Protecting multiple cores in a single FPGA design

Saar Drimer[1], Tim Güneysu[2], Markus G. Kuhn[1], Christof Paar[2]

[1]Computer Laboratory, University of Cambridge, UK

http://www.cl.cam.ac.uk/users/{sd410,mgk25}

[2]Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany

{gueneysu,cpaar}@crypto.rub.de

DRAFT

Abstract

As field programmable gate arrays become larger and more complex, system developers look to purchase off-the-shelf cores from specialized core vendors to save on development costs. The FPGA "cores industry" is rapidly growing but is hampered by the inability of cores owners to enforce licensing, which leads to a less than ideal blanket licensing arrangements which are based on personal trust and paper agreements. This works well for large reputable companies that can front the money in advance and have a lot to lose from breach of contract, though less well for new companies, hobbyists and academics. In this paper we describe a new cryptographic method that allows system developers to integrate externally-developed cores into their own designs while still letting cores vendors enforce licensing terms. Our scheme only requires modest additions to the FPGA's configuration logic and software tools while only using established primitives. By partitioning the design ahead of time and using currently available design methods, the system developer allocated portions of the FPGA off-the-shelf cores. Bitstream portions corresponding to these partitions are encrypted by the cores vendors, assembled by the system developer into a single bitstream and are decrypted in the FPGA using symmetric and asymmetric cryptography methods.

1 Introduction

Field programmable gate arrays (FPGA) are generic semiconductor devices made up of interconnected functional blocks that can be programmed, and reprogrammed, to perform user-described logic functions. Since the early 2000s, FPGA vendors have gradually absorbed into the FPGAs functions that previously required peripheral devices. 2008's FPGAs have embedded processors, giga-bit serial transceivers, clock managers, analog-to-digital converters, digital signal processing blocks, Ethernet controllers, megabytes of memory, and other functional blocks beyond the arrays of basic logic elements they started out with in the mid 1980s. We are also seeing a process of industry-specific sub-families

within a single family of devices that cater to embedded, DSP, military and automotive applications; this means that the distribution of the various embedded blocks is different across family members.

Design protection schemes suggested up to now dealt with the protection of a complete bitstream, not bitstreams consisting of several cores from several independent parties. In this contribution, we extend the design protection scheme for FPGAs presented by Güneysu et al. [13] to provide core vendors with means to determine exactly how many copies of their design were made, even if their cores are part of a larger designs. Through a relatively simple cryptographic exchange, we are able to provide core owners to control the number of copies made of their core without requiring changes to the FPGA use model of secure software components operating on the system developer's servers. We do not require a cryptographic components for key establishment in the static configuration logic, but instead use the user logic for a setup process based on public key cryptography.

This contribution is organized as follows: we start by introducing the main issues of cores protection for FPGA devices, define the principals involved, and discuss the basic security primitives we use. Next, after reviewing prior work and existing solutions for FPGA core protection, we present our scheme for single and many-core designs in Sections 3 and 4. In Section 5 we discuss the necessary requirements and assumptions and evaluate our scheme in Section 6. Finally, we present implementation details of a first proof-of-concept solution which shows the feasibility of our proposal even with today's small FPGAs.

1.1 The cores distribution problem

Cores are ready-made functional blocks that allow system developers to save design cost and time by purchasing them from "external" designers and integrate them into their own design. A single core can also occupy the entire FPGA to create a "virtual application specific standard products" (vASSP) [16]. Core designers sell their cores in the form of hardware description language (HDL) or complied netlists, while FPGA vendors provide some cores for free (they profit from FPGA sales, not cores) as do open-source core designers. There exists commercial or free cores for nearly every popular digital function.

In a constraint-free world, an ideal distribution model will allow us to distribute cores from multiple vendors to a single system developer such that he can evaluate, simulate, and integrate them into his own design while 1) the cores' confidentiality and authenticity are assured, and 2) the core vendor is able to enforce a limit on the number of instances made of any core, and make every instance of it restricted to operate only on a specific set of devices. We must first reason, however, why this model is better than what is available today. The growth in both capacity and application space of FPGAs has started a now flourishing industry for the sale of ready-made designs that are sold to system developers to be integrated into their own FPGA designs. Performance optimized (in both size and throughput) FPGA design now requires specialized skill, and in many cases it makes economic sense to purchase certain functions rather than designing them from scratch in-house.

The "design-reuse" industry has dealt with cores distribution by mostly relying on social means such as "trusted partners" and reputation, which works well for established corporations but much less so for unrecognized start-ups in, for example, Asia. An industry-

wide panel discussion in early 2007 about encrypted software flow for secure ASIC cores distribution (with FPGA considerations added on as an attachment) provides some insight into how large corporations view the issue [25, 26]. The representatives from these large corporations agreed that the trust-based system is working well for *them*, and a better solution is desirable for the long-run, but is not necessarily urgent. This conclusion is quite understandable for these companies, who are large enough to still design and fabricate ASICs, though the situation for FPGAs is quite different.

Many system developers opt to use FPGAs precisely because they cannot afford to produce an ASIC or have the money to pay up-front for a core before their products are successful. Their trustworthiness is determined by the cores vendors based on their own risk perception after assessing the locale and people behind the company. One FPGA cores vendor we spoke to confirmed that this is indeed the case, and that a "small number" of initial request are rejected on the basis of trust. After the initial development of a core, roughly half of the effort is invested in pre-sale support with a potential customer. This includes discussing the needs; educating the customer about the core's function; preparing a simulation model and an evaluation core for integration with the developer's design; and, supporting the simulation and integration process. The effort invested up to this point by the vendor may not yield any returns if the customer decides not to purchase the core after all. The risk is not only losing the time spent, but also that the evaluation core supplied be used without compensation. When there is a purchase, the customer pays a blanket license for the core and legal documents are signed for its protection. What the customers get is an unrestricted HDL of the core, which can be fully integrated into their own designs. The rest of the effort is put in post-sale support: full customization of core, last minute changes and requirements, and so on.

Most of what the customer pays for, then, is customization and support, rather than for the initial development costs.

1.2 Principals

In this section we will introduce principals as well as notation used throughout the paper. "A principal is an entity that participates in a security system. This entity can be a subject, a person, a role, or a piece of equipment, such as a PC, smartcard, or card-reader terminal" [3, p.9]. FPGAs, FPGA vendors, designers, programming cables *etc.* are principals interacting in a system whose security we are concerned with. Introduced below are the principals that partake in the design and distribution of FPGA products, along with their security requirements.

FPGA vendors (FV) introduces a new FPGA family roughly every 12 to 18, each costing many millions of dollars do design, fabricate, and test for production. The amount of transistors that can fit on a die is limited, so vendors only introduce embedded functions that are needed by the majority of their customers or several large ones, or are in-line with a long-term marketing strategy. Customers must be willing to pay for all the functional blocks on the die (by accepting the price of the FPGA) even if those end up being unused in the design. This is important to remember when we consider the addition of security features to FPGAs, as they must be financially profitable to the FPGA vendors.

Security-wise, FPGA vendors have two dominant concerns. Firstly, they are interested in

protecting their own proprietary designs and technology from being reverse engineered, copied, exposed, or modified. Secondly, they want to provide their customers means to protect their own designs throughout the design flow and in the field. They also have the incentive to provide means to enable secure distribution of cores since that translates into increased FPGA sales, by making FPGA designs more widely accessible.

System developers (SD). incorporate FPGAs into their system, and can be divided into two groups based on their security needs and views. *Cost-conscious* are the designers of commercial products who have the goal of meeting the product's specifications at the lowest cost while maintaining reliability. Often, there is a trade-off between system performance and cost, with a general tendency by engineers/management to resist additional components, design delays, and decrease in reliability that translates into higher maintenance and support costs. Therefore, a design protection scheme must make economic sense, and may only need to be effective for the lifetime of the product, from months to few years. *Security-conscious* designers and security-industry system developers are concerned with protecting designs, methods of operation and communications for a substantial length of time — from years to decades — while cost considerations may be secondary if those imply compromise of security.

In this paper we concentrate on the cost-conscious developers that is comfortable buying and using off-the-shelf components and together with the cores vendors is looking for a easy-to-use and logistically cheap protection scheme.

We have already introduced the **cores vendors (CV)**, and finally, there are **trusted parties (TP).** In order to reach the security goals of a protocol, a principal that is trusted by all other principals is often required for storing, processing and transferring data and keys. It is easy to add a trusted parties to a protocol though they are best avoided since implementing them in reality is difficult. The centralized nature of a trusted party makes it vulnerable to denial of service attacks, and a lucrative target for attackers. In addition, it may not even be possible to find one principal that is mutually trusted by all others. More practical issues such as location, trusted personnel, physical security and so on are also problematic.

1.3 Security

We define the following set of principals,

$$Z = \{\mathrm{FV}, \mathrm{CV}, \mathrm{SD}\},$$

in addition to FPGA itself, which is an active participant in the protocol. A symmetric key defined by principal $z \in Z$ is denoted as K_z, and an asymmetric public and private pair of keys are noted as PK_z and SK_z, respectively. Then index i will denote multiple principals of the same kind. Now we briefly discuss the cryptographic primitives we will use to describe ours and others' protocols.

Cryptographic hash function. Cryptographic hash functions are one-way functions which take arbitrary length string as input and map it onto a fixed length bit string. A cryptographic hash of a message can be considered its static-length signature or unique fingerprint. This is because cryptographic hash functions provide strong collision resistance and preimage intractability, which means that it is nearly impossible to find two

different inputs that produce the same hash and find the input that yielded a given hash. Most security applications employ cryptographic hash functions as utility functions for data compression and aggregation to reduce the computational complexity of subsequent operations like the generation of digital signatures.

Symmetric cryptography: For design protection and configuration confidentiality, some FPGAs already implement symmetric encryption based on well-known block ciphers like AES and 3DES. Such cryptographic functionality can be used for more than just confidentiality: in the CMAC construction [10], a computational process mostly identical to that of CBC encryption with a block cipher is used to generate a one-block message authentication code (MAC). Thus, to keep the footprint of the cryptographic component small, an implementation of a single block cipher can be used both for decryption and for MAC verification, using a CBC scheme in both cases. Using separate block cipher keys, a combination of such a MAC with encryption can achieve *authenticated encryption* [6]: we simply have to provide a MAC of the CBC-encrypted ciphertext. We can consider the pair of such keys a single (longer) key k. Throughout this paper, we will write $E_k(x)$ for authenticated encryption of a value x (the plaintext) yielding a ciphertext including a MAC value, and $E^{-1}_{k(y)}$ for the reverse step of decryption of a ciphertext y while also checking for an authentication error based on the MAC value provided as part of the ciphertext.

The use of CBC for confidentiality with CMAC for authentication is just an example of a convenient scheme. Alternatively, we could use any suitable symmetric cryptographic scheme that provides authenticated encryption in an appropriate sense. As an implementation note regarding the particular combined scheme using CBC with CMAC, note that an implementation of either block cipher encryption or block cipher decryption is sufficient in the FPGA: we can arrange to use the block cipher "in reverse" for one of the two cryptographic steps, e.g. use a CMAC based on block cipher decryption rather than on block cipher encryption.

Asymmetric cryptography: Using asymmetric (or public-key) cryptography, a pair of keys, public (PK) and private (SK), can be used for secure communication over insecure channels without prior key establishment. Since asymmetric cryptography is much more computationally demanding than symmetric cryptography, it is commonly used to establish symmetric keys at the initiation of the communication and then use the generally faster symmetric cryptographic. The first publicly known example of public-key cryptography was the Diffie-Hellman (DH) scheme [8], which can be used to establish keys for symmetric cryptography.

In conjunction with a key derivation function (KDF) based on a cryptographic hash function (H), Diffie-Hellman remains state of the art. For example, given any two principals A and B with their corresponding public and private keys $(PK_{\mathrm{A}}, SK_{\mathrm{A}})$ and $(PK_{\mathrm{B}}, SK_{\mathrm{B}})$ and an additional bit string data, where the keys were generated based on common domain parameters, the parties can determine a common symmetric key. This is done by A performing

$$K_{\mathrm{AB}} = \mathrm{KDF}(SK_{\mathrm{A}}, PK_{\mathrm{B}}, \mathrm{data}) = \mathrm{H}(\mathrm{DH}(SK_{\mathrm{A}}, PK_{\mathrm{B}}), \mathrm{data}),$$

by first computing the Diffie-Hellman result based on the key pairs, and then applying the KDF using a secure hash function. Principal B then computes

$$K_{\mathrm{AB}} = \mathrm{KDF}(SK_{\mathrm{B}}, PK_{\mathrm{A}}, \mathrm{data}) = \mathrm{H}(\mathrm{DH}(SK_{\mathrm{B}}, PK_{\mathrm{A}}), \mathrm{data}),$$

to get the same key on his end. The result is that both principals derive the same symmetric key without revealing they private keys; all they had to do was to before communication was to agree on domain parameters, and exchange data, which can be considered public.

The recommendations in [5] for static-key Diffie-Hellman settings additionally require the use of a nonce in the KDF input for each invocation of the key establishment scheme. This use of non-repeated random values ensures that different results can be obtained based on otherwise identical inputs. However, we do not need this nonce here: for our application, the reproducibility of key establishment results is a feature, not a deficiency. A variant of Diffie-Hellman uses elliptic curve cryptography [7], ECDH; a key establishment scheme based on Diffie-Hellman, including the ECDH variant is described in detail in NIST publication 800-56A [5].

2 Prior work and protection schemes

At this stage it is important to distinguish between two types of design protection. "Product protection" is when the system developer protects his own product — which may or may not contain cores from other parties — from cloning and reverse engineering by malicious users. "Core protection" is when cores vendors protect their designs from cloning and reverse engineering by malicious users *and* the system developers. The difference is important for the rest of our discussion.

In response to cloning of FPGA bitstreams, FPGA vendors introduced bitstream encryption as a "product protection" measure. This works by the system developer programming a key into the FPGA, encrypting the bitstream with the same key, and when the FPGA is programmed with this bitstream, an internal decryptor decrypts and the FPGA is configured. Bitstream encryption is now common in high-end FPGAs. Keys are stored in either battery-backed volatile or non-volatile memory, and some devices provide both, though currently this functionality is restricted to a single key. Cloning deterrents that rely on the secrecy of bitstream encoding are also a measure for "product protection". These are not cryptographically secure but may increase the cost of cloning sufficiently to be useful for some designers, and are also available for those devices without bitstream encryption. In 2000, Kessner [17] was the first to propose an FPGA bitstream theft deterrent with a CPLD sending a keyed LFSR stream to the FPGA for comparison to an identical local computation to verify that it is mounted on the right circuit board. More recently, both Altera and Xilinx proposed very similar challenge-response schemes [1, 4] as cloning deterrents where the bitstream authenticates the board it is "running" on by a challenge-response exchange with a cryptographic processor placed on the board. The shared key is stored on the processor, but is only hidden in the bitstream that is loaded onto the FPGA. Kean [14, 15] proposed schemes based on a static secret key already inserted during the manufacturing process. The issue of key transfer is solved by including cores both for encryption and for decryption in the FPGA: an FPGA specimen containing the appropriate key can be used to encrypt a configuration file, yielding a configuration that will work for itself and for other FPGAs sharing the same fixed key. Kean [16] also proposed a more complex token-based protocol which requires additional resources in the configuration logic. More crucially, it requires the participation of the FPGA manufac-

turer (as a trusted party) whenever a bitstream is to be encrypted for a particular FPGA, which means that such transactions cannot be kept just between the IP vendor and the customer.

The scheme described by Güneysu et al. [13] presents a two-stage protocol for establishing keys in an FPGA without personal interaction of the FPGA vendor but it protects, and allows the license of a single core from a single vendor. In fact, it assumes the vASSP scenario where a system developer licenses a complete bitstream that occupies the whole device without adding any contribution of his own. When this is the case, we can achieve the same results as in Güneysu et al. if the cores vendor programs the decryption keys into the FPGA's non-volatile key storage and sells the devices, complete with bitstream, directly to the system developer. However, we believe that virtual ASSPs are only a minor part of the market, and the majority of the business is in licensing cores such that they can be integrated into the system developer's design. In this context, a challenging scenario is to realize a protection scheme covering multiple cores from multiple vendors assembled together into a single design by the system developer. In this paper we address these scenarios by describing a mechanism that can enable it, which still maintaining the same properties set by Güneysu et al.

Several FPGA families have hard embedded processors, while soft ones, such as Nios and MicroBlaze, can be instantiated using the regular HDL flow in user logic. These are used to execute instruction sets corresponding to the architecture, as would otherwise be possible if the processor was an external device. The code is written in a high level language such as C, compiled, and becomes part of the bitstream as RAM; this compiled code may be updated without fully reconfiguring the rest of the user logic. Simpson and Schaumont [23] address the scenario where the system developer creates a design that uses either a hard or soft processor, but is able to accept updates to the compiled code from third parties. Their protocol is able to authenticate and decrypt code based on keys and challenge-response pairs derived from an embedded PUF. This requires the FPGA vendor to collect many challenge-response pairs of vectors from the PUF and enroll those, along with the FPGA's unique identifier, with a trusted party. Then, cores vendors enroll their compiled code, along with a unique identifier, with the trusted party. Through several exchanges between principals and the mutually trusted party, the compiled code is encrypted such that it can only be used in FPGAs that can reproduce the correct PUF response given a certain challenge vector. The scope to which this scheme applies is limited to the secure distribution of compiled code for embedded processors and does not apply to cores that use the FPGA's user logic. Guajardo et al. [12] suggest enhancements to this protocol and also describe an implemented PUF, as discussed in Section 6, which was originally simulated by an AES core by Simpson-Schaumont.

3 Single core protection

For clarity, a list of principals and notations in given in Table 1.

We now describe in more detail the scheme proposed by Güneysu et al. [13] for a single FPGA. In this scheme we describe the business use case where a system developer likes to license a virtual ASSP from a core vendor, i.e., the core vendor provides the design as a bitstream which should be installed on SD's FPGA products. Our protocol is principally

Notation	Meaning
FPGA	The physical FPGA
FV	FPGA vendor
CV	Cores vendor
SD	System developer
FID	A unique FPGA identification number
K, $\{PK, SK\}$	Symmetric and {private, secret} key pair
DH	Diffie-Hellman key establishment
H	Cryptographic hash function
KDF	Key Derivation Function
E, E^{-1}	Symmetric encryption and decryption
CORE, CORE_{K_z}	Bitstream as plaintext and ciphertext
PB, PB_{K_z}	Personalization bitstream in plaintext and ciphertext

Table 1: Principal and notation summary.

based on the idea of establishing a shared secret key K_{CV} between the FPGA and the cores vendor in the untrusted domain of the system designer since this is a "cores protection" problem. The key K_{CV} is used by the cores vendor to encrypt his core, which can then only be decrypted by the specific FPGA which is able to derive the same secret key using a key establishment protocol. Note that the following description covers the steps performed by each party according to an ideal protection protocol, i.e., assuming that all parties behave as desired without any fraud attempts or protocol abuse. Our scheme consists of of four main stages, as follows:

A. **SETUP.** This initial step has to be done by the FPGA vendor only once when a new family or class of FPGA devices is launched. For a group of devices, the FPGA vendor generates a public key pair first and creates a specific bitstream containing an algorithm for establishing symmetric keys using public-key cryptography. We denote this bitstream as a *Personalization Bitstream* (PB) since it will be used to install a secret key K_{CV} in the FPGA's keystore which can be used by the core vendor to bind the FPGA design *specifically* to this device using symmetric encryption of the bitstream. Since the PB contains secret key information from the FV required for this process, only an encrypted version of the PB is distributed, together with the public key of the FPGA vendor.

B. **LICENSING.** Assume a system developer to license a vASSP bitstream from a core vendor. Then, the core vendor generates a public key pair for this specific customer and provides the public key part to the system developer. In turn, the SD transfers a unique device identifier *FID* of each FPGA to the core vendor on which the SD intends to use the licensed IP. In his trusted domain, the core vendor then derives a unique symmetric and device-specific key K_{CV} which is derived using public-key crypto involving the keys from FV and CV as well as the received device *FID*. The exact specification of the involved Key Derivation Function (KDF) will be explained later on. For each distinct key K_{CV} (and thus FPGA), the core vendor encrypts the licensed design FPGA and sends the encrypted bitstreams to the system developer.

C. **PERSONALIZATION.** Next, the same keys, which have been previously used by the core vendor to encrypt the vASSP bitstream, are established on the corresponding FPGA devices at the system developer's place. The system developer loads the PB(and CV's public key) on each FPGA device which has been specified to the core vendor before with its individual *FID*. With the PB the key establishment scheme installs the same device-specific key K_{CV} in the FPGA based on the his and FV's key information and the received device *FID*.

D. **CONFIGURATION.** Finally, due to the binding of the key K_{CV} to a device, the SD can use each specifically encrypted bitstream only on that FPGA on which the appropriate key is available.

Let us now consider the details. For the protocol to work, the FPGA vendor is required to provide the following. Firstly, each FPGA must have an embedded unique, non-secret, identification number (*FID*). This can be done by the foundry or at the FPGA vendor's facility, though there must be certainty ID's are unique for each FPGA. Secondly, each FPGA must have embedded in it a unique symmetric key, K_{FPGA}, in non-volatile key storage that is readable only by the decryption core; this is done in a secure facility by the FPGA vendor. This key, K_{FPGA}, is used to decrypt the encrypted (and authenticated) personalization bitstream, PB later on. Thirdly, as already explained, the FPGA vendor generates a public key pair, $(SK_{\mathrm{FPGA}}, PK_{\mathrm{FPGA}})$ for a family of devices. Finally, the vendor creates the PB that contains the private key SK_{FPGA} shared among the same device family and the circuit required to derive a symmetric key used to establish K_{CV} using a Key Derivation Function (KDF). In the actual licensing process, the FPGA vendor does not play an active role and only participates in the setup stage. Hence, the information exchange between the parties is relatively simple. Figure 1 shows the data flow between the participants.

The personalization bitstream, shown in Figure 2, is vital to the security of our scheme because it contains the FPGA vendor's secret key SK_{FPGA} (for a group of FPGAs), with which an attacker (or FPGA owner) could establish duplicate keys. Thus, this bitstream is encrypted (and authenticated) using K_{FPGA} to produce $\mathrm{PB}_{K_{\mathrm{FPGA}}} = E_{K_{\mathrm{FPGA}}}(\mathrm{PB})$. As explained above, the core vendor generates a public key pair (SK_{CV}, PK_{CV}), e.g., for each individual system developer SD and IP core, denoted as CORE. Using this key and the *FID* of the FPGA, the CV generates an shared encryption key K_{CV} for each device by computing

$$K_{\mathrm{CV}} = \mathrm{KDF}(SK_{\mathrm{CV}}, PK_{\mathrm{FV}}, FID)$$

which is used to encrypt CORE to $\mathrm{CORE}_{K_{\mathrm{CV}}} = E_{K_{\mathrm{CV}}}$ in a next step.

Now, to prepare the FPGA for running the licensed core, the system developer configures it with PB and initiates the key establishment by loading PK_{CV} from the cores vendor to the device. The key establishment within the PB generates the key using the following function

$$K_{\mathrm{CV}} = \mathrm{KDF}(SK_{\mathrm{FV}}, PK_{\mathrm{CV}}, FID)$$

is produced and stored in the fabric writable register (only readable by the decryption cipher) in the device. Now, the FPGA can be programmed with the licensed core $CORE_{K_{\mathrm{CV}}}$.

A. SETUP :

FV		**SD**
$E_{K_{FPGA}}(\text{PB})$		
	$\xrightarrow{FID,\, PK_{\text{FPGA}},\, \text{PB}_{K_{\text{FPGA}}}}$	

B. LICENSING :

CV		**SD**
	$\xleftarrow{FID,\, PK_{\text{FPGA}}}$	
verify certificate PK_{FPGA}		
$K_{\text{CV}} = \text{KDF}(SK_{\text{CV}}, PK_{\text{FPGA}}, FID)$		
$E_{K_{\text{CV}}}(\text{CORE})$		
	$\xrightarrow{PK_{\text{CV}},\, \text{CORE}_{K_{\text{CV}}}}$	
		verify certificate PK_{CV}

C. PERSONALIZATION :

SD		**FPGA**
	$\xrightarrow{\text{PB}_{K_{\text{FPGA}}}}$	
		$E^{-1}(\text{PB}_{K_{\text{FPGA}}})$
	$\xrightarrow{PK_{\text{CV}}}$	
		$K_{\text{CV}} = \text{KDF}(SK_{\text{FPGA}}, PK_{\text{CV}}, FID)$

D. CONFIGURATION :

SD		**FPGA**
	$\xrightarrow{\text{CORE}_{K_{\text{CV}}}}$	
		$E^{-1}(\text{CORE}_{K_{\text{CV}}})$

Figure 1: The protocol description. Note that the system developer and core vendor must verify the certificate of public keys in order to prevent attackers from being able to decrypt the cores or insert malicious functions into designs.

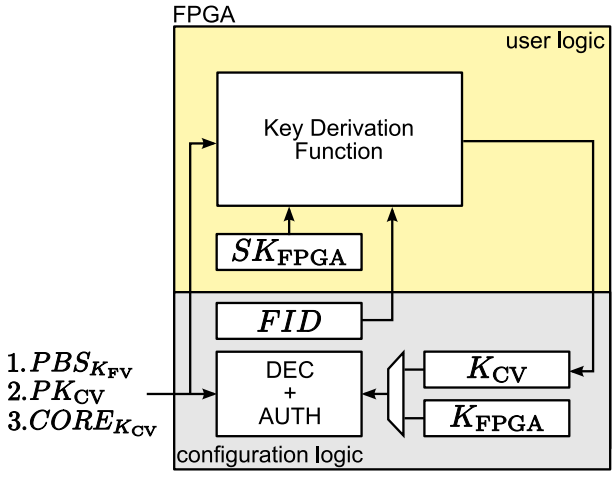

Figure 2: Step 1 of the "personalization" process begins by configuring the FPGA with $\text{PB}_{K_{\text{FPGA}}} = E_{K_{\text{FPGA}}}(\text{PB})$ decrypted and authenticated by the stored K_{FPGA}. In Step 2 CV's public key PK_{CV} is sent through the configuration port and together with SK_{FPGA} and *FID* contained in the FPGA and the PB, the shared key K_{CV} is computed and stored. Finally, in step 3 the encrypted core is sent to the FPGA and decrypted using K_{CV}.

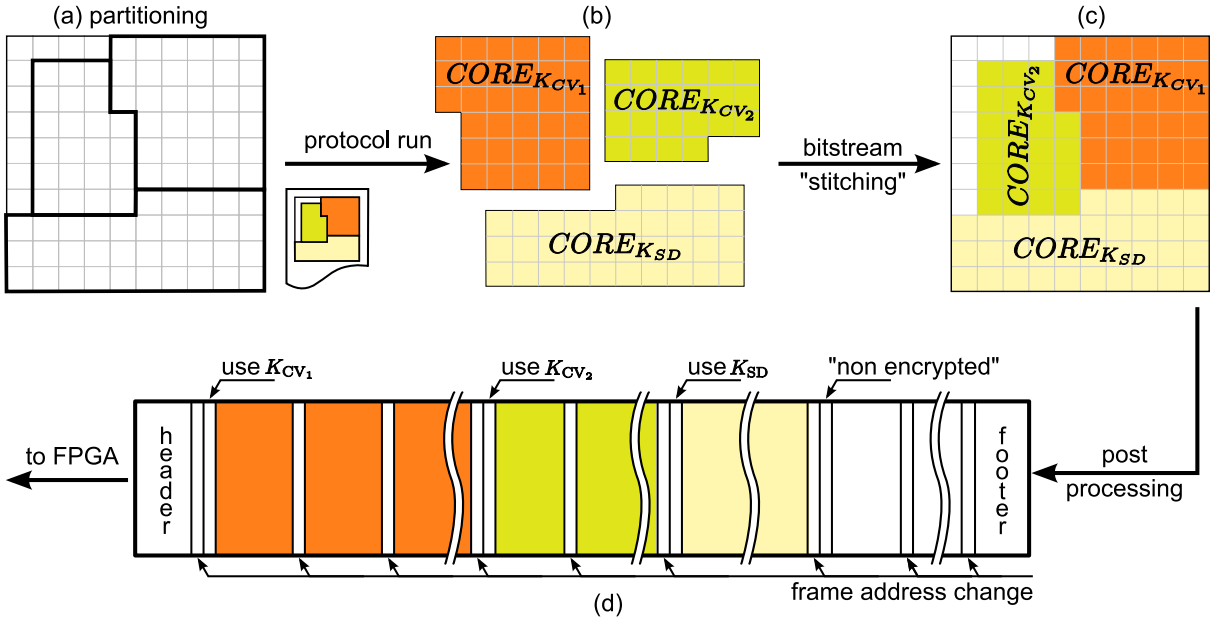

Figure 3: Initially, the system developer budgets the design and assigns partitions to cores vendors; this is done using a partitioning report made by the tools. At the end of the protocol run the cores are "stitched" together (or superimposed on a full bitstream). A post processing tool creates a bitstream that has contiguous frames based on key domains.

4 Many-core protection

In the previous section we described one solution to the virtual ASSP problem, where a single licensed bitstream is loaded onto the device with no contribution from the system designer. However, we need a solution of wider scope where the system developer incorporates cores from multiple vendors into a single design with his own contribution. We now extend the previous scheme such that multiple cores from multiple sources can be protected while allowing the system developer to add his own circuit into the design *and* the cores vendors to enforce the licensing agreement.

The primary observation is that the further up we go in the development flow, bitstreams being the lowest level, the harder it is to protect licensed designs. This is because the higher we go more tools and principals are involved, all of which need to be trusted (to various degrees) in order to implement the protection scheme. For example, delivering encrypted netlists requires the synthesizer and all the tools used thereafter to maintain the confidentiality of the design. When these tools come from various vendors it becomes increasingly harder to have a ubiquitous protection scheme that is also secure, especially if the software is required to be run on an untrusted client. Our conclusion, therefore, is that the most appropriate place where the combination of cores can occur is at the bitstream level, where decryption only takes place within the FPGA itself.

An overview of the scheme is shown in Figure 3. The system developer starts by partitioning the design into functional blocks and assigning physical allocations for them inside of the FPGA. The diagrammed "FPGA" is divided into an array of blocks, which provide the granularity of the partitioning; those blocks can be thought of as the FPGAs "frames"

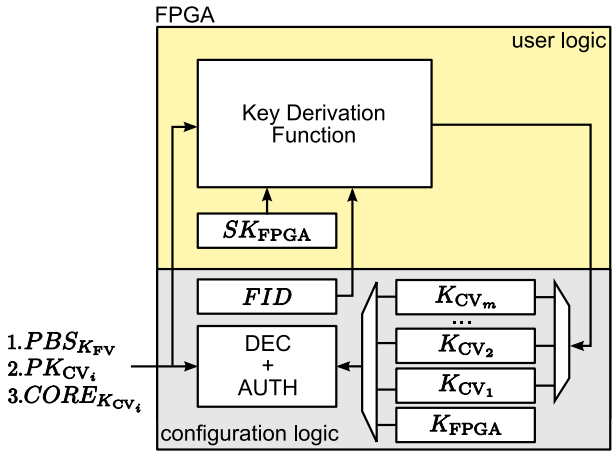

Figure 4: For multi-core protection, the personalization bitstream configuration logic can handle multiple keys K_{CV_i}. In addition to the FPGA vendor's FPGA key, there is storage for up to m decryption keys.

that are the configuration bitstream's building blocks. For each of these partitions the developer decides if it is going to be purchased from a core vendor or designed internally. At this stage, the interface between the blocks must also be defined (preferably using widely established macros though this is not strictly necessary if the interface can be agreed upon between the parties). This information is processed by a tool that produces a "partitioning definition" file containing all the information required for the FPGA software flow to confine a design to the allocated partition from HDL to the bitstream. With this file all parties can synthesize, place-and-route, and create a bitstream portion that corresponds to the allocated FPGA partition. The definition file is distributed in the licensing stage of the protocol, only this time with multiple core vendors. Now the system developer has bitstream portions corresponding to each partition, and those are encrypted under the respective core vendors' keys, $CORE_{K_{\mathrm{CV}_1}}, \ldots, CORE_{K_{\mathrm{CV}_m}}$, as shown in Figure 3(b). These portions are processed by a tool that "stitches" them together to produce a full bitstream, shown in Figure 3(c) (the stitching process may simply be superimposing the partial bitstreams onto a complete bitstream). Moreover, the system developer also has the core vendors' public keys corresponding to the design, $PK_{\mathrm{CV}_1}, \ldots, PK_{\mathrm{CV}_m}$.

The system developer now loads the personalization bitstream onto the FPGA and then sends iteratively the PK_{CV_i} which produces the respective K_{CV_i} in the configuration logic; this process is shown in Figure 4. The bitstream of Figure 3(c) cannot be loaded yet (as normal) because of the different key domains. Thus we require an additional command in the bitstream that tells the configuration logic which key to use to decrypt the bitstream. Another command we require is the one telling the configuration logic where to load the next frame. Using the partitioning file, a tool then rearranges the bitstream such that frames belonging to each key domain are contiguous, with the addition of the "use K_{CV_i}" instruction at the beginning of the block, and then "go to frame address X" between sub-blocks such that these are configured at the right place. Such a bitstream is shown in Figure 3(d).

For this scheme to work, we require the software to support a complete modular design

flow from HDL to bitstream generation, and to be able to "stitch" bitstreams based on particular partitions. In most part, this functionality is already present in some tools. Currently, the Xilinx and Altera tools support modular design to help developers cope with large designs that involve multiple teams working separately. Modular design allows the partitioning of the design such that portions can be held constant while others are re-synthesized in order to help portions of large designs be implemented by separate teams. When all partitions are complete, a "final assembly" maps, places and routes all of them together to generate a single bitstream. Bitstream-level partitioning is already supported by the Xilinx tools for partial reconfiguration, where portions of bitstreams are created that can reprogram portions of the device while the rest is operating as normal. In order to achieve this, interfaces between the partitions are defined such that the process of partial reconfiguration does not create contention.

In terms of additional hardware functionality, our scheme requires additional volatile or non-volatile (one-time programmable) symmetric key storage, one each for each core requiring protection. As with K_{CV} these keys are user-logic-writable but only readable by the decryption core; these keys are called K_{CV_j}, where j ranges from 0 to m. They are generated, as before, by using the personalization bitstream to process K_{CV_j}'s up to m. If the keys are stored in non-volatile memory then this process only happens once by the system developer. If volatile storage is used, the shared keys need to be regenerated on every power-up (PK_{CV_i}'s loaded via PROM). Note also that in this case a larger external configuration memory is required to store the $\mathrm{PB}_{K_{FPGA}}$ *and* application bitstream. The advantages of the latter is that keys are only generated when needed, and can be erased after configuration. In addition, volatile storage allows changing of the keys and is less prone to mishaps of wrong key configuration.

4.1 Key management

In Section 3 we discussed having a single symmetric key K_{CV} per *FID*. In the multi-core licensing model, we can establish multiple keys for each individual core still using only a single *FID*. More precisely, each contributing CV will generate a distinct key pair (SK_{CV_i}, PK_{CV_i}) for use with his individual CORE. Due to the different key pairs as input to the KDF totally different key K_{CV_i} will be created by the PB and written to the key store of the FPGA - although all relying on the same *FID*:

$$\begin{aligned} K_{\mathrm{CV}_1} &= \mathrm{KDF}(SK_{\mathrm{FV}}, PK_{\mathrm{CV}_1}, FID) \\ &\dots \\ K_{\mathrm{CV}_m} &= \mathrm{KDF}(SK_{\mathrm{FV}}, PK_{\mathrm{CV}_m}, FID) \end{aligned}$$

Note that our scheme supports very fine-grain group-key management. A CV can decide that a limited number of FPGAs are combined to a set that share a common key pair (SK_{CV_i}, PK_{CV_i}), e.g. all FPGAs that are licensed to a specific system developer. Hence, by defining distinct classes for each SD incorporating the IP core, the CV can make sure that — in case that the secret key SK_{CV_i} for a specific SD is revealed by accident — the ability to clone devices applies only to that system developer rather than to the entire system. In particular, the abuse of this secret key SK_{CV_i} is simple to trace back to the origin, hence the corresponding system developer for which this secret key was issued can be called to account.

5 Discussion of requirements and assumptions

5.1 Requirements

Our scheme requires the FPGA vendor to implement certain functions as hard blocks in addition to allowing access to various keys.

Bitstream encryption and authentication: Bitstreams must be cryptographically authenticated in addition to decryption as they are processed by the configuration block. This is done to avoid any tampering of the bitstreams by an attacker, e.g., to modify security-relevant components in a way to learn the key from some integrated information leakage. Note that many high-end FPGA devices already include an integrated symmetric decryption core to deal with encrypted bitstreams. The advantages of having both is discussed by Drimer [9], along with the benefits of separating the two functions to allow bitstream to be authenticated without being encrypted.

Secure keystore: Non-volatile storage for both *FID* and K_{FPGA} has to be provided with the former being readable from the fabric and the latter only readable by decryption the engine. Note that both parameters do not necessarily require to be changed so that one-time programmable memory can be used. Key storage should be designed such that physical or side-channel attacks are impractical. It is also important for the write access to the key memory to be atomic, in the sense that it only allows all key bits to be replaced at once. Write access to parts of the key memory, such as individual bytes, could be used to exhaustively search for the write operation that does not change the behavior of the key and thereby reveal individual key bits.

Partial reconfiguration: In general, partial reconfiguration defines distinct portions in an FPGA design which can be reconfigured while the rest of the device remains in active operation. We do not required partial reconfiguration for our scheme, though the software to generate these partial bitstreams would be used as well as the ability to address specific configuration frames within the FPGA. In addition to that we need a new bitstream command that tells the configuration logic which key to use.

5.2 Assumptions

In addition to requirements on the FPGA devices, we make the following assumption.

Trusted parties: We assume that the FPGA vendor is a trusted party to all participates. Everyone must trust that the FPGA vendor is not doing anything malicious inside of the personalization bitstream that creates K_{CV_i} and does not divulge any keys it is storing. System developers already trust the FPGA vendor to provide them with a functional FPGA that will not compromise their systems. That said, if the scheme is widely adopted or concerns arise about the FPGA vendor having too much control over fielded systems, a vendor-independent entity could program and store the keys, and encrypt the personalization bitstreams. This is at a logistical cost of having the FPGAs go through another hand on the way to the system developer, though a dedicated entity managing keys can guarantee the security of keys better than vendors.

Cryptographic implementations: We also assume that the cryptographic primitives and security parameters chosen (key and hash lengths) are computationally secure such

that no attacker will be able to compromise the system by brute force and other known attacks. Most modern FPGAs use the strong AES-256 for bitstream decryption. For the recommendation of bit sizes for public-key cryptography, we follow Lenstra and Verheul [21] and assume ECC algorithms and cryptographic hash functions with 256-bit as secure, e.g. ECC over NIST-256 prime and SHA-2 hash. The implementation of these primitives are assumed to be fault-tolerant and secure against side-channel attacks, ones that may allow the attacker to extract keys by observing timing variations, power consumption, or electromagnetic emanations while the device is in operation. Physical attacks are also assumed to be out of reach of the attacker such that keys cannot be extracted, or disabled functions be re-enabled (such as readback).

Communication between parties: The communication between the system developer and cores vendors should take place over secure and authenticated channels for to ensure non-repudiation and prevent man-in-the-middle attacks. This can be achieved by using a common Public-Key Infrastructure (PKI). Note that the public key pair issued by the FPGA vendor must be certified under the PKI as well. This avoids that an attacker can easily replace the original with a forged personalization bitstream (using the attacker's own public key pair) which, for example, could support the extraction of the generated key K_{CV} from the FPGA *before* it is written to the key store. Having a certified key pair from the FPGA vendor, the core vendor can be assured that all the issued and encrypted COREs can be only used with FPGAs on which keys using the original PB have been established.

6 Evaluation

This section covers aspects of the protection scheme with respect to advantages and disadvantages over other proposals as well as its general practicability in terms of additional overhead and costs.

6.1 Advantages

Our scheme has some notable advantages over current proposals.

Small, scalable and optional. The additions to the FPGA by the vendor are minimal and the current use model for FPGAs is unchanged. Bitstream configuration and encryption is still used as before, so from the system developer's point-of-view, this is an opt-in feature that can simply be ignored if not used. Furthermore, developers can use the scheme for their own designs by having portions of their designs encrypted and authenticated under different keys based on their content. This is particularly interesting for designs having an integrated soft core microprocessor. Soft and hard processors use RAM blocks to store compiled code, so an application designers could use a dedicated key for RAM block to protect their embedded software application. FPGA vendors can gradually introduce the scheme across several generations starting with adding bitstream authentication and then gradually adding additional key storage for more than a single core encryption. The vendors are not limited to any particular KDF, and the personalization bitstream can use any type that maintains the security properties while using the latest architectural advances in new generations of FPGAs.

Use of established primitives. Our protection model relies on well-established crypto primitives like Diffie-Hellman key exchange and cryptographic hash functions for which many FPGA-based implementations have been proposed in the open literature. As an alternative, Schaumont and Guajardo, for example, proposed design protection schemes which use physical unclonable functions (PUF) for unique key generation. PUFs seem attractive for authenticating devices, as they can produce a key on-demand based on the physical properties of the device. Their main advantages are: creating a model for faking PUFs is designed to be hard given that is it based on physical properties of the individual die; derived keys "exist" only when needed, and are not permanently stored; invasive tampering changes the properties of the PUF such that the correct key can no longer be reproduced; there is no need to program a key, and it does not even need to leave the device; random challenge-response pairs can be created with some PUFs such that a unique string is generated for the purpose of authentication; PUFs should be scalable in that more of them can be generated according to security needs; and finally, for FPGAs, several of the proposed PUF structures can already be realized in existing devices.

When used for identification and to produce keys for cryptographic applications, PUFs' output should be *unique* for each die and *reproducible* for a given die irrespective of temperature and voltage variation. These properties are quite challenging to achieve and pose the most difficulty in the generation of consistent, yet randomly distributed, bit strings. PUFs are not a mature technology, and may not be so for a while to come, so we opted to use established techniques in order to increase the likelihood of adoption by the risk-averse FPGA vendors. That said, we are able to use PUFs in our scheme for the generation of unique IDs and keys once those become reliable.

Moreover, additionally to keys generated by PUFs, True Random Number Generators (TRNG) could also be used to establish random keys K_{CV_i} inside the FPGAs. A few implementations for cryptographically secure RNGs in FPGAs are already available [19, 24]. Secret values created by TRNGs can be installed as K_{CV_i} in the keystore during personalization stage and thus can even replace the necessity of a fixed *FID* inside the device. The identification of the device is then solely based on the uniqueness of the key which, however, must be transferred confidentially to the core vendor. One solution can rely on public key encryption like RSA or ElGamal with which the generated key K_{CV_i} is encrypted using the CV's public key PK_{CV} inside the PB. Next, the encrypted key $E_{PK_{\mathrm{CV}}}(K_{\mathrm{CV}_i})$ leaves the FPGA device and is sent back to the CV who, in turn, then decrypts K_{CV_i} with this and generates the specifically encrypted CORE. In this case, the personalization bitstream employs a TRNG and a public-key encryption instead of a key derivation function based on Diffie-Hellman. Although we could use TRNGs, we think that the system would be more robust if we can achieve the same results using primitives that are less prone to tampering, as TRNGs are.

Configuration times. If we take the time it takes for today's FPGAs to process an encrypted bitstream as a baseline, our scheme does not adversely affect it. The transition from one key domain to another and the changing of frame addresses will only introduce very short delays. This is the case where keys are permanently stored in the FPGA (after the personalization bitstream has been "executed" at the developer's). However, in the case where volatile key storage is used (for CV_i's) the personalization bitstream would need to be loaded on every power-up, significantly reducing power-on times. We also assumed that bitstreams are authenticated, which is a function that does not exists

today, so it too may introduce delays to configuration times depending on how it is implemented.

Incentives. Security in practice is best when the parties that can enhance, or are in charge of, security are held accountable in cases of breach. It is therefore important to consider the incentive structure of any scheme in order to make sure that those are aligned correctly. In our scheme, the system developer can only compromise his own keys, so there is a strong incentive for keeping those safe. Similarly, compromise of the cores vendors' keys damage the cores vendors themselves. Both, however, rely on the the symmetric keys K_{FPGA_j} and secret keys SK_{FPGA_j} kept by the FPGA vendor, who is considered a trusted party. Compromise of these keys will invalidate the security of the whole scheme. The FPGA vendor has reputation to lose, though this may not be enough to make sure keys are kept safe. Therefore, we propose that when system developers and core vendors enroll for the scheme, the FPGA vendor will guarantee compensation in case keys are compromised.

6.2 Disadvantages

Of course, no system is perfect; here we discuss the disadvantages of the presented scheme.

Loss of optimization. Normally, when functional blocks are combined at the HDL level the synthesizer can optimize the design for performance and resources, but since this will not be possible with our scheme some resources will be potentially "lost". In addition, it is inevitable that some small portions of the allotted partition will not be fully used due to the frame resolution and initial partitioning, so the overall utilization may not be optimal. That said, two things can improve the situation. Firstly, careful planning and precise information from both the system developer and cores vendor can minimize the unused resources. Secondly, if the cores vendor's design is already optimized (for either throughput or area) further optimization by the EDA tools will not give significantly better results, and the design will already have been compacted to the tightest area possible.

Reliable PB operation in an untrusted domain. The key establishment of K_{CV} inside of the FPGA takes place in the untrusted domain of the system developer. In the "core protection" scenario the system developer is a potential adversary, so the implementation of cryptographic elements in the personalization bitstream must be fault-tolerant and tamper-resistant. Countermeasures against device tampering are required to check if the module is executed under unconventional conditions, e.g, over-voltage, increased temperature or clock frequency. A fairly easy approach to detect operational faults caused by such conditions is to use multiple, identical cryptographic components in the personalization module operated at different clocks domains, or insert randomization. If sufficient countermeasures are introduced [2, 20], we can assume that this will satisfy the cost-conscious developer.

Bandwidth and availability. An important consideration is the amount of network bandwidth required between the system developer and cores vendors since every core has to be individually encrypted for each FPGA and transferred between both parties. Since the data is encrypted compression will not be beneficial, so a few megabytes of data is transferred for each instance; large volume transaction may require gigabytes of data to

be transferred. We do not view this as a major concern, but something that must be addressed as part of the scheme's evaluation. The protocol exchanges can be made in one lump for a batch of FPGAs, as opposed to requiring a challenge response exchange, which means that data can be delivered on magnetic or optical media by a courier; there is nothing in our protocol that prevents this from happening. This property also defends against a denial of service attack targeting the cores vendors' servers.

We can minimize the bandwidth by issuing encrypted tokens to each FPGA instead of the encrypted core. The protocol is used to produce K_{CV} as before, but instead of the CV sending SD the encrypted core, it sends a token encrypted using K_{CV} that contains the the key needed to decrypt $K_{\mathrm{CV}'}$. The advantage of this extension is that only tokens need to be distributed rather than the cores themselves, and only a single core instance, encrypted using $K_{\mathrm{CV}'}$ is distributed. Of course, $K_{\mathrm{CV}'}$ is limited to a particular core from a core vendor and system developer's FPGA FIDs.

Simulation and Trust. Although we suggest that a crippled version of the core (as a bitstream matching the partition) be delivered to system developers for integration and testing, there is no way for them to check that the core does not contain malicious functions or Trojan horses. Since the core is a black box to the system developer he can only test it using input and output vectors, which quite possibly will not detect any extra functionality. Even if our scheme allows the vendor to test the core as a black box once it is loaded onto the FPGA, the fact that each core is encrypted using a different key requires testing each one, which may be costly. To what extent is this a problem? We assumed a cost-conscious developer, one that is interested in low cost solutions and is generally trusting (unless the developer is well funded, this is a pre-requisite as there is no way to verify that all EDA and hardware used are trust-worthy). The developer relies on social means such as contracts, agreements and the reputation of the vendor he purchases hardware and software from. One can say that trust has a price; if the developer wanted to verify the core, he would need to pay to see it. We can also have a commitment scheme where the software is able to cryptographically commit to a particular output, though that would require more trust in the software and complicate the protocol, which we wanted to avoid.

7 Implementation considerations

We will give some brief suggestions and implementation details how to realize the participating components, and discuss their expected system costs.

7.1 Implementing the personalization module

In the following we will demonstrate the feasibility of the personalization module that incorporates the implementation of a key establishment scheme.

For this proof-of-concept implementation, we realized an ECC core over prime fields, which forms the basis for an Elliptic Curve Diffie-Hellman (ECDH). Parameters for this implementation have been chosen by a trade-off of long-term security and efficiency: We designed an ECDH component over $GF(p)$ where p is a 160-bit prime, which is sufficient for mid-term security nowadays.

We take the smallest Virtex-4 (FX12) FPGA with an integrated AES-256 bit stream decryption core as a reference device. For the implementation of the KDF according to [5], we integrated an implementation of SHA-1 as the cryptographic hash function $h(x)$ with an output of 160 bits. During key generation, the SHA-1 hash function is executed twice using three different inputs: Given two 32-bit counter values (effectively, two constants) c_0, c_1, a 128-bit *FID* and the ECDH result E, a 256-bit AES key K_{FPGA} can be derived as follows:

$$\begin{aligned} H_i &= h(c_i \parallel E \parallel FID) \text{ for } i \in \{0,1\} \\ K_{\text{FPGA}} &= S(H_0 \parallel H_1), \end{aligned}$$

where $S(x)$ is a selection function choosing the first 256 out of 320 input bits and where "$\parallel$" denotes concatenation.

Component	4-input LUT	Flip flops	BRAM	Frequency
ECDH Core	5,674 (51%)	767 (7%)	5	65 MHz
SHA-1 Core	1102 (9%)	689 (6%)	0	76 MHz

Table 2: Implementation details for crypto components of the PB synthesized for a Xilinx Virtex-4 FX12-12 FPGA

It should be remarked that all implementations have been developed for an optimal area-time product so that reductions in logical elements can still be achieved if data throughput is not a primary issue. To compute the the ECDH key with 160 bit based on the implementation presented above, it takes 205,200 clock cycles on average. The SHA-1 requires 96 clock cycles to produce a 160 bit output for a single input block, i.e., for two block computations 192 cycles in total. Assuming all components in the PB to run at a clock frequency of at least 50 MHz, the generation of a single AES-256 key will take 4.1 ms. This run-time is negligible in case that a single setup is performed establishing the K_{CV} in non-volatile memory of the key store. However, considering a volatile key store where the key establishment must take place on each power-up prior to loading the application, this time must be taken into account in addition for each key to be established to the an initial latency before the application is fully configured.

Concluding, the implementations at hand are small enough to fit even the smallest Virtex-4 device (providing a total of 5,472 slices with 10,944 LUTs and flip flops each) with some remaining logical resources to add functionality providing tamper resistance and fault tolerance.

Beside the KDF implementation using the Diffie-Hellman key exchange, the PB can make use of a combination of TRNG and public-key encryption. Many public-key encryption algorithms have successfully been implemented in FPGAs, e.g., low footprint RSA implementations have been proposed in [11, 18, 22] and are already available in ready-to-use IP cores by FPGA vendors like Altera. Moreover, experimental TRNGs have been presented in [19, 24] so that this alternative for the PB should be possible as well.

8 Conclusions

We proposed a new many-core protection scheme for FPGA designs that provides an appealing solution to the FPGA "cores distribution problem", though with only mod-

est modifications to current FPGAs and their use models. Starting with the FPGA vendor programming the device with an "FPGA key", a resource-preserving personalization bitstream, temporarily located in the reconfigurable logic, is used for one-time key-establishment. In our model we not only cover the protection of a "virtual application specific standard product" where a ready-made core needs to be secured as a whole, the scheme is also applicable to designs consisting of many cores from multiple core vendors. For each individual core a pay-per-use license model can be used where each single core vendor can enforce how many of cores are used in the field by the system developer.

Acknowledgments

Saar Drimer's research is funded by Xilinx Inc.

References

[1] Altera Corp. *FPGA design security solution using MAX II devices*, September 2004. `http://www.altera.com/literature/wp/wp_m2dsgn.pdf`

[2] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic processors – a survey. *Proceedings of the IEEE*, 94(2):357– 369, Feb 2006

[3] R. J. Anderson. *Security engineering: A guide to building dependable distributed systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001. ISBN 0471389226

[4] C. Baetoniu and S. Sheth. *XAPP780: FPGA IFF copy protection using Dallas Semiconductor/Maxim DS2432 Secure EEPROM*, Xilinx Inc., August 2005. `http://www.xilinx.com/support/documentation/application_notes/xapp780.pdf`

[5] E. Barker, D. Johnson, and M. Smid. *FIPS 800-56: Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography*, NIST, U.S. Department of Commerce, March 2007

[6] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976, pages 531–545, 2000

[7] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press, 2006

[8] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976

[9] S. Drimer. Authentication of FPGA bitstreams: why and how. In *Applied Reconfigurable Computing*, volume 4419 of *LNCS*, pages 73–84, March 2007

[10] M. Dworkin. *Special Publication 800-38B: Recommendation for block cipher modes of operation: The CMAC mode for authentication*, NIST, U.S. Department of Commerce, May 2005

[11] J. Fry and M. Langhammer. RSA & Public Key Cryptography in FPGAs. Technical report, Altera Corp., 2005

[12] J. Guajardo, S. S. Kumar, G. J. Schrijen, and P. Tuyls. FPGA intrinsic PUFs and their use for IP protection. In *Cryptographic Hardware and Embedded Systems Workshop*, volume 4727 of *LNCS*, pages 63–80, September 2007

[13] T. Güneysu, B. Möller, and C. Paar. Dynamic intellectual property protection for reconfigurable devices. pages 169–176, November 2007

[14] T. Kean. Secure configuration of field programmable gate arrays. In *11th International Conference on Field-Programmable Logic and Applications, FPL 2001*, 2001

[15] T. Kean. Secure configuration of field programmable gate arrays. In *IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2001

[16] T. Kean. Cryptographic rights management of FPGA intellectual property cores. In *Field Programmable Gate Arrays Symposium*, pages 113–118, 2002

[17] D. Kessner. *Copy protection for SRAM based FPGA designs*, May 2000. `http://web.archive.org/web/20031010002149/http://free-ip.com/copyprotection.html`

[18] Ç. K. Koç. RSA hardware implementation. Technical report TR801, RSA Data Security, Inc., Aug. 1995

[19] P. Kohlbrenner and K. Gaj. An embedded true random number generator for FPGAs. In R. Tessier and H. Schmit, editors, *FPGA*, pages 71–78, 2004

[20] O. Kömmerling and M. G. Kuhn. Design principles for tamper-resistant smartcard processors. In *USENIX Workshop on Smartcard Technology*, 1999

[21] A. Lenstra and E. Verheul. Selecting Cryptographic Key Sizes. *Journal of Cryptology*, 14 (4):255–293, 2001

[22] A. Mazzeo, L. Romano, G. PaoloSaggese, and N. Mazzocca. FPGA-based implementation of a serial RSA processor. *Design, Automation and Test in Europe Conference and Exposition*, pages 10582–10589, 2003

[23] E. Simpson and P. Schaumont. Offline hardware/software authentication for reconfigurable platforms. In *Cryptographic Hardware and Embedded Systems Workshop*, volume 4249 of *LNCS*, pages 311–323, October 2006

[24] B. Sunar, W. J. Martin, and D. R. Stinson. A provably secure true random number generator with built-in tolerance to active attacks. *IEEE Transactions on Computers*, 56.1: 109–119, January 2007

[25] R. Wilson. *Panel unscrambles intellectual property encryption issues*, January 2007. `http://www.edn.com/article/CA6412249.html`

[26] R. Wilson. *Silicon intellectual property panel puzzles selection process*, February 2007. `http://www.edn.com/article/CA6412358.html`