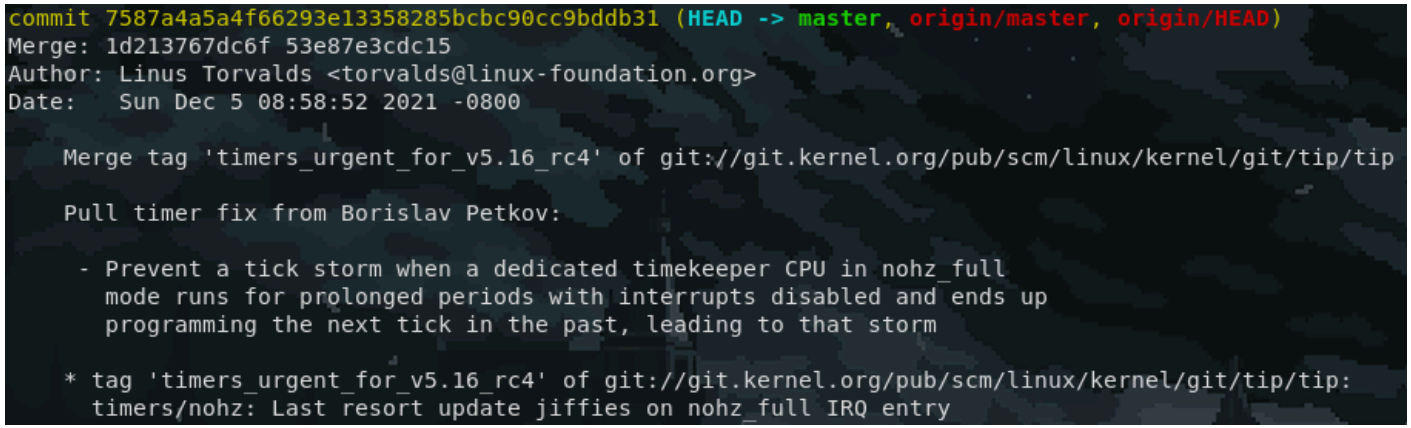
01. [25p] Commit signatures

When looking at the `git log` of a repository, you would normally see a sequence of entries such as this:



Each commit is identified via a 40-character long **hexstring**. A hexstring is the hexadecimal representation of a sequence of bytes, where every nibble (i.e.: 4 bits) is represented by a character ranging from *'0'* (= 0b0000) to *'F'* (= 0b1111). But how is this identifier calculated?

Well… this commit identifier is called a **hash value** or a **digest**, and is the output of a hash function [https://en.wikipedia.org/wiki/Hash_function]. A hash function takes an arbitrary amount of data and outputs a fixed-size bit array that is representative of the input. Normally, one would be weary of collisions: if the function's domain is virtually infinite and the co-domain is not only finite but rather small in comparison, wouldn't it be possible to create two commits with the same digest? Possible – yes, likely – no. Git uses **SHA1**, a cryptographic hash function [https://en.wikipedia.org/wiki/Cryptographic_hash_function]. What makes a cryptographic hash function so special is that it provides certain guarantees. For example, it should be impossible to calculate potential messages from a digest (meaning that the function is non-invertible). Moreover, any change in the input – no matter how small, should change the hash value so extensively that the new value and the old should appear uncorrelated. Consequently, being able to *craft* a commit such that it's not only comprehensible, but also creates a certain desired digest is so unlikely that it occurring naturally should not pose any risk.

There is, however, a more significant risk here. What guarantee do you have that the author of the commit above is actually Linus himself? Using something like git-blame-someone-else [https://github.com/jayphelps/git-blame-someone-else], you could overwrite commits and change their author only to then `git push --force` and replace the remote history (don't do this on the master branch if you're not working alone). The answer to this problem is **commit signing**. Cryptographic algorithms can be loosely categorized an **symmetric** and **asymmetric**. Symmetric algorithms like AES [https://en.wikipedia.org/wiki/Advanced_Encryption_Standard] use the same key for both encryption *and* decryption. Asymmetric algorithms like RSA [https://en.wikipedia.org/wiki/RSA_(cryptosystem)], on the other hand, utilize two keys. One for encryption and one for decryption. Usually, the one used for encryption is called the private key and the one used for decryption, the public key. The private key is your identity, so you don't share it with anyone. The public key you configure on remote servers to give them the ability to verify your identity (e.g.: configuring SSH keys on fep.grid.pub.ro). As a rule, you use asymmetric cryptography in cases where you need to prove your identity to a remote host, establish secure communication channels over an untrusted network, etc. The reason for this is that asymmetric algorithms are orders of magnitude slower than their symmetric counterparts. While encrypting the output of a **SHA256** function (32 bytes) with a 4096-bit RSA key takes about 5ms on regular CPUs, it takes almost a full minute on an Arduino Mega (with a MCU running at 16MHz). AES-256 on the other hand, encrypts the same amount of data in less than 5 microseconds. The downside is that you need to share your key with other systems for them to extract the plaintext message.

In the following tasks we will introduce **GNU Privacy Guard (gpg)** an open source encryption and signing tool. **git** can use **gpg** to sign your commits as you create them. For it to work, you will also have to upload your public key to github [https://github.com/settings/keys].

[15p] Task A - GNU Privacy Guard

As we mentioned before, **gpg** is an encryption and signing tool. Additionally, it also handles key management. When you first create a keypair, it will also construct you a **key ring**. This key ring can store both private keys (i.e.: your keys – you can have more than one) and multiple public keys. These public keys can belong solely to you, or to persons that you trust.

Let's say that you have *Person A*'s public key and *Person B* has your public key. If you want to introduce *Person A* to *Person B*, you can give his public key (that you have on your key ring) to *Person B*. If you're doing this over a secure channel (e.g.: you write it on a piece of paper and then hand it to *Person B*) all's well and good. However, if you send it over an untrusted medium (e.g.: the Internet), you should sign *Person A*'s public key with your private key. As a result, *Person B* can validate the integrity of *Person A*'s key with your public key.

This example with *Persons A and B* is characteristic of the Web of Trust [https://en.wikipedia.org/wiki/Web_of_trust] model. In it, every participant is responsible not only for his own key, but for those that he decides to add to his key ring – and thus, implicitly trust. As you can imagine, validating each new person (or server) that you communicate with can become taxing. Consequently, it's no surprise that the most prevalent model today is the Public Key Infrastructure [https://en.wikipedia.org/wiki/Public_key_infrastructure]. Further discussing the differences between these two models falls beyond the scope of this lab. However, this is a topic that you should research sooner rather than later (so don't wait for it to be mentioned during lectures).

For now, let's get you started by generating a brand new 4096-bit RSA keypair. Make sure to introduce your real name and email. At some point, you will be prompted for a password. That password is to secure your key ring. Its only purpose is to prevent attackers that have gained access to your computer from using your keys. Otherwise, the password will not factor in any of the cryptographic operations that **gpg** performs.

```
# generate key
$ gpg --full-generate-key

# list keys
$ gpg --list-public-keys
$ gpg --list-secret-keys
```

Now, let's upload your public key to https://github.com/settings/keys [https://github.com/settings/keys]. Click on `New GPG key` and copy paste the output of the following command:

```
# export public key
$ gpg --armor --export ${YOUR_MAIL_GOES_HERE}
```

Notice the `--armor` flag. Without it, the command above would write your key in a binary format to *stdout*. With it, **ASCII armor** is applied to the output. ASCII armor is a technique that encodes your binary data using algorithms such as base64 [https://en.wikipedia.org/wiki/Base64] in order to make it ASCII-printable. Because only a subset of 8-bit values are printable, a number of bits smaller than 8 (e.g.: for b64 encoding that number is 6) are expanded to a full ASCII printable byte. Naturally, this leads to output inflation but at the very least you can copy paste data or send it via email without attachments.

From now on, you can add the `-s, --signoff` flag to `git commit` and it will automatically add a *"Signed by …"* message at the end of your commit message and sign it with your default private key. Alternatively, if you don't want that trailing message, you can use `-S, --gpg-sign[=<keyid>]`. If you want to configure **git** to automatically sign all commits with a certain key, you can do so thusly:

```
# get your private key fingerprint (long upper case hexstring)
$ gpg --list-secret-keys
```

```
# configure your default signing key (git user.{name,email} must match those in key)
$ git config --global user.signingkey ${PRIVATE_KEY_FINGERPRINT}

# enable autosigning
$ git config --global commit.gpgsign true
```

[10p] Task B - Your first signed commit

Create a public repository on GitHub and add your music bot from lab 3 as a first commit. Make sure you use `git commit -s`, then push your changes to remote. Go to the *"github.com/…/commits"* page of your repo and check that your commit has a green *"Verified"* tag next to it.

Troubleshooting

If `git commit -s` fails, there are usually two reasons. Prepend `GIT_TRACE=1` to the command and run it again. Look for the last command executed in the background by **git** before the error occurred (most likely an invocation of **gpg**) and run it separately. This will give you a more verbose error that will let you determine the problem. Here are the usual suspects:

- The `user.name` and `user.email` values that you previously configured with `git config [--global]` differ from what you specified when creating the **gpg** public keypair. Remember that `user.name` is not the same as your account username on GitHub. It is the name under which you publish your commits and can be whatever you want it to be, but be consistent!
- **gpg** has no idea how to ask you for the passphrase needed to decrypt your keyring. This might be an issue when invoking it indirectly from **git**, usually from wsl [https://docs.microsoft.com/en-us/windows/wsl/install]. Add `export GPG_TTY=$(tty)` to your *.zshrc* or *.bashrc* and source it. This environment variable will let **gpg** know that it should use an ncurses [https://en.wikipedia.org/wiki/Ncurses#:~:text=ncurses%20(new%20curses)%20is%20a,runs%20under%20a%20terminal%20emulator.]-based terminal prompt. Note that after you input your passphrase once, it will be temporarily cached by a daemon named **gpg-agent**. As a last resort, try running `gpg --sign ${SOME_FILE}` just so `git commit -s` won't ask you for it afterwards. Again, this is only a workaround, not a solution. Ask your assistant for help!