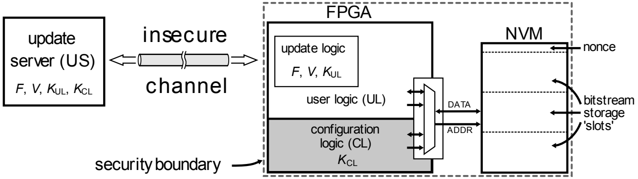
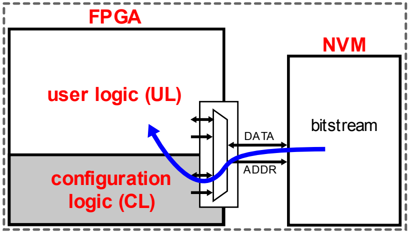
# A protocol for secure remote updates of FPGA configurations

Saar Drimer and Markus G. Kuhn
www.cl.cam.ac.uk/~sd410



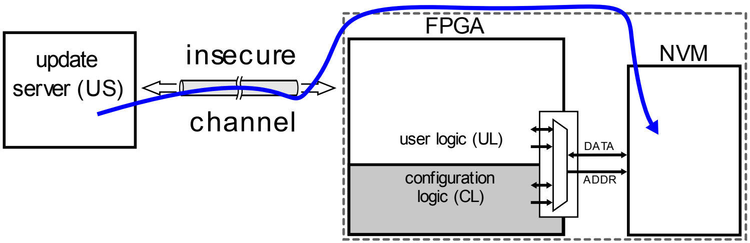
UNIVERSITY OF CAMBRIDGE
Computer Laboratory

## The challenge is to remotely reprogram the non volatile memory (NVM) with a new configuration



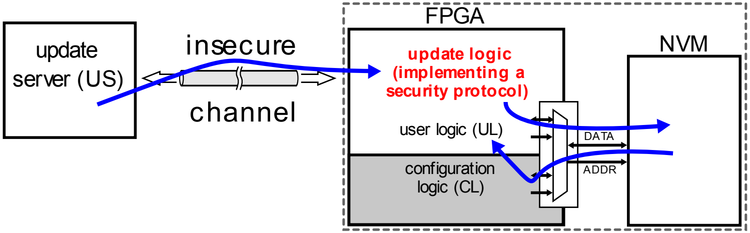
Bitstreams are loaded on every power-up from a local NVM through the configuration logic (CL) in order to program the user logic (UL); we use dual purpose I/Os

## The challenge is to remotely reprogram the non volatile memory (NVM) with a new configuration



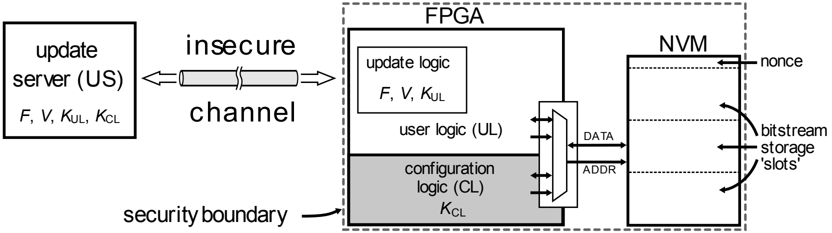
We need to reprogram the NVM through an insecure network with **no additional devices**

## The challenge is to remotely reprogram the non volatile memory (NVM) with a new configuration



The idea is to **enforce a security policy in user logic** so it programs the NVM and protects the system from attackers

## The challenge is to remotely reprogram the non volatile memory (NVM) with a new configuration



The complete system includes a **nonce in the NVM** and bitstream storage slots; a **security boundary** defines protected areas.

## Previous solutions did not consider security, or required extra configuration logic functionality

- Late 1990s Xilinx suggests "Internet Reconfigurable Logic" (security wasn't considered)
- Altera and Xilinx now have functionality to support insecure updates
- I proposed a rough outline of the current paper in late 2007
- Bardignans et al. then proposed a solution that requires changes to configuration logic

**Our solution is flexible because it is implementable in user logic, and can work with today's FPGAs**

Drimer: "**Volatile FPGA design security – a survey**", available on-line
Bardignans et al.: "**Secure FPGA configuration architecture preventing system downgrade**", FPL 2008

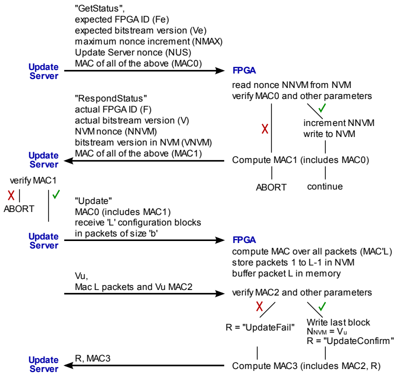
## Fundamental security concepts

- **Encryption**: provides confidentiality to data
- **Message Authentication Code** (MAC): provides identification and integrity
  - Both rely on a shared secret key between the sender and receiver
- **Nonce**: a number that is used **only once** in a MAC so the receiver knows that the message is **fresh**
  - Nonces are non-secret and can be random numbers, time stamps, or monotonically increasing counters

## Our assumptions and requirements

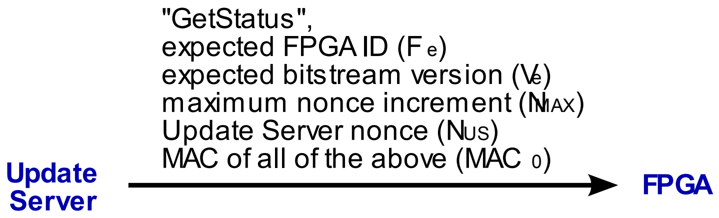
- A unique, non-secret, FPGA identifier; this could be an embedded ID or present in every instance of a bitstream for a particular FPGA
- Key $K_{\mathrm{UL}}$ stored in the bitstream and is unique to each device
- Key $K_{\mathrm{CL}}$ stored in configuration logic (where applicable)
- Designs do not load unless they are complete
- Cryptographic functions that resist cryptanalytic attacks

## An overview of the protocol


"GetStatus",
expected FPGA ID (Fe)
expected bitstream version (Ve)
maximum nonce increment (NMAX)
Update Server nonce (NUS)
MAC of all of the above (MAC0)
Update Server
FPGA
read nonce NNVM from NVM
verify MAC0 and other parameters
✓
increment NNVM
write to NVM
X
"RespondStatus"
actual FPGA ID (F)
actual bitstream version (V)
NVM nonce (NNVM)
bitstream version in NVM (VNVM)
MAC of all of the above (MAC1)
Update Server
Compute MAC1 (includes MAC0)
ABORT
continue
verify MAC1
X
✓
ABORT
"Update"
MAC0 (includes MAC1)
receive 'L' configuration blocks
in packets of size 'b'
Update Server
FPGA
compute MAC over all packets (MAC'L)
store packets 1 to L-1 in NVM
buffer packet L in memory
Vu,
Mac L packets and Vu MAC2
verify MAC2 and other parameters
X
✓
R = "UpdateFail"
Write last block
NNVM = Vu
R = "UpdateConfirm"
Update Server
R, MAC3
Compute MAC3 (includes MAC2, R)

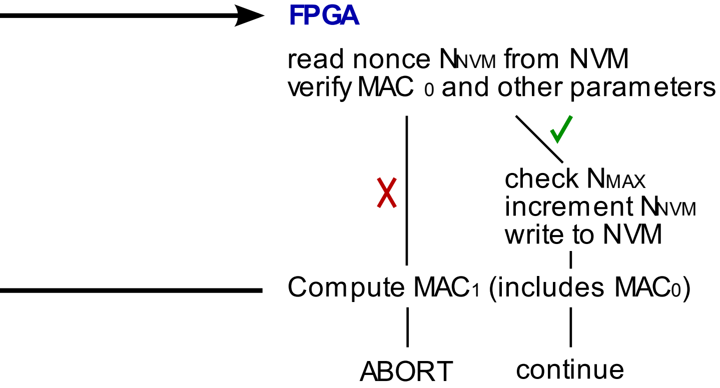
An update server and FPGA execute a security protocol implemented in the FPGA's user logic

## The update server (US) initiates an update


"GetStatus",
expected FPGA ID ($F_e$)
expected bitstream version ($V_e$)
maximum nonce increment ($N_{MAX}$)
Update Server nonce ($N_{US}$)
MAC of all of the above ($MAC_0$)
Update Server
FPGA

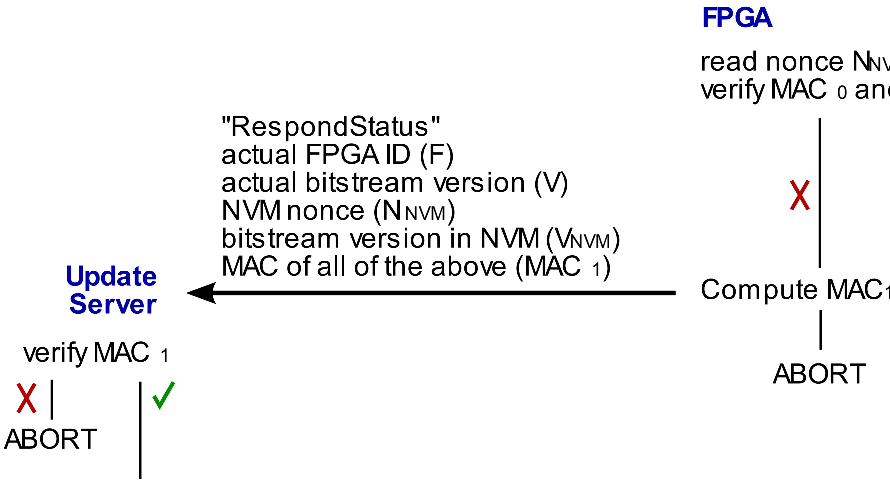
Expected system parameters ($V_e$, $F_e$) prevent an attacker from recording messages and re-sending them to other systems. $N_{\mathrm{MAX}}$ prevents the NVM nonce from being exhausted if the message is send multiple times.

## FPGA verified the MAC


FPGA
read nonce $N_{NVM}$ from NVM
verify $MAC_0$ and other parameters
✓
✗
check $N_{MAX}$
increment $N_{NVM}$
write to NVM
Compute $MAC_1$ (includes $MAC_0$)
ABORT
continue

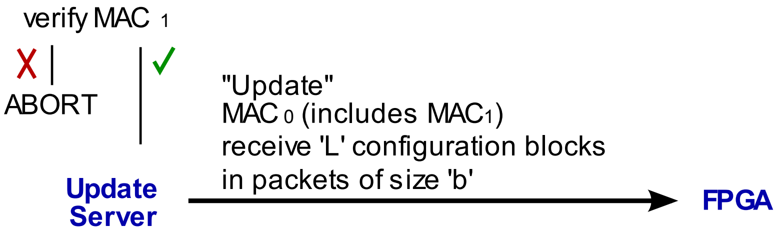
The FPGA reads the nonce from the NVM and uses it to compute the MAC. If the MAC and parameters match, the logic is ready to receive a new bitstream. In either case, a response is sent back to the Update Server.

## The FPGA sends an acknowledgment



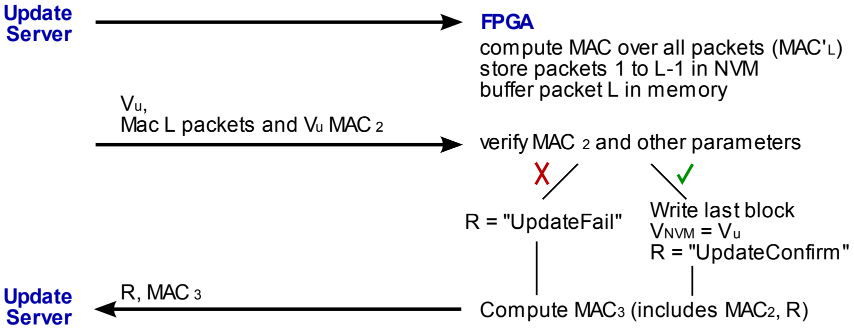
The FPGA computes a MAC with the parameters stored in logic *and* the previous MAC. The Update Server verifies the MAC in order to continue; the exchange up to now also serves as “remote attestation”

## The NVM is updated



'L' bitstream packets of size 'b' are sent; the FPGA MACs and writes them to the NVM as they arrive. *The last packet is stored in a buffer, and not written to the NVM*.

## Finalize update and send acknowledgment to US

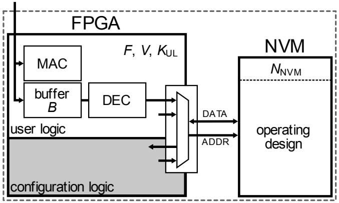

Update Server
FPGA
compute MAC over all packets (MAC'L)
store packets 1 to L-1 in NVM
buffer packet L in memory
Vu,
Mac L packets and Vu MAC 2
verify MAC 2 and other parameters
✗
✓
R = "UpdateFail"
Write last block
VNVM = Vu
R = "UpdateConfirm"
Update Server
R, MAC 3
Compute MAC3 (includes MAC2, R)

After receiving 'L' packets, $MAC_2$ is computed; if correct, the last block is written to NVM. *The size of b should be such that without the last block, the bitstream will not load.*

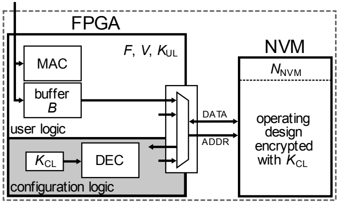
## Application scenarios

Our protocol can operate on existing FPGAs and adapt to their capabilities: without bitstream encryption; with bitstream encryption; or, in the future if bitstream authentication becomes available.

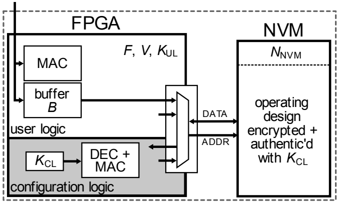
If bitstreams are stored unencrypted, and/or an attacker has physical access to the system, tamper proofing may be needed. This will protect against readout of plaintext bitstreams and system downgrade.



CL: no decryption or authentication



CL: decryption without authentication



CL: decryption and authentication

## To prevent “bricking” we use multiple NVM slots

- A single bitstream “slot” is vulnerable to denial of service attack during an update (after the bitstream is erased from the NVM).
- To prevent this, we propose using multiple slots that alternate between active and temporary slot with every update.
- Most recent FPGAs have the capability of supporting multiple slots in a way that prevents denial of service (e.g., “Fallback MultiBoot”, “SPIm Mode”, “Remote System Upgrade Mode”).

## Thanks!

Many more details in the paper:

`http://www.cl.cam.ac.uk/~sd410/papers/remoteupdates.pdf`

Thanks to XILINX® for funding my research!