3.1 Headers & Objects

We start with the `/lib` directory. These files are included by most steps of this implementation and are therefore rather important.

3.1.1 `/lib/const.h`

The `const.h` file contains a list of definitions which are used by a large part of the implementation.

Encrypt and Decrypt Constants According to [1], we adhere to the PKCS#1 standard. A public/private keypair is used to encrypt/decrypt a shared session key for data encryption. OpenSSL provides a library called "OpenSSL cryptographic library" [2], which is what we used to adhere to this standard. We used the `.PEM` file format to store public/private keypairs, which were read using the OpenSSL crypto functions `PEM_read_PUBKEY()` and `PEM_read_PrivateKey()`. Data was encrypted and decrypted using OpenSSL's "high-level cryptographic functions" [3].

The `CIPHER` constant in `const.h` tells which symmetric encryption scheme is used. We currently use the standard DES-EDE3-CBC cipher as provided by OpenSSL. Depending on the chosen cipher type, an initialization vector is needed to encrypt/decrypt the first data block. Since our implementation uses a static buffer

to store this initialization vector (IV), the length of IV is defined in `IV_SIZE`. This should match `EVP_CIPHER_iv_length(CIPHER)`. If a different encryption cipher is wanted, required actions include updating these two definitions, recompiling, and restarting registration.

`KEY_SIZE_BITS` is the size of the asymmetric public key in bits, `KEY_SIZE` is the public key size in bytes. These values are needed so that the buffer for the encrypted secret key can be declared static as well. An encrypted secret key (eskey) is the shared symmetric key mentioned above, being encrypted by an asymmetric public key. The value of `KEY_SIZE` should match `EVP_PKEY_size(crypto.pubkey[0])`. Here, `crypto` is a struct (defined in `type.h`) and `crypto.pubkey` is an array of pointers to `EVP_PKEY` types, which is the OpenSSL type for a public key. In short: the encrypted secret key used for symmetric data encryption/decryption will never be larger than the size of the used asymmetric key.

The algorithm that we use for creating digital signatures and hashes are defined by `SIGN_DIGEST_TYPE` and `DIGEST_TYPE` respectively. These are currently both set to SHA256. `SIGN_DIGEST_SIZE` and `DIGEST_SIZE` tell us how long (i.e. how many bytes) a signature or hash will be. What follows are two import definitions regarding the security level of the voting machine. First comes the salt size (`SALT_SIZE`) that tells us how many bytes will be added to a hashed user password before it is encrypted. This mechanism protects the password against brute force attacks. We currently set `SALT_SIZE` to $256 - $ `DIGEST_SIZE` bytes. Next comes `S_SIZE`, which defines the number of bytes that will be used for the generation of S_1, S_2 and S, currently set to 256 bytes.

File Locations Constants These are likely to change or disappear in future implementations. For now, they are defined in such a way that they can be used within s(n)printf() statements. The `sprintf(filename,P_PUBKEY_LOC(42))` call will write the location of the public key for precinct 42 into the string `filename`.

Database Prefixes Most (intermediate) data files are stored as plain text. Values are written to files using simple `fprintf()` statements:

```
fprintf(fp,"%s%d\n",PREFIX_DB_VID,voter->vid);
```

When data must be read again, `fgets()` can be used to read one entire line of data, followed by `strncmp()` to find out which data was fetched.

Remaining Definitions The remaining definitions are described in `const.h`.

3.1.2 `/lib/type.h`

Three main types are defined in `type.h`: `crypt_t`, `voter_t` and `session_t`. The crypt type consists of OpenSSL types to store keys (private, public and encrypted

secret) as well as encrypted and decrypted buffers (dbuf and ebuf) and their corresponding counters (dbuf_len and ebuf_len). The crypt_t struct is used by most steps below to avoid numerous variable declarations.

A voter type is used to store data about a specific voter. Besides personal information like name, address and password, this type also contains an array of MAX_RACES elements to store each vote of this voter. Once all votes are casted, this array is used to write the voting record of this voter to disc.

The session type is used only in step 7 & 8 and holds data about the election day. An important field is the ballot_t ballot, which holds information about the races and candidates as well as a counter for each candidate to be able to print the final results quickly when election day ends. Other interesting fields are blacklist (a list of voters who already logged in), the unique array and base. The unique array contains every unique number that was created during election day. A newly generated unique value is checked against this array to ensure its uniqueness. The contents of base will be prepended to each unique value. By using a different 'base' value for each voting machine, we avoid interfering unique values between two different voting machines. Adding the time of boot completion to this base value will avoid duplicates on the same machine after a system crash.

3.1.3 /lib/util.*

The most important parts of this project are those where user input must be parsed and stored in a buffer. This is where buffer overflow exploits can occur. A significant part of util.c consists of functions that parse user input. To simplify user input and avoid buffer overflow exploits, we have built a simple getch() function which may be used by higher level functions when user input is requested. Our getch() implementation does no more than the following:

- Change the terminal attributes using tcsetattr() in such a way that (1) input is immediately available (i.e. no line-delimiter character is needed, this is also known as non-canonical mode); (2) no characters are printed to the screen; and (3) INTR, QUIT, SUSP, or DSUSP are discarded.
- Read one byte from standard input using fread().
- Reset the terminal settings to its defaults.

Hence, functions that use getch() need to do their printing and/or deleting of characters manually. Note that this is a more accurate approach than the standard getchar() implementation [4]. Currently, getint(), getstring() and getoption() are functions that use getch() to get their user input. getint(), for example, is implemented such that only an integer value may inserted that lies between two pre-known bounds.

Other functions in util.c concern the gathering of public/private keys from a .PEM file (get_public(), get_private()), parsing plain-text data files (seek_

`vid()`, `parse_nextline_*()`), basic cryptography functionality (`hash()`, `sign()`, `simple_decrypt()`), and other minor functions.

Encrypted, signed or hashed data is written to a file as a hexadecimal bytestring (`etoh()` — encrypted to hexadecimal). This gives more flexibility when reading the encrypted data in a later stage and makes our files text-only (handy for testing). Note that the crypto library provides similar functionality with 'BIOs', but we opted for this approach for simplicity [5].

3.1.4 `/lib/cdrom.*`

The CD-R initialization and finalization steps are implemented in `cdrom.c`. The initialization procedure consists of a while loop that loops until a blank CD-R is inserted into the tray. Any media other than CD-R is discarded, making it impossible to use media types that allow the removal of written data. Once a CD-R is inserted, the drive is locked and it is set for packet writing. Initialization also performs a `DIOCRESERVE` ioctl on the inserted disc to reserve two large tracks for data storage and one small track for error recovery.

Finalization is more straightforward: close all tracks and finalize the disc before unlocking and ejecting it.

Besides start-up and shutdown procedures, `cdrom.*` also contains code that writes data to a track. The `cdrom_write()` function expects a `struct cd_write_buffer` (defined in `/usr/src/include/minix/cdrom.h`), which consists of a 16KB data buffer and an integer identifying which track should be used for writing this data.

3.2 Step 1: Precinct Master Key Generation and Distribution

This step is an operational concerning the procedural generation and distribution of precinct keys.

Besides the generation of a precinct key, [1] makes a note about the use of a county key as well: "*... It encrypts S_{i1} and S_{i2} with a county-generated public key* ...". The creation of this key was not stated explicitly, however. We feel that step 1 would be a good place to do this, but one may argue that a separate step is required.

There are numerous ways to generate keys that contain a public and private part. Our implementation assumes that RSA encryption is used and that the public and private keys are stored in a `.pem` file. Since our implementation uses the crypto library provided by the OpenSSL project, it makes sense to use OpenSSL for key generation as well. An example of creating keys using OpenSSL is shown Listing 1.

The rest of step 1 is signing and secret sharing the multiple key parts (if desired). These steps are omitted in our implementation.

Listing 1 Example OpenSSL calls for key generation

```
#Generate a 2048 bit county key using RSA:
openssl genrsa -out county.private.pem 2048
#Generate a public subkey of above key:
openssl rsa -in county.private.pem -pubout -out \
    county.public.pem

#Generate a 2048 bit key using RSA for precint i:
openssl genrsa precinct_i.private.pem 2048
#Generate a public subkey of above key:
openssl rsa -in precinct_i.private.pem -pubout -out \
    precinct_i.public.pem
```

3.3 Step 2: Voter Registration

Voters may now come to the county office to register themselves. Only registered voters are allowed to cast their vote on election day. Their personal information is stored in a database, `voter.db` (used in later steps). An example implementation of this registration procedure can be found in `register.c`.

When looking at `register.c`, the first thing we see is an if-statement that checks whether the initialization vector size `IV_SIZE` stored in `const.h` equals the expected initialization size. We saw this earlier in Section 3.1.1. If one decides to use a different cipher by adjusting the `CIPHER` constant in `const.h`, it may be necessary to update `IV_SIZE` as well. If these numbers do not match, the program behavior is undefined. Hence, it is important that this check is done in each program that uses the `crypt_t` type.

A few lines down in the code, right after loading the necessary `.pem` files, similar checks are done to make sure that `const.h` contains the correct key sizes. When longer keys are generated in step 1, it is necessary to update the `KEY_SIZE_BITS` value in `const.h`.

3.4 Step 3: Proof of Registration Mailed to the Voters

In this step, a proof of registration is mailed to the voters. Attached is the value of S_{i1}. The source in `sends1.c` searches for record i in `voter.db` and decrypts the encrypted S_{i1} value using the county private key. This value is then converted to a hexadecimal format and written to a file called i`.s1`.

3.5 Step 4: Voting Machines are Prepared

This step concerns the preparation of the voting machines. Voters registered themselves in step 2 by going to the county office. Hence, `voter.db` contains a list

of voters who will vote at a precinct within the county. `votersperprecinct.c` contains the source code that splits the `voter.db` into each precinct's list.

Once `voter.db` is split into multiple (decrypted) voter per precinct (VPPD) files, these files are encrypted and stored in voter per precinct (VPPE) files, using the precinct public key to prevent tampering in storage or transit.

3.6 Step 5: Key Assembly at Each Precinct

This step concerns assembling the keys at each precinct. Since we assumed in step 1 that no key splitting techniques are used, this step does not contain any relevant procedures.

3.7 Step 6: Voters Show up and Check in

In Step 6, voters go to the polling place to vote. Before voting, he must check in by going to a poll worker and provide the poll worker with S_{i1}. An example implementation of the check-in software running at the poll worker machine can be found in `checkin.c`.

This program decrypts the VPPE for a provided precinct, asks for a voter-id, looks up the voter, prints his personal information on the screen, asks for a confirmation, gets S_{i2}, computes S = S_{i1} XOR S_{i2} and creates a `token` file which contains the voter-id and S.

3.8 Step 7 & 8: Voters Cast Their Votes and Tabulating the Votes

Step 7 and step 8 are merged in the implementation. This is because step 8 is a procedural step of shutting down the voting machines.

We implement only a subset of the authentication interface between the voter and the machine. In the normal case, a voter was given a smartcard during the check in procedure that contains his token as described in section 3.7. This smartcard would be inserted into the voting machine so that the machine can authenticate the voter before it starts the voting procedure. In our implementation, the token that is needed for authentication is not fetched from a smartcard, but read from a file. The voting machine code makes simplifying assumptions on where previously entered data can be found. For example, the token described here is assumed to be in a file called `token` in the `current_data` directory. Notes about where the voting machine expects these files can be found in the code itself. They are also printed on screen when the machine is started.

`vm.c` contains the initialization, finalization and main loop of the voting machine implementation. A significant part of initialization and finalization concern the setting up of the CD-ROM device for packet writing. Section 4 provides more information about how this was achieved. Beside device initialization, the `init_session()` also fetches a candidate list by parsing a EML file. See Section 3.8.1 for its corresponding BNF notation.

The task of the main loop is easy: wait for input from the user and start the attestation or voting procedure. If a special key is pressed, the exit procedure is called.

The voting procedure itself consists of 12 substeps of which descriptions can be found in the comments of `vm.c`.

3.8.1 EML Notation

During initialization, the voting software parses a XML file to fetch the candidate list. The file is parsed by the BNF notation shown in Listing 2.

Listing 2 BNF notation of accepted EML input

```
LF      = <US-ASCII LF, linefeed (10)>
SP      = <US-ASCII SP, space (32)>
HT      = <US-ASCII HT, horizontal-tab (9)>
DQ      = <US-ASCII double-quote mark (34)>

WS      = *(LF | SP | HT)
LWS     = *(SP | HT)

UPALPHA = <any US-ASCII uppercase letter "A".."Z">
LOALPHA = <any US-ASCII lowercase letter "a".."z">
ALPHA   = UPALPHA | LOALPHA
DIGIT   = <any US-ASCII digit "0".."9">

CHAR    = ALPHA | DIGIT | SP

QSTRING = (DQ *CHAR DQ)

name_attribute = (LWS "name" LWS "=" LWS QSTRING LWS)

candidate = (WS "<candidate>" *CHAR "</candidate>" WS)
race      = (WS "<race" [name_attribute] ">" *candidate "</race>" WS)
ballot    = (WS "<ballot>" *race "</ballot>")
```

The used notation is meant to be a subset of the Election Markup Language (EML) [6]. To keep our codebase small, we do not need to support the entire EML specification (see `parser.c`).

3.9 Step 9: Publishing the Result

Before the election results can be published, it may be necessary to recount the votes that were entered earlier. This is done by reading all vote records from disc

and verifying them, which is implemented in `getresults.c`. We experienced some problems in MINIX when doing this (see Section 4.4.4), so this program can currently only be used in a Linux environment.

The principle of `getresults.c` is fairly simple: just reverse everything that we did so far. The encrypted voting records on disc are fetched by simple `fread()` and `fseek()` calls. These records are then decrypted using the precinct private key. Next, each subrecord (containing the vote for one race) will be parsed and its signature will be verified. When the signature check passes, the vote is stored in memory and the next subrecord is parsed.

When all votes are read, a result screen is printed to standard output.

3.10 Utils

We used a single CD-RW disc for our CD-R test purposes. A CD-RW has the same properties as a plain CD-R disc, with the advantage that it can be erased after every write session. However, there was no blanking program for CD-RW implemented on MINIX 3 yet, which is why we included one in the `utils` directory.

The utility program uses a new ioctl, `DIOCBLANK`, which performs a 'quick' blanking operation on the requested disc. More information about blanking types can be found in the "Blanking Command" section of MMC-6 [7].

4 CD-R Implementation

As outlined in [1], a write-once, read-many medium is specified to store votes, and they recommend a CD-ROM as a medium to achieve this. The main problem for an implementation is that MINIX 3 does not support CD-R. In this section, we describe how a CD-R driver for MINIX 3 can be implemented.

The driver will be responsible for sending commands to the device as well as receiving data from it. Possible commands are defined in the Multi-Media Command - 6 (MMC-6) document. This is a standard that defines a SCSI based command set needed to access multi-media features [7]. More commands are defined in the SCSI Primary Commands - 3 (SPC-3) document. This standard defines the SCSI commands that are basic to every device model and the SCSI commands that may apply to any device model [8].

Most CD drives nowadays are ATAPI devices. ATAPI allows the ATA interface to carry SCSI commands and responses. To issue a command, the driver has to construct an ATAPI packet containing the desired SCSI command from MMC-6 or SPC-3. This packet is then sent over the ATA interface to the device. Depending on the command and its arguments, the device sends back a response (sense data) and waits for more requests [9]

Fortunately, MINIX 3 already supports sending ATAPI packets and receiving responses: `atapi_sendpacket()` and `atapi_intr_wait` in the general AT driver (`/usr/src/drivers/at_wini/at_wini.c`) implement this functionality.

For more information about the physical structure of a CD-ROM, Philips' Red Book would be the best reference. Regrettably, this book is rather expensive, and a confidentiality agreement is required to get it. An 'approximation' of the Red Book is the international standard published by the International Electrotechnical Commission (IEC) as IEC 60908. Much of the information from the IEC 60908 is repeated in the freely available ECMA-130 CD-ROM specification [10, 11].

Below, we first explain why we can't simply port existing burning software to MINIX 3. Next, notes about the actual implementation can be found, followed by difficulties that occurred during implementation and some ideas for future work on MINIX 3.

4.1 Why porting is not sufficient

To add CD-R support to MINIX, one could choose to port an existing implementation or to write a new implementation from scratch. We now describe why porting is not sufficient:

1. Current implementation architectures significantly differ from that of MINIX 3.
2. Current implementations do not support our requirements.

First, focusing on the MINIX 3 ideology, we see that current *NIX burning software products rely on direct write access to the CD drive [12, 13]. This means that the SCSI commands for manipulating the device are sent to it directly by writing bytes to `/dev/cdrom`. Such direct communication between regular user programs and hardware brings some security risks. For example, it is possible to send a blocking command to the device which can make all other running processes (web server, database server, ...) become unresponsive. As MINIX 3 aims to be a secure and reliable operating system, using direct write access to `/dev/cdrom` violates the MINIX 3 design goals.

Second, existing burning software does not support our requirements. Below are the three main reasons why current implementations are problematic.

We need the ability to write more than 99 data chunks on one disc We see that most burning software is able to write data to a disc only in Track-At-Once (TAO), Session-At-Once (SAO) or Disc-At-Once (DAO) write mode. In TAO mode, the device expects data to arrive in a contiguous flow, possibly by using the device's internal buffer. When the flow is non-contiguous or when the buffer becomes empty, a buffer underrun occurs. Most devices have a buffer underrun protection mechanism which allows restarting the writing procedure after a buffer underrun occurred. However, although this is not stated by the MMC explicitly, our experiments showed that when a buffer underrun occurred, the current write track i is closed by the device. Once data becomes available again, the device creates a new track $i+1$ and starts writing to this new track.

The fact that a disc may consist of at most 99 tracks [9], implies that TAO mode is not sufficient since it limits the number of votes that can be stored on one disc to 99. SAO or DAO are also not sufficient, since their finest granularity is at least one track[14]. One could think of another scheme in which TAO write mode may be sufficient: not writing the votes to disc immediately, but storing them in memory first and writing them all at once when the election day ended. This would require only one track which can be written in TAO mode. In this scheme, a system crash or power loss destroys all entered votes up to that point in time. Variants of this scheme wherein x votes (for $x > 1$) are written to disc directly, suffer from the same vote loss possibility.

The solution lies in packet writing. This technique is also known as incremental writing and is mainly used by UDF like file systems to let CD-RW discs act like regular floppy discs. Unfortunately, we were unable to find any open source software products that provide such flexibility.

We need the ability to write votes to a "random" location This is another requirement of the design. No current implemenation exists that supports this. Because of this specific requirement, a custom implementation is needed.

We need the ability to recover an unfinished disc after a system crash This requirement is not defined by Paul and Tanenbaum. However, a disc failure could lose all previously unfinished votes. As with our search for drivers that support CD-R random write access, this feature may even be a more specific objective. Hence, we can safely state that current implementations are not sufficient and that we need a CD-R driver that is written from scratch to support our requirements.

4.2 Actual Implementation

In this section, we examine the actual implementation of the new CD-R driver for MINIX 3.

4.2.1 Packet Writing

Setting up a CD drive for packet writing (a.k.a. incremental writing) requires similar steps as setting it up for TAO mode. A good overview of how to write data to a disc in TAO mode can be found in the TAO Multi-Session CD Cookbook [15]. Setting the device for packet writing, writing data to it and closing the disc concerns the following commands (from MMC-6):

1. (optional) Set the drive write speed using `CD SET SPEED`.
2. Set up a Write Parameters mode page and send it to the device using `MODE SELECT`.
3. Use `READ TRACK INFORMATION` to locate the next writable address (NWA).
4. Write data to the media using `WRITE`.

5. (optional) Loop back to 3 to write more data.
6. Finalize the track using `CLOSE TRACK SESSION`.
7. Finalize the disc using `CLOSE TRACK SESSION`.

The important step here is sending the Write Parameters mode page. By setting the `Write Type` field of the mode page to $0x00$, the device shall perform Packet/Incremental writing when `WRITE` commands are issued. We also set `FP` to 1 and `Packet Size` to $0x08$ to use a fixed packet size of 8 sectors for each write.

One could choose to use variable packets instead of fixed size packets. However, by using variable packets, it is no longer possible to use the `READ TRACK INFORMATION` command to locate the NWA. This is because when variable packet sizes are used, the device cannot keep track of the sectors used for overhead, since it does not know the size of the previously written packet. Using variable packets thus forces us to manually calculate the NWA for each `WRITE`. This is feasible, but fixed-size vote records yields a simpler implementation.

To read a burned disc, we close the track and disc as our last two steps. Unfinalized discs do not have a valid TOC and such discs cannot be properly accessed by other devices.

For a discussion about fixed packets versus variable packets, see issue subject 6-3 of [14]. Issue [7] for details of the Command Descriptor Blocks (CDBs) used for each command.

4.2.2 Random Write Access & Recovery

We can now write data to a disc in packet mode, which allows us to write many data chunks to a single track. We do not have the ability to write data to a random location. To achieve random writes, we use the `RESERVE TRACK` previous to the first write sequence. By writing some information to the lead-in of the disc, this command is able to reserve a number of blocks for a track. After reservation of x sectors for the first track, it is possible to write to sector 0 (track 1) or to sector $x+1$ (track 2). This allows us to successfully implement random write access: we reserve two tracks during initialization and, for each vote, the TPM chip generates a random number between 0 and 1. Depending on this value, we append the vote to track 1 or 2. If necessary, more tracks could be reserved to increase randomness.

4.2.3 Recovery

The `RESERVE TRACK` commands are helpful for recovery. For example, consider when vote i is written to disc correctly, but the system crashes before vote $i+1$ is written. A crash will ultimately result in a reboot of the machine, resetting the device's buffer and causing information about where data on the disc can be found to be lost. This data would have been written to disc during the `CLOSE TRACK SESSION` commands. Now, we are unable to get vote i from disc, although we are sure that it is somewhere on it.

Since the RESERVE TRACK command needs to write some information to the disc about the ending of the previous track, we can use this command to implement a recovery function: by reserving i tracks, ejecting the disc without finalizing it (simulating a system crash), inserting it and finalizing tracks $1 \ldots (i-1)$ as well as the disc itself using the CLOSE TRACK SESSION command, we are able to read up to $i-1$ tracks. Thus, by simply reserving an extra, empty track, we can assure that written data to any previous track can still be accessed after a power outage.

Using the above technique, it is possible to add a `check_for_recovery()` function to the initialization procedure of the voting machine software. Here, we can test whether an inserted disc is empty, and, if not, we may try to finalize it using CLOSE TRACK SESSION commands. If the disc was from an unfinished previous session, we ensure that all votes can be read from this disc in a later stage.

One could think of other situations that may cause trouble:

- A crash during finalization (CLOSE TRACK SESSION) (at the end of the election day or during recovery) may result in broken lead-in/lead-out areas, causing the entire disc to be corrupted.
- A crash during the vote-writing procedure (WRITE) may corrupt the vote that is currently being written. The worst that can happen here is recovery failing as well, resulting in an unreadable disc.

Note that crashes during the initialization procedure are negligible: without previously entered votes, no votes can be lost. If we print paper ballots before votes are written to disc, no votes are lost in the latter scenario.

Unfortunately, CD-ROM media are likely to become corrupt. They are vulnerable to scratches and direct sunlight. Even when special care is taken to address these issues, some CD-R discs have bad sectors immediately after the burning procedure. However, the design states that the paper ballots are the actual votes, not those that are stored on CD-R. Thus, CD-R recovery is a nice feature to have but ultimately unnecessary.

4.2.4 Command List

Since the current ATAPI implementation of MINIX 3 can be found in `/usr/src/drivers/at_wini/at_wini.c`, we added our CD-R driver code to `at_wini.c`.

Beside the core functions described before, some more useful commands were implemented. A list of all newly implemented commands and their uses can be found below. Note that most commands are able to execute different subcommands by adjusting their CDB. Below, only the commands that were used in our implementation are shown.

- REQUEST SENSE in `request_sense()`.
  This is a SPC-3 command which forces the drive to provide sense data on the previous command. Although there already existed a `sense_request()` in `at_wini.c`, this new function uses a struct defined in `/usr/src/include`

/minix/cdrom.h to store sense data. This function may be used in future implementations to give a user program detailed information about what happened after a command was issued.

- START STOP UNIT in start_stop_unit().
  This command is used to eject media.
- TEST UNIT READY in test_unit_read().
  This command is used to test whether media is inserted and the drive is ready to process commands. By issuing a REQUEST SENSE after TEST UNIT READY, the sense key can be read to find out what state the device is in.
- PREVENT ALLOW MEDIUM REMOVAL in prevent_allow_medium_removal().
  This command is used to lock and unlock inserted media.
- GET CONFIGURATION in get_configuration().
  This command is used to get the device's configuration. This allows us to check which type of disc is inserted (CD-R, CD-RW, DVD, ...) and also shows us the capabilities of the device (can only read CD, can read CD/DVD and write CD-R, ...).
- SET CD SPEED in set_cd_speed().
  This command tries to set the read and/or write speed of the device.
- MODE SENSE in mode_sense().
  This command is used to read the Write Parameters mode page of the drive. This is currently only used to determine the page length which we need for MODE SELECT.
- MODE SELECT in mode_select().
  This command is used to update the Write Parameters mode page of the drive.
- RESERVE TRACK in reserve_track().
  This command is used to reserve a track of a specific size.
- READ DISC INFORMATION in read_disc_information().
  This command is used to get disc information of the inserted disc. The returned data is stored in a struct that is defined in /usr/src/include/minix/cdrom.h. We use it to determine whether a disc is really empty and not erasable.
- READ TRACK INFORMATION in read_track_information().
  This command is used to get information of a specific track on the inserted disc. The returned data is stored in a struct that is defined in /usr/src/include/minix/cdrom.h. We use it to determine the next writable address of a track.
- WRITE in cd_write().
  This command performs the actual write command and writes data to the disc.
- CLOSE TRACK SESSION in close_track_session().
  This command closes a track or disc.
- BLANK in blank().
  This command blanks a CD-RW in fast blank mode.

After issuing the selected command, each function described above does a `request_sense()`. The sense data that is returned will be stored in a sense struct which must have been provided by the caller.

Since the goal of this project was not to implement a steady CD-R driver, but implementing a voting machine, the return values of above functions are not explicitly defined. At this moment, above functions return one of the following values:

- `-3`: There was an error in the parameter list.
- `-2`: The driver was unable to send the ATAPI packet to the device, or I/O (`sys_insw()` or `sys_outsw()`) failed. Maybe the drive was busy.
- `-1`: The device reports an error. Examine the sense data for more details.
- `OK`: The device reports no error. Examine the sense data for more details.

To allow user processes to execute these commands, disc ioctl codes were added to `/usr/src/include/sys/ioc_disk.h` and `w_other()` in `at_wini.c` was updated. A list of new ioctls that a user can call after installing the new `at_wini` driver can be found below.

- `DIOCLOCK` and `DIOCUNLOCK` lock and unlock the drive.
- `DIOCEJECT` opens the tray (if any) and ejects the disc.
- `DIOCBLANK` blanks the inserted disc, if possible.
- `DIOCREADY` returns 1 if the device is ready, 0 if not.
- `DIOCISBLANKCDR` returns 1 if the device contains a blank CD-R, 0 if not. For this, `get_configuration()` is called to get the current profile of the drive. If this is not CD-R, 0 is returned. If the current profile is CD-R, `read_disc_information()` is called and the returned struct is inspected to ensure that the inserted media is blank and not erasable.
- `DIOCSETSPEED` tries to set the read and write speed of the device to its maximum.
- `DIOCMODESELECT` sets up the device for packet writing with a fixed packet size of 8 sectors (16 KB).
- `DIOCRESERVE` reserves 3 logical tracks on the current disc. Two tracks of 100 MB and a smaller third track of 1 MB.
- `DIOCWRITE` expects a struct `cd_write_param` contains user data and a track number and writes this data to the requested track.
- `DIOCCLOSE` closes the three logical tracks followed by closing the entire disc.

4.3 Over 10.000 Votes on One Disc?

We now examine how many votes we can store on a single disc. We saw in previous subsections that three tracks will be reserved. Two equally big ones for data storage and a smaller one for recovery functionality. Given the characteristics of optical disc media, it is impossible to resume writing at the exact same spot on the disc as we left it before. This means that a certain amount of overhead is inevitable, which

causes a delta between the actual size of a track (i.e., user space) and its reservation size.

If a fixed packet size is used, it is possible to compute the overhead beforehand. [7] states that each written packet will take up an extra of 7 sectors for run-in, run-out and link purposes. This gives us two functions. The first allows in calculation of the available user space depending on the chosen packet size and reservation size. The second one can tell us how many sectors we should reserve, given a packet size and the amount of user space that is desired.

$$ActualSize = (PacketSize + 7)\,\frac{ReservationSize}{PacketSize} - 5$$

$$ReservationSize = (ActualSize + 5)\,\frac{PacketSize}{PacketSize + 7}$$

In the above, *ActualSize*, *ReservationSize* and *PacketSize* are measured in sectors (1 sector = 2048 bytes).

A single track must consist of at least 300 sectors. Using a 700MB disc, which contains 360.000 sectors, our actual size for each data track is now limited to $\frac{360.000-300}{2} = 179.850$. We round this value to 179.000 to conservatively avoid errors.

Naturally, a larger packet size reduces the overhead and provides more actual available user space. Unfortunately, larger packet sizes also result in less available packets, which narrows the available number of votes that can be stored on a disc. This is shown in figure 1.

Notice that we cannot permit a packet to contain multiple votes, since each vote must be written to disc directly after it was casted. Writing only a half packet is not possible. Waiting for another vote to fill a packet is not desired, since a crash can destroy the first vote.

Although we can divide a vote over multiple packets, each vote will be the approximate same size. This forces every vote to be divided over multiple packets. To avoid packet storage overhead, it is better to increase the packet size. We choose a packet size where each vote can fit in a single packet. Hence, the number of packets limits the number of available votes.

A packet size of 8 or 16 blocks, allowing 16KB or 32KB per vote respectively, will limit the number of votes to around 10.000. This is sufficient for any type of election.

4.4 Issues during Implementation

Below, we outline the interesting issues that future MINIX 3 developers may find useful.

4.4.1 Prepare Issue (solved)

We experienced some strange behavior during tests of a first version of the driver. We tested a program that first ejected the tray, followed by waiting for a blank

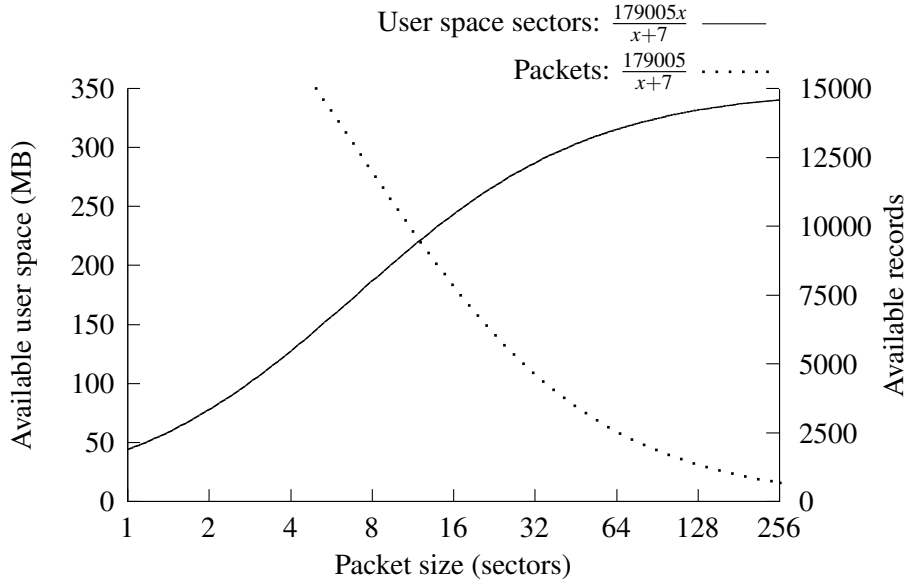

Figure 1: A larger packet size reduces the overhead (left y-axis), but also the number of available votes (right y-axis)

CD-RW to be inserted. `sleep(1)` was called between each `TEST UNIT READY` command. We executed this program two times in a row. Strangely, the first iteration worked perfectly fine, while the second timed out.

After some debugging, we found the issue in the `w_prepare()` function in `at_wini.c`. Each time we wish to perform device I/O, we must call this function. If not — as we did — `at_wini` accesses the device that was used during previous hardware I/O. Somehow, between the first and second iteration of our test program, another process accessed the hard disk after which our driver tried to access the hard disk as well instead of the CD-ROM drive. Hence, by doing a `w_prepare(m->DEVICE)` before sending bytes to the device, this issue was solved.

4.4.2 Time-out Issue (solved)

Most implemented commands discussed earlier set the `IMM` field in their CDB to 0. This implies that the device does not return control to the operating system as long as the requested command is not completed. Since closing a disc may take some time, we needed to increase the `WAKEUP_SECS` time-out value defined in `at_wini.c` from 30 seconds to 600 seconds, assuming that 5 minutes should be long enough to close any disc at any speed.

When the `IMM` field is set to 1, one must issue `TEST UNIT READY` commands periodically to determine completion of the current operation. Devices may provide detailed sense data to report a command's completion percentage. Since `WRITE` commands do not take a lot of time to complete and `CLOSE TRACK SESSION`

is supposed to be called once at the end of election day, adding this functionality to our implementation provides little benefit.

4.4.3 Large Write Buffer Issue (unsolved)

The `WRITE` command allows us to send any number of bytes to the device. This behavior is implemented in different steps. First, we send a 'write request' to the device, telling it how many bytes we want to send, say s. The devices replies with the number of bytes it is willing to accept at once, say n. We now write n bytes to the device, after which we wait for a reply again. This is repeated as long as $n < s$. The value of n is determined by the internal buffer size of the device. To burn data at higher speeds, it is desired to use this buffer extensively and send large data chunks, so that the overhead of processing `WRITE` CDBs is reduced to a minimum.

Unfortunately, we were unable to use data chunks larger than 8 blocks (16 KB). Although the sending of a `WRITE` CDB with LBA = 16 resulted in a 'positive' reply from the device, writing 16 KB of data to the device using the `sys_outsw()` function triggered a MINIX crash.

Debugging showed an issue with the `rep 016 outs` instruction in `phys_outsw()` (in `/usr/src/kernel/i386/klib386.s`). Adjusting `phys_outsw()` so that only one `outw` was executed, showed that the crash occurs when the `outw` instruction is executed more than 8240 consecutive times.

To eliminate the chances of this crash being caused by faulty hardware, we posted a "request for testing" on the MINIX 3 google group[16]. Unfortunately, we got no replies. Hence, we cannot say anything about the origin of this problem. Future MINIX 3 CD-R implementers need to take this behavior into consideration.

For this project, using a smaller buffer size is sufficient. However, since this problem limits the maximum write speed of the device, write support for DVD or Blu-Ray may be impossible to implement.

4.4.4 Strange `read()` Behavior (unsolved)

While implementing step 9, where data must be read from the written disc again, we experienced some strange `read()` behavior in MINIX. Reading data from the first track (`lseek(cdrom_fd,0,SEEK_SET)`, followed by `read()` calls) were successful. However, reading data from the second track failed. Running `cdrskin -toc` under Linux showed that the second track was located at sector 96145. This would require a `lseek(cdrom_fd,96145*2048,SEEK_SET)`, followed by `read()` calls to start reading data from this track. The data read, however, was not the data that we expected. It seemed that we read from sector 96153 instead (the next packet).

We did some more tests using the `dd` command which allow direct data copies from CD-ROM to disk. We found that we needed to start reading from sector 96140 or 96138 to obtain the record that was supposedly located at sector 96145. Linux'

`dd` implementations showed no problems whatsoever, which is why we ported the `getresults.c` program to Linux, using stream I/O.

More investigation into this issue is necessary.

4.5 Future Work on CD-R Support for MINIX 3

DVD read support and Blu-Ray read support are the next logical steps for MINIX 3 media support.

In addition to DVD-read support, needed abilities in the CD-write driver include writing discs in TAO, DAO, SAO or packet writing mode. One would have to think about how to provide all the possible options (e.g., a MODE SELECT). A solution may lie in the implementation of a new library which goal is to parse CD-R requests from user space.

Future CD/DVD implementers should take into consideration that the current implementation does not have DMA support.

5 Future Work

A few more things remain before the completion of a trustworthy voting system implementation. One gap is the issue of machine attestation. Implementing attestation requires knowledge of TPM programming and is outside the scope of this work.

In addition to attestation, Step 7 & 8 needs two updates before deployment. By adding bit blitting code to MINIX, simple bitmap images can be read and displayed on the screen. Because bit blitting is fast and requires a small code base, it meets our need for a simple voting machine GUI.

While the attestation and bit blitting are useful contributions, stripping down unneeded voting machine functionality would contribute to a smaller code base (and thus to reliability). Before stripping down MINIX, porting the current implementation to the latest version of MINIX 3 will incorporate new bugfixes. Except for our modified `at_wini.c`, which may require some minor adjustments before it compiles on later releases, other source files should compile without any modification.

Apart from the voting machine, more work may be done to extend the CD driver. Up till this moment, there is no secure mechanism in any OS to write data to CD's. We saw in Section 4 that current *NIX based CD-R burning tools let users interact with the hardware directly. This can lead to instable systems. A different approach in where a new system layer is added that takes care of sending commands to the device and retrieving replies may be an approach that is more secure. This design could disallow users to send a CDB with the IMM bit set to 1, ensuring continuation of other system processes.

In MINIX 3, this new layer would be an entirely new driver, probably heavily based on and partially replacing the current `at_wini.c` implementation. Future

source code that will support more optical disc features (DVD/Blu-Ray support) should be placed outside `at_wini.c` to avoid code bloat.

References

[1] Nathanael Paul and Andrew S. Tanenbaum. The design of a trustworthy voting system. Computer Security Applications Conference, Annual, 0:507–517, 2009.

[2] OpenSSL Cryptographic Library. `http://www.openssl.org/docs/crypto/crypto.html`.

[3] OpenSSL EVP Manpage. `http://www.openssl.org/docs/crypto/evp.html`.

[4] Sean Barnum. GETC. `https://buildsecurityin.us-cert.gov/bsi-rules/home/g1/746-BSI.html`.

[5] OpenSSL BIO Manpage. `http://www.openssl.org/docs/crypto/bio.html`.

[6] Organization for the Advancement of Structured Information Standards. Election Markup Language (EML). `http://xml.coverpages.org/eml.html`, 2008.

[7] Bill McFerrin. Multi-Media Commands - 6 (MMC-6), 2006.

[8] Ralph. O. Weber. SCSI Primary Commands - 3 (SPC-3), 2005.

[9] Dal Allan, Tom Hanan, and Devon Worrell. Information Specification for ATA Packet Interface for CD-ROMs, 1996.

[10] Chip Chapin. Chip's CD Media Resource Center: CD-DA (Digital Audio). `http://www.chipchapin.com/CDMedia/cdda1.php3`, 2005.

[11] Ecma. Data interchange on read-only 120 mm optical data disks (CD-ROM), 1996.

[12] Jörg Schilling. CDRecord. `http://cdrecord.berlios.de`.

[13] Thomas Schmitt. Cdrskin. `http://libburnia-project.org/`.

[14] Andy McFadden. CD-Recordable FAQ. `http://www.cdrfaq.org/`, 2010.

[15] Thomas Schmitt. Optical Media Rotisserie Recipes. `http://www.libburnia-project.org/browser/libburn/trunk/doc/cookbook.txt`, 2010.

[16] Victor van der Veen. Request for cd-write testing. http://groups.google.com/group/minix3/browse_thread/thread/d911936ef8e16259/7f3a4a244a384e18#7f3a4a244a384e18, April 2010.