A Protocol for Secure Remote Updates of FPGA Configurations

Saar Drimer, Markus G. Kuhn

Computer Laboratory, University of Cambridge,
http://www.cl.cam.ac.uk/users/{sd410,mgk25}

Abstract. We present a security protocol for the remote update of volatile FPGA configurations stored in non-volatile memory. Our approach can be implemented on existing FPGAs, as it sits entirely in user logic. Our protocol provides for remote attestation of the running configuration and the status of the upload process. It authenticates the uploading party both before initiating the upload and before completing it, to both limit a denial-of-service attack and protect the integrity of the bitstream. Encryption protects bitstream confidentiality in transit; we either decrypt it before non-volatile storage, or pass on ciphertext if the configuration logic can decrypt it. We discuss how tamper-proofing the connection between the FPGA and the non-volatile memory, as well as space for multiple bitstreams in the latter, can improve resilience against downgrading and denial-of-service attacks.

1 Introduction

Networked FPGA-based systems gain particular flexibility if remote configuration updates are possible. The question of how to secure such updates against malicious interference may seem easily answered at first glance: many existing cryptographic authentication protocols protect the confidentiality, integrity and authenticity of transactions, such as overwriting a file (e.g., an FPGA bitstream) in a remote computer. They can be applied easily if the FPGA is merely a peripheral device and the remote update of its configuration bitstream can be handled entirely by software running on the system's main processor. Here, however, we consider designs that lack a separate trustworthy main CPU, where the FPGA configuration itself becomes fully responsible for securing its own update.

Two constraints of volatile FPGAs pose particular problems here: they lack memory for storing more than a single configuration at any time, and they retain no state between power-cycles. These, in turn, have two main implications. Firstly, FPGAs have no notion of the "freshness" of received configuration data, so they have no mechanism for rejecting old or revoked configuration content. Secondly, unless a trusted path is established with devices around the FPGA (such as non-volatile memory), they have no temporary memory to store the bitstream while it is being authenticated or decrypted before being loaded into configuration memory cells. In other words, merely initiating a reconfiguration will destroy the current state.

J. Becker et al. (Eds.): ARC 2009, LNCS 5453, pp. 50–61, 2009.

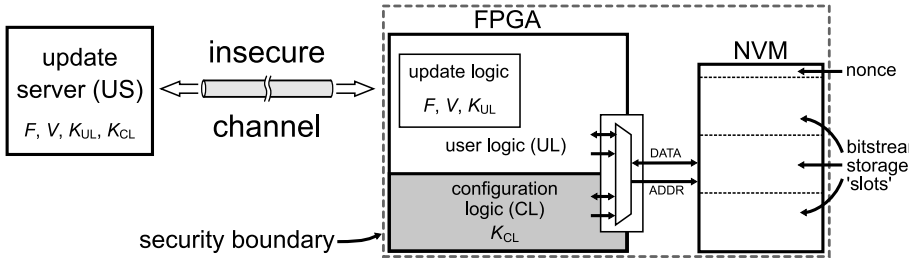

Fig. 1. An update server (US) installs a new bitstream in a system's NVM over an insecure channel by passing it through update logic in the FPGA's user logic (UL). After reset, the hard-wired configuration logic (CL) loads the new bitstream.

The key observation we make is that the FPGA's user logic can be used to perform security operations on the bitstream before it is stored in external non-volatile memory (NVM) and loaded into the FPGA's configuration cells. We then rely on system-level properties, such as tamper proofing and remote attestation, to compensate for the lack of cryptographic capabilities and non-volatile memory in the FPGA's configuration logic. Our solution does not require that FPGA vendors add any hard-wired circuits to their devices' configuration logic, and therefore can be implemented with existing products.

We first list our assumptions (Section 2.1) and then present our secure remote update protocol (Section 2.2), which meets the following goals as far as possible: no additions to the configuration logic; use of the user logic; bitstream confidentiality and authenticity (Section 2.3); prevention of denial-of-service attacks; no reliance on bitstream-encoding obscurity; and, finally, prevention of replay of older, revoked bitstream versions. We then outline a more robust variant of the protocol for systems where the NVM can hold multiple bitstreams (Section 2.4). Finally, we discuss the security properties of our protocol (Section 3) and place it into the context of related work (Section 4).

2 Secure Remote Update Protocol

Our secure update protocol defines an interactive exchange between an *update server* (US), the entity in charge of distributing new bitstreams to FPGA systems in the field, and an *update logic*, the receiving end, implemented in the *user logic* (UL) of each FPGA (Figure 1). Bitstreams are loaded into configuration memory cells by the *configuration logic* (CL), which is hard-wired into the device by the FPGA vendor.

2.1 Assumptions

We require a unique, non-secret, FPGA identifier F, which the authentication process will use to ensure that messages cannot be forwarded to other FPGAs.

If an embedded device ID is available (such as "Device DNA" in some of Xilinx's FPGAs), then that can be used. Otherwise, at the cost of having to change the parameter for every instance of the bitstream intended for a different FPGA, it can also be embedded into the bitstream itself.

We require a message-authentication function $\mathrm{MAC}_{K_{\mathrm{UL}}}(\cdot)$ and a block cipher function $E_{K_{\mathrm{UL}}}(\cdot)$, both of which we assume to resist cryptoanalytic attacks. Even if we use both notationally with the same key, we note that it is prudent practice not to use the same key for different purposes and would in practice use separate derived keys for each function.

The secret key K_{UL} is stored in the bitstream. It should be individual to each device, such that its successful extraction from one device does not help in attacking others.

Device-specific keys can be managed in several ways. For example, K_{UL} can be independently generated and stored in a device database by the update server. Or it could be calculated as $K_{\mathrm{UL}} = E_{K_{\mathrm{M}}}(F)$ from the device identifier F using a master key K_{M} known only to the update server. As a third example, F could contain a public-key certificate that the update server uses to securely exchange the secret key K_{UL} with the update logic.

Where the configuration logic also holds a secret key, K_{CL}, it could be stored in battery-backed volatile or in non-volatile memory, as long as it cannot be discovered through physical attacks or side channel analysis.

We assume that each FPGA of a given model and size loads only bitstreams of fixed length $L \times b$ bits, where b bits is the capacity of the memory block B that the update logic uses to buffer an incoming new bitstream before writing it to NVM. The size b must be large enough to guarantee that the FPGA configuration logic will not load a bitstream from NVM if its last block of b bits is missing. This is normally ensured if any checksum that the FPGA's configuration logic verifies is entirely contained in the last b bits of the loaded bitstream. (In practice, b might also be the minimum amount of data that can be written efficiently to NVM.)

The system needs to be on-line on demand or within a reasonable amount of time, for both update and/or remote attestation. Our protocol handles both malicious and accidental (transmission errors, packet losses, etc.) corruption of data. However, it merely aborts and restarts the entire session if it detects a violation of data integrity, rather than trying to retransmit individual lost data packets. Therefore, for best performance on unreliable channels, it should be run over an error-correcting protocol layer (TCP, HDLC, LAPM, etc.), which does not have to be implemented inside the security boundary.

2.2 The Protocol

Algorithm 1 shows the implementation of the update-logic side of our protocol, which forms a part of the application that resides in the FPGA's user logic. We focus our discussion here on the FPGA side, as this is the half of the protocol that runs on the more constrained device. It supports a number of different policies that an update server might choose to implement.

Algorithm 1 Update-logic state machine

Constants:

K_{UL}	key shared with update server	L	length of bitstream (blocks)
V	version ID of operating bitstream	F	FPGA chip unique ID

Variables:

V_{NVM}	version ID of NVM bitstream	$V_{\mathrm{e}}, F_{\mathrm{e}}$	expected value of V, F
V_{u}	version ID of uploaded bitstream	N_{NVM}	NVM counter value
N_{US}	nonce generated by update server	$N_{\max}$	upper bound for N_{NVM}
B	b-bit buffer for a bitstream block	M, M'	MAC values

1: $V_{\mathrm{NVM}} := V$
2: Receive$(C, V_{\mathrm{e}}, F_{\mathrm{e}}, N_{\max}, N_{\mathrm{US}}, M_0)$
3: **if** $C \neq$ "GetStatus" **then** Send("Abort"); **goto** 2
4: ReadNVM$_{\mathrm{N}}(N_{\mathrm{NVM}})$
5: $S := [M_0 = \mathrm{MAC}_{K_{\mathrm{UL}}}(C, V_{\mathrm{e}}, F_{\mathrm{e}}, N_{\max}, N_{\mathrm{US}})] \wedge (V_{\mathrm{e}} = V) \wedge (F_{\mathrm{e}} = F) \wedge (N_{\mathrm{NVM}} < N_{\max})$
6: **if** S **then**
7: $\quad N_{\mathrm{NVM}} := N_{\mathrm{NVM}} + 1$
8: $\quad$ WriteNVM$_{\mathrm{N}}(N_{\mathrm{NVM}})$
9: **end if**
10: $R :=$ ("RespondStatus", $V, F, N_{\mathrm{NVM}}, V_{\mathrm{NVM}}$)
11: $M_1 := \mathrm{MAC}_{K_{\mathrm{UL}}}(M_0, R)$; Send$(R, M_1)$
12: **if** $\neg S$ **then goto** 2
13: Receive(C, M'_0)
14: **if** $M'_0 \neq \mathrm{MAC}_{K_{\mathrm{UL}}}(M_1, C)$ **then goto** 2
15: **if** $C =$ "Update" **then**
16: $\quad V_{\mathrm{NVM}} := 0$
17: $\quad$ WriteNVM$_{\mathrm{B}[1\ldots L]}(0)$
18: $\quad$ **for** $i := 1$ to L **do**
19: $\quad\quad$ Receive(B_i)
20: $\quad\quad M'_i := \mathrm{MAC}_{K_{\mathrm{UL}}}(M'_{i-1}, B_i)$
21: $\quad\quad$ **if** $i < L$ **then** WriteNVM$_{\mathrm{B}[i]}(B_i)$
22: $\quad$ **end for**
23: $\quad$ Receive(V_{u}, M_2)
24: $\quad$ **if** $M_2 = \mathrm{MAC}_{K_{\mathrm{UL}}}(M'_L, V_{\mathrm{u}})$ **then**
25: $\quad\quad$ WriteNVM$_{\mathrm{B}[L]}(B_L)$
26: $\quad\quad V_{\mathrm{NVM}} := V_{\mathrm{u}}$
27: $\quad\quad R :=$ ("UpdateConfirm")
28: $\quad$ **else**
29: $\quad\quad R :=$ ("UpdateFail")
30: $\quad$ **end if**
31: $\quad M_3 := \mathrm{MAC}_{K_{\mathrm{UL}}}(M_2, R)$; Send$(R, M_3)$
32: **else if** $C =$ "Reset" **then**
33: $\quad M_2 := \mathrm{MAC}_{K_{\mathrm{UL}}}(M'_0,$ "ResetConfirm"); Send("ResetConfirm", M_2)
34: $\quad$ ResetFPGA()
35: **end if**
36: **goto** 2

In addition to the unique FPGA identifier F, the update logic also contains, as compiled-in constants, the version identifier $V \neq 0$ of the application bitstream of which it is a part, and a secret key K_{UL} that is only known to the update server and update logic.

Each protocol session starts with an initial "GetStatus" message from the update server and a "RespondStatus" reply from the update logic in the FPGA. This exchange serves two functions. Firstly, both parties exchange numbers that are only ever used once ("nonces", e.g. counters, timestamps, random numbers). Their reception is cryptographically confirmed by the other party in subsequent messages. Such challenge-response round trips enable each party to verify the freshness of any subsequent data packet received, and thus protect against replay attacks. The nonce N_{US} generated by the update server must be an unpredictable random number that has a negligible chance of ever repeating. This prevents attackers prefetching a matching reply from the FPGA. The nonce N_{NVM} contributed by the update logic is a monotonic counter maintained in NVM (avoiding the difficulties of implementing a reliable and trusted source of randomness or time within the security boundary). To protect this counter against attempts to overflow it, and also to protect against attempts to wear out NVM that only lasts a limited number of write cycles, the update logic will only increment it when authorized to do so by the update server. For this reason, the update server includes in the "GetStatus" message an upper bound $N_{\max}$ (of the same unsigned integer type as N_{NVM}) beyond which the NVM counter must not be incremented in response to this message. The protocol cannot proceed past the "RespondStatus" message unless the NVM counter is incremented.

The second purpose of the initial exchange is to ensure that both parties agree on values of F and V. The update server sends its expected values V_{e} and F_{e}, and the update logic will not proceed beyond the "RespondStatus" message unless these values match its own V and F. This ensures that an attacker cannot reuse any "GetStatus" message intended for one particular FPGA chip F and installed bitstream version V on any other such combination. The update server might know V and F already in advance from a database that holds the state of all fielded systems. If this information is not available, the update server can gain it in a prior "GetStatus"/"RespondStatus" exchange, because both values are reported and authenticated in the "RespondStatus" reply.

All protocol messages are authenticated using a message-authentication code (MAC) computed with the shared secret key K_{UL}. This is done in order to ensure that an attacker cannot generate any message that has not been issued by the update server or update logic. In addition, with the exception of the initial "GetStatus" message, the calculation of the MAC for each message in a protocol session incorporates not only all data bits of the message, but also the MAC of the previously received message. This way, the MAC ensures at any step of the protocol that both parties agree not only on the content of the current message, but also on the content of the entire protocol session so far. This mechanism makes it unnecessary to repeat in messages any data (nonces, version identifiers, etc.) that has been transmitted before, because their values are implicitly carried

forward in each message by the MAC chain. In the presentation of Algorithm 1, M, M' and B are registers, not arrays, and their indices merely indicate different values that they store during the execution of one protocol session.

Note that any "GetStatus" request results in a "RespondStatus" reply, even without a correct MAC. This is to allow update servers to query F even before knowing which K_{UL} to apply. However, an incorrect MAC in a "GetStatus" will prevent the session from proceeding beyond the "RespondStatus" reply. This means that anyone can easily query the bitstream version identifier V from the device. If this is of concern because, for example, it might allow an attacker to quickly scan for vulnerable old versions in the field, then the value V used in the protocol can be an individually encrypted version of the actual version number $\hat{V}$, as in $V = E_{K_{\mathrm{UL}}}(\hat{V})$. Whether this option is chosen or not does not affect the update-logic implementation of the protocol, which treats V just as an opaque identifier bitstring.

The protocol can only proceed beyond the "RespondStatus" message ($S =$ true) if the update-logic nonce has been incremented and both sides agree on which FPGA and bitstream version is being updated. If the update server decides to proceed, it will continue the session with a command that instructs the update logic either to begin programming a new bitstream into NVM ("Update"), or to reset the FPGA and reload the bitstream from NVM ("Reset"). The MAC M'_0 that comes with this command will depend on the MAC M_1 of the preceding "RespondStatus" message, which in turn depends on the freshly incremented user-logic nonce N_{NVM}, as well as V and F. Therefore, the verification of M'_0 ensures the user-logic of both the authenticity and the freshness of this command, and the same applies to all MAC verifications in the rest of the session.

The successful authentication of the "Update" command leads to erasing the entire bitstream from the NVM. From this point on, until all blocks B_1 to B_L have been written, the NVM will contain an invalid bitstream. Therefore, there is no benefit in authenticating each received bitstream data block B_i individually before writing it into NVM. Instead, we postpone the writing of the last bitstream block B_L into NVM until the message authentication code M'_L that covers the entire bitstream has been verified. This step is finally confirmed by the update logic in an "UpdateConfirm" message.

Since a system that can store only a single bitstream in its NVM must not be reset before all blocks of the bitstream have been uploaded, our protocol also provides an authenticated status indicator V_{NVM} intended to help recover from protocol sessions that were aborted due to loss or corruption of messages. After a successful reset, the update logic sets $V_{\mathrm{NVM}} := V$ to indicate the version identifier of the bitstream stored in NVM. Before the process of erasing and rewriting the NVM begins, it sets V_{NVM} to the reserved value 0, to indicate the absence of a valid bitstream in NVM. After the last block B_L was written, the update logic receives from the update server the version identifier V_{u} of the bitstream that has just been uploaded into NVM, and sets register V_{NVM} accordingly. Should the update session be aborted in any way, then the update server can always initiate a new session with a new "GetStatus"/"RespondStatus" exchange, where it will

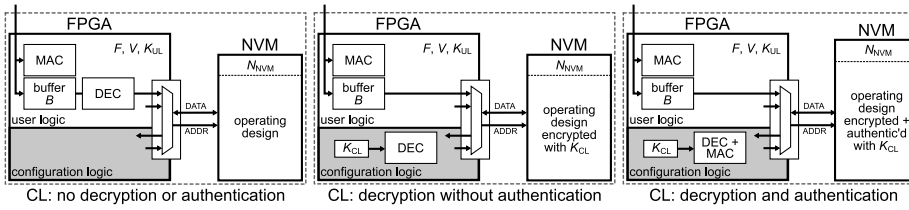

Fig. 2. Scenarios for different configuration logic capabilities.

learn from the V_{NVM} value in the "RespondStatus" message the current status of the NVM, that is, whether the old bitstream is still intact, the new bitstream has been uploaded completely, or it contains no valid bitstream. It can then decide whether to restart the upload or perform a reset.

If no messages were lost, the update server will automatically receive an authenticated confirmation. "UpdateConfirm" indicates that the "Update" command has been processed, and its MAC M_3 confirms each received data byte, as well as the old and new version identifiers, FPGA ID, nonces, and any other data exchanged during the session. The "ResetConfirm" command can only confirm that the reset of the FPGA is about to take place; any attestation of the successful completion of the reset must be left to the new bitstream.

The initial "GetStatus"/"RespondStatus" exchange can, besides for initiating a new session or restarting after an aborted one, also be used for remote attestation of a system. For this purpose, "GetStatus" is simply sent with $N_{\max} = 0$ and the values of V_{e} and F_{e} do not matter. This will not affect the NVM counter, but results in authenticated fresh values of V, F, and V_{NVM} in "RespondStatus".

2.3 Bitstream Encryption and Authentication

Some FPGAs can decrypt bitstreams in their configuration logic, using embedded (or battery-backed) keys, while others lack this capability. Parelkar and Gaj [1] and Drimer [2] have also proposed adding bitstream authentication. Algorithm 1 can be used with FPGAs that support any combination of these functions (three are shown in Figure 2), provided that the user logic compensates for those that are missing.

For confidentiality, bitstreams should always be transmitted encrypted between the update server and update logic. Where the configuration logic is able to decrypt a bitstream while loading it from NVM, the update server can encrypt the bitstream under a secret key K_{CL} shared with the configuration logic, leaving the update logic and NVM merely handling ciphertext. If the configuration logic cannot decrypt, the update server has to encrypt the bitstream under a key derived from K_{UL} and the user logic has to decrypt each block B_i before writing it to NVM (using some standard file encryption method, such as cipher-block chaining). If the configuration logic also performs authentication, the requirements above do not change; the authentication in the update logic is still necessary to prevent denial-of-service attacks that attempt unauthorized

overwriting of NVM content. Again, the last buffered block B_L must contain the MAC of the bitstream that the configuration logic will verify, such that the bitstream will not load without the successful verification of M_2.

2.4 Multiple NVM Slots

NVM that provides only a single memory location ("slot") for storing a bitstream can seriously limit the reliability of the system. There will be no valid bitstream stored in the NVM from when the update logic starts erasing the NVM until it has completed writing the new bitstream. A malicious or accidental interruption, such as a power failure or a long delay in the transmission of the remaining bitstream, can leave the system in an unrecoverable state. Such single-slot systems are, therefore, only advisable where loss of communications or power is unlikely, such as with local updates with a secure and reliable power source.

Otherwise, the NVM should provide at least two slots, such that it can hold both the old and the new bitstream simultaneously. The FPGA's configuration logic will then have to scan through these NVM slots until it encounters a valid bitstream. It will start loading a bitstream from the first slot. If the bitstream is invalid (i.e., has an incorrect checksum or MAC), it will load another bitstream from the next slot, and so on, until all slots have been tried or a valid one has been found.

The additional slot is then used during the update as a temporary store, in order to preserve the currently operating design in case of an update failure. The role of the two slots – upload area and operating bitstream store – can alternate between updates, depending on how multiple slots are supported by the configuration logic. The update process may be modified as follows.

At manufacturing, slot 1 is loaded with an initial design whose update logic can only write new bitstreams into the address space of slot 2. Before the $V_{\mathrm{NVM}} := V_{\mathrm{u}}$ changeover is made (line 26), the update logic erases slot 1, which then allows the configuration logic to find the new bitstream in slot 2 at the next power-up. The new bitstream, now in slot 2, will write its next bitstream into slot 1, and erases slot 2 when that update is complete, and so on. If an update is aborted by reset, one slot may remain with a partially uploaded bitstream. If this is slot 1, the configuration logic will fail to find a correct checksum there and move on to load the old bitstream from slot 2, from where the update process can then be restarted. If the partial content is in slot 2, then the configuration logic will never get there because the bitstream in slot 1 will be loaded first.

If the configuration logic tells the user logic which slot it came from (through status registers), then the user logic can simply pick the respective other slot and there is no need to compile one bitstream for each slot. If not, then the update server must ensure, through remote attestation, that each newly updated bitstream is compiled to write the next bitstream to the respective other slot. This might be aided by encoding in V which slot a bitstream was intended for. A third slot might be provided with a fallback bitstream that is never overwritten, and only loaded if something goes wrong during the NVM write process

without the update logic noticing (e.g. data corruption between the FPGA and the NVM). This could leave both slot 1 and 2 without a valid bitstream and cause the fallback bitstream to be loaded from slot 3. Alternatively, the FPGA may always load, as a first configuration, a bootloader bitstream that controls from which slot the next bitstream is loaded from.

Recent FPGAs, such as Virtex-5 [3, UG191, Chap. 8, "Fallback MultiBoot"] or LatticeECP2/M [4, TN1148, "SPIm Mode"], implement a multi-slot scan in the configuration logic. Some, including Stratix IV [5, SIV51010-2.0, "Remote System Upgrade Mode"] or Virtex-5 [3, UG191, Chap. 8, "IPROG Reconfiguration"], provide a "reload bitstream from address X" command register, so that a user-designed bootloader bitstream can implement a multi-slot scan and checksum verification (but such a bootloader itself cannot necessarily be updated remotely). Alternatively, external configuration logic could be added to do the same.

3 Analysis

Algorithm 1 alone does not prevent the configuration logic from loading old bitstreams from NVM. In order to maintain our security objective of preventing attackers from operating older, outdated FPGA configurations, we also need to rely on either tamper proofing or binding the provision of online services to remote attestation. With Algorithm 1, if attackers can either feed the FPGA with N_{NVM} values of previously recorded sessions, or have access to its configuration port, they can replay older bitstreams. Therefore, these interfaces must be protected: the FPGA, NVM and the interface between them (configuration and read/write) must be contained within a tamper-proof enclosure. Manufacturing effective tamper-proof enclosures can be challenging, although there are now a number of strong off-the-shelf solutions available, such as tamper-sensing membranes that trigger zeroization of keys stored in battery-backed SRAM when penetrated. But protection against highly capable attackers is not necessary for all applications. Sometimes, it may suffice to deter attackers by using ball-grid array packages and routing security-critical signals entirely in internal printed circuit board layers without accessible vias. Some manufacturers may not be concerned if a few of their systems are maliciously downgraded with great effort in a laboratory environment, as long as the financial damage of the attack remains limited and it is impractical to scale it up by creating a commercial low-cost kit that allows everyone to repeat the attack with ease. For example, in consumer electronics, an attractive attack cannot require risky precision drilling into a printed circuit board or desoldering a fine-pitch ball-grid array. New stacked-die products, where the NVM is attached to the top of the FPGA die inside the same package (such as the Xilinx Spartan-3AN family) also make tamper proofing easier.

In some applications (e.g., online entertainment set-top boxes), the device's only use is to provide a service by interacting with a central server. Here, the provision of the service can be made conditional to a periodic successful remote attestation of the currently operating bitstream, in order to render the device

inoperable unless it has an up-to-date bitstream loaded in the NVM. The remote attestation facility can provide the service provider authenticity assurances even where no tamper proofing exists, though the system must be on-line at short intervals.

If the bitstream is kept in the NVM encrypted under an FPGA-specific key (K_{CL}), then neither bitstream reverse engineering nor cloning will be possible, even if the tamper proofing of the NVM link fails. We already assume that such ciphertext can be observed in transit between the update server and update logic. If the plaintext bitstream is kept in NVM, because the configuration logic lacks decryption capability, we rely on NVM tamper-resistance to protect against cloning and the risk of bitstream reverse engineering. Projects for the latter, such as "ULogic" by Note and Rannaud [6], illustrate the risk of merely relying on the obscurity of the bitstream's syntax for security.

3.1 Parameter Sizes

As the NVM counter is well protected against overflow attacks by the $N_{\max}$ parameter (controlled by the update server), a size of 32 bits appears more than sufficient for most applications. Since an attacker can keep asking for a response for any value of N_{US} during remote attestation, N_{US} should be large enough to make the creation of a dictionary of responses that can be replayed impractical. For instance, using a uniformly distributed 64-bit word for N_{US} will ensure that an attacker who performs 10^3 queries per second will fill less than 10^{-7} of the dictionary within a decade. MAC values (M, M') of 64-bit length provide an equally generous safety margin to brute-force upload attempts.

4 Related Work

The Xilinx "Internet Reconfigurable Logic" initiative from the late 1990s discussed how remote updates can be performed, though the program was short lived [3, App. Note 412]. Remote reconfiguration using embedded processors has also been proposed [3, App. Note 441]. A patent by Trimberger and Conn [7] describes a remote update through a modem, an FPGA controller (in addition to the main FPGA) and "shadow PROMs" for recovery from failed writes. Altera describes how field updates can be performed for Stratix and Cyclone devices using a soft processor in the user logic and a hard logic interface using a non-volatile memory device, with the ability to revert to a "factory bitstream" stored in the NVM [5, User Guides SII52008, SIII51012, SIV51010, CIII51012]. However, the security aspects of remote update are not considered in any of the above.

Castillo et al. [8] propose a solution based on an OpenRISC1200 processor implemented in the user logic, together with an RSA engine for remote configuration on every start-up. However, the cryptographic key on which its security depends is obfuscated in a non-encrypted bitstream stored in the local NVM. Fong et al. [9] propose a security controller based on the Blowfish block cipher for encryption and CRC for data correctness. Attached to a "Media Interface", the FPGA is able to receive updates that are programmed into the configuration logic through the internal configuration port. The authors point out the

vulnerabilities of their scheme: key obfuscation within the bootstrap bitstream, but more significantly, lack of data authentication with freshness, opening the system to replay attacks. Both the above schemes require an on-line connection at start-up to receive the operating design, while ours performs a secure remote update once, stores the bitstream locally, and loads it into the FPGA at start-up without on-line connectivity.

Replay attacks despite bitstream encryption and authentication were described by Drimer [10, p. 21], who suggested adding a non-volatile counter as nonce for bitstream authentication, or remote attestation in user logic as countermeasures. Motivated by this, Badrignans et al. [11] proposed additions to the hard-coded configuration logic in order to prevent replay of old bitstreams. They use a nonce in the authentication process, in addition to a mechanism for alerting the developer of its failure. Our solution of using user-logic resources instead of adding hard-wired configuration logic functionality is more flexible: our update logic can also update itself in the field. More importantly, our approach can be used with existing FPGAs, although it can equally benefit from additional security features in future ones. We also discuss denial-of-service attacks and failed updates, how the system can recover, and specify the detailed behaviour of our update logic, ready for implementation.

5 Conclusions

We have proposed a secure remote update protocol that maintains the confidentiality, integrity and freshness of bitstreams to the extent possible. In contrast to other proposals, our solution requires no additions to the configuration logic of existing FPGAs and uses the user logic for most security functions. The required cryptographic primitives consume some user-logic resources, but they can be reused by the application. Even local attackers can be deterred from restoring old and outdated bitstreams, which the update server may want to suppress (e.g., due to known vulnerabilities), by tamper proofing the NVM connection.

The update logic proposed here can be implemented either in software on a soft processor, or as a logic circuit. The design and presentation of our protocol was influenced by our experience with an ongoing logic-circuit implementation on a Virtex-5 evaluation board, using the publicly available AES design by Drimer et al. [12].

Possible extensions for the protocol presented here include role- and identity-based access control (beyond the current single role of "update server"), as well as receiving on-line partial configuration content at start-up, to be programmed into memory cells using an internal configuration port.

Acknowledgments

Saar Drimer's research is funded by Xilinx, Inc. We thank Steven J. Murdoch, Sergei Skorobogatov and the anonymous reviewers for valuable comments and suggestions.

References

1. Parelkar, M.M., Gaj, K.: Implementation of EAX mode of operation for FPGA bitstream encryption and authentication. In: Field Programmable Technology. pp. 335–336 (December 2005)
2. Drimer, S.: Authentication of FPGA bitstreams: why and how. In: Diniz, P.C., Marques, E., Bertels, K., Fernandes, M.M., Cardoso, J.M.P. (eds.) ARC 2007. LNCS, vol. 4419, pp. 73–84. Springer, Heidelberg (2007)
3. Xilinx Inc.: `http://www.xilinx.com`.
4. Lattice Semiconductor Corp.: `http://www.latticesemi.com`.
5. Altera Corp.: `http://www.altera.com`.
6. Note, J.B., Rannaud, É.: From the bitstream to the netlist. In: ACM/SIGDA Symposium on Field Programmable Gate Arrays, pp. 264–264. ACM, New York (2008)
7. Trimberger, S.M., Conn, R.O.: Remote field upgrading of programmable logic device configuration data via adapter connected to target memory socket. United States Patent 7,269,724. (September 2007)
8. Castillo, J., Huerta, P., Martínez, J.I.: Secure IP downloading for SRAM FPGAs. Microprocessors and Microsystems 31(2), pp. 77–86. (2007)
9. Fong, R.J., Harper, S.J., Athanas, P.M.: A versatile framework for FPGA field updates: an application of partial self-reconfiguration. In: IEEE International Workshop on Rapid Systems Prototyping, pp. 117–123 (2003)
10. Drimer, S.: Volatile FPGA design security – a survey (v0.96) (April 2008), `http://www.cl.cam.ac.uk/~sd410/papers/fpga_security.pdf`
11. Badrignans, B., Elbaz, R., Torres, L.: Secure FPGA configuration architecture preventing system downgrade. In: Field Programmable Logic, pp. 317–322 (September 2008)
12. Drimer, S., Güneysu, T., Paar, C.: DSPs, BRAMs and a pinch of logic: new recipes for AES on FPGAs. In: IEEE Symposium on Field-Programmable Custom Computing Machines. IEEE, Los Alamitos (2008)