HUSKY RECORDS MUSIC PLAYER DESIGN DOCUMENTATION

Version 1.1a
Final Release

Cameron Kennedy
Christopher Brown
Dennis Giese
Erik Uhlmann
Trey Del Bonis

www.huskyrecords.net

Advised by: Prof. Guevara Noubir
(Northeastern University)

 German Engineering

 Made in USA

Overview

System design idea

Our first line of defense is our Terms of Service and EULA, where we sue everyone who dares to hack our platform or to copy the music. Our big idol here is the very famous Recording Industry Association of America (RIAA). We spent most of our design time and budget in discussing the EULA and deciding on the team name[1], until we actually figured, that the EULA is not enforceable and not legally binding. The remaining time we tried quickly to implement a technical defense against attacks.

We have created a cryptographic scheme for protection and verification of song data as well as auxiliary data such as user information. Our scheme attempts to cover as much of the requirements as possible using only basic cryptographic primitives such as Blake2b hashes, XChaCha20 block encryption, and EdDSA signatures, ensuring that the only way to decrypt songs is with the knowledge of the appropriate user and region secrets. Additionally, we use Certified Web Scale Blockchain Technology™ to verify the integrity of song data as it is playing, ensuring that it is impossible to play an unauthorized song.

In the event that unintended operation of the secure processor occurs, we invoke the DAB mechanism (Data and execution Access 🅱️revention) in order to halt future operation of the secure processor until the device is reset. We also perform a DAB invocation whenever we detect other internal errors. Due to our aggressive application of the design recipe, these kinds of errors should not occur under normal operation. We also make use of the Xilinx brownout detection and other security features to trigger a DAB in the event of glitching attacks. We have disabled hardware debug modules in order to disallow trivial memory inspection, disabled hardware readback of the FPGA bitstream in the build configuration, and have made efforts to prevent usage of ptrace and other revere engineering utilities in the miPod player.

Security goals

Music Integrity Protection

We protect our system against playing unauthorized songs by mandating that all songs are signed by the factory at multiple levels. Forging these signatures requires knowledge of a private key that never leaves the factory and is not present on customer devices. We also ensure that the key material to decrypt region locked songs does not exist on devices for other

[1] By reading this document you confirm to have read our EULA, not too seriously, which can be found here: https://www.huskyrecords.net/eula.txt

regions. Users that do not have access to songs additionally do not have the keys to decrypt the metadata either.

User-specific keys are generated only on an as-needed basis using Argon2id, and do not exist up until this point.

We also avoid needing to sign each individual song block by instead signing a Merkle root and generating inclusion proofs for each block against this Merkle root. This requires that the untrusted side have the entire song available as the song being loaded is required to generate these proofs.

Data and execution Access 🅱revention

The Data and execution Access 🅱revention (DAB) mechanism is a bleeding edge new tool activated when certain anomalous behaviors are identified. The secure processor will clear all buffers, sensitive and nonsensitive, then reset itself. Performing a DAB is a defense procedure that should not occur when using the Husky Records standard miPod music player. Use of other players is not supported and Husky Records is not liable for any damages, real or perceived, from using nonstandard players.

Protection of flags

In many places keys are encrypted and payload data is signed. The only place any of these layers of encryption are unwrapped is on the secure MicroBlaze processor. The structure of these keys can be found in the Data Structures section.

Region Lock

“play a song from a region that the player is not provisioned for”
When the device is provisioned for a region it’s assigned a region specific key used to decrypt the region data block corresponding with that region. This block includes the song decryption key used to decrypt the song stream. If a player doesn’t have the key for any of the regions of the song then it simply doesn’t have enough information to decrypt the song at all.

Unauthorized Play

“play a song that the user does not have access to”
Similarly to the region locking, users must also have the decryption key for the songs they own in order to be able to play them. So, if a user does not have access to a song, they are unable to decrypt the grant information containing the song decryption key. Any attempt to bypass the miPod userland application results in a reset and temporary lockdown of the DRM module.

Pin Extraction

“obtain the pin for another user”
User pins are never stored on the device at rest, and user pins are only stored transiently during hashing. The username and pin are used to create HMACs using the secret keys of the miPod application and the DRM module. The resulting HMAC is hashed with a salt using Argon2ID in order to verify the pin against the user information file. The Argon2ID execution is done on the insecure ARM core, and then transferred to the secure processor for verification and then transformation into the user key. After the user logs out or the device is reset the user keys are destroyed.

Music Tamper

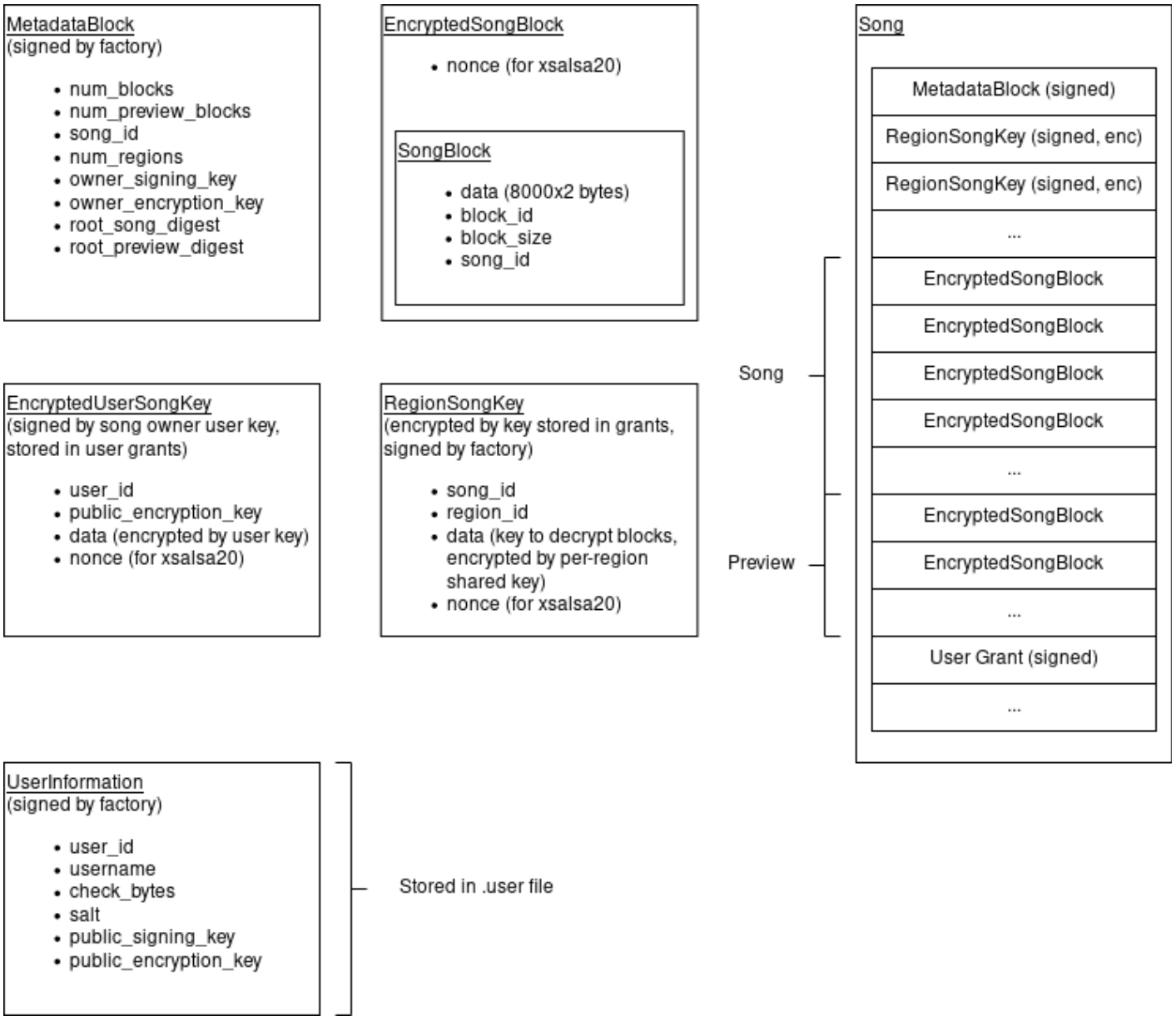
“modify a protected song”
In the song metadata block are the Merkle roots for the song data block sequence and the song preview block sequence. In addition to the blocks themselves, the player must provide the secure processor a Merkle proof of each block, proving that the block is committed to in the Merkle root. The metadata block must be signed by the factory so this also requires that the Merkle roots be blessed. Additionally, the song_key (which may be derived from the user_song_key and the region_song_key) is trusted because both of its components are signed by the factory and bound to the song via the song_id (which itself is signed and stored within the song metadata).

Custom Music

“protect a custom song, optionally using a custom player program”
A user would need to know multiple encryption keys and to forge multiple signatures in order to play unauthorized music. The miPod application is protected against reverse engineering. Every attempt to bypass it, results in a temporary lockdown of the DRM.

Data Structures

These are the core data structures, for a full description of all data structures including the different wrapping layers, see `mb/crypto_types.h`.



Features

miPod supports the following WAV files features

- Songs up to 128 megabytes.
- Songs with a sample rate of 48000 Hz.

miPod supports the following functionality

- Play: plays the song
- Stop: stops the song
- Pause: pauses the song
- Restart: restarts the song
- Fast forward: seeks forward
- Rewind: seeks backwards
- Share: share a song with another user
- Query: get the users and regions a song is provisioned for
- Multi-language support (German, English)

Implementation

Keys and Configuration

- Factory
    - **factory_enc_sk, factory_enc_pk:** Ed25519XChacha20 keypair; used to encrypt **region_song_keys** in drm song files.
    - **factory_sign_sk, factory_enc_pk:** Ed25519SHA512 Keypair; used to sign **region_song_keys** and **user_song_keys** to ensure that they are valid, and (transitively) prove that the derived **song_key** is also valid.
- Device
    - **mixkey:** 16-byte Prince key; used in the key derivation process as a step for deriving private keys within the microblaze component from seeds provided from the miPod application.
    - **hmac_key**: 32-byte HMAC key; for creation of the HMAC of the user login data and later the final step of the key derivation process, used to derive private keys from the result of applying the **mixkey** to the miPod-provided seed.
- Region
    - **region_sk, region_pk:** Ed25519XChacha20 keypair; A keypair used in conjunction with the factory encryption keypair used to encrypt and decrypt **region_song_keys** stored in drm song files.
- User
    - **Pin:** 8-64 digit pin; used by users to authenticate with the miPod application and the starting point for deriving their private keys.

    - **Seed**: 68-byte result of argon2kdf; derived from applying argon2kdf to user pins in combination with a unique salt as the first step in the key derivation process.
    - **user_enc_sk, user_enc_pk:** Ed25519XChacha20 keypair; used in conjunction with those of other users to decrypt **grants** issued by the song owner in order to play songs.
    - **user_sign_sk, user_sign_pk:** Ed25519SHA512 Keypair; used by song owners to sign **grants** they create for other users so that those users may listen to songs.
- Song
    - **song_key**: XChavha20 key; used to encrypt **song blocks**.
    - **region_song_key**: 32-bytes random data; combined with **user_song_keys** via a PRINCE decryption to derive **song_keys**. These are first signed by **factory_sign_sk** and then encrypted with a specific **region_enc_pk** and the **factory_enc_sk** and placed in drm song files at provision time. They are unique per song.
    - **user_song_key**: 32-bytes random data; combined with **region_song_keys** via a PRINCE decryption to derive **song_keys**. These are first signed by **factory_sign_sk** and then encrypted with the song's owner's **user_enc_sk** and the grant recipient's **user_enc_pk**. They are unique per song.

Special Modules and Security features

Kingcrab

Custom FPGA core that adds security. It binds key derivation computation to the FPGA using a key that is generated at build time. As it is implemented in the FPGA fabric, and not the Microblaze, reverse engineering is more difficult. The modules are used to transform the key derivation data which was sent by the miPod application. This prevents external brute forcing of pins and enforces rate limiting. The Kingcrab module is based on Prince encryption.
Hint: Running the Synthesis will report a timing issue with the Kingcrab FPGA Core. This is expected behaviour and is part of one of our security measurements.

Custom seeded crypto

Seeds and constants for hash and crypto function generated at build time. These seeds can be only extracted by side channels or reverse-engineering.

Required presence of components for key derivation

For the login and key derivation the presence of the protected miPod app and the DRM module is required. Each component contains secret keys which are used in the process. An offline generation of seeds or key material is therefore not possible. The DRM module and the miPod app enforce rate-limiting individually.

Environmental monitoring

The Husky Records Music Player uses state of the art hardware monitoring features, which are provided by the Zynq 7000. The XADC FPGA core reads the current chip temperature and various voltages and resets the DRM module in case it violates defined thresholds.

Glitching and Timing attacks prevention

“To serve and reset”

The well proven “Protectonator” script from last year's design is reused this year again. It adds random delays and computations into the code at build time and verifies the correct execution at runtime. In case of mis-computations, the Microblaze memory will be purged and the DRM will be reset. To limit the rate of resets, the design enforces a 3-5 seconds waiting period after loading the DRM module.

Binary packing

The miPod binary is packed by the factory. This helps save space and makes it difficult for attackers to reverse engineer the miPod application. The binary is packaged with UPX and sstrip is run on the resulting binary. This results in UPX being unable to unpack the binary from the command line.

I2S tweaking

The design uses customized clock frequencies for the ARM core, the DDR memory and the PL. Due to the implementation of the given I2S module, we need to modify the clock frequency for I2S. The default 24,576 MHz[2] needs to be multiplied with the ratio to the current PL frequency and 100MHz. In case of 111 MHz the I2S clock frequency needs to be 27.306MHz. Without the clock adjustment, the sound output is too slow or too fast.

Multi-language support

The implementation of the software supports the usage of different language files. These can be selected at build time. For convenience, the language files for English and German are provided.

Requirements

The provisioning requires Python3.7 and several python modules. They are listed in requirements.txt and can be installed with pip. Also, the merkle_bindings C module has to be built. This can be done by running `python3.7 build_merkle_bindings.py` in the tools folder. Building the miPod application requires UPX and sstrip.

[2] See https://reference.digilentinc.com/reference/pmod/pmodi2s2/reference-manual

Provisioning Process

The provisioning process is largely the same as the reference implementation. See above for the kinds of keys that we generate during the provisioning process.

NOTE: All intermediate build files (aside from media files to be distributed to customer devices) are to be kept confidential and must not be distributed. This is especially true for secrets files that have not been properly packaged!

createRegions

Syntax: “createRegions --region-list <regions> --outfile <region.secret>”

Description: This script creates the region secrets file. It creates an entry for every region provided and writes a file to the given output

Output: A json file with the fields “regions”, “factory”, and “device”

- regions
    - Maps a region name to the region’s name, id, public key, and secret key
- factory
    - The factory’s enc_sk, enc_pk, sign_sk, sign_pk, sign_sk_seed
- device
    - The preview key

createUsers

Syntax: “createUsers --user-list <user:pin> --outfile <user.secret>”

Description: This script creates the users secrets file. --user-list is a list of “user:pin” combinations. It creates an entry for every user provided and writes a file to the given output

Output: A json file with the fields users and device

- users
    - Maps a username to the user’s name, id, secret key, public key, sign secret key seed, sign public key, salt, and check bytes
- device
    - The key used by the kingcrab FPGA block

protectSong

Syntax: “protectSong --region-list <regions> --region-secrets-path <region.secret> --outfile <song.drm> --infile <song.wav> --owner <user> --user-secrets-path <user.secret>”

Description: This script protects a given file song file. The song is owned by the user given and can only be played on devices provisioned with a region in the region list.

Output: An encrypted song. The song format can be found at the end of this document.

createDevice

Syntax: “createDevice --region-list <regions> --region-secrets-path <region.secret> --user-list <users> --user-secrets-path <user.secret> --device-dir <device>”

Description: This script creates files needed for a device. It creates files that are used by the DRM and miPod.

Output:

- `mb/secrets.h`
    - The region signing keys (if the device is not provisioned for a region the key is 0)
    - The factory encryption public key
    - The factory signing public key
    - The preview key
- `miPod/secrets.h`
    - Provisioned regions as an array of booleans
    - The factory signing public key
- `miPod/user.user` (a file is created for each provisioned user)
    - The user’s id
    - The username
    - The user’s encryption public key
    - The user’s signing public key
    - Check bytes
    - Salt
- `miPod/regions`: List of all regions created
- `miPod/users`: List of all users created
- `xor`: Mix key used by the “kingcrab” FPGA block

buildDevice

Syntax: “buildDevice -p <dev_path> -n <project_name> -bf <build_flag> -secrets_dir <device>”

Description: This script builds the bitstream and the miPod application. It outputs the files into the device folder given by -secrets_dir. Specific sections can be run by using the build flag. The sections are cs (copy secrets), cp (create project), gb (generate bitstream), bm (build microblaze and miPod), cb (combine bitstream), or all.

Output: The bitstream containing the DRM program and the obfuscated miPod program

packageDevice

Syntax: “packageDevice <template.bif> <miPod.bin> <download.bit>”

Description: Creates a miPod.bin using the created bitstream

Output: The miPod.bin

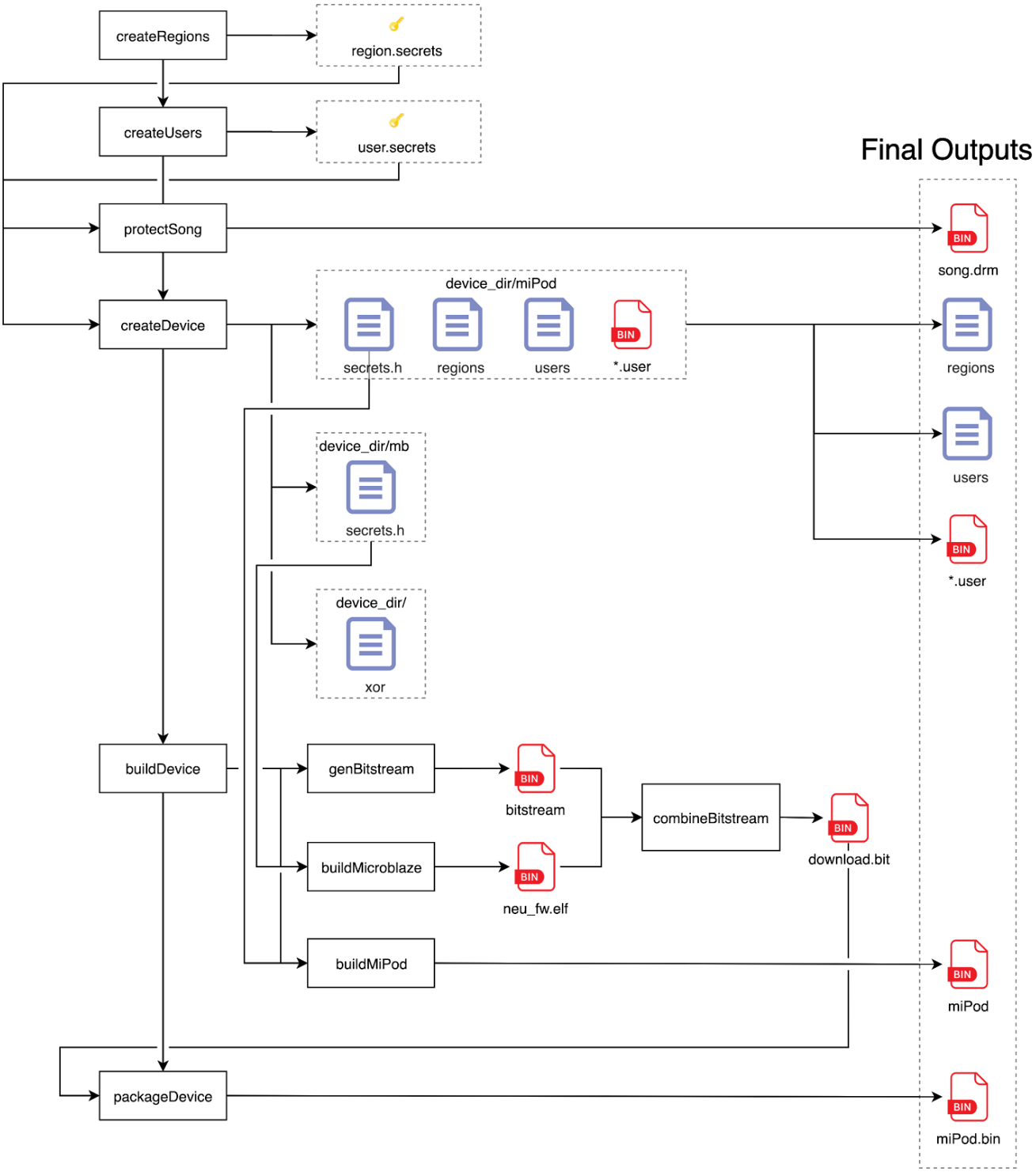
deployDevice

Syntax: “deployDevice <SD card> <boot.bin> <audio_folder> <miPod_folder> <image.ub> --mipod-bin-path <miPod.bin>”

Description: Deploys a device to a SD card. audio_folder contains all of the encrypted songs that are to be deployed and miPod_folder is the path to the miPod folder in the device folder.

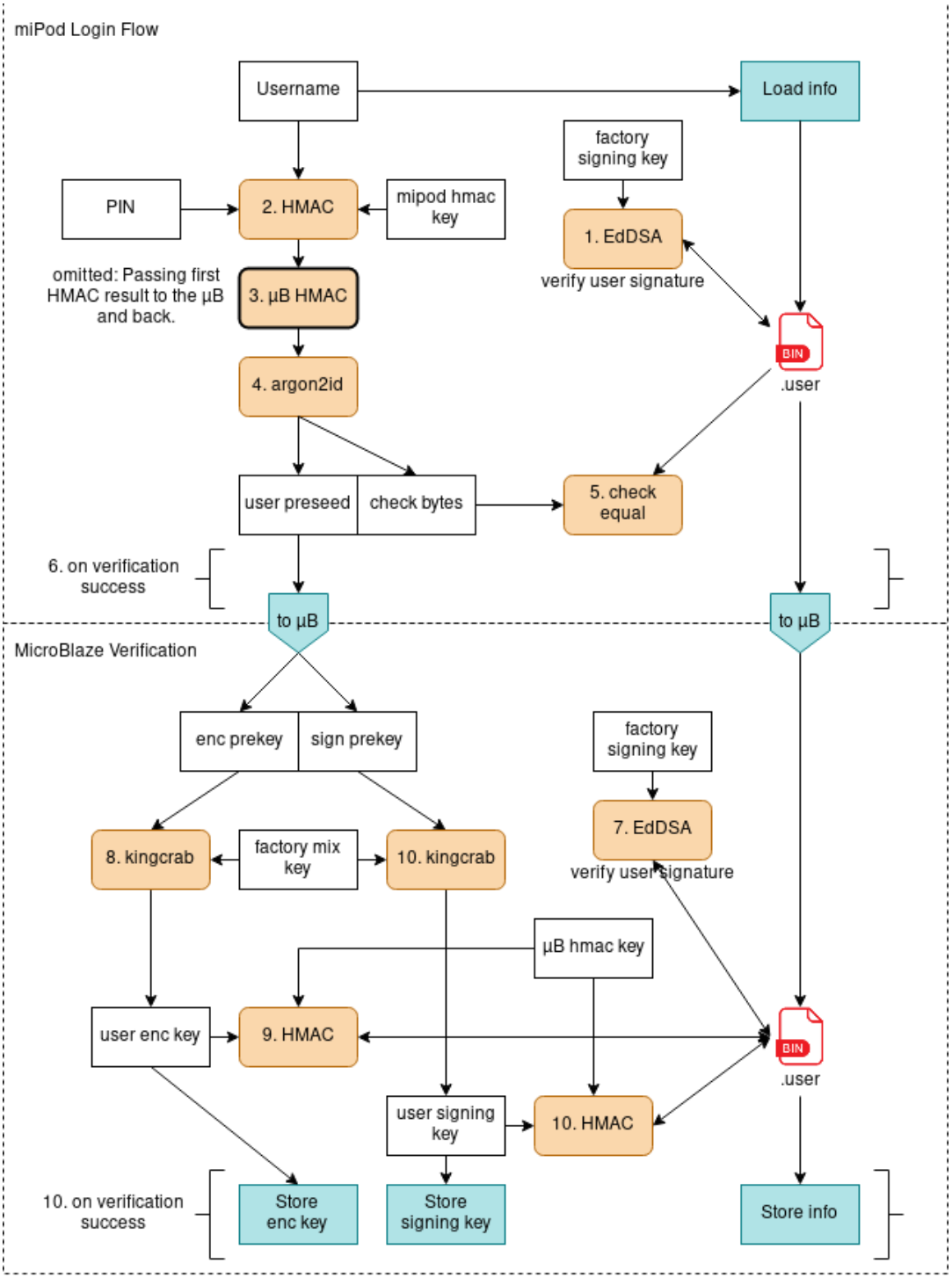
Output: A provisioned SD card

Provision Flowchart



Login Process

Numbers on orange boxes notate order of execution.



Login Detail

This is an elaboration of the above steps. The numbers show the general evaluation steps, but this is more specific on what's actually being computed.

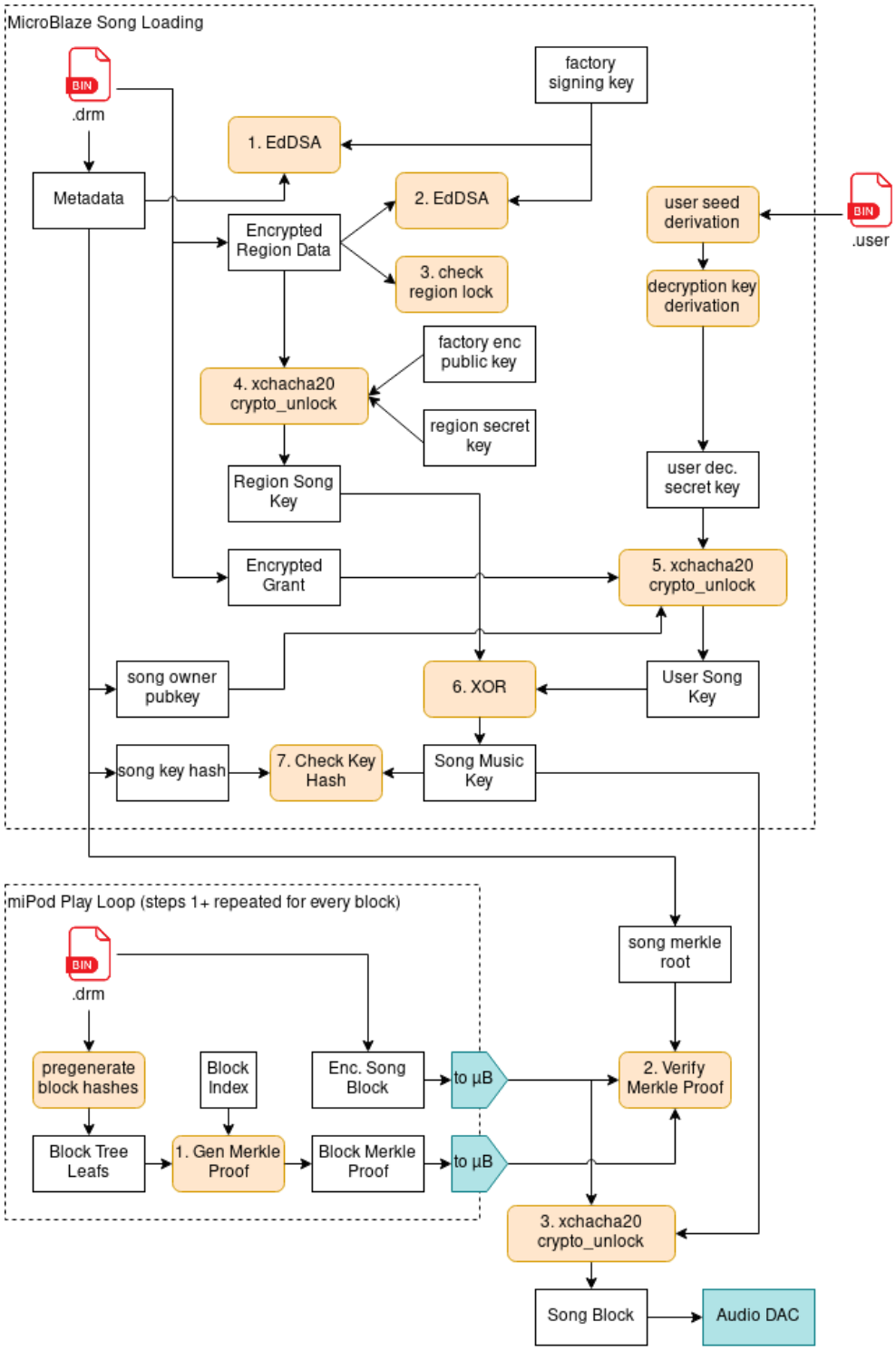
1. We first verify that the user's information file is signed by the factory. The distributor is the sole source of truth for this kind of information. It should not be possible to construct arbitrary users. We verify a signature on the user information committing to a fixed set of user parameters. See the user information structure above.
2. Compute Blake2b HMAC of the user PIN and username with a miPod-specfic HMAC key, verify against stored hmac in user information file.
3. Query the DRM module to compute an Blake2b HMAC with the DRM module secret key. (This is a proof of the presence of the DRM module.)
4. Apply Argon2ID on the HMAC to derive preseed.
5. Verify the part of the result against the check bytes in the user information file.
6. Pass the rest of the KDF result with the signed user information to the secure DRM processor.
7. On the secure processor, we verify the factory signature on the user information.
8. Run the first half of the preseed through the kingcrab module, computing the user encryption key.
9. Compute a Blake2b HMAC of the encryption key using a microblaze-specific HMAC key.
10. Run the second half of the preseed through the kingcrab module, computing the user signing key.
11. Compute a Blake2b HMAC of the signing key using a microblaze-specific HMAC key.
12. Store the results and mark the user as logged in.

Most of this logic is implemented here: /mb/neu_fw/src/crypto.c

If any of the verification steps on the secure processor fail, it performs a DAB. This file also is responsible for much of the logic used in the following section on the song play loop.

Music Playing Process

This diagram is similar for the preview play code path, except fewer decryption steps are used and a different merkle root for just the preview is used instead. The miPod is also responsible for loading the correct region and grants to the microblaze.



Song Load Detail

There's two main phases in playing songs. There's the load phase and the play phase. Some earlier validation steps happen in the miPod player, but the primary logic happens on the secure processor. If any verification step fails, the secure processor performs a DAB.

1. First we verify that the song metadata is authentic against a factory public key. Only the vendor should be able to issue songs. We commit to all of the blocks of the song (and the preview) through a Merkle root in this metadata header, so we indirectly are pre-committing to verifying the contents of the song (see later).
2. Secondly, we verify the region block corresponding to the region of the device. We verify this signature so ensure that we do not attempt to play songs we do not actually have a valid key for.
3. We verify that the region ID of this song is a region ID we have a key for. This is simply checking against an ID we have stored in the firmware.
4. We decrypt the region block with some stored key information to reveal the region key. This key forms half of the song key.
5. Using the users secret decryption key we derived during login, we decrypt the user grant that the miPod provided us to reveal the user key. The miPod is expected to only load a grant that we should be able to decrypt. The user key is issued to the song owner, but the owner can share the song to other users by issuing a grant, which stores a copy of the key (and a signature) encrypted under a different key.
6. XOR the region key and the user key together in order to produce the song key. This is the actual XChaCha20 key the song is encrypted under.
7. We verify the key against its hash stored in the metadata block. This is a final check to ensure that the key was generated properly.

Before playing, the miPod player must precompute a hash of each block in the song. These hashes are the leaves of a Merkle tree of all of the song blocks. The roots of these trees are stored in the metadata blocks of each song.

1. For each block, the miPod player must generate a Merkle proof, proving that the block is committed to in the Merkle root for the block header. In order to produce this proof, the player must have all of the leafs used in generating the root available, so this indirectly enforces that the whole sequence of blocks is available to the player when playing a song.
2. The secure processor is passed the song block and a proof of the song block. It hashes the song block and verifies the proof with it against the root stored in the header. If it fails, it performs a DAB.
3. We then decrypt the song block using the song key that was generated in the load phase. This step is not performed when playing a preview as previews are not encrypted. After decrypting, the samples stored in the song block are sent to the DAC to be rendered into sound.

Pause, Resume, Rewind, Fast-forward

There's special messages that we sent to the secure processor, directing it to update its internal state about which song block is next so it knows how to properly verify the next song block proof. Similarly, when pausing we tell it not to keep spinning and expecting new song blocks until we send the next play message. In order to avoid scrambled output, the DMA buffer gets flushed.

Song Sharing Process

The song sharing process is simple enough that it doesn't need to have a flowchart written out for it. The sharing process is executed on the microblaze and requires that the user already be logged in since it needs decryption keys that can only be derived from the user seed, which is only ever computed on the microblaze.

1. miPod loads and verifies metadata and owner's grant information for song
2. miPod directs secure processor to compute a grant for a user, providing their user information structure
3. Secure processor verifies that the currently logged in user is the real owner of the music file, by checking against some metadata
4. Secure processor performs basic validation on the target user information to verify it is authentic (verifying signature against factory key, etc.)
5. Secure processor computes up to step 5 of the song loading process in order to obtain the user song key, and then re-encrypts this and with the target user's public encryption key, packaging it into a new grant
6. New grant structure is passed back to miPod player
7. miPod appends grant information to the DRM file, so it can be read later by the target user

Troubleshooting

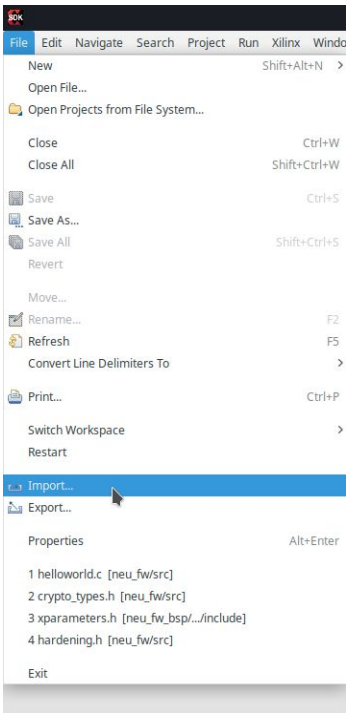
How to build Microblaze and Bitstream independently

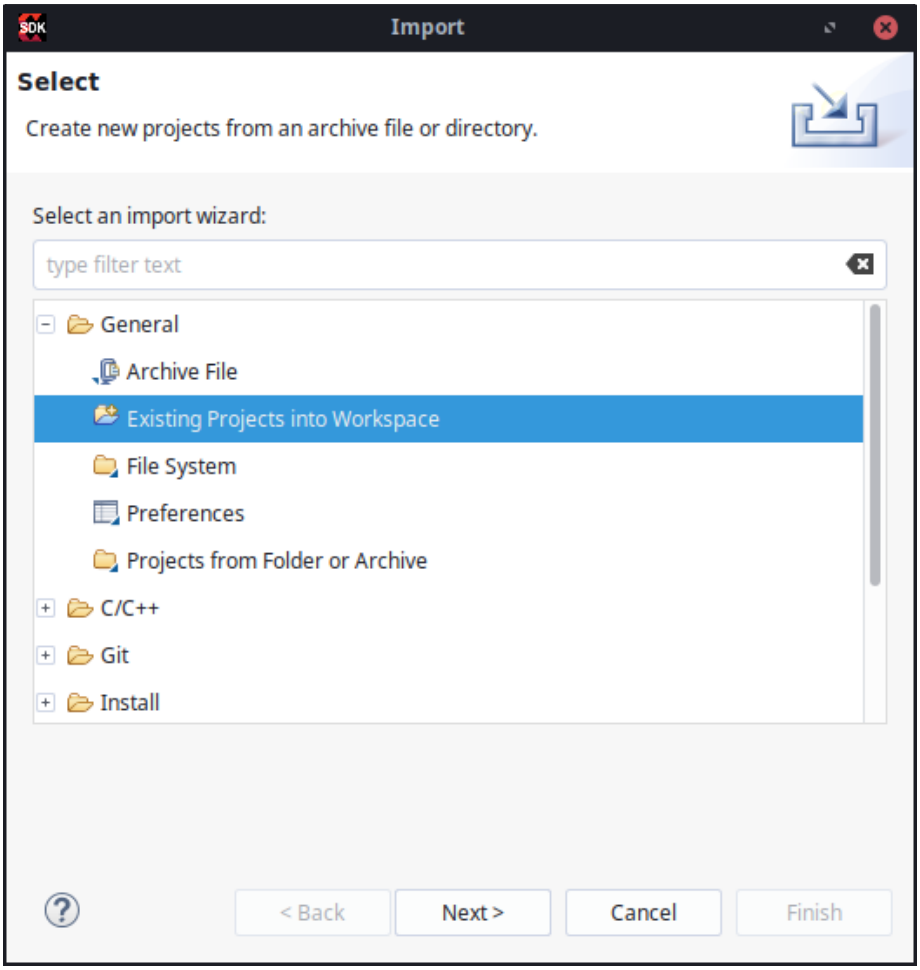
1. *Clone this repository to a known location (<project_directory>, you can use /ectf)*
2. *Open Vivado 2017.4*
3. *In the Tcl Console at the bottom of the screen, run the following commands*

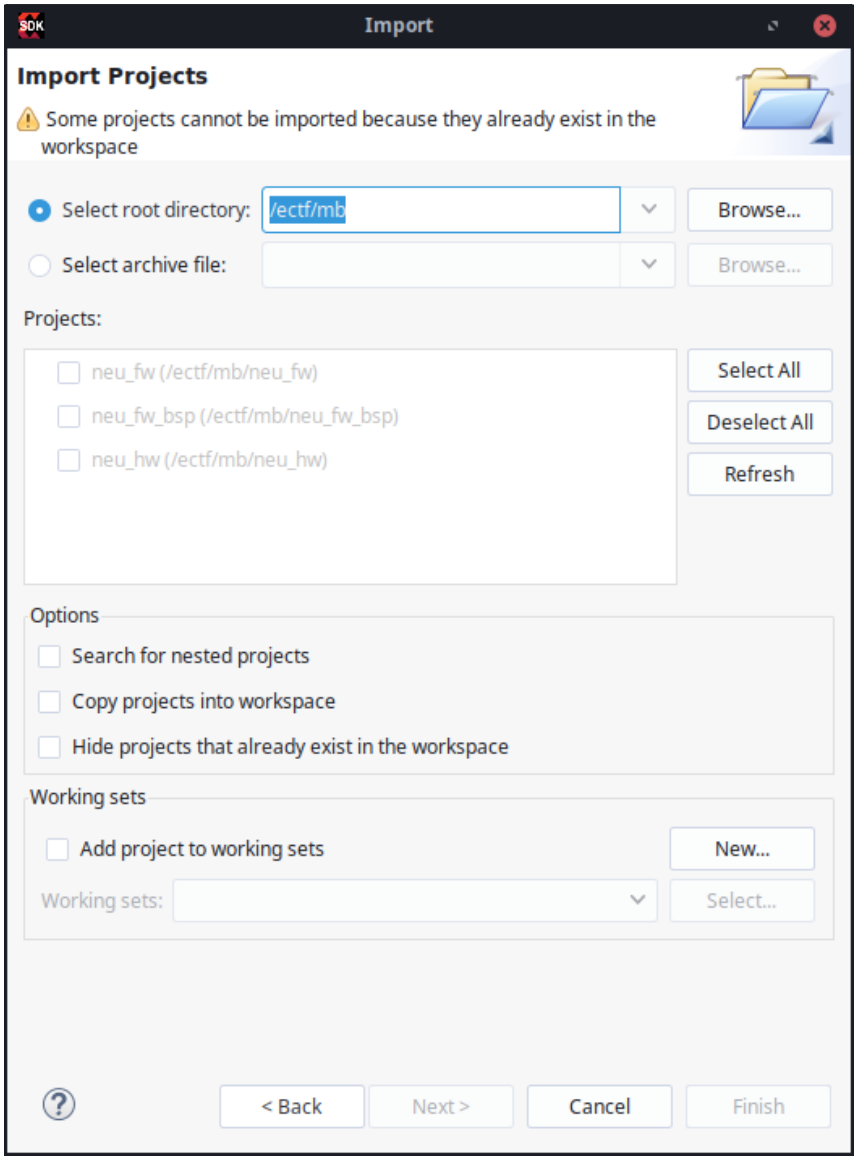
   ```
   cd <project_directory>/pl/proj
   source create_project.tcl
   ```

4. *Click “Generate Bitstream” and select OK on the dialog*
5. *Wait for the project to generate in Vivado.*
6. *Open Xilinx SDK 2017.4*
7. *Import the following projects by pointing the Import Projects wizard to:*

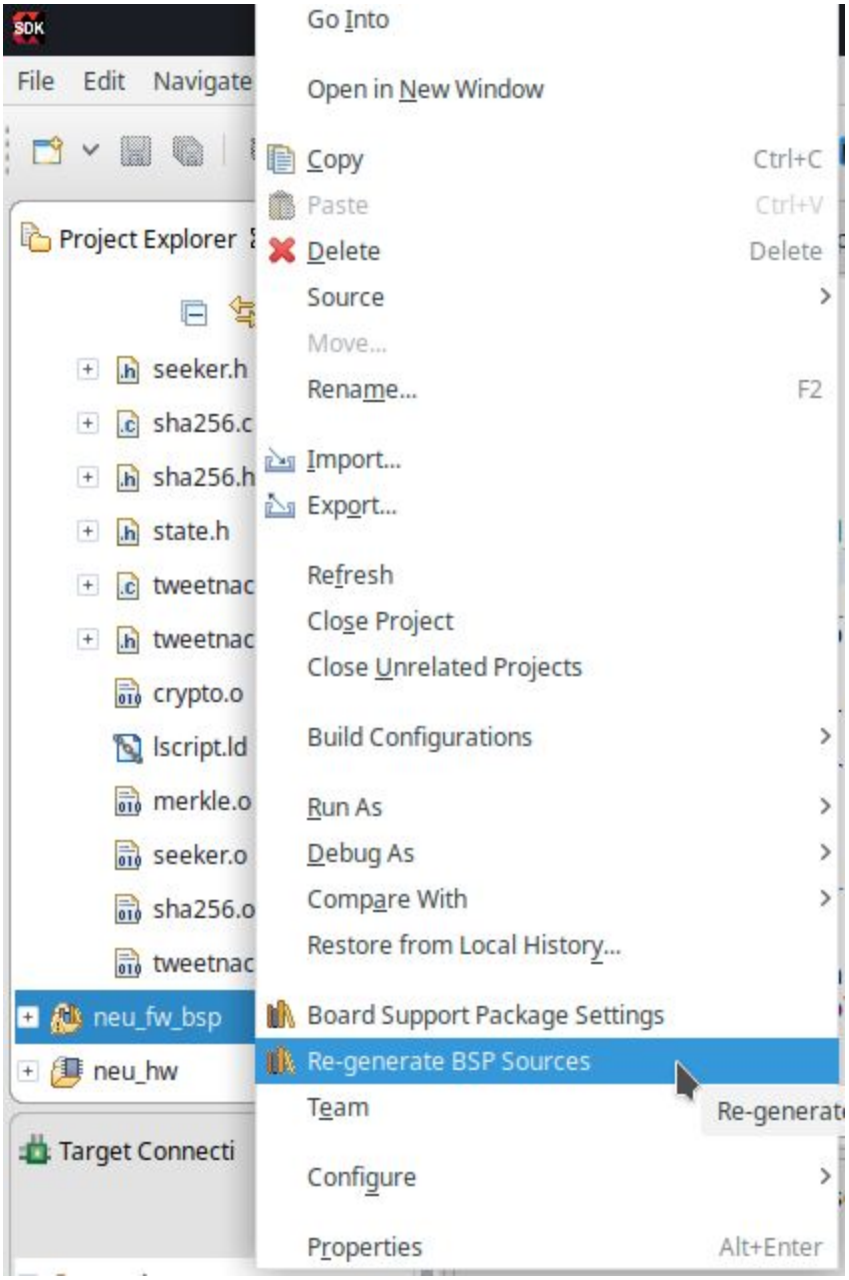
   ```
   <project_directory>/mb/
   ```


File Edit Navigate Search Project Run Xilinx Windo
New Shift+Alt+N
Open File...
Open Projects from File System...
Close Ctrl+W
Close All Shift+Ctrl+W
Save Ctrl+S
Save As...
Save All Shift+Ctrl+S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... Ctrl+P
Switch Workspace
Restart
Import...
Export...
Properties Alt+Enter
1 helloworld.c [neu_fw/src]
2 crypto_types.h [neu_fw/src]
3 xparameters.h [neu_fw_bsp/.../include]
4 hardening.h [neu_fw/src]
Exit


Import
Select
Create new projects from an archive file or directory.
Select an import wizard:
type filter text
General
Archive File
Existing Projects into Workspace
File System
Preferences
Projects from Folder or Archive
C/C++
Git
Install
< Back
Next >
Cancel
Finish



8. *If you get errors in neu_fw, regenerate sources in neu_fw_bsp by right-clicking the neu_fw_bsp project in the project explorer and selecting Re-generate BSP Sources.*


SDK
File
Edit
Navigate
Project Explorer
seeker.h
sha256.c
sha256.h
state.h
tweetnac
tweetnac
crypto.o
lscript.ld
merkle.o
seeker.o
sha256.o
tweetnac
neu_fw_bsp
neu_hw
Target Connecti
Go Into
Open in New Window
Copy Ctrl+C
Paste Ctrl+V
Delete Delete
Source
Move...
Rename... F2
Import...
Export...
Refresh
Close Project
Close Unrelated Projects
Build Configurations
Run As
Debug As
Compare With
Restore from Local History...
Board Support Package Settings
Re-generate BSP Sources
Team
Re-generat
Configure
Properties Alt+Enter

This Page Intentionally Left Blank