36th European Modeling & Simulation Symposium, 005
21st International Multidisciplinary Modeling & Simulation Multiconference

2724-0029 © 2024 The Authors.
doi: 10.46354/i3m.2024.emss.005

Comparison of ECDSA Signature Verification Implementations on Bare-Metal Embedded Systems

Philipp Holzer[1,*], Franz Leopold Wiesinger[2] and Michael Bogner[2]

[1]Photeon Technologies GmbH, Hintere Achmuehlerstrasse 1, Dornbirn, 6850, Austria
[2]University of Applied Sciences Upper Austria - Department of Embedded Systems Engineering, Softwarepark 11, Hagenberg, 4232, Austria

[*]Corresponding author. Email address: philipp.holzer@photeon.com

Abstract

The elliptic curve digital signature algorithm (ECDSA) is a cryptographic scheme used to generate digital signatures and to verify them. In the course of this research, two software libraries got implemented that perform an ECDSA signature verification. Those two implementations of the ECDSA signature verification are discussed and compared regarding their performance. Both implementations target a single core RISC-V CPU in a minimal simulated test environment. The first implementation is done purely in software, while the second implementation is done using a coprocessor to accelerate execution. To access this coprocessor, the RISC-V GNU Toolchain got extended with custom instructions during this research. This is done by reason of the ECDSA and its requirement for especially large numbers (e.g. 283 bit integers). Handling those numbers in software requires a relatively high amount of execution time, especially on single core systems with low clock frequency. For those systems, the coprocessor library is very well suited for most scenarios. If the systems clock frequency is respectively high, then the pure software implementation might fit one's requirements as well without the need for additional hardware. Furthermore, if the number of signature verifications is very low (e.g. just once at application startup), then the coprocessor would require chip area that is mostly unused during runtime.

Keywords: Cryptographics, ECDSA, RISC-V, Coprocessor, Performance Evaluation

1. Introduction

In today's manufacturing facilities, as many production steps as possible are automated in order to minimize down-time of machines and maximize the output. This kind of automation requires a lot of communication between different kinds of machines or production robots, as well as monitoring and controlling of the whole production process, and maintaining the machines. Despite all the advantages that Industry 4.0 brings, it also comes with the risks of cyber security breaches. Every interface to the system forms a potential attack vector that may be exploited by an attacker. For example, a maintenance interface of a machine that communicates over some wireless protocol (e.g. Bluetooth or WiFi) with a mobile device, like a tablet or smart phone, could form an attack vector. Normally, the maintenance interface is used to track some diagnostic data, configure some parameters for controlling the production process, or maybe even to perform a software update. However, if no proper security mechanisms are implemented, an attacker that gained access to the facility, e.g. by joining a guided tour, could use a mobile device to establish a connection to a production machine and access the maintenance interface. The attacker would then be able to obtain some production data or even bring that machine to halt, by changing some configuration parameters, or upload a new software image, and thereby disrupt the production process.

A solution for the described scenario would be to restrict access for any unknown device to the maintenance interface. This could be done by using public key cryptography to authenticate the mobile device, before granting access to the the maintenance interface. The "Elliptic Curve Digital Signature Algorithm" (in short ECDSA) would be an option to implement this restriction. In the described scenario, the mobile device would be asked to sign a random message, and send the signature to the production machine. The production machine is able to verify the signature. If the signature is valid, then access to the maintenance interface is granted, otherwise further communication with the mobile device is refused by the production machine. Therefore, only known mobile devices that are dedicated to controlling and maintaining the production machines, are able to access the maintenance interface.

This article's focus is set on the implementation of the signature verification according to the ECDSA scheme for embedded systems. The ECDSA signature verification algorithm got implemented two times. The first implementation is done purely in software, the second implementation is using a coprocessor, to accelerate the execution of the ECDSA signature verification. The reason for those two implementations, is the complexity of the ECDSA signature verification algorithm. Since the ECDSA verification requires arithmetic operations with relatively large integers (up to 283 bits, details follow in Chapter 2), the pure software implementation is expected to need relatively much execution time on small embedded devices with low clock frequency. On bigger platforms, like for example systems running Embedded Linux or industrial PCs, the coprocessor might not be needed. Both implementations are discussed and compared in this article. The coprocessor has been implemented and integrated in a 32-bit RISC-V CPU by (Jahn, 2023).

In this article the necessary basics to understand the ECDSA as well as the aim of this research are introduced in the section "State of the art". Afterwards the "Materials and Methods" section follows, which handles the details of the target platform, the implementation of the two ECDSA software libraries and how the performance measurements have been recorded. Those performance measruements are then discussed in the next section called "Results and Discussion". At the end the main findings are summarized and a prospect for future work is given in "Conclusions".

2. State of the art

In 1976, public key cryptography has been introduced by Whitfield Diffie and Martin Hellman. The idea, behind public key cryptosystems, is that each participating party generates a pair of keys. Those keys are called "public key" or "enciphering key" and "private key" or "deciphering key". As the names already suggest, the private key must not be known to any other party, while the public key can be accessed by anyone. All participating parties use the same set of functions to transform plain text into cipher text (encrypt or encipher) and the other way around (decrypt or decipher) (Diffie and Hellman, 1976).

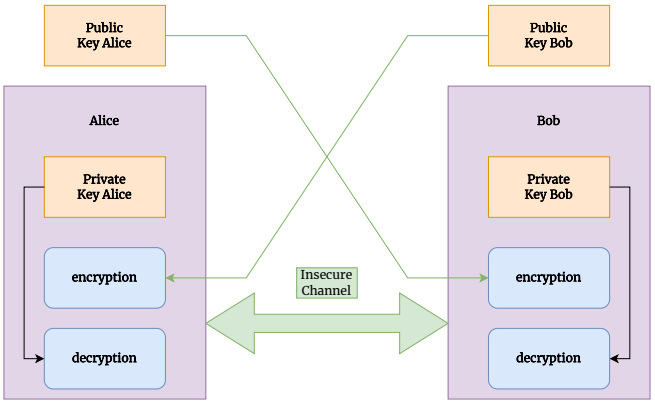

Figure 1. Secure communication using public key cryptography (Mühlberghuber, 2011).

Figure 1 shows how a communication with a public key cryptosystems works. In this particular case there are two communicating parties called Alice and Bob. If Alice wants to send a message to Bob over an insecure channel, then Alice uses Bob's public key to encrypt her message and sends it to Bob. Bob is then able to decrypt that message, since he knows the corresponding private key. Any other eavesdropper on the insecure channel can not decrypt the encrypted message Alice sent to Bob (Mühlberghuber, 2011).

2.1. Digital signatures

The described scenario used public key cryptosystems to exchange data over an insecure channel. However, public key cryptosystems can also be used to generate digital signatures. According to (Johnson et al., 2001), a digital signature is a number that depends on some secret known only to the signer, and on contents of a message that is getting signed. This secret is the private key of an entity in the public key crypto system (Johnson et al., 2001). For example, Alice wants to be sure that she is communicating with Bob. In this case Alice would send a random message m to Bob and ask Bob to sign it. Bob then calculates the signature that is depending on m and his private key, using his encryption function. The result is Bobs signature of the message m. The signature is sent back to Alice, who then is using Bob's public key to "decrypt" the received signature. If the result of the decryption is then equal to the original message m, Alice can be sure that the received signature was sent by Bob, since only Bob knows the corresponding private key, to his public key (Diffie and Hellman, 1976).

2.2. Elliptic curve digital signature algorithm

The elliptic curve digital signature algorithm (in short ECDSA) is a cryptographic scheme that uses public key cryptography in order to generate and verify digital sig-

natures. The ECDSA is a variant of the so-called "Digital Signature Algorithm" (in short DSA). Both schemes can be used for digital signature generation and verification, however it is way harder to calculate a private key from a public key for an ECDSA key pair than a DSA key pair. Therefore, ECDSA key pairs can achieve the same cryptographic security as DSA key pairs with a smaller bit length. This results in smaller elliptic curve parameters, faster computations and less memory usage compared to the DSA (Johnson et al., 2001).

2.2.1. Finite fields

Finite fields or Galois fields form the base of every ECDSA implementation. Each elliptic curve that is used to implement an elliptic curve cryptography (in short ECC), has an underlying finite field. This means that the coordinates of points on those elliptic curves are elements of the underlying finite field. Finite fields consist of a finite set of elements. Furthermore, every field element is invertible, and two operations are defined on finite fields, the addition and the multiplication of field elements. The result of those operations is always another element of the field. Let $\mathbb{F}_q$ be a finite field with $q = p^m$, where p is prime number and $m \in \mathbb{N}$. In this case p is called the characteristic of $\mathbb{F}_q$ and m is called the "extension degree" of $\mathbb{F}_q$ (Johnson et al., 2001).

There are two large groups of finite fields commonly used in modern cryptography, the so-called "prime fields" $\mathbb{F}_q$ or $\mathcal{GF}(q)$ where the order of the field is an odd prime ($q = p$) and the "binary fields" $\mathbb{F}_{2^m}$ or $\mathcal{GF}(2^m)$, where the order is a power of two ($q = 2^m$) (Johnson et al., 2001). According to (Wenger and Hutter, 2012), hardware ECC implementations based on binary fields have better runtime and energy performance, than those based on prime fields.

For this article the most important difference between binary and prime fields, is the representation of field elements. In prime fields the elements simply are natural numbers from zero to $(q-1)$. In binary fields there are multiple ways to represent field elements. For this article the "polynomial basis representation" is of interest. In this representation, the field elements represent polynomials of the form

$$f(x) = \sum_{i=0}^{m-1} c_i x^i \tag{1}$$

where $c_0, c_1, ..., c_{m-1} \in \mathbb{F}_2$. Since the coefficients of the polynomial can only be zero or one, a polynomial can be represented as bit string of the width m. The position of the bit in the bit string represents the order of the corresponding x (Johnson et al., 2001).

2.2.2. Elliptic curve parameters

For the ECDSA to work, some specific domain parameters have to be defined. Those parameters depend on the concrete elliptic curve that has been chosen for the implementation of the ECDSA application (Johnson et al., 2001). The elliptic curve used for this ECDSA implementation, is the so-called "Curve B-283" that got defined by (Chen et al., 2023). Curve B-283 is defined by the equation

$$y^2 + xy = x^3 + ax^2 + b \tag{2}$$

and the domain parameters described in Table 1. The underlying finite field is a binary field.

Table 1. Parameters and their meaning of the elliptic curve B-283 according to (Chen et al., 2023).

Parameter name	Description
m	Bit length of field elements
a	Elliptic curve equation coefficient
b	Elliptic curve equation coefficient
$G(x, y)$	Base point
n	Order of the base point

The bit length m of the field elements for Curve B-283 equals 283. The coefficient a of the curve equation equals one and the coefficient b is a non-zero element of $\mathcal{GF}(2^m)$ (Chen et al., 2023). The base point $G(x, y)$ is required to define the relationship between a public and private key. The relation between the public key Q, which is a point on the elliptic curve, and the private key k, which is a natural number is

$$Q = G * k \tag{3}$$

where the $*$-operator represents a multiplication of a point on the elliptic curve with a natural number (Johnson et al., 2001). According to (Liu et al., 2021) the security of the ECC relies on the elliptic curve discrete logarithm problem (in short ECDLP). This means it is easy to calculate the public key Q if one knows the base point G and the private key k, but it is very hard to calculate the private key k if one knows the public key Q and the base point G. As soon as one would be able to calculate the private key k, an attacker could fake digital signatures using that private key.

2.2.3. ECDSA signature generation

The procedure to generate a digital signature according to the ECDSA scheme is shown in Algorithm 1. First a hash (e.g. SHA-256) has to be calculated over the random message m. If the result of the hash function has a greater bit length, than the finite field elements, then only the i-leftmost bits are needed for the rest of the algorithm. Next a random integer has to be chosen between one and $n - 1$, where n is the order of the base point. Then the private key k is multiplied with the base point G. Finally the integers r and s that form the digital signature are calculated (Johnson et al., 2001).

2.2.4. ECDSA signature verification

The procedure to verify a digital signature according to the ECDSA scheme is shown in Algorithm 2. Again a hash

Algorithm 1 ECDSA signature generation (Johnson et al., 2001).

Require: Message m
Ensure: Digital signature (r, s) of m
$e = HASH(m)$
Define l_i as the i-leftmost bits of e.
Choose $t \in [1, n-1]$ at random.
$(x_1, y_2) = k * G$
$r = x_1 \bmod n$
$s = t^{-1}(l_i + rk)$
return (r, s)

Algorithm 2 ECDSA signature verification (Johnson et al., 2001).

Require: Digital Signature (r, s), Message m
Ensure: *valid*
$e = HASH(m)$
Define l_i as the i-leftmost bits of e.
$u_1 = l_i s^{-1}$
$u_2 = rs^{-1}$
$(x_1, y_1) = u_1 * G + u_2 * Q$
$r_{verify} = x_1 \bmod n$
return $r_{verify} == r$

has to be calculated over the message m. It is crucial that the hash function is the same function that has been used to create the signature. Next the integers u_1 and u_2 get calculated, which depend on the input signature. The base point gets multiplied with u_1 and the public key Q gets multiplied with u_2. The results of those multiplications are then added to form the point (x_1, y_1), which is again a point on the elliptic curve. Only if $x_1 \bmod n$, where n is the order of the base point is equal to r, then the signature is valid (Johnson et al., 2001).

2.3. Aim of this research

An ECDSA signature generation has been implemented in hardware by (Pittner, 2022). The implemented elliptic curve cryptography used for this implementation relies on the Curve B-283 defined by (Chen et al., 2023). Furthermore, a coprocessor that is used to accelerate the ECDSA signature verification has been implemented and integrated in a RISC-V CPU by (Jahn, 2023).

During this research, two software libraries have been implemented. Both libraries provide an implementation of the ECDSA signature verification. The first implementation is done purely in software, while the second implementation utilizes the previously mentioned coprocessor. Since the coprocessor is directly integrated into the RISC-V CPU and not connected like a peripheral (for further details see (Jahn, 2023)), custom assembler instructions are necessary to access the coprocessor. Therefor, the RISC-V GNU Toolchain got extended with custom instructions that provide the functionality to access the coprocessor in the course of this research.

Of course, the coprocessor implementation is expected to be faster than the pure software implementation. However, the coprocessor requires additional chip area (for details see (Jahn, 2023)). To spend this additional chip area for the coprocessor might not be necessary, if the ECDSA signature verification in software is done in a reasonable amount of time. On small single core embedded systems with bare-metal applications, the execution speed heavily depends on the systems clock frequency. The aim of this research is to determine, in which scenarios the use of the coprocessor or respectively the pure software library for the ECDSA signature verification is more beneficial on those small single core systems.

3. Materials and Methods

Both implementations of the ECDSA signature verification have been run on the same simulated target platform. The execution of the performance measurements was done in a cycle accurate simulation. For performance measuring the needed cycles for the whole ECDSA signature verification have been recorded.

3.1. Target platform

The target RISC-V CPU is the so-called "Ibex RISC-V Core" that got developed by (lowRISC, 2023). The Ibex core has a number of optional hardware features that improve performance. All of them have been deactivated, except for the hardware support for multiplications and divisions, and the instruction prefetch buffer. Additionally the coprocessor for the ECDSA signature verification has been integrated into the Ibex core by (Jahn, 2023). This modified Ibex core is integrated in a small simulation environment (the so-called "Ibex Simple System"), consisting of a memory for instructions and data, a peripheral for writing ASCII output into a log file and a timer peripheral. For details about the Ibex Simple System see (lowRISC, 2022).

3.2. Pure software library

The pure software library consists of three layers. The first layer implements the finite field arithmetic for the field elements of the elliptic curve's binary field. This means this layer provides the necessary arithmetic operations for 283 bit wide bitstrings that represent polynomials. Furthermore all required arithmetic operations for the regular 283 bit wide integers (e.g. r, s, u_1, u_2, see Algorithm 2) are implemented in this layer as well. The second layer is on top of the finite field arithmetic layer and provides two functions, one to multiply a scalar value with a point on the elliptic curve (e.g. $u_1 * G$) and a function to add two points on the elliptic curve (e.g. $u_1 * G + u_2 * Q$, see Algorithm 2). The third layer implements the whole ECDSA signature verification using the other layers.

Another approach would have been to use open-source

libraries like "Mbed TLS" or "libecc" for comparison instead of implementing an own library. However, this approach was not chosen, since those libraries only support elliptic curves with an underlying prime field. This means the low level arithmetic in the software would have been different, compared to the one implemented in the coprocessor, since the field elements of the prime fields are not represented as polynomials.

3.3. Coprocessor library

The memory map of the coprocessor is shown in Table 3. The coprocessor library only has to initialize the coprocessor and retrieve the result from it. The whole ECDSA signature verification is implemented in the coprocessor, except for the hash calculation. The hash calculation is done in parallel by the coprocessor library to save chip area. According to (Jahn, 2023), the coprocessor performs the multiplication $u_2 * Q$ before the multiplication $u_1 * G$ and the calculation of u_1 (see Algorithm 2). The result of the hash calculation is only needed to calculate u_1, which means while the coprocessor is calculating $u_2 * Q$, the CPU has time to calculate the hash value and write it into the coprocessor. Therefore, chip area is saved, without affecting the overall execution time, given the condition that the hash value has been written into the coprocessor, before the multiplication $u_2 * Q$ is finished. The coprocessor library implements the following procedure:

1. Write the digital signature (r and s) and the public key Q into the coprocessor.
2. Calculate the hash value for the message m.
3. Write the hash value into the coprocessor and set the "hash value sync" bit in the coprocessor's control register.
4. Poll the "busy bit" of the coprocessor's status register.
5. Read the "signature valid" bit, once the "busy bit" is cleared.

To access the coprocessor integrated in the Ibex core, the RISC-V GNU Toolchain got extended with custom instructions. Those instructions are all based on the RISC-V I-type instruction format (see Figure 2). The bits 31 down to 20 form a twelve bit immediate, the bits 19 down to 15 are used to encode a CPU source register and the bits 14 down to twelve encode the type of operation (in this case load word, move data from CPU register to coprocessor register and the other way around). Furthermore, the bits

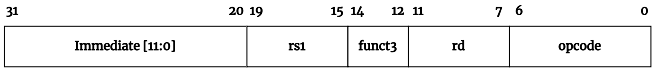

Figure 2. RISC-V I-type instruction format (Waterman et al., 2019).

Table 2. Custom instructions for accessing the coprocessor.

Name	31:28	27:22	21:20	19:15	14:12	11:7	6:2	1:0
lw.ecdsa	0000	rd_ecdsa	00	rs1	000	00000	00010	11
wcr.ecdsa	0000	101100	00	rs1	001	00000	00010	11
rsr.ecdsa	0000	101101	00	00000	010	rd	00010	11

Table 3. Coprocessor memory map according to (Jahn, 2023).

Address offset	Description	Bitfields		Access
0x00-0x20	r value of signature	0-281	value	write
		282-287	reserved	
0x24-0x44	s value of signature	0-281	value	write
		282-287	reserved	
0x48-0x8C	Public key Q	0-282	x-coordinate	write
		283-287	reserved	
		288-570	y-coordinate	
		571-575	reserved	
0x90-0xAC	Calculated hash e	0-255	value	write
0xB0	Control register	0	start verification	write
		1	reset unit	
		2	hash value sync	
		3-31	reserved	
0xB4	Status register	0	busy	read
		1	signature valid	
		2	error invalid input	
		3-31	reserved	

eleven down to seven encode a CPU source register and the bits six down to zero represent the instructions opcode (Waterman et al., 2019). Table 2 shows the three added custom instructions. The immediate (bits 31 down to 20) is always used to reference a coprocessor register. CPU source registers are referenced with the "rs1" operand and CPU destination registers are referenced with the "rd" operand. The "lw.ecdsa" instruction is used to load a four byte word from the main memory directly into a coprocessor register. The register referenced by the operand "rs1" contains the base address of the to be transferred data. The operand "rd_ecdsa" is a register index. It forms the register address together with the rest of the immediate. The register index of a desired register, is the address according to the memory map divided by four. To determine the memory address of the four byte word that should be loaded into the coprocessor, the address offset of the target coprocessor register is added to the base address stored in the register referenced in "rs1". The instruction "wcr.ecdsa" is used to access the control register of the coprocessor. It writes the content stored in the CPU register referenced by "rs1" into the coprocessor's control register. The "rsr.ecdsa" instruction reads the content of the coprocessor's status register and stores it in the CPU register referenced by "rd".

3.4. System simulation and testing

To verify the correct functionality of the two software libraries, as well as the correct implementation and integration of the coprocessor and the modification of the RISC-V GNU Toolchain, an automated system test has been set up in cooperation with (Jahn, 2023). The following description of the system test is visualized in Figure 3. At the beginning of the system test, random ECDSA signatures get generated by a script, with the help of a golden model implementation in Python of the whole ECDSA scheme (signature generation and verification). Those random generated signatures, get passed into a header file for the test software. The test software containing one of the two implemented ECDSA signature verification libraries, de-

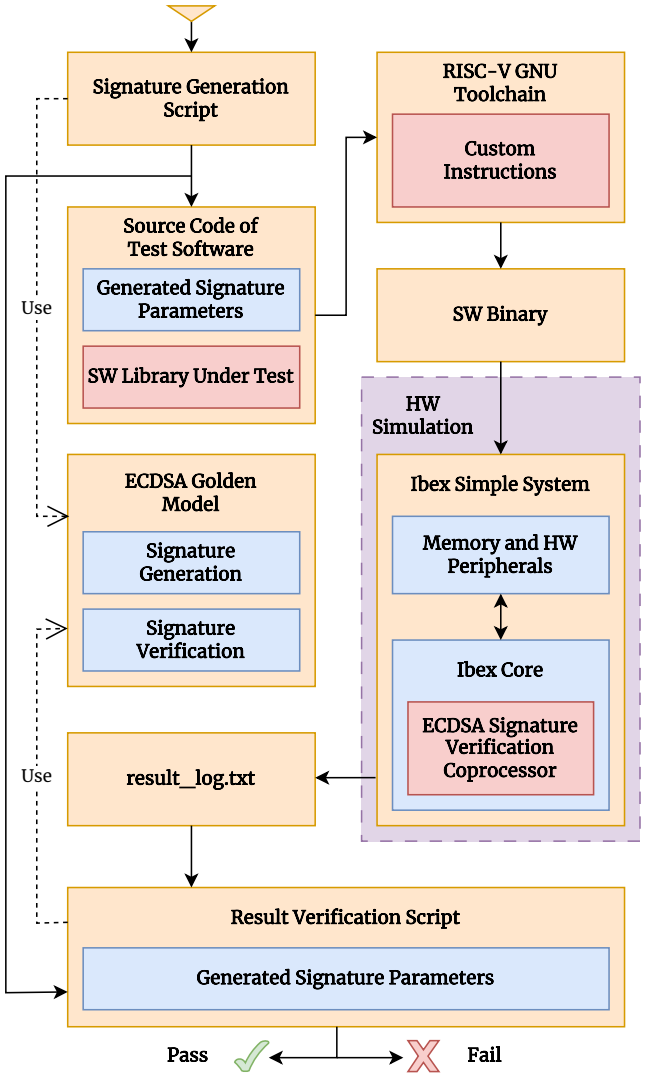

Figure 3. Flow chart describing the procedure of a system test.

pending on which library should be tested, gets built using the modified RISC-V GNU Toolchain. The software binary, gets then loaded as memory content at the start of a cycle accurate hardware simulation of the Ibex Simple System. The Ibex Core in that particular Ibex Simple System contains the coprocessor. During the simulation of the Ibex Simple System, executing the test software, a log file gets generated that contains the results of all performed ECDSA signature verifications. After the simulation is finished a script gets executed that verifies the results produced by the simulation. This is done by taking the signature parameters that got generated for the test software and use the golden model of the ECDSA signature verification to verify those signatures again. If all results that are produced by the golden model match the results produced by the simulation, then the system test passes.

Table 4. Averaged numbers of required clock cycles to perform an ECDSA signature verification.

Implementation	Clock cycles	Time[s] (50 MHz Clock)
Pure software	798,634,348	15.97
Coprocessor	515,374	0.01031

3.5. Performance measurements

In order to record the performance measurements the same principle as described in Section 3.4 or respectively shown in Figure 3 has been used in general. The only difference is that for the performance measurements the signature generation script, generated ten random and valid signatures that got reused for all measurements instead of generating new random signatures on each run. Also the test software that is running during the simulation on the Ibex Core, slightly differs from the test software that got used during the system test. Those modifications were necessary to record the measurement results. However, those changes do not affect any of the ECDSA signature verification libraries. Table 4 shows the needed clock cycles for the corresponding ECDSA signature verification implementation. The usage of the coprocessor brings a speedup of approximately 1550. The clock cycle values are averaged values. Each implementation verified ten random and valid signatures, to ensure the whole ECDSA procedure is executed, and no error handling, or parameter validation would terminate the verification at an earlier time.

Additionally, the needed clock cycles to calculate the hash value, have been measured as well, it takes 7068 clock cycles. According to (Jahn, 2023), the following steps are executed by the coprocessor after setting the "start verification" bit in the control register, before the coprocessor needs the hash value:

1. Calculation of the inverse of s. Needs 899 clock cycles.
2. Multiplication of s^{-1} with r. Needs 494 clock cycles.
3. Multiplication of u_2 $(r \cdot s^{-1})$ with Q. Needs 248781 clock cycles.

This results in a total of 250147 clock cycles that pass, before the coprocessor needs the hash value. Since the concrete hash calculation (SHA-256) in software needs only 7068 cycles, the CPU has enough time to write the hash value into the coprocessor and set the "hash value sync" bit, without blocking the coprocessor.

4. Results and Discussion

A speedup of the given size (1550) was expected since a software implementation got compared with an almost pure hardware implementation. Given those implementations would run on a single core embedded system, without any kind of operating system, and a clock frequency of 50 MHz, the pure software implementation of the ECDSA signature verification would take 15.97 seconds (see Table 4). In general this is too much execution time, for a

Table 5. Execution times in seconds for various clock frequencies.

Clock Frequency [MHz]	Pure software [s]	Coprocessor [s]
50	15.97	0.01031
100	7.99	0.00515
200	3.99	0.00258
300	2.66	0.00172
400	2.0	0.00129
500	1.6	0.00103

signature verification, therefore the pure software implementation is not suitable for applications that run on small systems with low clock frequency. However, if the clock frequency would be increased to e.g. 500 MHz, then the execution time reduces time 1.6 seconds (see Table 5). This may already be an acceptable time to complete a device authentication. For example, if the ECDSA signature verification only needs to be executed at the systems startup, 1.6 seconds could be a reasonable execution time. However, in a scenario where a lot of signature verifications have to be executed, 1.6 seconds are still a lot of time, since the CPU is completely occupied during the verification. As soon as for example a real time operating system is used for the application, the ECDSA verification will be further interrupted by context switches and other tasks, which makes the situation even worse.

The coprocessor on the other hand performs very well over all clock frequencies, compared to the pure software library. The overall ECDSA signature verification time of 10.31 milliseconds with a clock frequency of 50 MHz has good chances to fit the needs of most applications regarding signature verification. Furthermore, the coprocessor is expected to perform also well, if a real time operating system is used. This is because most parts of the ECDSA signature verification are done in parallel to the CPU. Once the hash value is written into the coprocessor, it would be possible to execute a different task on the CPU to reduce the polling time of the coprocessor's busy bit. The major drawback of the coprocessor, is the required additional chip area. The worst scenario for the coprocessor, is when the systems clock frequency is quite high and only one signature has to be verified at startup like already described above. Then the additional chip area is spent, but most of the time not used. Furthermore, depending on the concrete clock frequency a pure software implementation might be fast enough as well.

5. Conclusions

In general, for small single core embedded systems, the coprocessor library is better suited than the pure software library. However, if the system's clock frequency is respectively high, and the number of signature verifications is very small, then the pure software library might be better suited for the ECDSA signature verification, since the coprocessor would need chip area that is mostly unused during the application's runtime. The bigger the systems on the other hand (e.g. systems capable of running Embedded Linux), the more suitable the pure software implementation becomes, since the execution time of the pure software library will decrease.

5.1. Prospect

For the future one could invest some time in additional tuning of the pure software library, to increase the pure software library's performance in contrast to the documented implementation. Additionally, the pure software library could be ported to a dual-core platform and be refactored to use parallelization, and compare the performance again against the coprocessor on the single core system. This would increase the performance of the pure firmware library, and furthermore the whole application could utilize the second core. In the best case scenario the second core would need just as much or less chip area, than the coprocessor.

References

Chen, L., Moody, D., Regenscheid, A., Robinson, A., and Randall, K. (2023). Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters. Technical report, National Institute of Standards and Technology, Gaithersburg, MD.

Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6).

Jahn, S. (2023). Extension of a risc-v architecture with custom instructions to perform an elliptic curve digital signature algorithm verification.

Johnson, D., Menezes, A., and Vanstone, S. (2001). The elliptic curve digital signature algorithm (ecdsa).

Liu, S.-G., Chen, W.-Q., and Liu, J.-L. (2021). An efficient double parameter elliptic curve digital signature algorithm for blockchain. *IEEE Access*, 9:77058–77066.

lowRISC (2022). Ibex simple system. URL: https://github.com/lowRISC/ibex/tree/master/examples/simple_system, Accessed: 04-05-2023.

lowRISC (2023). Ibex risc-v core. URL: https://github.com/lowRISC/ibex, Accessed: 20-03-2024.

Mühlberghuber, M. (2011). Comparing ecdsa hardware implementations based on binary and prime fields.

Pittner, D. (2022). Design and hardware implementation of an elliptic curve cryptography.

Waterman, A., Asanovic, K., and Foundation, R.-V. (2019). The risc-v instruction set manual, volume i: User-level isa, document version 20191213. https://riscv.org/technical/specifications/, Accessed: 02-03-2023.

Wenger, E. and Hutter, M. (2012). Exploring the design space of prime field vs. binary field ecc-hardware implementations. In Laud, P., editor, *Information Security Technology for Applications*, pages 256–271, Berlin, Heidelberg. Springer Berlin Heidelberg.