Implementation of a Server Architecture for Secure Reconfiguration of Embedded Systems

Yannick Verbelen*, An Braeken*, Serge Kubera*, Abdellah Touhafi*, Jo Vliegen† and Nele Mentens†
*Erasmushogeschool Brussel, Brussels, Belgium
Email: {yannick.verbelen, an.braeken, serge.kubera, abdellah.touhafi}@ehb.be
†Katholieke Hogeschool Limburg, Hasselt, Belgium
Email: {jo.vliegen, nele.mentens}@khlim.be

Abstract—Field reconfigurable logic finds an increased integration in both industrial and consumer applications. A need for secure reconfiguration techniques on these devices arises as live firmware updates are essential for a guaranteed continuity of the application’s performance. Ideally, a wide variety of different reconfigurable devices in a range of applications should be configurable with suitable firmware from a central location, since outdated or wrong configuration data could potentially cause irreversible damage to the device. At the same time eavesdropping must be made unfeasibly difficult to keep the intellectual properties of the application provider secured.

This work proposes a software architecture for a server platform allowing secure bidirectional communication over TCP/IP with reconfigurable logic in the field. Moreover a performance comparison between C# and Java is discussed for the different cryptographic algorithms applied in the application.

Index Terms—Server Architecture, Embedded System, FPGA, CRU

1 INTRODUCTION

The increased presence of reconfigurable logic devices such as Complex Programmable Logic Devices (CPLDs) and Field Programmable Gate Arrays (FPGAs) in secure applications originates the need for a mechanism to securely reconfigure these devices with a revised bit stream. In the project STRES (Secure Techniques for Remote reconfiguration of Embedded Systems), a complete solution is developed for secure remote reconfiguration of an FPGA-based embedded system by means of a central reconfiguration unit (CRU).

This solution consists of three different parts, as can be identified in Figure 1. The first part is the underlying communication protocol that ensures mutual authentication of client and server and data integrity and confidentiality. The second component represents the software implementation of the CRU. Finally, the last component consists of a synthesizable VHDL core that can be integrated into any existing application’s VHDL design. This core is developed with a focus on compactness and simplicity for integration. Especially this last property implies that during the design of the application, less attention must be given to reconfiguration since this capability can at release time be added to the application’s design.

![Fig. 1. Structural model of FPGA and CRU in the STRES project.](image)

Since the VHDL code is fundamentally hardware independent (given the condition that enough reconfigurable space is available in the device) [3], only one hardware feature is required on the client side, being a communication port to the CRU. Although technically any interface connectable to the reconfigurable device can be used, the wide availability of the Internet inspired the limitation of the STRES core communication...
capabilities to TCP/IP only. This eliminates the need of implementing rarely used interfaces, thus allowing for a smaller overall design, and potentially a cheaper end product since logically smaller (and cheaper) reconfigurable devices can be used. The downside of this approach however is the necessity to implement the complex TCP/IP communication stack, yet this issue falls outside the scope of this paper and will not be covered here. The reader is referred to [2] for more details concerning the hardware implementation at the embedded system side.

On server side (CRU), a software application is required to manage bit streams for different reconfigurable platforms, to keep track of the different clients in the field, and to organize communication with clients to transfer bit streams when necessary. By convention, in the STRES layout, clients always initiate bit stream updating sequences as regular firewall setups in secure environments normally allow outgoing connections to be made (from client to CRU) but block incoming connections before they reach the secure network level where STRES enabled secure applications are typically operated. Essentially, the CRU will be listening for incoming connections, and reconfiguration procedure will be initiated as soon as an incoming client connection is received.

The paper is organized as follows: Section 2 explains the STRES protocol between server and client, while Section 3 discusses the server architecture. In Section 4 implementation details are provided for which Section 5 presents a proof of concept. Finally, Section 6 concludes the paper.

2 STRES PROTOCOL

The communication between client and CRU as designed for STRES is based on a well established cryptographic protocol, enabling mutual authentication of client and server, and ensuring confidentiality and integrity of the transferred data. Firstly the cryptographic protocol will be introduced, followed by a discussion of various practical implementation aspects primarily aimed at improving the communication security.

2.1 Cryptographic Protocol

In order to enlarge the credibility of the system, a standardized cryptographic protocol was chosen for exchanging data. The Station-to-Station (STS) protocol, based on the classic Diffie-Hellman protocol, fulfills all the necessary security requirements. This protocol defines a shared secret session key $K$. After the key agreement, $K$ can be used for further communication based on symmetric key encryption.

The STS protocol applied in STRES is based on elliptic curve operations. Elliptic curve cryptography (ECC) relies on the ability to compute a point multiplication and the inability to compute the multiplicand using the original and product points [4], creating a one-way function. Although the existence of one-way functions is still open for debate and as of 2011 no mathematical proof of security has been published for elliptic curve cryptography, it is regarded as safe for protection of information with a minimum key length of 384-bit keys [5]. For STRES, this bit length is extended to 512 bits [2]. Note that the choice for ECC was mainly driven by the fact that a compact implementation at hardware side was needed. ECC requires smaller key sizes, less storage, less power, less memory, and often less bandwidth than other public key systems for an equivalent amount of security. These properties make ECC well suited for application in embedded systems.

![Fig. 2. STRES cryptographic protocol for authentication and key exchange.](image-url)
reference to Figure 2 for a schematic overview of the protocol.

Both the CRU and the client’s embedded system start with a generator \( P \) and a key pair consisting of a private and public key respectively, \((a, A)\) for the CRU and \((b, B)\) for the client. When the client initiates the key agreement protocol, the server reacts by randomly choosing a number \( k_1 \) and executing the elliptic curve point multiplication \( Q_1 = k_1 \cdot P \). The resulting point \( Q_1 \) has two coordinates \((Q_1x, Q_1y)\) each of length 256 bit. Likewise, the client also chooses a random number \( k_2 \) and multiplies it with the same generator resulting in \( Q_2 = k_2 \cdot P \). Using \( k_2 \), the client can calculate \( k_2 \cdot Q_1 \), resulting in the same value \( k_1 \cdot Q_2 \) which will be later calculated by the CRU. This value, \( K \), is the result of the STS protocol, and can now be used as session specific symmetric key. \( K \) is a point on the elliptic curve with a coordinate pair \((Kx, Ky)\), each having a length of 256 bits. Therefore, it must first be reduced by means of a hash function before it can be used as symmetric key. The hash function halves the key length from 512 to 256 bits. This string is then split into two parts. The 128 most significant bits \( K' \) are used as symmetric key for encryption. The 128 least significant bits are used as secret key for the message authentication code (MAC) to ensure the integrity of the message.

However, to establish a legitimate connection from the CRU’s point of view, \( Q_2 \) is combined with \( Q_1 \) and signed using the client’s private key \( b \). Signature generation is denoted by \( \varsigma \) in Figure 2, while signature verification is symbolized by \( \Sigma \). The signature is then encrypted \((\epsilon)\) with the symmetric key \( K' \) along with the client’s public key \( B \) and the unique FPGA ID. In the next step, \( K \) (and thus \( K' \)) is calculated on the CRU side, and used to decrypt the message sent by the client. The signature can then be verified using \( B \), and the same process repeats when the CRU echoes an encrypted signature of \( Q_1 \) and \( Q_2 \) back to the client along with its public key \( A \). In the last step the client decrypts and verifies this signature after which both have agreed upon a session key for symmetric key cryptography and message authentication and have verified each other’s origin.

The Advanced Encryption Standard (AES) was chosen as symmetric key algorithm because it is the standard cryptographic encryption algorithm offering 128 bit security [6]. As hash algorithm, the Secure Hash Algorithm (SHA)-256 function was chosen which is one of the few unbroken and well established hash functions with 128-bit security. Since most elliptic curve operations are readily available from the key agreement scheme, ECDSA was chosen as signature algorithm [7].

### 2.2 Practical security

The communication link itself is theoretically secure due to the difficulty of elliptic curve logic trapdoor operations. However, this concept is only effective on the condition that clients are able to access the CRU. If a client is unable to retrieve an updated bit stream from the CRU to reconfigure itself with, an attacker could be given the time to complete a malicious operation because the faulty bit stream cannot be patched. This implies that measures must be taken to prevent the CRU from going offline, getting overloaded or inaccessible to the client by any other means [9]. The most significant threat uncovered by the cryptographic protocol is a Denial Of Service (DoS) attack, which attempts to consume all available server or client resources rendering them unavailable to the legitimate users. No proven countermeasure exists other than the installation of backup servers, high capacity links on ISP level etc. Fortunately, the STRES mutual authorization system in combination with the socket interface provided by the TCP/IP communication protocol, can help making DoS attacks a harder task. Note that other protocols can be found in literature typically designed to offer increased resistance to DoS attacks [10]. However, we have chosen to use a well established security protocol.

In order to make DoS attacks even more difficult, the communication flow was implemented based on an easy mechanism called the port forwarding mechanism. In a classic server setup, any client might be allowed to connect to any of the \( 2^{16} \) logical UDP ports of the system. As long as neither of both actively terminates the connection, it stays open and thus keeps the assigned server port in use. In the hypothetical case that \( 2^{16} \) malicious clients connect to that server and keep the connections open indefinitely, any incoming connection request from a genuine client will be refused, resulting in a DoS scenario. To prevent this critical security hazard, the STRES communication protocol allows clients to connect only to a single fixed server port which can be chosen at random in the \( 2^{10} \) to \( 2^{16} − 1 \) range. This not only eliminates the risk of all available server ports to be taken hostage almost uncontrollably, but is also from a resource point of view a more profitable setup since listening to every port would require \( 2^{16} \) active sockets, which
is especially for a client population size $\ll 2^{16}$ a waste of resources.

Since probing the server for its active communication port would result in a poor performance, the chosen port should preferably be coded into every client as well because no drop in security is created by the exposure of this information. Depending on the connection speed, this implies a possible maximum connection lifetime per client exceeding incoming request time outs. Rather than extending the time out configuration on TCP/IP level, the communication protocol attempts to structurally solve this problem by forwarding incoming clients to other ports as soon as possible. When a client connection is received on the fixed port $x$, the CRU reacts by opening a socket on a randomly chosen free port and returning this port number $y$ to the client. Next, the CRU terminates this connection on port $x$. To ensure a maximum availability of the CRU, a new socket is opened on port $x$ which starts listening for other incoming connections and the client is given the opportunity to connect to the issued port $y$. In this setup at any time only $n + 1 < 2^{10}$ sockets are active simultaneously, with $n$ being the number of connected (or connecting) clients. After reconnecting to port $y$, the cryptographic protocol is initiated.

Despite the more efficient resource management of the communication protocol thanks to the port forwarding mechanism, it is still possible to perform an effective DoS attack on this configuration. One just needs to connect to port $y$ and follow the protocol definition steps until authentication and then stalling its execution. To decrease the effectiveness of this approach, a watchdog timer is implemented to limit the allowed time span to successful authentication with a countdown magnitude being function of the network response (ping).

### 3 Server Architecture

#### 3.1 Namespace overview

To avoid excessive entanglement of the application with the STRES CRU core itself, the top level is split into two categories of namespaces. The application namespace contains all code unique to a single specific application. The STRES core namespaces contain routines for cryptography, database management, and server operations. They are called SecurityAndSafety, Database, and Postoffice namespace respectively (see Fig. 1). This organization prevents editing of the STRES core code for modal applications, thus directly avoiding the potential insertion of security leaks. Secondly, a transparent implementation of the STRES core also allows for quicker and cheaper integration of the STRES system into existing or new applications.

#### 3.2 Cryptographic Libraries

A second measure to increase the STRES core reliability is the choice for reuse of as much predefined code as possible, under the form of cryptographic libraries with proven integrity. Virtually all currently available platforms implement cryptographic base functionality to some degree. However, for the implementation of client - server applications, C# and Java stand out to be particularly strong in this field. C# is backed up by the .NET Framework, an extensive library containing a multitude of cryptographic algorithms. Java supports JCE (Java Cryptographic Extension) and JCA (Java Cryptographic Architecture) as a source of basic cryptographic functionality [14]. Unluckily neither are sufficient to implement all the cryptographic functionalities as defined in the STRES protocol, hence an additional library is required.

Fortunately, the Bouncy Castle cryptographic library, which implements all non licensed cryptographic algorithms known to date (excluding the IDEA algorithm), is available for both the C# and Java platforms [13]. Moreover, it is a complete open source reference library. An important difference between the two platforms is that Bouncy Castle for Java behaves as a black box in code, while for C# the source code is much easier tweakable due to its better availability. It must also be noted that a larger online knowledge base for the use of the Bouncy Castle library with Java is available than there is for C#. This situation can mainly be explained because relatively a higher fraction of applications implement connections between platforms which both run a Java platform, rather than communicating between a Java platform and low level hardware as is the case for the STRES layout.

After careful comparison of both platforms against a list of all features required for the implementation of the STRES CRU, a few flaws in Java were found that turned out to be prohibitive for using Java for the development of the STRES CRU core. The most important problem is the lack of unsigned integers in Java [15], which makes TCP/IP communication with clients significantly more complex. Note that this problem can
be solved by means of the JBits library. However since the development of JBits is discontinued and recent high end FPGAs are unsupported (e.g. up to Xilinx Virtex 2), no further attempts were made to integrate JBits in the FPGA communication code.

Consequently, it was decided to implement both CRU core, database routines and front end using C#. To make a comparison with respect to speed and code complexity still possible, multiple CRU classes were implemented in both languages, and in each of these situations the superiority of C# over Java was confirmed. Section 5 provides details with respect to this comparison.

4 IMPLEMENTATION

As previously mentioned, the STRES core is an independent unit separated from any application code, acting like a black box and providing a secure shell around the different building blocks composing it. It consists of a namespace bundling all cryptography and safety routines called SecurityAndSafety namespace, another one managing the database connections called Database namespace, and a third one responsible for network communications Postoffice namespace. Since many implementation aspects of the CRU are rather trivial for experienced software architects, only the most important implementation aspects are discussed.

4.1 SecurityAndSafety namespace

The SecurityAndSafety namespace has a double function. It groups on the one hand classes together implementing the STRES specific cryptographic and security code as well as their auxiliary routines. On the other hand, it also provides an entry point for externally called cryptographic routines from the Bouncy Castle library. Although these external cryptographic calls imply a security threat if the library is altered, this is considered negligible since full access to the server file system would be required. However, it is possible to fully integrate the Bouncy Castle routines necessary for STRES in the STRES core since the complete C# source of Bouncy Castle is publicly available [13]. The Safety part of the namespace’s name reflects the presence of lower importance routines such as CRC-checking of bit files and verification of bit file platform identifiers to prevent the transmission of incompatible bit files to clients.

4.1.1 Protocol implementation

A first important class in the SecurityAndSafety namespace is the STRESProtocol class which implements the IHandshakeProtocol interface specifying the sequence of different steps required to authenticate a client and agree on a session key. In other words, STRESProtocol implements the cryptographic protocol as described in Section IIa. An implementation of the IHandshakeProtocol interface can be seen as a state machine calling cryptographic operations in a protocol-specific order. Requests are relayed to the Cryptographer class which acts as a proxy for the four main algorithms used by the cryptographic protocol: Diffie Hellman Elliptic Curve key agreement (DHEC), Elliptic Curve DSA (ECDSA), AES-128 and SHA-256. Security settings for encryption are part of the Cryptographer configuration stack, and passed down to the underlying algorithm implementations. The native availability of a SHA implementation in the .NET Framework for instance allows for a selection of either the SHA digest algorithm in .NET’s Security namespace, and the digest algorithm of Bouncy Castle. Furthermore, both Bouncy Castle as C# provide a (pseudo)random number generator specifically developed for cryptography with a period of $2^{19937} - 1$ (19937 being a Mersenne prime), sufficiently secure for application in the STRES cryptographic protocol. This information can subsequently be used for the construction of an elliptic point multiplication algorithm, the previously discussed one-way algorithm on which the security of DHEC is based.

Entries for both .NET and Bouncy Castle implementations have been made accessible from the Cryptographer and selectable where possible, primarily executed in parallel as a means of fast verification of end results (if for the same input both outputs match it is accepted, else it is rejected and logged). This parallel processing and comparison makes tampering with cryptographic routines significantly more challenging for attackers. The Cryptographer handles this parallelization transparently and only returns a value to the calling routine, the protocol state machine in most cases, when both values match.

Since every cryptographic protocol implements the IHandshakeProtocol interface and almost any cryptographic algorithm is directly available in either the .NET Framework, Bouncy Castle or both, new protocols can easily be constructed. Actions as simple as rearranging the steps of existing inherited protocols guarantee to slow down attackers, while the CRU’s cryptographic strength can continuously be updated.
throughout the life time of connected products by inserting the newest cryptographic algorithms into the handshake mechanism.

4.1.2 Key and data handling
Data blocks and keys are 128 and 256 bits in length respectively, while the largest unsigned integer in C# only has a length of 64 bits. Fortunately, the Biginteger structure extends the base type 64 bit integer and allows for the creation of arbitrary sized integers. The BigInteger structure provides basic arithmetic operations on the integer value as a whole, while also allowing bit level operations such as XOR of two BigIntegers of equal length, bit inversions etc. Both the .NET Framework as the Bouncy Castle library provide an implementation of the BigInteger structure with comparable functionality. While the value of BigIntegers can be set from a multitude of base types including a string with known radix (allowing hexadecimal user input, for example), a byte vector is preferable in the STRES application context since bytes are directly ready for transmission to the client. Likewise, BigIntegers representing keys can be constructed directly from incoming byte data. BigIntegers play a double role in the STRES core: they represent AES data blocks and hold elliptic curve point coordinates.

4.1.3 Elliptic Curve implementation
As with many trapdoor functions in key agreement schemes, elliptic curve operations require a set of parameters needed for initialization [5]. For elliptic curve cryptography this includes, but is not limited to, the definition of the curve itself which takes the shape of an equation of the form \( y^2 = x^3 + ax^2 + b \), parameterized by the variables \( a \) and \( b \). Secondly, a field prime is needed, and care must be taken when choosing it since the security of later elliptic curve operations will mainly depend on this parameter. Hence a large number of primes have been constructed historically, many of which are described in NIST publications [8] and recommended for use with elliptic curve cryptography as published in appearing in X9.62 and FIPS PUB-186-2. The curve selected for the STRES cryptographic protocol is P-256 with prime \( p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \).

Fortunately, Bouncy Castle features a list of NIST recommended curves in the NistNamedCurves class, providing the GetName function to retrieve all necessary parameters from P-256 as well as many others and thus eliminating the need of hard coding prime, \( a \) and \( b \) parameters directly in the Cryptographer class.

The computation of the elliptic point multiplications is implemented using the shift-and-add algorithm, which loops through the bits of \( a \) when multiplying \( a \) with \( b \) and adds \( b \) to a nullified multiplicand when a given bit in \( a \) is 1. After each cycle, regardless of the adding step, the multiplicand is shifted to the left for one bit, doubling its value. \( a \), \( b \) and the multiplicand are all elliptic points, thus requiring more advanced arithmetic operations than can be used for regular integers. The Bouncy Castle library provides the ECPoint class however, from which the instances represent points on elliptic curves. ECPoint implements an Add function which can be used to add another ECPoint to a given ECPoint object, and a Twice function for doubling the value of a point using the predefined constant points ECPoint.Zero, ECPoint.One and ECPoint.Two. For the exact implementation of the Add and Twice functions, the reader is directed to the published Bouncy Castle source.

While it is possible to retrieve the \( x \) and \( y \) coordinates from ECPoint objects to avoid redundant serialization of the objects for transmission, the inverse operation is defined. For example, the ECPoint class does not have a public constructor taking \( x \) and \( y \) coordinates as arguments. The most important reason for this complication is the inability to construct an ECPoint object capable of performing the arithmetic operations described above solely from the \( x \) and \( y \) coordinates. Information about the curve on which the point is situated on is also required. As a logical consequence, arbitrary points built from coordinates are created by a point generator integrated in the ECCurve class which provides a portal for prime and both \( a \) and \( b \) parameters. Its function CreatePoint is essential for converting received point coordinates \( (Q_2) \) from the client to an ECPoint object which can be used for calculations.

4.2 PostOffice namespace
Since any server application must be capable of making different connections simultaneously, from which none are allowed to stall the server, intuitively a multithreaded code structure is necessary. Therefore in the STRES PostOffice namespace, the ClientLink thread is responsible for managing the connection with a single client FPGA bidirectionally. At any given time, as many ClientLink threads are running...
as there are FPGA’s connected to the server. Every active ClientLink thread consumes a single logical port on the host, which is used for both transmitting data (more specifically bit streams) to the client FPGA and receiving commands from it. Routines in a ClientLink instance are encapsulated within the running thread, and only accessible in the context of the specific associated client FPGA. Depending on the state of the ClientLink as defined in the LinkState enumeration, additional non shared code can be retrieved as object instances. For example the handshake routine is not needed before a connection with the client is established, and the Authorizing state is reached in the state diagram. The isolation of state specific routines contributes to saving server resources as obsolete object instances can be disposed of without severing the ClientLink itself (e.g. when the state diagram reaches the state Authenticated, the handshake routine is no longer needed and can be safely removed from the working set.

Fig. 3. Software layout of the Postoffice namespace.

For a factory pattern to operate efficiently, a supervising monitor is mandatory. This purpose serves the LinkMaster class, which continuously listens to a fixed port for incoming connections and negotiates a port number to which the client can be delegated. When an incoming connection is received, the primary task of the LinkMaster is to execute its factory pattern constraints to spawn a new ClientLink instance by calling its constructor. In the ClientLink constructor, a port number is requested from the system’s network, and returned to the LinkMaster using a PortMonitor object. This mutex locks the port exclusively for the associated ClientLink object, and as a dependent resource the LinkMaster is stalled until a free port number can be obtained. The LinkMaster can then retrieve the reserved port number from the PortMonitor and launch the ClientLink thread before sending the port number to the client FPGA. Although creating a small overhead, this setup ensures that the server will be listening to the reserved port before the FPGA can attempt to establish a connection there, thus preventing a possible deadlock situation. Immediately upon passing the port mutex for a specific port number to its corresponding ClientLink thread, an independent watchdog timer is started by the LinkMaster to control irregular ClientLink behavior. This timer must be actively reset by the ClientLink by progressing through its state diagram, else a time out will be generated, causing the LinkMaster to terminate the ClientLink process. The most critical goal of the watchdog timer is to reduce the likeliness of DoS attacks, by releasing resources on server side that are illegitimately occupied. Without the watchdog timer, a rogue client could for example continuously establish connections, thus reserving port numbers and spawning ClientLink threads, but stalling them by not initiating the STRES authentication procedure. In that case however, the watchdog will expire at LinkMaster level regardless whether or not the ClientLink is stalled, and trigger the necessary events to terminate the thread and recover its resources. Since communication between ClientLink and LinkMaster happens asynchronously, multiple events are more efficient than passing in a set of conditions using a single notification event. The events exposed by the LinkMaster are ConnectionAccepted, ConnectionTerminated and ConnectionAuthenticated. The interval until time out before a ConnectionTerminatedEvent will be triggered unless a ConnectionAccepted or ConnectionAuthenticated event is fired by the ClientLink thread can be configured as a server setting and is vital for server performance optimization. The time out period will be proportional to the available server resources, but will also be a function of the expected network delays between client and server. Figure 3 displays the layout of the Postoffice namespace, as explained above.

4.3 Database namespace
As a data driven application, a repository is needed for both descriptive information about the client (such as its location, IP address, platform, current bit file

Fig. 3. Software layout of the Postoffice namespace.
version etc.) and the bit files themselves. Although from the point of view of a relational database design, a bit file is also a property of the entity Client. Therefore, it was chosen to exclude the bit files from management by the Database Management System (DBMS) (PostgreSQL was chosen for STRES). Because a bit file contains all the data needed to systematically reconfigure the client FPGA regardless of its occupancy ratio (e.g. how much logic is actually used by the design), its size will consequently outnumber the other combined client properties by a factor typically being $10^4$ - $10^6$. Instead, saving the bit files directly in the file system requires no overhead due to the nature of the information as a file. This allows a reduction of the size of the database with at least a factor $10^4$. Since bit file synthesizers natively deposit output in the host file system, it also circumvents the obsolete step of inserting or updating existing bit files in the database.

The primary key uniquely identifying the clients in the database is the FPGA ID. This ID number is retrieved in the mutual authentication handshake, during the initial connection phase of the client. It is then used by the CRU to retrieve all client’s properties as well as the absolute file path to the most recent bit file for that particular client. Next, the bit file is loaded from the disk as soon as the FPGA ID is known to the CRU to conserve time. It is unloaded by the LinkMaster when the ClientLink requests resource recovery by firing the ConnectionTerminated event.

5 Proof of Concept

To verify the operation of the STRES reconfiguration system, a test case requiring most of the STRES core elements was chosen in function of the quantity of reconfigurable logic available on an average 2010 high end FPGA system (Xilinx Virtex 5, Spartan 6 or equivalent FPGA classes). The selected user application includes intensive image processing on client side (thus implying the necessity of dedicated FPGA hardware to accelerate this operation) and secure transmission of data extracted from the images to the CRU.

The application was demonstrated on January 27, 2011 as part of the concluding lectures of the STRES project. The reconfiguration capabilities of the STRES system were proven by the live real time reconfiguration of the image processing unit, adding background subtracting functionality to the design. All tests were performed on a Xilinx SP605 FPGA test platform. Comparison of the independent computation results on the same input data of the STRES core and the Magma computer algebra system [16][17], which was used extensively to verify the correct operations in the Cryptographer class in early stages of STRES development, reveal a perfect match for all cryptographic levels.

In order to give an idea about the efficiency and complexity of the code for both C# and JAVA, two important metrics for Java with respect to C# are compared: speed and lines of code (LOC).

5.1 Speed

The speed is computed for 250 independent runs of several cryptographic algorithms that were both implemented in JAVA and in C#. A first observation is the difference in execution speed of C# and Java. The average speed turned out to be 25 % higher for ECDH key agreement benchmarks and around 20 % for ECDSA signature generation and verification. Interestingly, a wide range of measurements can be observed when running Java benchmarks, even on identical systems (see Fig. 4). C# results are more tightly concentrated around a pole value whereas Java results are spread out. This phenomenon exists for both elliptic curve operations, and currently no explanation can be given for it other than the inaccurate timing functionality of the Java platform for small time spans. It is essential to note here that both Java and C# compile into managed code, and hence cannot execute real time operations. The speed advantage of C# seems to persist when benchmarking on other hardware, though again wide variations are possible (Figure 4 shows benchmarking results on an Intel P7450 @ 2.13 GHz). From comparison of our measurements with speed benchmark results in literature, it can be concluded that the speed difference between Java and

![Fig. 4. Comparison of elliptic curve benchmarking in C# and Java on Intel P7450 2.13 GHz CPU.](image-url)
C# in this particular setup is attributed entirely to the additional delays induced by Java by creating key and parameter objects from data bytes while this step is not needed in C#.

Secondly, both C# and Java are able to encrypt and decrypt data blocks faster than calculating elliptic curve instructions, respectively 0.6 ms on average for C# and 0.9 ms on average for Java, resulting in a 50% performance difference. However, it must be noted that also in this test values around 0.7 ms have been observed for Java, which feeds the assumptions that the real performance difference is rather negligible and the wide variations are indeed caused by inaccurate time measurements. All data was measured when running a sequential encryption and decryption pass on a chunk of 1024 bits data, equal to the chunk size used to transfer bit files in STRES (10,000 passes were measured at once), using cipher-block chaining (CBC) mode with IV = 0. It is possible that testing of larger data chunks might uncover completely different trends, but these tests were not performed since they do not contribute to the performance analysis of the STRES cryptographic framework. Although the AES symmetric key algorithm was implemented in both C# and Java with and without help of Bouncy Castle and despite observation of minor speed differences, these were found to be statistically insignificant and thus will not further be discussed. It is assumed that Bouncy Castle, JCE/JCA and .NET Framework all use a very similar implementation of a highly optimized AES algorithm because of its wide usage. It is however important to mention here that the algorithm access in Java, e.g. the application of Policy Files, does cause a small delay on first request. Since this delay is subsequently spread over 250 sequential runs, it passes unnoticed in the execution of benchmarking tests.

5.2 LOC

Finally, the complicity of code can be measured using the LOC parameter, although this is obviously highly dependent on the coding style of the software developer. However since Java and C# code have both been developed by the same author in this case, a reasonable foundation for comparison can be legitimized. A function for AES encryption in Java requires for example around 50 LOC using JCA/JCE while the same can be done in C# using the .NET Framework in less than 10 LOC. Most other routines are much harder to compare due to Java’s black box nature when it comes to cryptography, as well as the addition of code in C# to import and export keys as byte vectors (the latter being very hard to implement in Java). Generally, it can be concluded that a routine with similar functionality implemented in Java will require more lines of code compared to C# because of initialization (the GetInstance call to retrieve an algorithm from a security provider, for example) and other overhead such as the registration of security providers and policy files themselves.

6 CONCLUSION

A server architecture implemented in C# was demonstrated as a functioning solution against interception of reconfiguration bit streams for embedded systems. Elliptic curve key agreement and DSA schemes, AES and SHA-256 cryptographic routines from the Bouncy Castle library as well as the native .NET Framework Security namespace have proven to be successful to implement a secure connection between server and FPGA client to exchange both data and reconfiguration bit streams.

ACKNOWLEDGMENT

The STRES project has been supported financially by the IWT - Flemish Agency for Innovation by Science and Technology under Tetra.

REFERENCES


