UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIA DA COMPUTAÇÃO

Lucas Mayr de Athayde

Implementing a library to provide Winternitz signatures with lightweight primitive using the Rust Programming Language

Florianópolis
2021

Lucas Mayr de Athayde

Implementing a library to provide Winternitz signatures with lightweight primitive using the Rust Programming Language

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciência da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciência da Computação.
Orientador: Prof. Ricardo Felipe Custódio, Dr.
Coorientador: Lucas Pandolfo Perin, Me.

Florianópolis
2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Athayde, Lucas
Implementing a library to provide Winternitz signatures with lightweight primitive using the Rust Programming Language / Lucas Athayde ; orientador, Ricardo Felipe Custódio, coorientador, Lucas Pandolfo Perin, 2021.
67 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2021.

Inclui referências.

1. Ciências da Computação. 2. Assinaturas baseadas em resumo criptográfico. 3. Esquema de assinatura única Winternitz. 4. Criptografia pós-quântica. 5. Criptografia de baixo custo. I. Felipe Custódio, Ricardo. II. Pandolfo Perin, Lucas. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

Lucas Mayr de Athayde

Implementing a library to provide Winternitz signatures with lightweight primitive using the Rust Programming Language

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo curso de Graduação em Ciência da Computação.

Florianópolis, 05 de Maio de 2021.

Prof. Alexandre Gonçalves Silva, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Ricardo Felipe Custódio, Dr.
Orientador
Universidade Federal de Santa Catarina

Lucas Pandolfo Perin, Me.
Coorientador
Universidade Federal de Santa Catarina

Prof. Martin Augusto Gagliotti Vigil, Dr.
Avaliador
Universidade Federal de Santa Catarina

Gustavo Zambonin, Me.
Avaliador
University of Ottawa

This thesis is dedicated to my Mother.

ACKNOWLEDGEMENTS

I would like to thank my advisors, Dr. Prof. Ricardo Felipe Custódio and Lucas Perin for their advice in this work. My colleagues from LabSEC, for listening and discussing topics related to this subject at length. Lastly, but certainly not least, I would like to express my most sincere thanks to my family, for without their teachings and unending support I would not have reached this far.

"Once you’ve got a task to do, it’s better to do it than live with the fear of it."
(ABERCROMBIE, 2006)

RESUMO

Criptografia pós-quântica se refere a algoritmos considerados seguros mesmo quando atacados usando técnicas desenvolvidas para computadores quânticos. A maioria dos algoritmos clássicos baseiam sua segurança na dificuldade de problemas matemáticos, a fatoração de números inteiros grandes e logarítmo discreto. Shor (1999) mostra um método efetivo de ataque a estes problemas usando um computador quântico que, teoricamente, quebraria a segurança de qualquer algoritmo dependente destes problemas. A segurança dos esquemas de assinaturas digital baseadas em resumo criptográfico não são quebradas pelos métodos introduzidos por Shor, dependendo apenas da segurança do algoritmo usado no resumo criptográfico. Merkle (1989) introduz o esquema de assinatura única Winternitz (WOTS, *Winternitz one-time signature scheme*), um esquema de assinatura baseado em funções de resumo criptográfico onde o resumo sucessivo de blocos de mensagem é computado como uma forma de assinar dados. A família de resumo criptográfico usada pode ser trocada sem a perda de segurança desde que elas sigam algumas propriedades de segurança. Bernstein *et al.* (2017) introduz GIMLI em (BERNSTEIN et al., 2017) como uma alternativa de baixo consumo para a geração de funções de resumo usando construções esponja. GIMLI é desenvolvido com o intuito de prover portabilidade e eficiência para dispositivos embarcados. Neste trabalho, o esquema de assinatura Winternitz com função de resumo criptográfico subjacente GIMLI é implementado na linguagem Rust, uma linguagem portátil com ferramentas de suporte a desenvolvimento de aplicações embarcadas.

Palavras-chave: Assinaturas baseadas em resumo criptográfico. Esquema de assinatura única Winternitz. Criptografia pós-quântica. Rust. Criptografia de baixo custo.

RESUMO EXPANDIDO

Introdução

Esquemas de assinatura digital clássicos como RSA baseiam sua segurança em problemas matemáticos difíceis como logaritmos discretos e fatoração de números inteiros grandes. Acredita-se que estes problemas sejam da classe de dificuldade NP-Completo e computacionalmente inviáveis de serem resolvidos em computadores clássicos. Uma das maneiras usadas para se quebrar a segurança de um esquema é encontrar um método de resolver tais problemas em tempo polinomial.

Shor apresenta algoritmos que resolvem estes problemas em tempo polinomial através da utilização de computadores quânticos (SHOR, 1999), efetivamente quebrando a segurança destes algoritmos clássicos assim que um computador quântico avançado o suficiente seja construído. Este esforço cria uma corrida contra o tempo para encontrar uma alternativa a estes métodos criptográficos, mudando o foco da criptografia para um mundo resistente a computadores quânticos, ou "pós-quântico".

Assinaturas baseadas em resumos criptográficos foram propostas como uma alternativa aos modelos clássicos. A segurança de tais esquemas de resumo se baseia na resistência de funções de resumo convencionais, funções estas que são amplamente usadas na criptografia clássica, contornando os atrasos e gastos de um método completamente novo de certificação digital (MERKLE, 1989).

Funções de resumo criptográfico são, por definição, fáceis de serem computadas, mas computacionalmente inviáveis de serem revertidas. Para uma função F de resumo, tal que $y = F(x)$, tem-se que com o conhecimento de F e x, y é trivial, mas caso um agente tenha conhecimento apenas de F e y, descobrir o x original deve ser impraticável.

A primeira forma de assinatura deste tipo, o esquema de assinatura única Lamport-Diffie (LD-OTS, *Lamport-Diffie one-time signature scheme*), foi apresentado por Lamport *et. al* como uma maneira de assinar documentos usando apenas funções de resumo criptográfico (LAMPORT, 1979). A chave privada x deste esquema consiste em dois grupos de 256 bits cada. A chave pública y é o resumo correspondente de todos os 512 bits da chave privada. Assina-se uma mensagem através deste esquema revelando partes da chave secreta de um ou outro conjunto dependendo do bit original da mensagem.

O esquema de assinatura WOTS foi introduzido em (MERKLE, 1989) como uma alternativa ao esquema proposto por Lamport. Em WOTS a aplicação das funções de resumo são executadas múltiplas vezes. Assim como no esquema LD-OTS, temos uma chave privada x, uma função de resumo F e chave pública y, mas é introduzido um novo parâmetro público w. O parâmetro w controla o número de aplicações da função de resumo para cada bloco a ser assinado.

Assim como Merkle trata as funções de resumo criptográfico como o alicerce de tais esquemas de assinatura digital, deve-se mencionar alguns principais tipos de funções de resumo criptográfico. SHA-2 é uma família de algoritmos de resumo clássico definida pelo NIST com uma construção feita sobre blocos de cifra. SHA-3 é um subconjunto da família de funções conhecidas como Keccak, onde a maior diferença em comparação com SHA-2 é o método de construção, sendo construído através de funções esponja, resultando em uma maior proteção contra ataques de extensão de largura. Ressalta-se que SHA-3 não é uma melhoria direta de SHA-2, são duas famílias de algoritmos com características e usos distintos. Uma nova construção baseada em funções esponja é a função de resumo Gimli, participante da terceira fase da competição organizada pela NIST sobre criptografia de baixo custo. Gimli promete uma construção altamente portátil entre arquiteturas diferentes e de baixo custo.

Sistemas embarcados estão cada vez mais presentes no dia a dia e não aparentam parar de

crescer com o avanço relacionado à internet das coisas. Muitas das aplicações embarcadas em tais sistemas se comunicam com o mundo fora do seu sistema, criando uma necessidade de autenticação e verificação de integridade. Assinaturas baseadas em resumo são uma alternativa para suprir esta necessidade em um mundo pós-quântico, principalmente se a geração de chaves é delegada para um servidor fora do sistema embarcado. MANIFAVAS et al. analisa e compara várias técnicas de assinatura digital, clássicas e pós-quantum. Nota-se que o esquema baseado em resumo é menor em tamanho de código e mais rápido que o algoritmo clássico RSA. Os autores especulam que sistemas baseados em resumo só têm a ganhar com a evolução de funções de resumo com designs voltados ao baixo custo computacional MANIFAVAS et al..

Códigos de referência e implementações embarcadas com relação a criptografia são geralmente escritos nas linguagens de programação C e código de máquina (*assembly*) para melhorar a eficiência dos algoritmos. Entretanto, estas são duas linguagens de baixo nível que podem ser uma barreira para novos programadores interessados em desenvolvimento embarcado. A evolução dos sistemas embarcados e o aumento do interesse nesses sistemas com o advento da internet das coisas proporcionou alternativas para a programação em embarcados com linguagens de mais alto nível. Tais linguagens se tornaram atrativas e viáveis, inclusive, alguns códigos de referência modernos já incluem implementações em linguagens de alto nível (BERNSTEIN et al., 2017).

Rust é uma linguagem de alto nível que se contém uma robusta ferramenta de análise estática, produto da decisão de design onde variáveis não podem ser compartilhadas (*ownership*), resultando em referências e ponteiros que não podem ser duplicados. Onde outras linguagens encontram problemas com acesso a memória inválida, ponteiros nulos e afins, Rust as proíbe de sequer compilar, a não ser que sejam usadas em blocos declarados como não seguros. Rust também oferece uma maneira clara de incorporar códigos em linguagem C e código de máquina através destes blocos inseguros, o que resulta em uma linguagem atrativa para desenvolvimento embarcado.

Neste trabalho implementa-se o esquema de assinaturas baseado em funções de resumo WOTS e uma função de resumo criptográfico Gimli em puro Rust tentando evitar ao máximo a utilização de blocos inseguros e mantendo compatibilidade com o maior número de arquiteturas embarcadas possível. Para os propósitos de testes será utilizado um emulador para o processador Cortex-M4 e devido às restrições do emulador não será levado em considerão medidas de consumo de energia nem da eficiência do programa.

Objetivos

Pretende-se implementar uma biblioteca portátil compatível com Cortex-M4 contendo o esquema de assinatura baseado em funções de resumo WOTS, incorporando a mesma com a função de resumo de baixo custo Gimli usando a linguagem de programação Rust.

Objetivos Específicos

- Analisar implementações de referência WOTS e Gimli.
- Implementar o esquema WOTS e Gimli sem a utilização da caixa `std`.
- Implementar o esquema WOTS e Gimli utilizando blocos de código seguro.
- Oferecer o suporte para a troca de funções de resumo.
- Testar as implementações quanto a corretude usando códigos de referência.

Metodologia

Para alcançar os objetivos propostos, primeiramente fez-se o levantamento das principais contribuições quanto às assinaturas digitais baseadas em funções de resumo e suas implementações. Em seguida foi feito uma pesquisa sobre as implementações da função de resumo Gimli.
Para a implementação dos algoritmos WOTS e Gimli, fez-se um estudo sobre o fluxo utilizado em aplicações similares e sobre o armazenamento de informações necessárias para os esquemas. Por fim, foram realizados testes usando implementações de referência para confirmar a corretude dos códigos.

Estrutura

Este trabalho é dividido nas seguintes seções:
Na seção 1, introduz-se o trabalho, apresentando o problema a ser tratado, as fontes que originaram as questões e dão base para a execução do mesmo. São introduzidos elementos fundamentais para o entendimento do trabalho como funções de resumo, uma breve evolução temporal de esquemas de função baseadas em resumo criptográfico e os problemas atuais.
Nas seções 2 e 3 expande-se sobre a teoria necessária para o entendimento do trabalho. A seção 2 foca nas primitivas criptográficas relevantes como assinaturas digitais, funções de resumo criptográfico e detalha mais a fundo a construção destas funções e as particularidades do algoritmo Gimli. A seção 3 detalha o funcionamento de assinaturas baseadas em resumo, seus problemas e soluções propostas.
Na seção 4, discute-se as dificuldades e restrições na implementação bibliotecas voltadas para dispositivos embarcados e a abertura de novas opções com o avanço da tecnologia destes sistemas na seção 4. Além da argumentação geral sobre embarcados, é também discutida a escolha da linguagem Rust e suas propriedades úteis voltadas para sistemas seguros e sistemas embarcados.
Por fim, A seção 5 apresenta o desenvolvimento da biblioteca desenvolvida EWOTS, seus resultados, uma breve comparação entre algoritmos implementados e o pseudocódigo de referência.
A seção 6 apresenta as considerações finais e trabalhos futuros.

Resultados e Discussão

Desenvolveu-se uma biblioteca em Rust com implementações seguras sem blocos inseguros dos algoritmos WOTS e Gimli. A ausência de blocos *unsafe* provê a garantia de um código seguro com seguranças providas diretamente da linguagem utilizada, a não existência de problemas de memória decorrentes de ponteiros nulos e condições de corrida. Restringe-se também a utilização da caixa *core* no desenvolvimento do projeto, descartando a caixa *std* do Rust e provendo uma implementação que pode ser utilizada em ambientes embarcados que oferecem suporte a Rust. A adição de recursos (*features*) para a seleção de algoritmos não só melhora o desempenho geral como abre a possibilidade de implementações alternativas para partes do código de forma simples.
Foram realizadas algumas decisões durante o desenvolvimento do projeto que merecem ser destacadas. Em especial sobre o armazenamento das chaves privadas na parte WOTS e o armazenamento do estado atual da permutação Gimli. Quanto ao armazenamento das chaves privadas, a alternativa é manter a *seed* usada na geração das chaves e gerar cada chave privada sob demanda. Esta alternativa leva a um requerimento de memória menor durante a execução do programa mas introduz uma complexidade maior ao código. A segunda decisão em relação ao Gimli é parecida à primeira, ao optar-se por manter o estado Gimli em uma estrutura, aumentamos o custo de memória relacionado à execução do código, mas evitamos erros com relação ao passar a estrutura de método a método.

Considerações finais

Mesmo com foco do trabalho estando voltado na criação de uma implementação segura dos algoritmos WOTS e Gimli usando a análise sintática da linguagem Rust, existem várias implementações em *assembly* de passos importantes seguros tal que a inclusão de blocos inseguros contendo estas implementações não degradaria muito a segurança da biblioteca. A inclusão destes trechos de código aumentaria a eficiência do código. O custo desta inclusão seria pago em garantias de segurança e portabilidade, já que é necessária a inclusão de blocos diferentes para arquiteturas diferentes.

A utilização de um emulador para a execução dos testes implica em algumas limitações quanto a medição de eficiência da biblioteca, em especial foram as limitações quanto a medição de consumo de energia, contagem de ciclos e consumo de memória. Outro aspecto que foi levemente explorado é o tamanho dos binários em Rust quando comparados com o tamanho de programas similares em C, a comparação não foi realizada pois levaria o foco do trabalho para a otimização de uma biblioteca em Rust.

Um outro aspecto que foi deixado de fora da análise do trabalho é o custo energético de implementações embarcadas, esta limitação se deve a falta de equipamentos para realizar tal medição. Tal medição, principalmente quando relacionada a implementação de baixo custo Gimli, seria um acréscimo considerável para trabalhos futuros.

Palavras-chave: Assinaturas baseadas em resumo criptográfico. Esquema de assinatura única Winternitz. Criptografia pós-quântica. Rust. Criptografia leve.

ABSTRACT

Post-quantum cryptography refers to cryptography algorithms that are considered secure even against quantum computer attacks. Most conventional algorithms depend on the hardness of integer factorization and discrete logarithm mathematical problems. Shor (1999) presents a method to solve both of these problems using a large enough quantum computer, effectively breaking the security those algorithms depend on. Hash-based signature schemes are shown to be secure against attacks using Shor's algorithms. These signatures relying on the security of the underlying cryptographic hash function, avoiding the delays and cost of a new certification effort (MERKLE, 1989). Merkle (1989) introduces WOTS as a hash-based signature scheme where the cryptographic function is applied to the message blocks repeatedly as a way to sign messages using minimal security assumptions, the underlying family of hash functions can be exchanged as long as they fulfill some security properties. Bernstein et al. (2017) introduces Gimli, a lightweight option for building hash functions through a sponge construction, aimed towards embedded devices and portability. In this paper we present EWOTS, a library containing the Winternitz signature scheme and the Gimli hash function through the Rust programming language.

Keywords: Hash-based signatures. Winternitz one time signature scheme. Post-quantum cryptography. Lightweight cryptography. Rust.

LIST OF FIGURES

Figure 1 – Merkle–Damgård construction 34
Figure 2 – Sponge construction 35
Figure 3 – Gimli State Representation. From (BERNSTEIN et al., 2017) 36
Figure 4 – Gimli Non-linear layer 36
Figure 5 – Gimli linear layer. 37

Figure 6 – Winternitz Signature Operation 42
Figure 7 – Merkle Tree Authentication 42
Figure 8 – Merkle Tree Authentication Pathing 43

Figure 9 – Rust ownership transfer between variables 47
Figure 10 – Rust ownership transfer to function 47
Figure 11 – Rust variable transfer through borrow 48

Figure 12 – Project dependencies 51
Figure 13 – Hash feature selection 52
Figure 14 – Block size constants 52
Figure 15 – EWOTS Features from Cargo.toml 52
Figure 16 – Gimli Structure 53
Figure 17 – Gimli Non-linear layer 53
Figure 18 – Gimli Linear layer 54
Figure 19 – Gimli constant addition 54
Figure 20 – Gimli Hash 55
Figure 21 – Gimli Block Absorb step 55
Figure 22 – Gimli Padding 56
Figure 23 – Gimli Squeeze 56
Figure 24 – Gimli hash output 57
Figure 25 – EWOTS Structures 58
Figure 26 – EWOTS Digest 58
Figure 27 – EWOTS Secret Key 59
Figure 28 – EWOTS Public Key 59
Figure 29 – EWOTS Sign 60
Figure 30 – EWOTS Verify 61
Figure 31 – EWOTS debug output 62

LIST OF ALGORITHMS

Algorithm 1 – GIMLI permutation. From (BERNSTEIN et al., 2017) 67

LIST OF SYMBOLS

$\xleftarrow{\$}$	Random sample
$\wedge$	Logical and
$\vee$	Logical or
$\neg$	Logical not
$(\ldots)$	Tuple
$w_1 \parallel w_2$	Concatenation of words
$\langle w \rangle$	Size of *w* in bits
W^{nxm}	*n* by *m* array of words
$<<$	Bit shift left
$<<<$	Bit rotation left

CONTENTS

1 INTRODUCTION 27
1.1 OBJECTIVES 29
1.1.1 Specific Objectives 29
1.2 METHODOLOGY 29
1.3 STRUCTURE 29
1.4 LIMITATIONS 30

2 PRIMITIVES 31
2.1 ASYMMETRIC CRYPTOGRAPHY 31
2.2 DIGITAL SIGNATURES 31
2.3 ONE-WAY FUNCTIONS 32
2.4 BLOCK CIPHERS 33
2.5 SPONGE FUNCTION 34
2.5.1 SHA3 35
2.5.2 Gimli 35

3 HASH-BASED SIGNATURE SCHEMES 39
3.1 LAMPORT-DIFFIE OTS 39
3.2 WINTERNITZ OTS 40
3.3 MERKLE SIGNATURE SCHEME 42

4 EMBEDDED DEVICES 45
4.1 RUST 46
4.1.1 The future 49
4.1.2 Embedded Rust 49

5 EWOTS 51
5.1 CONFIGURATION 51
5.1.1 Features and Constants 51
5.2 GIMLI IMPLEMENTATION 52
5.2.1 Gimli Permutation 53
5.2.2 Gimli Hash 53
5.2.3 Gimli Outputs 55
5.3 WOTS IMPLEMENTATION 56
5.3.1 Key Pair 58
5.3.2 Signing 59
5.3.3 Verify 59

6 CONCLUSION 63

6.1 FUTURE WORK 63

BIBLIOGRAPHY 65

APPENDIX A – GIMLI PERMUTATION ALGORITHM 67

APPENDIX B – EWOTS 69

APPENDIX C – SBC PAPER 85

1 INTRODUCTION

Classical signature schemes like RSA base their security on the hardness of its one-way function, which in turn depends on the hardness of the integer factorization and discrete logarithm problems. Those problems are believed to be NP-Complete and computationally impractical to solve on a classical computer. One way to break the security of these signature schemes is to find an algorithm that solves those problems in polynomial time.

Peter W. Shor shows algorithms that solve these problems on a quantum computer (SHOR, 1999), effectively creating a scenario where those signature schemes are broken. Believing that quantum computers are not a thought exercise but something that will come to be, his work creates a deadline where there needs to be a shift from classical signature schemes to *post-quantum* signature schemes.

Hash-based cryptography has been proposed as an alternative to classical signature schemes. The security of hash-based signatures depends on the safety of conventional cryptographic functions, avoiding the delays and costs of a new certification effort (MERKLE, 1989).

Hash functions are, by definition, easy to compute but hard to invert functions. If we have a hash function F such as that $y = F(x)$, then given knowledge of F and x we can compute y with ease, however, if we are given y and F it shouldn't be feasible to find x.

Lamport-Diffie one time signature scheme (*LD-OTS*) was first presented in (LAMPORT, 1979) as a way of using these functions to sign data. The private key x consists of two randomly generated sets of 256 bit-strings. The public key y is the hash of each of the bit-strings from the private key. Signing the message consists of revealing parts of your private key, when the corresponding message bit is a zero, reveal the corresponding key from one set, when it's a one, reveal from the other.

Winternitz one time signature scheme (*WOTS*) was introduced as an alternative to LD-OTS, where both signatures and public key generation involve multiple applications of the one-way function (MERKLE, 1989). In LD-OTS we have $y = F(x)$, public (F, y) and a private x. On WOTS we now have $y = F^w(x)$, (F, y) is still public and x is still private, but there is the introduction of the public w parameter, called Winternitz parameter, we apply the one-way function F a number w times and compute the public key. To sign a message using WOTS we compute $F^m(x)$ where m is the message to be signed. To verify a signature we can simply compute $y = F^{w-m}(x)$ and compare with the public key y. By signing blocks from the message instead of bit-by-bit we reduce the signature size.

Merkle also introduced the Merkle signature scheme (*MSS*), a signature scheme with a tree authentication algorithm that consists of a binary tree where every leaf node corresponds to a WOTS public key and the root node is the public key for every leaf (MERKLE, 1989). Meaning that a signer would only need to publish a single public key instead of multiple. However, this protocol is not without its drawbacks, the structure of this authentication tree requires that a signer generate all public keys for its leaves before obtaining its root node, which leads to an increased time in key generation and signature. The fact that we need to generate every public

key before publishing the root node means that a Merkle scheme has a finite amount of keys that can be used. One other consequence is that because of the nature of WOTS signatures, this scheme must be stateful in order to preserve its security.

As Merkle wrote in (MERKLE, 1989), cryptographic hash functions are the heart of hash based cryptography schemes, and for that reason we would be remiss not to mention a few of the most used hash functions. SHA-2 is a staple when it comes to hash functions, defined by NIST in 2001, being mandatory for any application that requires collision resistance. Keccak (BERTONI et al., 2013) is a family of hash functions based on sponge constructions, where SHA-3 is a subset of. The main difference between SHA-2 and SHA-3 is that unlike SHA2, which uses a block cipher, SHA-3 and the Keccak family uses the sponge structure, making SHA-3 resistant to length extension attacks (BERTONI et al., 2011). Another upcoming hash is based on Gimli (BERNSTEIN et al., 2017), a 384-bit permutation that can be used to create hash functions while maintaining *cross-platform performance*, something that we also strive to achieve in this work.

Embedded systems as a paradigm have not stopped growing in the last years and with enough computational capability, any real-world object could be equipped with a microchip to exchange information with its environment. Many of these applications use either the internet or a LAN to share messages with the outside world, creating a need for applications to authenticate and verify messages with each other. Hash-based signatures can fill that gap, especially when key generation and signing are done externally from the embedded device, leaving only the responsibility of verifying the messages for its embedded partner. Improvements by (PERIN et al., 2018) fit this case and could lend themselves as an argument for using hash-based signatures in embedded devices.

In (MANIFAVAS et al., 2014), the authors analyse and compare various asymmetric cryptography schemes for embedded devices, including classical RSA systems and alternative ones such as hash, lattices and code based. Special note should be given to MSS, found to be smaller in code size and faster than RSA. The authors also speculate that MSS based schemes would only gain more ground with the evolution of lightweight hash function designs.

Reference codes with regards to cryptography are usually written in both C and Assembly to increase efficiency in both space and time, however. Those are very low-level languages that can be somewhat of a barrier of entry to the world of cryptography implementations, even more so when dealing with IoT and embedded designs. However, recent reference codes have begun a shift to higher-end languages as seen in (BERNSTEIN et al., 2017), Rust in particular provides a good toolkit for embedded programming.

Rust as a language prides itself in its static analysis, byproduct of a design decision where variables cannot be shared, only borrowed or given to other functions. Rust also offers a clear way to incorporate existing C code into its own, resulting in a low-level enough language when necessary while maintaining the simplicity of high-level abstractions.

Rust separates itself from other languages by embedding implicit rules of programming in itself, where other languages have the programmer take care of not using invalid point-

ers. One of the most stand-out features of Rust is the concept of *borrowing* and *lifetimes*. There are no duplicate pointers in Rust, only one variable can have the true reference to an object.

In this paper we implement hash-based signature schemes in pure embedded Rust trying to use as many *safe* blocks as possible, sacrificing some efficiency for portability between devices. For the purpose of testing, we have chosen to emulate a Cortex-M4 processor. Emulating a processor comes with restrictions, therefore, we won't be measuring energy, memory and time efficiency.

1.1 OBJECTIVES

In this thesis, we aim to implement both WOTS and Gimli into a portable library compatible with embedded devices through Rust.

1.1.1 Specific Objectives

- Analyse WOTS and Gimli reference implementations.
- Implement WOTS scheme and Gimli without the `std crate`.
- Implement WOTS and Gimli using `safe` blocks.
- Simplify the change of the underlying hash function.
- Verify the correctness of our implementation against the original author's reference code.

1.2 METHODOLOGY

To reach the desired objectives we start by reviewing the main literature for hash-based signatures and Gimli, searching for both reference implementations and theoretical details. To implement WOTS and Gimli, a detailed study was made for similar algorithms implemented in Rust, paying close attention to the correct flow of instructions and which information was stored in memory. Lastly, tests were made with reference code to verify our implementation.

1.3 STRUCTURE

Section 1 introduces the problem to be studied, the main works related to our research, and how we intend to tackle it. Sections 2 and 3 expand on the theory necessary to comprehend this thesis. Section 2 gives a brief explanation of digital signatures, exemplifies one-way functions, and how to turn them into cryptographic one-way functions. Section 2 also goes through the fundamentals of how hash functions are built. Section 3 details the works of hash-based signatures, their problems, and proposed solutions. Section 4 elaborates on the difficulties of embedded programming and how advances on embedded devices allow for new options when

writing code. We also discuss the benefits of using Rust as an embedded programming language and the properties that make it so. Section 5 presents EWOTS, our implementation of WOTS and Gimli as a Rust library, a brief comparison of our algorithm with the reference documents, and some outputs. Section 6 elaborates on our conclusions and how this thesis could be expanded in future works.

1.4 LIMITATIONS

By using an emulator to validate our project we become limited by its functions. Unfortunately, measuring energy consumption, counting processor cycles and memory consumption. One other aspect that was briefly explained but not thoroughly explored is the comparison of binary sizes between Rust and C, as it would require shift the focus of this work towards binary size optimization.

2 PRIMITIVES

2.1 ASYMMETRIC CRYPTOGRAPHY

Cryptography is the practice and study of techniques of securing information in a way that an adversary cannot meddle with the information without leaving traces. One of those cryptosystems is called *Asymmetric Cryptography* or *Public-key cryptography*. Introduced in (DIFFIE; HELLMAN, 1976), the idea behind asymmetric cryptography is that if it is possible to have two distinct keys, one could be used to cipher documents while the other could be used to decipher messages, also called the public and private keys correspondingly. In this manner, the public key could be published to everyone, but only the holder of the secret key would be able to read the messages encrypted by the public key.

Public-key cryptosystems do not rely on a secure channel and remove the need for two parties to agree on the same private key, as is needed by symmetric cryptography, but introduces the need to authenticate published public keys. Signing is a slow operation for large messages, in those cases the signature is only done to the hash output of the message and then appended to the original, to verify the signature, the user must first calculate the original hash.

Asymmetric cryptography security depends largely on the ease of uncovering the secret key with the information present on the public key. The difficulty of revealing the secret key depends on the scheme used. As an example, RSA depends on the hardness of integer factorization. Those problems are believed to be computationally impractical to solve on classical computers. Many attempts on breaking RSA have been made (BONEH et al., 1999), but it still holds strong in the classical world. However, Shor (1999) and Grover (1996) shows us that, in theory, quantum computers could solve these problems in a non-trivial manner, effectively breaking the security of schemes relying on those problems.

2.2 DIGITAL SIGNATURES

A digital signature, first proposed in (DIFFIE; HELLMAN, 1976), conjectured that if a one-way function were to exist, a signature scheme would derive from it. Shortly after, RSA was introduced in (RIVEST; SHAMIR; ADLEMAN, 1978) as one of the classical signature scheme.

Digital signature schemes work by using a private key in conjunction with a message to generate an output that can be verified using a public key. These schemes are usually comprised of three main steps, **Key generation**, which is responsible for creating the key pair, the **Message signing** itself and **Signature verification** which should provide the following set of assurances.

Authentication. When the private key of a message is bound to a user, verifying the signature of the document provides proof that the document was indeed signed by that user;

Integrity. Any changes made to the document after it was signed should invalidate the signature;

Non-Repudiation. The owner of the private key cannot deny having signed the document after the fact.

2.3 ONE-WAY FUNCTIONS

One-way functions are easy to compute but computationally hard to invert functions that can receive an arbitrarily large input to generate an arbitrarily large output as long as it is easy to compute, and given the output of one such function, it should be computationally hard enough to find out what the input was. The existence of this type of function has not yet been mathematically proven, but enough evidence suggests that they do. The existence of one-way functions would imply that the complexity classes P and NP are not equal (GOLDREICH, 2006).

As a formal way of describing one-way functions, let $F : \{0,1\}^* \rightarrow \{0,1\}^*$. Then F is a one-way function if and only if F can be computed in polynomial time and F has a negligible probability of being inverted by a randomized polynomial algorithm.

However, generic one-way functions like these will not fulfill every demand required when working with cryptography. To create more useful one-way functions we are going to add new requirements and constraints step-by-step to mold these generic one-way functions into something that can be used to provide security. We start by building what is called a hash function.

A **hash function** is a one-way function $F : \{0,1\}^* \rightarrow \{0,1\}^n$ where $n \in \mathbb{N}$ is a fixed output size. Fixing the output of the one-way function to n shrinks the co-domain from $\{0,1\}^*$ to $\{0,1\}^n$. From this restriction we can see that since $\{0,1\}^n$ is a subset of $\{0,1\}^*$, hash functions can compress arbitrarily large inputs to a smaller image, therefore creating unavoidable collisions.

It is important to note that while collisions are unavoidable by nature for hash functions, they should be extremely unlikely to happen. As a rule of thumb a hash function that outputs a message of N bits should be able to hash, on average, $\sqrt{2^N}$ messages before outputting a collision.

The idea behind hash functions is that the compressed value can act as a fingerprint to the document, representing it with a fixed size. Digital signatures usually use hash functions to shorten the size of the message being signed. Instead of signing the whole document, The hash is signed in its place and the signature appended to the document.

Finally, to create a **cryptographic hash function**, we need to add a set of security properties on top of our hash function. To illustrate these properties let's define **H** as follows:

$$x \in X; \;\; y \in Y;$$

$$H : x \mapsto y; \;\; y = H(x);$$

As previously stated, it is desirable for these hash functions that the only way to obtain the pair (x,y) is to first select an x and then compute y by applying H to it.

The first property is **Pre-image resistance**. Given H and y, it should be infeasible for an attacker to discover x. In other words, it should be computationally hard to *invert* our hash function.

The second property, **Second Pre-image resistance**, tells us that given H and a known x, it should be *computationally hard* to find an x' such that $x' \neq x$ and $H(x) = H(x')$. This property is especially important in digital signatures since a valid pair $(x', H(x))$ would mean a message had been forged under the key holder's name.

The third property, **Collision resistance**, says that, for a given H, find any pair (x,x') such that $x' \neq x$ and $H(x) = H(x')$. This is a more relaxed form of the second pre-image resistance property that aims to guarantee that a hash from a message is unique *enough*.

Lastly, it also desirable that our cryptographic hash functions also have the property known as the *Avalanche effect*, where modifying a single bit from the original message should, on average, modify half of the final hash output.

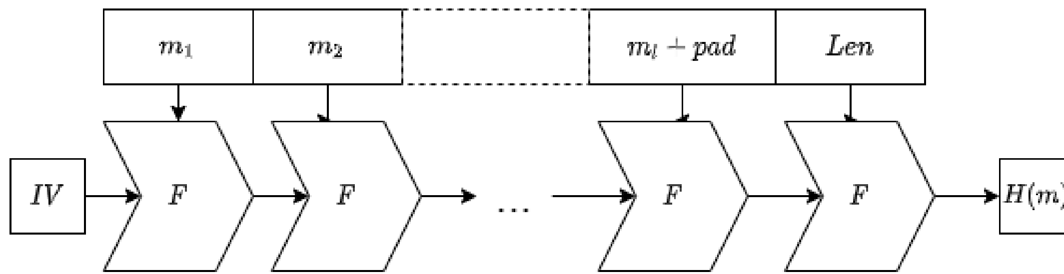
2.4 BLOCK CIPHERS

Block ciphers are important elements in cryptography thanks to their versatility, they are used to create symmetric key algorithms like DES and AES to being the founding block of hash algorithms like SHA-2. These symmetric ciphers work by using keys and encrypting fixed-length blocks of bits, called blocks. Security derives from two properties called *confusion* and *diffusion* (INFORMATION..., 2008). These properties are used to restrict as much as possible the use of cryptanalysis. *Confusion* aims to hide as much information about the key when analyzing the ciphertext, for that purpose a bit in the ciphertext should depend on as many bits as possible of the secret key. While confusion is responsible for hiding the secret key information, diffusion is responsible for obscuring the relationship of plain text and ciphertext, it is responsible for the *Avalanche Effect* where changing a single bit in the original message should change roughly half of the output. As an example, AES implements these properties with the use of a *Substitution-Permutation Network*.

No block cipher is perfect (MENEZES; OORSCHOT; VANSTONE, 2001), system and design constraints can demand more from a cipher than just security, often trading it for efficiency and suggesting that systems should be able to handle a host of block ciphers. In the context of embedded systems, especially those with code size requirements, being able to support various block ciphers can become a detriment.

Block ciphers can be used to build hash functions through one-way compression functions, these functions mix two inputs to generate a smaller output, just like block ciphers. Merkle–Damgård is a method for building cryptographic hash functions and is in fact used in

Figure 1 – Merkle–Damgård construction



Source: Original

many popular algorithms like SHA-1 and SHA-2. First, the message is padded to a set length, and a new block is added to the end containing the message length, this is called the *finalization* or *length padding* step and, while not obligatory, aims to increase security. The construction works by breaking down the padded message in blocks of fixed size and compressing them through a one-way function f at first with the starting *initialization vector* (*IV*) and then using the result of the last application in succession until the message has been fully parsed as seen in 1.

2.5 SPONGE FUNCTION

Sponge functions are a particular way to generalize hash functions through sponge construction (BERTONI et al., 2007), which iterates over a padded variable-length input over a fixed-length permutation to generate an arbitrarily large output, resulting in a more generalized function. The fact that the output can now vary means that this sponge function can also be used as a stream cipher.

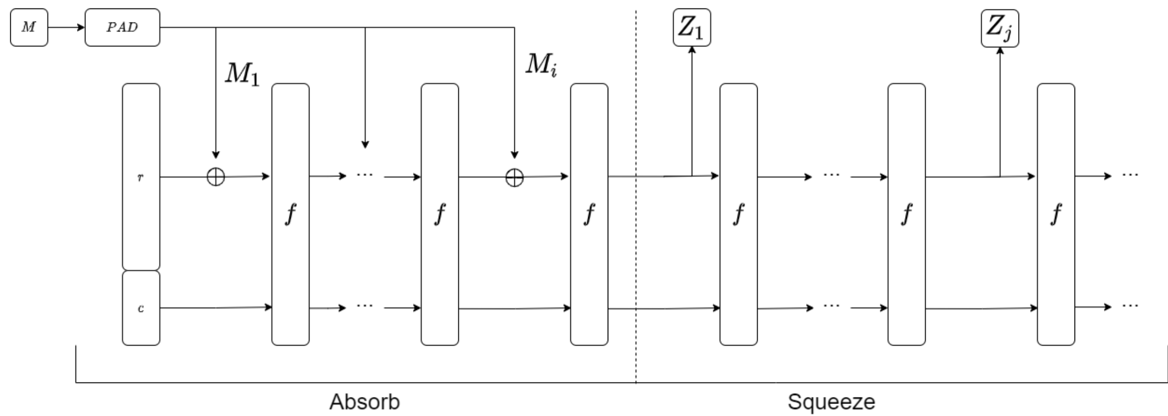
A sponge function is built iterating through a fixed-length permutation or transformation function f that operates on a state of b bits from a message with i blocks. The state of a sponge construction can be divided in 2 parts, the bitrate r and the capacity c such that $b = r + c$. At the beginning of the construction, the message is padded up to a multiple of b and the state is set to 0. The construction then proceeds to the absorbing phase and the squeezing phase.

In the absorb step, the input blocks are consumed through an *exclusive or* ($\oplus$) into the r-bit part of the state, between each input block, f is applied to the current state in between the blocks.

In the squeezing step, the user selects how many blocks are desired and the first r bits of the block are returned as output, interleaved by applications of f.

Sponge functions can be used to implement a host of cryptographic primitives by manipulating the internal function f. One of those primitives being cryptographic hashes. One example of a function used to build hashes is Keccak, which SHA3 is built upon.

Figure 2 – Sponge construction



Source: Original

2.5.1 SHA3

SHA-3 is the family of hash functions designed to complement SHA-2 and SHA-1 (DWORKIN, 2015). It is worth noting that SHA-3 is not a direct improvement on SHA-2, but an alternative. It is comprised of six functions that share the same underlying scheme of sponge construction, and therefore all six are sponge functions.

Keccak is the 1600 bit permutation on which SHA-3 is built upon. It is well known to be effective against side-channel attacks and boasts great energy efficiency (BERNSTEIN et al., 2017). The basic execution of the Keccak-f permutations is done in multiple rounds, this construction makes Keccak-f compact and leaves no space for trapdoors to be hidden (BERTONI et al., 2011).

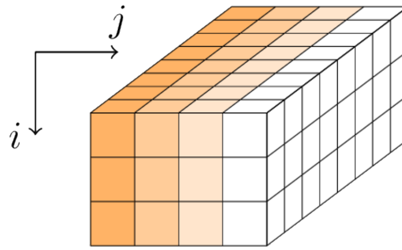
Bitwise logical operations and fixed rotations limit dependence on CPU support only for the rotations, coupled with the fact that Keccak implementations do not depend on large tables like SHA-2 diminish the risk of table-lookup-based cache miss attacks and timing attacks. It also sees gains in energy efficiency (BERTONI et al., 2013).

Keccak also offers good parallelization, leading to compact code and efficient co-processor designs for embedded devices. This parallelization also opens up space for fast hardware implementations by exploiting *single instruction, multiple data* designs, and pipelining in CPUs through the mapping of one-dimensional arrays to CPU words. These improvements drastically improve energy efficiency for embedded devices when compared to SHA-2 (BERTONI et al., 2013).

2.5.2 Gimli

Gimli (BERNSTEIN et al., 2017) is a permutation designed with cross-platform performance in mind. It seeks to mainly provide energy efficiency, compactness, side-channel attack protection through its sponge construction and compactness. It can be used as a building

Figure 3 – Gimli State Representation. From (BERNSTEIN et al., 2017)

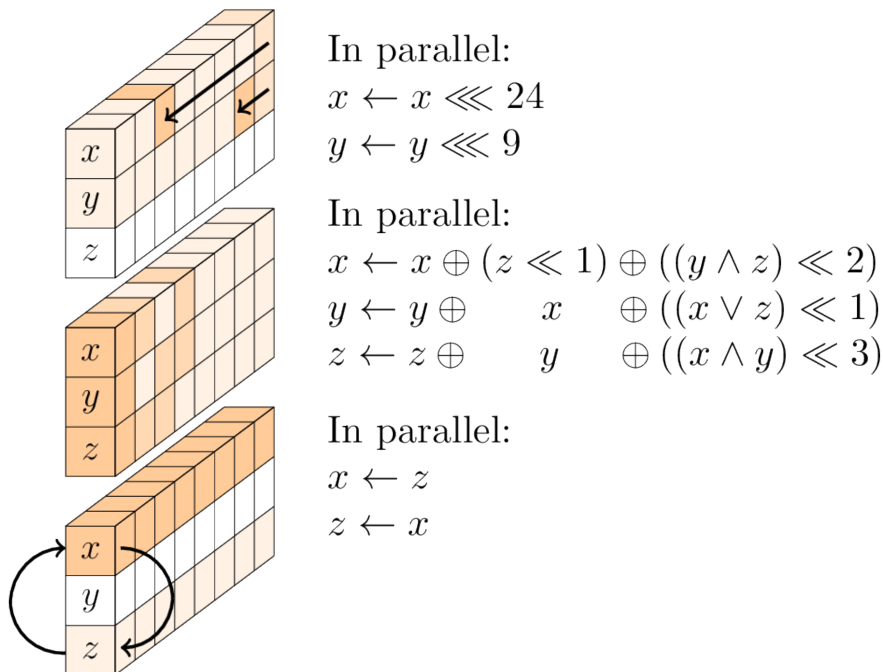


Source: (BERNSTEIN et al., 2017)

block for block ciphers, stream ciphers, message authentication codes, and secure hash functions.

Gimli works by applying a sequence of rounds to a 384-bit state represented by a 32x3x4 matrix. As is with sponge functions, this state is initialized as zero. Each Gimli round is separated into three steps. The non-linear step consists of rotations, a non-linear function T, and swap operations. These steps can be seen in detail in algorithm 1. The linear step consists of two swap operations, and lastly, a constant addition. Let $r \in \mathbb{N}$ be the current Gimli round, $W \in \{0,1\}^{32}$ be a word and $\{s \mid s \in W^{3x4}\}$ be the Gimli state.

Figure 4 – Gimli Non-linear layer

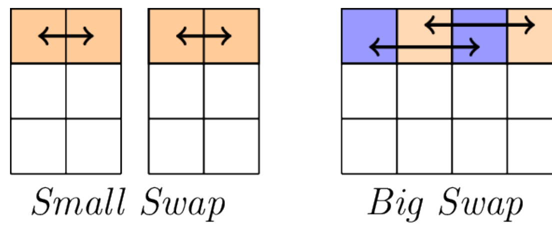


Source: (BERNSTEIN et al., 2017)

The linear step has two types of swaps, called *big swap* and *small swap*. A small swap is done every four rounds starting from the first and a big swap is done every four rounds starting

from the third.

Figure 5 – Gimli linear layer.



Source: (BERNSTEIN et al., 2017)

Creating a hash function from the Gimli permutation using the sponge function can be done by following the sponge construction detailed in the previous section. The only unique characteristic is the padding done, adding a single byte (`0x1`) to the last byte of the last block.

3 HASH-BASED SIGNATURE SCHEMES

3.1 LAMPORT-DIFFIE OTS

The Lamport-Diffie OTS scheme is based on the existence of secure one-way functions (LAMPORT, 1979), so much so that its only security assumption is that they exist, making its security very reassuring. The scheme itself is very simple but suffers from a few glaring flaws, mainly the size of the keys.

First, let's exemplify the Lamport-Diffie OTS with a simple problem. Ana wishes to sign a 1-bit message m using a one-way function F. First, she generates a pair of n-length strings (x_0, x_1), this is her *Private Key*. Afterwards she applies the one-way function F to each string, generating the public key $(y_0 = F(x_0), y_1 = F(x_1))$. The signing process is very simple can be seen as happening bit-by-bit without loss of generality, if the bit about to be signed is a 0, Ana reveals the corresponding x_0, otherwise, she reveals x_1. With public key and message in hands, anyone can verify the signature by computing $F(x) = y'$ and matching it with the public key.

We can now extrapolate this example for a m bit message. The private key now needs a pair of n-length strings for each bit in the message, and the public key now has a size of $m*$n-length bits, one $n-string$ for each bit. As an example for a 256bit security n-string signing a 256bit message we have $256*256*2 \approx 16$ KiB for the private key, increasing to 32 Kib while storing the public key as well. Even when taking into account techniques to lower the amount of memory needed to store Lamport-Diffie OTS keys they are still prohibitively large. As a more formal description of the Lamport-Diffie OTS scheme, Let $F : \{0,1\}^* \rightarrow \{0,1\}^n$ such that F is a cryptographic one-way hash function and n is a positive integer known as the security parameter.

Key generation. The private key consists of bit strings of length n chosen at random.

$$X = \begin{pmatrix} X_0 = (x_{0,1}, x_{0,2}, x_{0,3}, ..., x_{0,(n-2)}, x_{0,(n-1)}) \\ X_1 = (x_{1,1}, x_{1,2}, x_{1,3}, ..., x_{1,(n-2)}, x_{1,(n-1)}) \end{pmatrix}$$

we can clearly see that a private key needs $2n^2$ bits to be stored.

To generate the public key Y, we apply the hash function F to every string in X.

$$Y = \begin{pmatrix} Y_0 = (F(x_{0,1}), F(x_{0,2}), F(x_{0,3}), ..., F(x_{0,(n-2)}), F(x_{0,(n-1)})) \\ Y_1 = (F(x_{1,1}), F(x_{1,2}), F(x_{1,3}), ..., F(x_{1,(n-2)}), F(x_{1,(n-1)})) \end{pmatrix}$$

Signature generation. Before we start the signing process we must first hash the original message m, $m' = F(m)$. The signature σ is created by revealing the corresponding n-length string from one of the private key sets. For every bit in the message m' we reveal an n-length string from the corresponding private key set, revealing from X_0 if the bit is a 0 or revealing from X_1 otherwise.

$$\sigma = (\sigma_1, \sigma_2, \sigma_3, ..., \sigma_{n-2}, \sigma_{n-1})$$
$$\sigma = (X_{m'_i,1}, X_{m'_i,2}, \ldots, X_{m'_i,n-1})$$

Signature verification. To verify a signature σ from a message m we compute $m' = F(m)$ and select the corresponding public key y for each bit from m' to generate Y'.

$$Y' = (y_{m_0,1}, y_{m_1,2}, y_{m_2,3}, ..., y_{m_{(n-2)},n-2}, y_{m_{(n-1)},n-1})$$

we then apply the hash function F to every string from the signature.

$$\sigma = (F(\sigma_1), F(\sigma_2), F(\sigma_3), ..., F(\sigma_{n-2}), F(\sigma_{n-1}))$$

If we find that $\sigma = Y'$ we can affirm that this signature is indeed valid and invalid otherwise.

3.2 WINTERNITZ OTS

Robert Winternitz proposed in 1979 an improvement to the Lamport-Diffie OTS scheme that aimed to reduce signature and key size by signing multiple bits at once instead of every single bit separately. This change trades time for space, the more bits signed at once, the longer it takes to generate keys, sign and verify messages. Just like Lamport-Diffie OTS scheme, WOTS uses a cryptographic hash function F.

The Winternitz parameter w corresponds to the level of compression we wish to utilize. To compress the message in blocks of 4 bits we would have $w = 16$. The larger this parameter, the smaller the resulting keys, but the longer it will take to sign and verify any document.

In the following, we present the process of signing and verifying a WOTS signature for a message M bits long.

Key pair generation. First we must choose the Winternitz parameter $w \in \mathbb{N}, w > 1$, Effectively defining our compression level.

The private key consists of L strings of n bits chosen at random.

$$X = (x_1, x_2, x_3, ..., x_{l-1}, x_l) \xleftarrow{\$} \{0,1\}^{(n,L)}$$

The public key is generated by applying the hash functions to itself $w-1$ times for every x_i, giving us Y.

$$Y = (Y_1, Y_2, Y_3, ..., Y_L)$$
$$Y = F^{(w-1)}(x_1), F^{(w-1)}(x_2), F^{(w-1)}(x_3), \ldots, F^{(w-1)}(x_L)$$

Where L_1, L_2 and L are obtained as follows:

$$L_1 = \left\lceil \frac{M}{\log_2(w)} \right\rceil, \; L_2 = \left\lfloor \frac{\log_2((L_1)(w-1))}{\log_2(w)} \right\rfloor + 1 \text{ and } L = L_1 + L_2.$$

L_1 represents the number of blocks required to sign the whole message. L_2 represents the minimum amount of blocks required to calculate and store the checksum of the message which will be explained shortly. It follows then that L is the total amount of blocks required for the signature process.

Message signing. We now sign a message by breaking it down into L_1 blocks represented in `base-w`, such that $M = (m_1, m_2, m_3, ..., m_{L_1})$ where $m_i \in \{0, ..., w-1\}$.

Before we start signing our message we need to compute the checksum. To understand why this step is necessary let's imagine a message with a single block. The signing process happens by signing a message a few times and stopping somewhere between our secret key and our public key $F^0(x_i) \leq \sigma_i \leq F^{w-1}(x_i)$. We know that a hash function is computationally infeasible to invert, however, it is extremely easy for our attacker to compute $F(\sigma)$ and claim that we signed a message that we did not. To prevent this attack we compute a checksum that indicates the total remaining applications of our hash function. That means that to go *forward* in a block, the attacker would have to go *backward* on the checksum block.

We now calculate the checksum and represent it as *base-w*.

$$C = \sum_{i=1}^{L_1} (w - 1 - m_i)$$

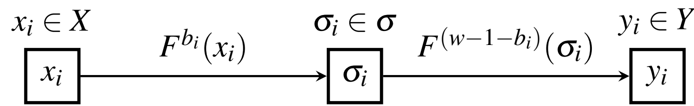
We concatenate both message and checksum $B = M||C = (b_1, b_2, b_3, ..., b_l)$ and can now sign the message by applying our hash function onto our corresponding x_i blocks, b_i times.

$$\sigma = (\sigma_1, \sigma_2, \sigma_3, ..., \sigma_L)$$
$$\sigma = (F^{b_1}(x_1), F^{b_2}(x_2), F^{b_3}(x_3), ..., F^{b_L}(x_L))$$

Signature verification. To verify the signature the user must recreate the base-w B string and compute the remaining hash functions to reach $w-1$ applications, comparing both the public key and the one just generated. If we have calculate the public key through the signature and compare it to the published public key Y we can verify that the signature is valid if those public keys match. Otherwise we know that the signature is false. This operation can be summarized in the following equation:

$$Y = (F^{w-1-b_1}(\sigma_1), F^{w-1-b_2}(\sigma_2) F^{w-1-b_3}(\sigma_3), ..., F^{w-1-b_L}(\sigma_L))$$
$$= (y_1, y_2, y_3, ..., y_L) = Y$$

Figure 6 – Winternitz Signature Operation



Source: Original

As a note on key sizes, both the private and public keys can be further reduced. Private keys are often generated as needed by using a pseudo random number generator and storing only the seed. The public key can be concatenated and hashed.

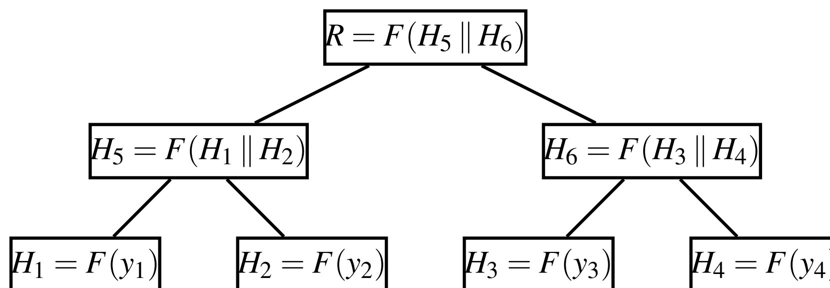
3.3 MERKLE SIGNATURE SCHEME

One of the problems that one time signatures have is that to sign a large number of messages we would either have to store and send a large number of public keys or generate a new one every time a signature would be requested, the former would require large amounts of space and the latter would be susceptible to attacks where an attacker could pass itself as the key holder and send out fake public keys.

Merkle proposed a protocol called "tree authentication" which can be used to authenticate a fixed number of public keys while using minimal space. This tree is constructed as a binary tree (MERKLE, 1989). The idea behind tree authentication is to lock nodes behind hash functions, every node is the hash from the concatenation of its children. Leaf nodes are generated by creating and hashing a WOTS public key.

The price we pay for this scheme can be seen in the signature, no longer we need only transmit a tuple containing public key and signature but now it must also include the **path** of authentication. We exemplify this using a tree with 4 WOTS signatures in figure 7.

Figure 7 – Merkle Tree Authentication

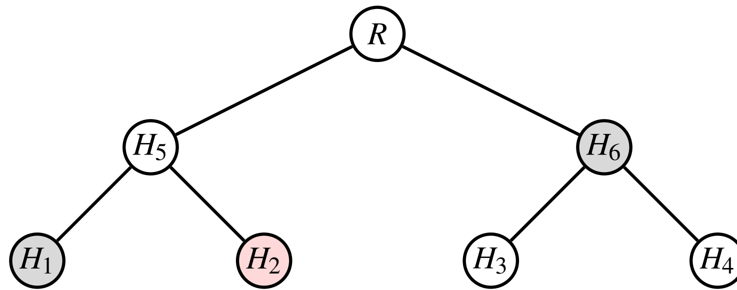


Source: Original

One of the downsides to this authentication method and those similar to it is that to generate our public key, the root node, we must first generate all the leaf nodes without exception. This leads to a big increase in key generation times, increasing exponentially with tree depth. Once every leaf node has been generated we can climb up the tree, generate and even publish our root public key.

Another problem that arises with the introduction of such a scheme is that if we don't keep all the nodes stored, we need to reconstruct part of the tree to generate the tree path which is part of the signature, that means we are paying the cost of key generation every time we want to sign a message. For example, if we wish to use H_2 in figure 8, we would need to calculate and send along (H_1, H_2, H_6), but to calculate H_6 we would need to generate (H_3, H_4) again.

Figure 8 – Merkle Tree Authentication Pathing



Source: Original

4 EMBEDDED DEVICES

Embedded systems are, in simple terms, mini-computers that often have the least amount of functions for a given task. They are comprised of a processor, memory, and peripherals as needed, usually microcontrollers. Since microcontrollers are often produced in big numbers, any small change in their design can have a big impact on the cost of production, be it an added feature driving its price up or the removal of unneeded features lowering its cost. The example given was of cost, but it should not be thought of as the only constraint embedded devices have. An embedded device can have many more constraints, ranging from the simple cost of production to clock frequency and even the volume of the chip.

With the advent of internet of things (*IoT*), the need for microprocessors to be able to operate with other devices through the internet at large has become noticeable. The fact that these microprocessors now must work together implies the need for exchanging messages between themselves, and that in turn makes it clear that there is a need for securing messages between these devices and the internet. Unfortunately, while IoT has been steadily rising in popularity, security for embedded devices has not kept pace (VIEGA; THOMPSON, 2012). A common example used is AES, where the high energy consumption of its algorithm makes it vulnerable to side-channel attacks (BERTONI et al., 2011). While there are implementations and papers that offer adaptations to these problems for existing protocols (GROSS; MANGARD; KORAK, 2017), many fall short when compared to new algorithms made specifically for embedded devices.

Following this shift in security requirements for embedded devices, a new branch of cryptography has been flourishing, namely lightweight cryptography. There are new lightweight cryptography (LWC) protocols being developed. The main concerns for LWC are energy, code size, and RAM usage. The the main goals of LWC are end-to-end communication efficiency, especially for battery-powered devices and applicability for low resource devices (KATAGI; MORIAI et al., 2008).

Securing embedded devices is hard. The problem with securing embedded devices comes from the range of knowledge one must have before attempting to pick the best approach (GEBOTYS, 2009). Not only must programmers ensure that their implementations are secure, but they must also keep in mind the multiple ways an attacker can try to break in. In general, a truly secure embedded device would have to be secure even against unforeseen attacks, coupled with the fact that most embedded devices use assembly and C as languages and a lot of responsibility falls under the programmer.

Even though embedded devices have been known for their weak processor and small memory sizes, advances in this field have made it possible for programmers to use higher-level languages for their projects, alleviating some weight from the programmers. By using a higher-level language, designers can trade efficiency for some guarantees that a higher-level language can offer.

4.1 RUST

Rust is an open-source multi-paradigm programming language that aims to provide safety, concurrency, and speed developed by the Mozilla Foundation. While superficially similar to C and C++ style languages, under the hood, Rust is akin to ML family-like Haskell where most parts of a method are expressions. One of the main objectives of the Rust language is to provide system-wide integrity.

Cargo is Rust's package manager. It can download dependencies, compile, distribute and upload packages to crate.io, Rust's package registry. Cargo can also use templates to initialize your project, an example that is commonly used in embedded programming for the Cortex-M family of microprocessors is `cortex-m-quickstart` which upstarts the memory mappings and sets up useful crates for developing embedded applications. Cargo is also used to install support for cross-compiling Rust code to other architectures.

Rust has a powerful static analysis, thanks to its design and linear types. Thanks to its type system, it prevents data races at compile time. Seen through the lens of embedded programming, this static analysis can also be used to catch errors in I/O configuration.

LLVM (LATTNER; ADVE, 2004) is a compiler infrastructure written in C++ containing a set of toolchains and compilers to aid the development of languages for any instruction set architecture, it is portable to most modern Unix-like environments and has great support for Windows as well. Rust is built atop the LLVM infrastructure, where Rust code goes through LLVM to be translated to the target architecture. This means that portability is defined by LLVM.

One of the biggest hurdles in programming is dealing with pointers, memory, and concurrency, so much so that higher-level languages usually don't deal directly with these topics, hiding them behind safe abstractions. Rust tackles this problem using its powerful static analysis through one of its defining features, *ownership*. *ownership* in Rust refers to the fact that no two scopes can share the same pointer. Thanks to *ownership*, Rust doesn't have a garbage collector, instead, when the scope that *owns* the resource ends, so does every object that belonged to it.

To prevent an object from being deleted, one must explicitly transfer *ownership* to another scope. It is also possible to reference objects and pass them along to other scopes without passing ownership by using the *borrow* feature, a borrowed object is not destroyed when the function returns and can only be used by it while it is still in scope. It cannot, for example, spawn a thread to use the borrowed object. These same primitives also deal with concurrency, ownership guarantees thread isolation and locks work by preventing data to be accessed, not code. An example using ownership and borrows can be seen in figures 9, 10 and 11.

Thanks to the ownership model and its data types, Rust can enable static analysis. The restrictions imposed through ownership make it possible to track the lifetime of an object at compile-time and to destroy it without the use of a garbage collector. Along with the lack of

Figure 9 – Rust ownership transfer between variables

```
let foo_a = Bar.new();
let foo_b = foo_a;
// foo_b is the new owner of Bar.new()
// and calls to foo_a would cause
// a compile error
```

Source: Original

Figure 10 – Rust ownership transfer to function

```
fun bar(data: Foo) -> usize {
    data.len()
    // data is destroyed when
    // this function returns
}

let foo = Foo.new();
let size = bar(foo);
// foo is given ownership to bar function
// and destroyed when it returns
// calls to foo would cause
// a compile error
```

Source: Original

a garbage collector, Rust also has zero-cost overhead abstractions like C++, coupled with its powerful analyzer, Rust has no runtime overhead. It is worth noting that the ownership model, while great for safety, incurs a steep learning curve due to its rigidness.

Rust aims to be as fast as C/C++ and it is already comparable to them in a number of benchmarks (KOSTYA, 2021), some of these results can be seen in table 2.

Table 2 – Time comparisons (seconds).

	Rust	C/gcc	C++/gcc
bench.b	2.156 ± 0.030	1.939 ± 0.048	0.892 ± 0.022
mandel.b	19.505 ± 0.307	13.477 ± 0.263	14.001 ± 0.347
Base64	1.328 ± 0.049	1.240 ± 0.029	2.512 ± 0.052
Matmul	3.438 ± 0.015	3.361 ± 0.032	N/A

Source: Adapted from: (KOSTYA, 2021)

Figure 11 – Rust variable transfer through borrow

```
fun bar(data: &Foo) -> usize {
    data.len()
    // data is not destroyed when
    // this function returns
}

let foo = Foo.new();
let size = bar(&foo);
// foo is borrowed to bar
// and is not destroyed when it returns
// calls to foo would not cause
// a compile error this time
```

Source: Original

Another feature, or lack thereof, that helps Rust with its safety is the **lack** of exceptions. Exceptions are known to be the topic of many books due to their complexity and wide-ranging implications. Rust instead uses its multi-valued return types (eg.: `Option<T>`) and the `match` operator to handle exception cases. This alternative to exception handling makes code be easier to analyze and guarantee there are no invalid states caused by a break in the middle of a function. It also helps programmers by restricting the errors they make, one does not need to worry about what exceptions a function can throw if there are no exceptions.

There are cases when code that could be flagged by the compiler as unsafe needs to be written. The keyword `unsafe` delimits a scope where the compiler turns a blind eye to whatever safety violation might happen inside it. This is most often seen in embedded programming where memory access has to be done manually to be efficient. The `unsafe` keyword can also be used to introduce Assembly and C code directly into a Rust application or from outside sources.

However, all of these features do not come for free, coupled with its intricate type system Rust's compiler has to spend quite a bit of time enforcing the constraints that make it so safe. Another problem is how Rust compiles its projects, while some languages like C have files as its compile unit, Rust compiles its crates as a whole, so modifying a single file in Rust can lead to a lot more compile-time than the alternatives. Another hurdle that Rust has to cross to improve its compile speed is its debt to the LLVM. Intermediate representation LLVM code generated from Rust is generally considered of poor-quality and therefore must be fixed by LLVM, leading to an increase to compile time. Rust introduced a mid-level intermediate representation to help clean up some of this debt, but there is still research being done to alleviate these problems.

4.1.1 The future

While priority in reference code is still focused in C/C++, it doesn't take long before Rust implementations are made available. These Rust implementations often come from the authors of the reference itself. Rust is one of the most community-driven languages, largely due to Cargo and the ease of publishing packages through Cargo, it's not long before a Rust implementation can be found, implemented by the community.

HACSPEC was a popular Python package used to specify cryptography primitives in a succinct and easy-to-read manner that lends itself to formal verification. The Python branch has been deprecated and the development of HACSPEC has shifted to Rust. Gimli used the older HACSPEC specification in their work to provide an easy-to-read and execute reference code. Bringing HACSPEC to Rust is a nice addition to the Rust safety toolkit when dealing with cryptography.

Constant Generics have been added in the beta version of Rust and are expected to be released shortly. This tool helps design compile-time safe APIs and is highly anticipated. These generic variables are arguments that range over constant values, allowing the programmer to parameterize types by primitive types. This addition would remove the need for some workarounds when designing software that needs the guarantee of compile-time safety.

4.1.2 Embedded Rust

While Rust suffers on one end from long compile times, in embedded programming, the worst offender to development is debugging and hard-to-find errors, especially memory ones, since debugging in embedded environments is fairly harder than in normal circumstances. While debugging tools for C and C++ are given preference to Rust ones, Rust's static analysis increases confidence in generated code and diminishes time spent debugging messy errors. The earliest a bug is caught in an embedded environment the better. Since Rust forces developers to adhere to safety guidelines through the ownership model, it is harder for bugs to slip through the cracks, leading to less debugging time and a lower chance of updating, or even recalling, products.

One big area of embedded programming is systems with very strict time and space requirements, these systems belong to a class called real-time. Such programs usually do not have the free space needed to handle all of the standard library size. `no_std` is a directive offered by Rust to not include any standard functionality and only includes those from the `core` crate. This limits the number of high-level abstractions we can use, but the safety guarantees are still in there. Rust has quite the selection of `no_std` crates to choose from the official registry, including most cryptographic primitives.

Rust depends on LLVM to support multiple architectures. LLVM offers support for most modern microprocessors, but older or cheaper devices might not have support for LLVM, and so it won't support Rust. C on the other hand is almost universally supported by any

microchip, bar the odd assembly only chip. The current trend is for chips to have support for LLVM but it is undeniable that C/C++ programs have a wider selection of supported microchips.

Executable sizes in Rust have known to be rather large, and while not entirely true, it requires some work to strip it down to C sizes. With no work whatsoever a simple Rust *Hello world* program is as large as 650711 bytes while the C counterpart is 8551 bytes. (LIFTHRASIIR, 2021) shows us the steps necessary to lower this bloat in size and where they come from. The major difference between Rust and most other languages is that Rust's standard library is compiled directly into its binaries, bloating them by a considerable amount, meaning that while Rust sizes can be magnitudes larger than C programs in usual programming, embedded Rust has a comparable size to its counterparts. In (LIFTHRASIIR, 2021), the author manages to reduce the size of the executable to 9248 by using pure Rust calls and to 6288 bytes by using system calls. While this example shows us a very simple program, comparisons between C and Rust binary sizes become blurry when moving to more intricate programs due to difference in structures and types, however, it is important to note that given enough work, Rust executables can become as small as other embedded languages.

5 EWOTS

We now present EWOTS, a `no_std` Rust crate that provides implementations of WOTS and Gimli that depend only on `core` crates. While debugging was done using specific Cortex-M4 crates, the `core` implementations do not rely on any external library and can theoretically be used on any device able to run Rust programs. The use of *features* to select implementations and constants makes it so that any new algorithm (e.g. a new hash function) can be easily added without interfering in the main functions as long as they have the same signature.

`Core` crate libraries present some problems as many high-level abstractions are only available in the standard crates, creating a more rigid environment. This effect creates a steeper curve to learning and using Rust as a language but it should not deter anyone from trying as there is plenty of material available for embedded Rust, both from the developers and the community. The gain we obtain from using only the `core` crate is seen in the binary size as explained at the end of section 4 and high portability.

5.1 CONFIGURATION

Configuration of embedded development is rather simple when using Cargo. Using a quick start for the selected architecture, we do not have to worry about memory layout, nor the initial setup for running and emulating our project.

We make use of `Features` and external packages through Cargo. `Features` allow us to select constants and implementations in compile time.

`Cargo.toml` is the main configuration file, it also manages the external and internal packages used in the project. The most important packages are listed in listing 12. We specify the path to our Gimli implementation in this file. An important crate for debugging is `cortex-m-semihosting`, included for proper debugging when using cortex processor.

Figure 12 – Project dependencies

```
gimli = { path = "./gimli" }
cortex-m-semihosting = "0.3.3"
```

Source: Original

5.1.1 Features and Constants

Rust has a feature system to select which parts of the program should be compiled. We make use of this system to set constant values depending on the kind of algorithm we wish to use. Another benefit of using this feature is that we can circumvent some ownership problems when dealing with constant values which would bloat some parts of our code. This type of

Figure 13 – Hash feature selection

```
#[cfg(feature = "Gimli")]
pub const N: usize = 32;
#[cfg(feature = "Gimli")]
pub const HASH: &str = "Gimli";
```

Source: Original

Figure 14 – Block size constants

```
#[cfg(feature = "16")]
pub const W: usize = 16;
#[cfg(feature = "16")]
pub const LOG2W: usize = 4;
#[cfg(feature = "16")]
pub const L: usize = 67;
```

Source: Original

Figure 15 – EWOTS Features from Cargo.toml

```
[features]
default = ["NO_DEBUG", "NO_DEBUG_SKEY"]
16 = []
Gimli = []
...
```

Source: Original

feature can be easily expanded to include any type of algorithm and constant. We can select debug levels, hash implementations, and constants.

These features must be declared in the `Cargo.toml` file. A set of features can be selected as default ones, however, to remove these features one must use `default-features = false` and select each one individually, therefore, it is recommended to select only the most common features for the default parameter. features added without setting `default-features = false` are added on top of the default ones.

5.2 GIMLI IMPLEMENTATION

We keep track of the Gimli state and modify it by using a struct that contains the current state of Gimli. Our functions and transformations do not return the state, but instead, alter it in place. While `Gimli struct` itself is public, the internal *state* is not, so a user can only alter its value by using the intended functions.

Figure 16 – Gimli Structure

```
pub struct Gimli {
    state: [u32;12]
}
```

Source: Original

5.2.1 Gimli Permutation

Our Gimli permutation follows the code shown in (BERNSTEIN et al., 2017) along with the reference implementations published by the authors. While every part is implemented in the same function, we can see each layer presented in the original work by itself. The non-linear layer seen in listing 17 involves the rotation of bits in our word, luckily, Rust offers the `rotate_left` function that does just that, saving us the trouble of implementing it. The second half of the non-linear layer is a set of bit operations, modifying the original state.

Figure 17 – Gimli Non-linear layer

```
    fn gimli(&mut self){
        ...
        // Non-linear layer
        x = self.state[j].rotate_left(24);
        y = self.state[j+4].rotate_left(9);
        z = self.state[j+8];
        self.state[j+8] = x ^ (z << 1) ^ ((y & z) << 2);
        self.state[j+4] = y ^ x ^ ((x | z) << 1);
        self.state[j] = z ^ y ^ ((x & y) << 3);
        ...
```

Source: Original

The second step, the linear layer, can be seen in figure 18. We shuffle the values into an auxiliary array and pass ownership of those values to the corresponding place of the state. Since we use every entry and don't repeat them, we can be sure no value will be lost to ownership. The final layer, constant addition can be seen in listing 19, is done after a small swap.

5.2.2 Gimli Hash

The implementation of Gimli Hash was shown in the NIST submission for the latest lightweight cryptography competition. It follows the usual Keccak structure through the use of *Absorb* and *Squeeze* functions with the underlying *Gimli Permutation*. We chose to implement both the permutation and the hashing process in the same structure since they both rely in the

Figure 18 – Gimli Linear layer

```
fn gimli(&mut self){
    ...
    // Small swap
    aux = [self.state[1],self.state[0],self.state[3],self.state[2]];
    self.state[0] = aux[0];
    self.state[1] = aux[1];
    self.state[2] = aux[2];
    self.state[3] = aux[3];
    ...
    // Big swap
    aux = [self.state[2],self.state[3],self.state[0],self.state[1]];
    self.state[0] = aux[0];
    self.state[1] = aux[1];
    self.state[2] = aux[2];
    self.state[3] = aux[3];
    ...
```

Source: Original

Figure 19 – Gimli constant addition

```
fn gimli(&mut self){
    ...
    self.state[0] ^= 0x9e377900 ^ r;
    ...
```

Source: Original

current Gimli state. While the authors do not force a single number of rounds, they strongly recommend 16 rounds, and as such, we fixed the rate number to 16 to keep it as close to the proposal as possible. The first step to our hash seen in listing 20 is to find out how many blocks we will need to absorb and how many bytes will need to be padded in the last block. Afterward, we absorb each block individually except the last one (unless there is no need for padding), since *blocks* is rounded down.

The next step is to handle the last block. Because we are dealing with immutable object `data` we can not directly increase its size to match our rate, therefore we instantiate a new block filled with zeros and transfer the values from data to this auxiliary array and write `0x1` to its last byte to complete the padding. We then absorb the last block into our state as we can see in 22.

Lastly, we squeeze our Gimli state twice with a Gimli permutation in between to generate the hash output. The squeeze function seen in listing 23 simply outputs 4 words from the top of the state into a single 16-byte array. These two squeezes are then concatenated to generate the final 32-byte hash.

Figure 20 – Gimli Hash

```
pub fn finalize(&mut self, data: &[u8]) -> [u8;32] {
    ...
    let len = data.len();
    let blocks = len/rate;
    let bytes_left = len%rate;

    for i in 0..blocks {
        self.absorb_block(&data[16*i..(16*(i+1))]);
    }
    ...
```

Source: Original

Figure 21 – Gimli Block Absorb step

```
fn absorb_block(&mut self, data: &[u8]) {
    self.state[0] ^= (data[0] as u32) |
        ((data[1] as u32) << 8) |
        ((data[2] as u32) << 16) |
        ((data[3] as u32) << 24);
    ...
    self.state[3] ^= (data[12] as u32) |
        ((data[13] as u32) << 8) |
        ((data[14] as u32) << 16) |
        ((data[15] as u32) << 24);
    self.gimli();
}
```

Source: Original

5.2.3 Gimli Outputs

These tests were done using the reference code from the second round NIST submission and can be found in (NIST, 2021). As the Gimli permutation is the focal point of the submission, we chose to separate our tests into two distinct sections. These results can be seen in table 3 and 4.

To test the permutation algorithm we feed known Gimli states and perform the full 24 round Gimli permutation in both the reference code and EWOTS, comparing each word.

Recently NIST has published a new list of finalists for its lightweight cryptography competition, alas, Gimli was dropped out. However, other candidates stand out as having similar strengths as Gimli had, in particular, Xoodyak seems like an attractive alternative, co-designed by one of the original authors of AES, it seems to have remarkable performance, simplicity, and

Figure 22 – Gimli Padding

```
pub fn finalize(&mut self, data: &[u8]) -> [u8;32] {
    ...
    let mut last_block: [u8;16] = [0;16];
    let mut start = len - bytes_left;
    for i in 0..bytes_left {
        last_block[i] = data[start];
        start += 1;
    }
    last_block[bytes_left] = 0x1;
    ...
    self.absorb_block(&last_block);
```

Source: Original

Figure 23 – Gimli Squeeze

```
fn squeeze(&self) -> [u8;16] {
    let mut out: [u8;16] = [0;16];
    out[0] = self.state[0].to_le_bytes()[0];
    out[1] = self.state[0].to_le_bytes()[1];
    out[2] = self.state[0].to_le_bytes()[2];
    out[3] = self.state[0].to_le_bytes()[3];
    ...
    out[12] = self.state[3].to_le_bytes()[0];
    out[13] = self.state[3].to_le_bytes()[1];
    out[14] = self.state[3].to_le_bytes()[2];
    out[15] = self.state[3].to_le_bytes()[3];
    out
}
```

Source: Original

a small code base.

5.3 WOTS IMPLEMENTATION

Our implementations of WOTS introduces three structures, namely, *Wots* that contains the internal states necessary for the WOTS algorithm, *WotsSign* that handles the signature itself and the public key. The `wots_const` seen here are defined in compile time through the use of *features*, as seen in section 5.1.1. By defining these constants in compile time, we can avoid using larger objects such `Box<T>` to instead use a lower level static array, giving us smoother handling of these variables.

The digest algorithms can be set using the appropriate feature, in listing 26 we can see

Figure 24 – Gimli hash output

```
pub fn finalize(&mut self, data: &[u8]) -> [u8;32] {
    ...
    let h1 = self.squeeze();
    self.gimli();
    let h2 = self.squeeze();

    for (to, from) in out.iter_mut().zip(h1.iter().chain(h2.iter()))
{ *to = *from };
    out
}
```

Source: Original

Table 3 – Gimli Permutation Test

Initial Gimli state			
0x6467d8c4	0x7dcf83b	0x3b0bb0d4	0x1b21364c
0x83431dc	0xefbbe8e	0x54e884	0x648bd955
0x4a5db42e	0xca0641cb	0x8673d2c2	0x2e30d809
Reference output			
0xf99da657	0xfd6a7878	0x8894eae9	0xfd59fd85
0x9f41cd12	0x266a1891	00xcf7ad831	0xf916b6e9
0xb9a3a5e8	0x3d7dee51	0xf50ce0fd	0xf102005e
EWOTS output			
0xf99da657	0xfd6a7878	0x8894eae9	0xfd59fd85
0x9f41cd12	0x266a1891	00xcf7ad831	0xf916b6e9
0xb9a3a5e8	0x3d7dee51	0xf50ce0fd	0xf102005e

Source: Original

Table 4 – Comparing Gimli Hash outputs

There's plenty for the both of us, may the best Dwarf win.
Reference Output
0xf456f2da6bf5bbded9b6033e63f7564537eec541c78f3c471eca11bffae90f55
EWOTS Output
0xf456f2da6bf5bbded9b6033e63f7564537eec541c78f3c471eca11bffae90f55

Source: Original

an example of GIMLI being used to generate a digest. Other algorithms can be set as long as they respect the function signature. The same logic can be applied to any algorithm with the feature flag. Since we can only set one of the digest algorithms without causing compiler errors, by setting a feature we can guarantee not only one algorithm, but the set of algorithms that are necessary to ensure the correctness of our code.

Figure 25 – EWOTS Structures

```
pub struct Wots<'a> {
    w: usize,
    s_key: SecretKey<'a>,
}

pub struct WotsSign {
    sign: [[u8;wots_const::N];wots_const::L],
    p_key: [u8;wots_const::N],
}

struct SecretKey<'a> {
    seed: &'a mut u64,
    s_key: [[u8;wots_const::N];wots_const::L],
}
```

Source: Original

Figure 26 – EWOTS Digest

```
#[cfg(feature = "GIMLI")]
fn digest(data: &[u8]) -> [u8; wots_const::N] {
    let mut hasher = Gimli::new().unwrap();
    let digest: [u8; wots_const::N] = hasher.finalize(data).into();
    digest
}
```

Source: Original

5.3.1 Key Pair

We generate secret keys based on the number of blocks through a PRNG function from a seed given by the user. We store both the seed and the private key to speed up the key generation process, but an alternative is to generate each private key on demand while storing the current key state, this, however introduces a few ownership problems that would have to be dealt with during key generation.

Following the WOTS algorithm we generate the public key by chaining the application of our hash function. We store the digest of the public key to save space without losing much efficiency.

Figure 27 – EWOTS Secret Key

```
    fn new(seed: &mut u64) -> Result<SecretKey, &'static str> {
        ...
        for i in 0..wots_const::L {
            aux = rng.rand_u64();
            ...
        }
```

Source: Original

Figure 28 – EWOTS Public Key

```
fn gen_pkey(&self) -> [u8;wots_const::N] {
        let mut out: [[u8;32];wots_const::L] = [[0;32];wots_const::L];
        for i in 0..wots_const::L {
            out[i] = chain_digest(&self.s_key.s_key[i], wots_const::W -
   1);
        }
        let pk = digest_pkey(&out);
        pk
    }
```

Source: Original

5.3.2 Signing

To sign a message, we first hash it so we can set some variables to a constant value and generate the key pair. We then allocate the necessary space for our signature as a static array, since this array size can be known at compile-time Cargo can optimize its location in memory. We calculate the `base w` representation of the digest and the checksum. The checksum function returns the correct `base w` representation for itself, so we can simply concatenate both arrays. We iterate through both digest and checksum to pass ownership to our auxiliary array `concat`. We then iterate through the secret keys to generate our signature, since our chain digest assumes at least one application of hash, we need to verify that the number of hash applications is higher than zero. Since we have that $H^0(x) = x$ for a hash function H we simply set the corresponding part of the secret key to the signature.

5.3.3 Verify

The verifying steps are similar to the signature, going through the same steps to generate the final array of (`digest`||`checksum`). We then use the same type of hash function to generate the public key through the signature and compress the public keys. This public key

Figure 29 – EWOTS Sign

```
pub fn sign(&self, data: &[u8]) -> Result<WotsSign, &'static str> {
        let mut concat: [u8;wots_const::L] = [0;wots_const::L];
        let digest = digest(&data);
        let p_key = self.gen_pkey();
        let mut sign: [[u8;32];wots_const::L] = [[0;32];wots_const::L];
        let base16 = to_basew(&digest);
        let cs = checksum(&base16);

        for (i,v) in base16.iter().chain(&cs).enumerate() {
            concat[i] = *v;
        }
        for (i,it) in concat.iter().enumerate() {
          if *it > 0 {
                sign[i] = chain_digest(&self.s_key.s_key[i], *it as
   usize);
            } else {
                sign[i] = self.s_key.s_key[i];
            }
        }
        Ok( WotsSign { sign, p_key } )
    }
```

Source: Original

is compared to the published public key byte by byte, returning either `true` or `false` when finished. Every object created in this function is deleted since we go out of scope.

Figure 30 – EWOTS Verify

```
pub fn verify(&self, data: &[u8]) -> bool {
        ...
        for (i,it) in concat.iter().enumerate() {
            if *it == 15 {
                verify[i] = self.sign[i];
            } else {
                verify[i] = chain_digest(&self.sign[i], (wots_const::W-*
  it as usize-1 as usize) as usize);
            }
        }
        let pk = digest_pkey(&verify);
        self.p_key.iter().eq(pk.iter())
```

Source: Original

Figure 31 – EWOTS debug output

```
++++++++++++++++++++++++++++++++++++++++++++++
Debugging WOTS with the following parameters:
Hash: "GIMLI"
Security: 32 bytes
Winternitz parameter: 16
L1: 64
L2: 3
L: 67
++++++++++++++++++++++++++++++++++++++++++++++

#=#=#=# Sign #=#=#=#
Data:
889bd1c3a01590f322db1d751b25976
3a72a9f570ecd352bc8450799f3178

Public Key:
d19da30f1f9576e6eb6ebd5336aa3e8
de6559ff047c6f6e18bc61d80ce35b3

Checksum: 5d2

Concatenated message:
889bd1c3a01590f322db1d0751b25976
3a072a9f5700ecd352bc8450799f31785d2

#=#=#=# Verify #=#=#=#
Data:
889bd1c3a01590f322db1d751b25976
3a72a9f570ecd352bc8450799f3178

Restored Public Key Y:
d19da30f1f9576e6eb6ebd5336aa3e8
de6559ff047c6f6e18bc61d80ce35b3

Signature: true
```

Source: Original

6 CONCLUSION

We set out to implement and combine two solutions for post-quantum cryptography, a tried-and-true scheme that is key in the modern hash-based signature scheme world, WOTS, and Gimli, a new lightweight cryptographic hash function from the current NIST's lightweight cryptography competition. As an addendum, we also aimed for this implementation to be portable and to retain as much as possible of Rust's safety guarantees. We have implemented both WOTS and Gimli using Rust as seen in section 5. by not using any `unsafe` code blocks, we keep the full certainty of Rust's intrinsic safety assurance, leading to a code free of race conditions and memory concerns. Restricting our external crates to `core` only creates we are able to require the minimum amount of implementations possible from the target architectures. Since `core` only crates are a small sub-set of the default library, in theory, any microprocessor that can rust Rust and has enough memory is able to run EWOTS. The use of `features` made EWOTS flexible enough to be integrated with more implementations of hash functions and constants.

While the implementations of WOTS and Gimli have been successful, some decisions that were made during the development phase of this project can be revisited to shift the focus from one priority to another, in particular, two decisions stand out. The first, from the WOTS implementations, is that we decided to store the secret keys and not only the seed. This improves the speed, simplifies the code, and avoids tricky ownership problems, the alternative would lower the amount of memory required to sign messages. The second decision was similar, as we keep track of the Gimli state we trade a little bit of space for convenience, the alternative would be to pass the Gimli state around when executing each step.

An aspect that was left out of this research is that of energy consumption. Mostly due to our lack of devices to measure. However, energy consumption is a very important metric when dealing with embedded devices and should be considered as a next step in measuring Rust's viability in an embedded scenario. Another worthwhile measurement would be to compare the signature timing when using other primitives such as SHA-2, SHA-3 and other finalists from the lightweight cryptography competition.

6.1 FUTURE WORK

While we focused on creating a safe implementation of WOTS and Gimli by using Rust's static analysis, there are quite a few assembly implementations of commonly used steps in cryptography (e.g. integer rotations) that can be used without much loss of safety for a substantial gain in performance. This addition can be done by adding `feature` options that select the assembly implementations for the target architecture in compile time. To provide good portability, several of these assembly blocks would have to be implemented.

Hash-based signature schemes based on tree authentication usually rely on WOTS signatures in their leaf nodes for the underlying signature. XMSS is a signature scheme with WOTS signatures that have been approved by NIST. It was shown that it is possible to im-

plement SPHINCS, a hash-based signature scheme based on tree authentication in an ARM processor (HÜLSING; RIJNEVELD; SCHWABE, 2016). SPHINCS+ is a similar version that uses WOTS signatures (BERNSTEIN et al., 2019). Implementing these schemes could be the next step for EWOTS to grow.

BIBLIOGRAPHY

ABERCROMBIE, J. **The Blade Itself**. [S.l.: s.n.], 2006.

BERNSTEIN, D. J. et al. The sphincs+ signature framework. In: **Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security**. New York, NY, USA: [s.n.], 2019. (CCS '19), p. 2129–2146. ISBN 9781450367479.

BERNSTEIN, D. J. et al. Gimli: a cross-platform permutation. In: SPRINGER. **International Conference on Cryptographic Hardware and Embedded Systems**. [S.l.], 2017. p. 299–320.

BERTONI, G. et al. Sponge functions. In: CITESEER. **ECRYPT hash workshop**. [S.l.], 2007. v. 2007, n. 9.

BERTONI, G. et al. Keccak. In: SPRINGER. **Annual international conference on the theory and applications of cryptographic techniques**. [S.l.], 2013. p. 313–314.

BERTONI, G. et al. The keccak reference. 2011.

BONEH, D. et al. Twenty years of attacks on the rsa cryptosystem. **Notices of the AMS**, v. 46, n. 2, p. 203–213, 1999.

DIFFIE, W.; HELLMAN, M. New directions in cryptography. **IEEE Trans. Inf. Theor.**, v. 22, n. 6, p. 644–654, set. 1976. ISSN 0018-9448.

DWORKIN, M. J. Sha-3 standard: Permutation-based hash and extendable-output functions. 2015.

GEBOTYS, C. H. **Security in embedded devices**. [S.l.]: Springer Science & Business Media, 2009.

GOLDREICH, O. **Foundations of Cryptography: Volume 1**. USA: Cambridge University Press, 2006. ISBN 0521035368.

GROSS, H.; MANGARD, S.; KORAK, T. An efficient side-channel protected aes implementation with arbitrary protection order. In: SPRINGER. **Cryptographers' Track at the RSA Conference**. [S.l.], 2017. p. 95–112.

GROVER, L. K. A fast quantum mechanical algorithm for database search. In: **Proceedings of the twenty-eighth annual ACM symposium on Theory of computing**. [S.l.: s.n.], 1996. p. 212–219.

HÜLSING, A.; RIJNEVELD, J.; SCHWABE, P. Armed sphincs. In: **Public-Key Cryptography–PKC 2016**. [S.l.]: Springer, 2016. p. 446–470.

INFORMATION Theory and Entropy. In: MODEL Based Inference in the Life Sciences: A Primer on Evidence. New York, NY: Springer New York, 2008. p. 51–82. ISBN 978-0-387-74075-1.

KATAGI, M.; MORIAI, S. et al. Lightweight cryptography for the internet of things. **Sony Corporation**, v. 2008, p. 7–10, 2008.

KOSTYA. **Rust benchmarks**. 2021. https://web.archive.org/web/20210517000134/https://github.com/kostya/benchmarks. Accessed: 2021-03-10.

LAMPORT, L. **Constructing digital signatures from a one-way function**. [S.l.], 1979.

LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: **Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)**. Palo Alto, California: [s.n.], 2004.

LIFTHRASIIR. **Why is a Rust executable large?** 2021. https://web.archive.org/web/20210515040228/https://lifthrasiir.github.io/rustlog/why-is-a-rust-executable-large.html. Accessed: 2021-03-10.

MANIFAVAS, C. et al. Lightweight cryptography for embedded systems – a comparative analysis. In: GARCIA-ALFARO, J. et al. (Ed.). **Data Privacy Management and Autonomous Spontaneous Security**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 333–349. ISBN 978-3-642-54568-9.

MENEZES, A. J.; OORSCHOT, P. C. van; VANSTONE, S. A. **Handbook of Applied Cryptography**. CRC Press, 2001. Disponível em: http://www.cacr.math.uwaterloo.ca/hac/.

MERKLE, R. C. A certified digital signature. In: SPRINGER. **Conference on the Theory and Application of Cryptology**. [S.l.], 1989. p. 218–238.

NIST. **Lightweight Cryptography**. 2021. https://web.archive.org/web/20210517000141/https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates. Accessed: 2021-03-22.

PERIN, L. P. et al. Tuning the winternitz hash-based digital signature scheme. **2018 IEEE Symposium on Computers and Communications (ISCC)**, p. 00537–00542, 2018.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. **Commun. ACM**, New York, NY, USA, v. 21, n. 2, p. 120–126, fev. 1978. ISSN 0001-0782.

SHOR, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. **SIAM review**, v. 41, n. 2, p. 303–332, 1999.

VIEGA, J.; THOMPSON, H. The state of embedded-device security (spoiler alert: It's bad). **IEEE Security & Privacy**, v. 10, n. 5, p. 68–70, 2012.

APPENDIX A – GIMLI PERMUTATION ALGORITHM

```
Input: s = (s_{i,j}) ∈ W^{3×4}
Output: GIMLI(s) = (s_{i,j}) ∈ W^{3×4}
  for r from 24 downto 1 inclusive do
    for j from 0 to 3 inclusive do
      x ← s_{0,j} ⋘ 24                                   ▷ SP-box
      y ← s_{1,j} ⋘ 9
      z ← s_{2,j}
      s_{2,j} ← x ⊕ (z ≪ 1) ⊕ ((y ∧ z) ≪ 2)
      s_{1,j} ← y ⊕    x    ⊕ ((x ∨ z) ≪ 1)
      s_{0,j} ← z ⊕    y    ⊕ ((x ∧ y) ≪ 3)
    end for
                                                         ▷ linear layer
    if r mod 4 = 0 then
      s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3} ← s_{0,1}, s_{0,0}, s_{0,3}, s_{0,2}   ▷ Small-Swap
    else if r mod 4 = 2 then
      s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3} ← s_{0,2}, s_{0,3}, s_{0,0}, s_{0,1}   ▷ Big-Swap
    end if

    if r mod 4 = 0 then
      s_{0,0} = s_{0,0} ⊕ 0x9e377900 ⊕ r                 ▷ Add constant
    end if
  end for
  return (s_{i,j})
```

Algorithm 1 – GIMLI permutation. From (BERNSTEIN et al., 2017)

APPENDIX B – EWOTS

Listing B.1 – EWOTS Cargo file (`Cargo.toml`)

```
[package]
authors = ["Lucas␣Mayr␣de␣Athayde␣<lucas.mayr.athayde@gmail.com>"]
edition = "2018"
readme = "README.md"
name = "ewots"
version = "0.1.0"
license = "MIT"
description = "This␣is␣a␣learning␣project␣for␣both␣Embedded␣and␣Rust␣
   development␣focusing␣on␣Cortex-M4."
exclude = [
    "examples/*",
    "target/*",
]

[features]
default = ["NO_DEBUG", "NO_DEBUG_SKEY"]
16 = []
SHA256 = []
GIMLI = []
DEBUG = []
NO_DEBUG = []
DEBUG_SKEY = []
NO_DEBUG_SKEY = []


[dependencies]
gimli = { path = "./gimli" }
sha2 = { version="0.9", default-features = false }
oorandom = "11.1.3"
cortex-m = "0.6.0"
cortex-m-rt = "0.6.10"
cortex-m-semihosting = "0.3.3"
panic-halt = "0.2.0"

[lib]
```

```
[profile.release]
codegen-units = 1
debug = true
lto = true
```

Listing B.2 – EWOTS Constants (`ewots_const.rs`)

```
// DEBUG
#[cfg(feature = "DEBUG")]
pub const DEBUG: bool = true;
#[cfg(feature = "NO_DEBUG")]
pub const DEBUG: bool = false;
#[cfg(feature = "DEBUG_SKEY")]
pub const DEBUG_SKEY: bool = true;
#[cfg(feature = "NO_DEBUG_SKEY")]
pub const DEBUG_SKEY: bool = false;


// HASH FUNCTION
// SHA-256
#[cfg(feature = "SHA256")]
pub const N: usize = 32;
#[cfg(feature = "SHA256")]
pub const HASH: &str = "SHA256";
// GIMLI
#[cfg(feature = "GIMLI")]
pub const N: usize = 32;
#[cfg(feature = "GIMLI")]
pub const HASH: &str = "GIMLI";

// WOTS 16
//#[cfg(feature = "16")]
//pub const M: usize = 256;
#[cfg(feature = "16")]
pub const W: usize = 16;
#[cfg(feature = "16")]
pub const LOG2W: usize = 4;
#[cfg(feature = "16")]
pub const L: usize = 67;
#[cfg(feature = "16")]
```

```
pub const L1: usize = 64;
#[cfg(feature = "16")]
pub const L2: usize = 3;
```

```
#![no_std]

//crypto;
extern crate oorandom;
use oorandom::Rand64;

use sha2::{Sha256, Digest};

use gimli::Gimli;

extern crate core;
use cortex_m_semihosting::{hprintln, hprint};

// WOTS Constants
mod wots_const;

// STRUCTS
#[allow(dead_code)]
pub struct Wots<'a> {
    w: usize,
    s_key: SecretKey<'a>,
}

pub struct WotsSign {
    sign: [[u8;wots_const::N];wots_const::L],
    p_key: [u8;wots_const::N],
}

#[allow(dead_code)]
struct SecretKey<'a> {
    seed: &'a mut u64,
    s_key: [[u8;wots_const::N];wots_const::L],
}

// IMPLEMENTATIONS
impl WotsSign {
```

```
    pub fn verify(&self, data: &[u8]) -> bool {
        if wots_const::DEBUG {
            hprintln!("\n#=#=#=#␣Verify␣#=#=#=#").unwrap();
        }
        let mut concat: [u8;wots_const::L] = [0;wots_const::L];
        let digest = digest(&data);

        if wots_const::DEBUG {
            hprint!("Data␣hash:␣").unwrap();
            for i in &digest {
                hprint!("{:x?}", i).unwrap();
            }
            hprintln!().unwrap();
        }

        let base16 = to_basew(&digest);
        let cs = checksum(&base16);

        for (i,v) in base16.iter().chain(&cs).enumerate() {
            concat[i] = *v;
        }
        let mut verify: [[u8;32];wots_const::L] = [[0;32];wots_const::L
];

        for (i,it) in concat.iter().enumerate() {
            if *it == 15 {
                verify[i] = self.sign[i];
            } else {
                verify[i] = chain_digest(&self.sign[i], (wots_const::W-*
it as usize-1 as usize) as usize);
            }
        }

        let pk = digest_pkey(&verify);
        if wots_const::DEBUG {
            hprint!("Restored␣Public␣Key␣Y:␣").unwrap();
            for i in &pk {
                hprint!("{:x?}", i).unwrap();
            }
```

```
            hprintln!().unwrap();
        }
        if wots_const::DEBUG {
            hprintln!("#=#=#=#␣Done␣#=#=#=#").unwrap();
        }

        self.p_key.iter().eq(pk.iter())

    }
}

impl Wots<'_> {
    pub fn new<'a>(seed: &'a mut u64) -> Result<Wots<'a>, &'static str>
   {
        if wots_const::DEBUG {
            hprintln!("\n++++++++++++++++++++++++++++++++++++++++++++++")
   .unwrap();
            hprintln!("Debugging␣WOTS␣with␣the␣following␣parameters:").
   unwrap();
            hprintln!("Hash:␣{:?}", wots_const::HASH).unwrap();
            hprintln!("Security:␣{:?}␣bytes", wots_const::N).unwrap();
            hprintln!("Winternitz␣parameter:␣{:?}",wots_const::W).unwrap
   ();
            hprintln!("L1:␣{:?}", wots_const::L1).unwrap();
            hprintln!("L2:␣{:?}", wots_const::L2).unwrap();
            hprintln!("L:␣{:?}", wots_const::L).unwrap();
            hprintln!("++++++++++++++++++++++++++++++++++++++++++++").
   unwrap();
        }

        if wots_const::DEBUG_SKEY {
            hprintln!("\n#=#=#=#␣Secret␣Key␣#=#=#=#").unwrap();
        }
        let s_key = SecretKey::new(seed).unwrap();

        if wots_const::DEBUG_SKEY {
            hprintln!("#=#=#=#␣Done␣#=#=#=#").unwrap();
        }
        Ok( Wots{ w:wots_const::W, s_key } )
```

```
    }

    pub fn sign(&self, data: &[u8]) -> Result<WotsSign, &'static str> {
        if wots_const::DEBUG {
            hprintln!("\n#=#=#=#␣Sign␣#=#=#=#").unwrap();
        }
        let mut concat: [u8;wots_const::L] = [0;wots_const::L];

        let digest = digest(&data);

        if wots_const::DEBUG {
            hprint!("Data␣hash:␣").unwrap();
            for i in &digest {
                hprint!("{:x?}", i).unwrap();
            }
            hprintln!().unwrap();
        }

        let p_key = self.gen_pkey();

        if wots_const::DEBUG {
            hprint!("Public␣Key␣Y:␣").unwrap();
            for i in &p_key {
                hprint!("{:x?}", i).unwrap();
            }
            hprintln!().unwrap();
        }

       let mut sign: [[u8;32];wots_const::L] = [[0;32];wots_const::L];

        let base16 = to_basew(&digest);

        if wots_const::DEBUG {
            hprint!("Base16:␣").unwrap();
            for i in &base16 {
                hprint!("{:x?}", i).unwrap();
            }
            hprintln!().unwrap();
        }
```

```
        let cs = checksum(&base16);

        if wots_const::DEBUG {
            hprint!("Checksum:␣").unwrap();
            for i in &cs {
                hprint!("{:x?}", i).unwrap();
            }
            hprintln!().unwrap();
        }

        for (i,v) in base16.iter().chain(&cs).enumerate() {
            concat[i] = *v;
        }
        if wots_const::DEBUG {
            hprint!("concat:␣").unwrap();
            for i in &concat {
                hprint!("{:x?}", i).unwrap();
            }
            hprintln!().unwrap();
        }

        for (i,it) in concat.iter().enumerate() {
            if *it > 0 {
                sign[i] = chain_digest(&self.s_key.s_key[i], *it as
    usize);
            } else {
                sign[i] = self.s_key.s_key[i];
            }
        }
        if wots_const::DEBUG {
            hprintln!("=+=+=␣Done␣=+=+=").unwrap();
        }
        Ok( WotsSign { sign, p_key } )
    }


    fn gen_pkey(&self) -> [u8;wots_const::N] {
```

```
        let mut out: [[u8;32];wots_const::L] = [[0;32];wots_const::L];
        for i in 0..wots_const::L {
            out[i] = chain_digest(&self.s_key.s_key[i], wots_const::W -
    1);
        }
        let pk = digest_pkey(&out);
        pk
    }
}

impl SecretKey<'_> {
    fn new(seed: &mut u64) -> Result<SecretKey, &'static str> {
        let mut out: [[u8;wots_const::N];wots_const::L] = [[0;wots_const
    ::N];wots_const::L];
        let mut rng = Rand64::new(*seed as u128);
        let mut hasher = Sha256::new();
        let mut aux;
        for i in 0..wots_const::L {
            aux = rng.rand_u64();
            hasher.update(aux.to_ne_bytes());
            out[i] = hasher.finalize_reset().into();
        }
        if wots_const::DEBUG_SKEY {
            for (i, key) in out.iter().enumerate() {
                hprint!("s_key␣{:?}:␣", i).unwrap();
                for byte in key {
                    hprint!("{:x?}", byte).unwrap();
                }
                hprintln!().unwrap();
            }
        }
        //hprintln!("S_KEY: {:?}", out);
        Ok( SecretKey{ seed, s_key:out } )
    }
}


// HASH
// GIMLI
```

```
#[cfg(feature = "GIMLI")]
fn chain_digest(data: &[u8], it: usize) -> [u8; wots_const::N] {
    let mut hasher = Gimli::new().unwrap();
    let digest: [u8; wots_const::N] = hasher.chain_digest(data, it).into
   ();
    digest
}

#[cfg(feature = "GIMLI")]
fn digest(data: &[u8]) -> [u8; wots_const::N] {
    let mut hasher = Gimli::new().unwrap();
    let digest: [u8; wots_const::N] = hasher.finalize(data).into();
    digest
}

#[cfg(feature = "GIMLI")]
fn digest_pkey(data: &[[u8;wots_const::N];wots_const::L]) -> [u8;
   wots_const::N] {
    let mut hasher = Gimli::new().unwrap();
    let mut concat: [u8;wots_const::N*wots_const::L] = [0;wots_const::N*
   wots_const::L];

    let mut i = 0;
    for pk in data {
        for bt in pk {
            concat[i] = *bt;
            i += 1;
        }
    }

    let digest: [u8; wots_const::N] = hasher.finalize(&concat).into();
    digest
}

// SHA256
#[cfg(feature = "SHA256")]
fn chain_digest(data: &[u8], it: usize) -> [u8; wots_const::N] {
    let mut hasher = Sha256::new();
    hasher.update(&data);
```

```
    let mut digest: [u8; wots_const::N] = hasher.finalize_reset().into()
    ;
    for _ in 1..it {
        hasher.update(&digest);
        digest = hasher.finalize_reset().into();
    }
    digest
}

#[cfg(feature = "SHA256")]
fn digest(data: &[u8]) -> [u8; wots_const::N] {
    let mut hasher = Sha256::new();
    hasher.update(&data);
    let digest: [u8; wots_const::N] = hasher.finalize().into();
    digest
}

#[cfg(feature = "SHA256")]
fn digest_pkey(data: &[[u8;wots_const::N];wots_const::L]) -> [u8;
    wots_const::N] {
    let mut hasher = Sha256::new();
    for d in data {
        hasher.update(&d);
    }
    let digest: [u8; wots_const::N] = hasher.finalize().into();
    digest
}

// WOTS16
#[cfg(feature = "16")]
fn to_basew(data: &[u8]) -> [u8;wots_const::L1] {
    let mut out: [u8;wots_const::L1] = [0;wots_const::L1];

    let mut it = 0;
    for i in data.iter() {
        out[it] = i >> wots_const::LOG2W;
        out[it+1] = (i << wots_const::LOG2W) >> wots_const::LOG2W;

        it += 2;
```

```
    }
    out
}

#[cfg(feature = "16")]
fn checksum(data: &[u8;wots_const::L1]) -> [u8;wots_const::L2] {
    let mut out: [u8;wots_const::L2] = [0;wots_const::L2];
    let mut aux: u16 = 0;

    for i in data {
        aux += wots_const::W as u16 - 1 - *i as u16;
    }
    let aux_bytes = aux.to_ne_bytes();
    out[0] = aux_bytes[0] >> wots_const::LOG2W;
    out[1] = (aux_bytes[0] << wots_const::LOG2W) >> wots_const::LOG2W;
    out[2] = aux_bytes[1];

    out
}
```

Listing B.3 – Gimli Cargo file (`Cargo.toml`)

```
[package]
name = "gimli"
version = "0.1.0"
authors = ["Lucas Mayr <lucas.mayr.athayde@gmail.com>"]
edition = "2018"

[dependencies]
cortex-m = "0.6.0"
cortex-m-rt = "0.6.10"
cortex-m-semihosting = "0.3.3"
```

Listing B.4 – Gimli lib (`lib.rs`)

```
#![no_std]

use cortex_m_semihosting::{hprintln, hprint};

pub struct Gimli {
    state: [u32;12]
```

```
}

impl Gimli {

    pub fn new() -> Result<Gimli, &'static str> {
        let state: [u32;12] = [0;12];

        Ok(Gimli { state })
    }

    fn absorb_block(&mut self, data: &[u8]) {
        self.state[0] ^= (data[0] as u32) | ((data[1] as u32) << 8) | ((
    data[2] as u32) << 16) | ((data[3] as u32) << 24);// as u32;
        self.state[1] ^= (data[4] as u32) | ((data[5] as u32) << 8) | ((
    data[6] as u32) << 16) | ((data[7] as u32) << 24);// as u32;
        self.state[2] ^= (data[8] as u32) | ((data[9] as u32) << 8) | ((
    data[10] as u32) << 16) | ((data[11] as u32) << 24);// as u32;
        self.state[3] ^= (data[12] as u32) | ((data[13] as u32) << 8) |
    ((data[14] as u32) << 16) | ((data[15] as u32) << 24);// as u32;

        self.gimli();
    }

    fn squeeze(&self) -> [u8;16] {
        let mut out: [u8;16] = [0;16];

        out[0] = self.state[0].to_le_bytes()[0];
        out[1] = self.state[0].to_le_bytes()[1];
        out[2] = self.state[0].to_le_bytes()[2];
        out[3] = self.state[0].to_le_bytes()[3];
        /////
        out[4] = self.state[1].to_le_bytes()[0];
        out[5] = self.state[1].to_le_bytes()[1];
        out[6] = self.state[1].to_le_bytes()[2];
        out[7] = self.state[1].to_le_bytes()[3];
        /////
        out[8] = self.state[2].to_le_bytes()[0];
        out[9] = self.state[2].to_le_bytes()[1];
        out[10] = self.state[2].to_le_bytes()[2];
```

```
        out[11] = self.state[2].to_le_bytes()[3];
        /////
        out[12] = self.state[3].to_le_bytes()[0];
        out[13] = self.state[3].to_le_bytes()[1];
        out[14] = self.state[3].to_le_bytes()[2];
        out[15] = self.state[3].to_le_bytes()[3];

        out
    }

    fn gimli(&mut self){
        let mut x;
        let mut y;
        let mut z;
        let mut aux: [u32;4];
        for r in (1..25).rev() {
            for j in 0..4 {
                x = self.state[j].rotate_left(24);
                y = self.state[j+4].rotate_left(9);
                z = self.state[j+8];
                ////
                self.state[j+8] = x ^ (z << 1) ^ ((y & z) << 2);
                self.state[j+4] = y ^ x ^ ((x | z) << 1);
                self.state[j] = z ^ y ^ ((x & y) << 3);
            }
            if r%4 == 0 {
                aux = [self.state[1],self.state[0],self.state[3],self.
state[2]];
                self.state[0] = aux[0];
                self.state[1] = aux[1];
                self.state[2] = aux[2];
                self.state[3] = aux[3];

                self.state[0] ^= 0x9e377900 ^ r;

            } else if r%4 == 2 {
                aux = [self.state[2],self.state[3],self.state[0],self.
state[1]];
                self.state[0] = aux[0];
```

```
                self.state[1] = aux[1];
                self.state[2] = aux[2];
                self.state[3] = aux[3];

            }
        }
    }

    pub fn finalize(&mut self, data: &[u8]) -> [u8;32] {
        let rate = 16;
        let mut out: [u8;32] = [0;32];

        let len = data.len();
        let blocks = len/rate;
        let bytes_left = len%rate;

       for i in 0..blocks {
            self.absorb_block(&data[16*i..(16*(i+1))]);
        }

        let mut last_block: [u8;16] = [0;16];

        let mut start = len - bytes_left;
        for i in 0..bytes_left {
            last_block[i] = data[start];
            start += 1;
        }
        last_block[bytes_left] = 0x1;

        self.state[11] ^= 0x01000000;
        self.absorb_block(&last_block);

        let h1 = self.squeeze();
        self.gimli();
        let h2 = self.squeeze();

        for (to, from) in out.iter_mut().zip(h1.iter().chain(h2.iter()))
    { *to = *from };
```

```
        out
    }

    pub fn chain_digest(&mut self, data: &[u8], it: usize) -> [u8;32] {
        let mut out: [u8;32] = [0;32];
        out = self.finalize(&data);
        self.reset();

        for i in 1..it {
            out = self.finalize(&out);
            self.reset();
        }
        out
    }

    pub fn reset(&mut self) {
        self.state = [0;12];
    }
}
```

APPENDIX C – SBC PAPER

Implementing a library to provide Winternitz signatures with lightweight primitive using the Rust Programming Language

Lucas Mayr de Athayde [1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 5040 – 88040-900 – Florianópolis – SC – Brazil

Abstract. *Most conventional algorithms depend on the hardness of integer factorization and discrete logarithm mathematical problems. Shor (1999) presents a method to solve both of these problems using a large enough quantum computer [Shor 1999]. Hash-based signature schemes are shown to be secure against attacks using Shor's algorithms [Merkle 1989]. Winternitz signature scheme (*WOTS*) was introduced as a hash-based signature scheme [Merkle 1989]. Gimli is a lightweight option for building hash functions [Bernstein et al. 2017]. In this paper we implement EWOTS, a library containing the Winternitz signature scheme and the Gimli hash function through the Rust programming language for the Cortex-M4.*

1. Introduction

Classical signature schemes like RSA base their security on the hardness of its one-way function, which in turn depends on the hardness of the integer factorization and discrete logarithm problems. Those problems are believed to be NP-Complete and computationally impractical to solve on a classical computer. One way to break the security of these signature schemes is to find an algorithm that solves those problems in polynomial time.

Peter W. Shor shows algorithms that solve these problems on a quantum computer [Shor 1999], effectively creating a scenario where those signature schemes are broken. Believing that quantum computers are not a thought exercise but something that will come to be, his work creates a deadline where there needs to be a shift from classical signature schemes to *post-quantum* signature schemes.

Hash-based cryptography has been proposed as an alternative to classical signature schemes. The security of hash-based signatures depends on the safety of conventional cryptographic functions, avoiding the delays and costs of a new certification effort [Merkle 1989].

Hash functions are, by definition, easy to compute but hard to invert functions. If we have a hash function F such as that $y = F(x)$, then given knowledge of F and x we can compute y with ease, however, if we are given y and F it shouldn't be feasible to find x.

Winternitz one time signature scheme (*WOTS*) was introduced as an alternative to Lamport-Diffie one time signature scheme (*LD-OTS*), where both signatures and public key generation involve multiple applications of the one-way function [Merkle 1989]. In LD-OTS we have $y = F(x)$, public (F, y) and a private x. On WOTS we now have $y = F^w(x)$, (F, y) is still public and x is still private, but there is the introduction of the public w parameter, called Winternitz parameter, we apply the one-way function F a

number w times and compute the public key. To sign a message using WOTS we compute $F^m(x)$ where m is the message to be signed. To verify a signature we can simply compute $y = F^{w-m}(x)$ and compare with the public key y. By signing blocks from the message instead of bit-by-bit we reduce the signature size.

As Merkle wrote in [Merkle 1989], cryptographic hash functions are the foundation of hash based cryptography schemes, and for that reason we would be remiss not to mention a few of the most used hash functions. SHA-2 is a staple when it comes to hash functions, defined by NIST in 2001, being mandatory for any application that requires collision resistance. Keccak [Bertoni et al. 2013] is a family of hash functions based on sponge constructions, where SHA-3 is a subset of. The main difference between SHA-2 and SHA-3 is that unlike SHA2, which uses a block cipher, SHA-3 and the Keccak family uses the sponge structure, making SHA-3 resistant to length extension attacks [Bertoni et al. 2011]. Another upcoming hash is based on Gimli [Bernstein et al. 2017], a 384-bit permutation that can be used to create hash functions while maintaining *cross-platform performance*, something that we also strive to achieve in this work.

In [Manifavas et al. 2014], the authors analyse and compare various asymmetric cryptography schemes for embedded devices, including classical RSA systems and alternative ones such as hash, lattices and code based. Special note should be given to hash based schemes, found to be smaller in code size and faster than RSA. The authors also speculate that hash based schemes would only gain more ground with the evolution of lightweight hash function designs.

Rust as a language prides itself in its static analysis, byproduct of a design decision where variables cannot be shared, only borrowed or given to other functions. Rust also offers a clear way to incorporate existing C code into its own, resulting in a low-level enough language when necessary while maintaining the simplicity of high-level abstractions.

In this paper we implement hash-based signature schemes in pure embedded Rust trying to use as many *safe* blocks as possible, sacrificing some efficiency for portability between devices. For the purpose of testing, we have chosen to emulate a Cortex-M4 processor. Emulating a processor comes with restrictions, therefore, we won't be measuring energy, memory and time efficiency.

2. Primitives

2.1. Asymmetric Cryptography

Cryptography is the practice and study of techniques of securing information in a way that an adversary cannot meddle with the information without leaving traces. One of those cryptosystems is called *Asymmetric Cryptography* or *Public-key cryptography*. Introduced in [Diffie and Hellman 1976], the idea behind asymmetric cryptography is that if it is possible to have two distinct keys, one could be used to cipher documents while the other could be used to decipher messages, also called the public and private keys correspondingly. In this manner, the public key could be published to everyone, but only the holder of the secret key would be able to read the messages encrypted by the public key.

Public-key cryptosystems do not rely on a secure channel and remove the need for two parties to agree on the same private key, as is needed by symmetric cryptography, but

introduces the need to authenticate published public keys. Signing is a slow operation for large messages, in those cases the signature is only done to the hash output of the message and then appended to the original, to verify the signature, the user must first calculate the original hash.

Digital signature schemes work by using a private key in conjunction with a message to generate an output that can be verified using a public key. These schemes are usually comprised of three main steps, **Key generation**, which is responsible for creating the key pair, the **Message signing** itself and **Signature verification** which should provide the following set of assurances.

Authentication. When the private key of a message is bound to a user, verifying the signature of the document provides proof that the document was indeed signed by that user;

Integrity. Any changes made to the document after it was signed should invalidate the signature;

Non-Repudiation. The owner of the private key cannot deny having signed the document after the fact.

2.2. Hash functions

One-way functions are easy to compute but hard to invert functions that can receive an arbitrarily large input to generate an arbitrarily large output as long as it is easy to compute, and given the output of one such function, it should be hard enough to find out what the input was. The existence of this type of function has not yet been mathematically proven, but enough evidence suggests that they do. The existence of one-way functions would imply that the complexity classes P and NP are not equal [Goldreich 2006].

A **hash function** is a one-way function $F : \{0,1\}^* \rightarrow \{0,1\}^n$ where $n \in \mathbb{N}$ is a fixed output size. Fixing the output of the one-way function to n shrinks the co-domain from $\{0,1\}^*$ to $\{0,1\}^n$. From this restriction we can see that since $\{0,1\}^n$ is a subset of $\{0,1\}^*$, hash functions can compress arbitrarily large inputs to a smaller image, therefore creating unavoidable collisions.

The idea behind hash functions is that the compressed value can act as a fingerprint to the document, representing it with a fixed size. Digital signatures usually use hash functions to shorten the size of the message being signed. Instead of signing the whole document, The hash is signed in its place and the signature appended to the document.

Finally, to create a **cryptographic hash function**, we need to add a set of security properties on top of our hash function. To illustrate these properties let's define **H** as follows:

$$x \in X; \quad y \in Y; H : x \mapsto y; \quad y = H(x);$$

The first property is **Pre-image resistance**. Given H and y, it should be infeasible for an attacker to discover x. In other words, it should be hard to *invert* our hash function.

The second property, **Second Pre-image resistance**, tells us that given H and a known x, it should be *computationally hard* to find an x' such that $x' \neq x$ and

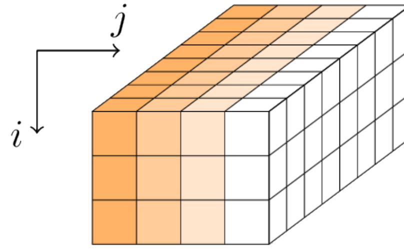


Figure 1. Gimli State Representation. From [Bernstein et al. 2017]

$H(x) = H(x')$. This property is especially important in digital signatures since a valid pair $(x', H(x))$ would mean a message had been forged under the key holder's name.

The third property, **Collision resistance**, says that, for a given H, find any pair (x, x') such that $x' \neq x$ and $H(x) = H(x')$. This is a more relaxed form of the second pre-image resistance property that aims to guarantee that a hash from a message is unique *enough*.

Lastly, it also desirable that our cryptographic hash functions also have the property known as the *Avalanche effect*, where modifying a single bit from the original message should, on average, modify half of the final hash output.

A sponge function is particular way to generalize hash functions [Bertoni et al. 2007]. They are by built iterating through a fixed-length permutation or transformation function f that operates on a state of b bits from a message with i blocks. The state of a sponge construction can be divided in 2 parts, the bitrate r and the capacity c such that $b = r + c$. At the beginning of the construction, the message is padded up to a multiple of b and the state is set to 0. The construction then proceeds to the absorbing phase and the squeezing phase.

In the absorb step, the input blocks are consumed through an *exclusive or* ($\oplus$) into the r-bit part of the state, between each input block, f is applied to the current state in between the blocks.

In the squeezing step, the user selects how many blocks are desired and the first r bits of the block are returned as output, interleaved by applications of f.

2.3. Gimli

Gimli [Bernstein et al. 2017] is a permutation designed with cross-platform performance in mind. It seeks to mainly provide energy efficiency, compactness, side-channel attack protection through its sponge construction and compactness. It can be used as a building block for block ciphers, stream ciphers, message authentication codes, and secure hash functions.

Gimli works by applying a sequence of rounds to a 384-bit state represented by a 32x3x4 matrix. As is with sponge functions, this state is initialized as zero. Each Gimli round is separated into three steps. The non-linear step consists of rotations, a non-linear function T, and swap operations. The linear step consists of two swap operations, and lastly, a constant addition. Let $r \in \mathbb{N}$ be the current Gimli round, $W \in \{0, 1\}^{32}$ be a word and $\{s \mid s \in W^{3x4}\}$ be the Gimli state.

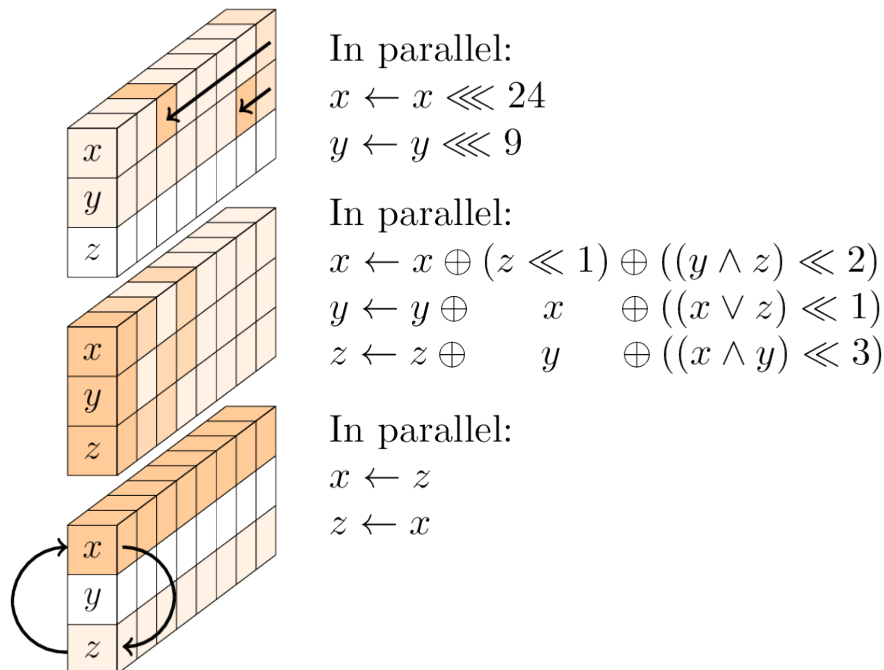


Figure 2. Gimli Non-linear layer. From [Bernstein et al. 2017]

The linear step has two types of swaps, called *big swap* and *small swap*. A small swap is done every four rounds starting from the first and a big swap is done every four rounds starting from the third.

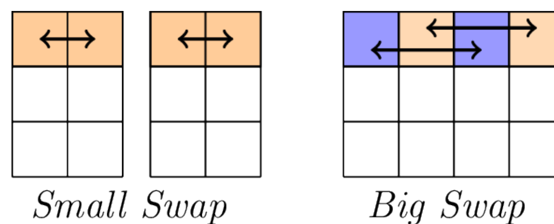


Figure 3. Gimli linear layer. From [Bernstein et al. 2017]

Creating a hash function from the Gimli permutation using the sponge function can be done by following the sponge construction detailed in the previous section. The only unique characteristic is the padding done, adding a single byte (`0x1`) to the last byte of the last block.

2.4. Winternitz signature scheme

Robert Winternitz proposed in 1979 an improvement to the Lamport-Diffie OTS scheme that aimed to reduce signature and key size by signing multiple bits at once instead of every single bit separately. This change trades time for space, the more bits signed at once, the longer it takes to generate keys, sign and verify messages. Just like Lamport-Diffie OTS scheme, WOTS uses a cryptographic hash function F.

The Winternitz parameter w corresponds to the level of compression we wish to utilize. To compress the message in blocks of 4 bits we would have $w = 16$. The larger this parameter, the smaller the resulting keys, but the longer it will take to sign and verify any document.

In the following, we present the process of signing and verifying a WOTS signature for a message M bits long.

Key pair generation. First we must choose the Winternitz parameter $w \in \mathbb{N}, w > 1$, Effectively defining our compression level.
The private key consists of L strings of n bits chosen at random.

$$X = (x_1, x_2, x_3, ..., x_{l-1}, x_l) \xleftarrow{\$} \{0,1\}^{(n,L)}$$

The public key is generated by applying the hash functions to itself $w - 1$ times for every x_i, giving us Y.

$$Y = (Y_1, Y_2, Y_3, ..., Y_L)$$
$$Y = F^{(w-1)}(x_1), F^{(w-1)}(x_2), F^{(w-1)}(x_3), \dots, F^{(w-1)}(x_L)$$

Where L_1, L_2 and L are obtained as follows:

$$L_1 = \left\lceil \frac{M}{\log_2(w)} \right\rceil, \; L_2 = \left\lfloor \frac{\log_2((L_1)(w-1))}{\log_2(w)} \right\rfloor + 1 \text{ and } L = L_1 + L_2.$$

L_1 represents the number of blocks required to sign the whole message. L_2 represents the minimum amount of blocks required to calculate and store the checksum of the message which will be explained shortly. It follows then that L is the total amount of blocks required for the signature process.

Message signing. We now sign a message by breaking it down into L_1 blocks represented in `base-w` $M = (m_1, m_2, m_3, ..., m_{L_1})$ where $m_i \in \{0, ..., w-1\}$.
Before we start signing our message we need to compute the checksum. To understand why this step is necessary let's imagine a message with a single block. The signing process happens by signing a message a few times and stopping somewhere between our secret key and our public key $F^0(x_i) \leq \sigma_i \leq F^{w-1}(x_i)$. We know that a hash function is computationally infeasible to invert, however, it is extremely easy for our attacker to compute $F(\sigma)$ and claim that we signed a message that we did not. To prevent this attack we compute a checksum that indicates the total remaining applications of our hash function. That means that to go *forward* in a block, the attacker would have to go *backward* on the checksum block.
We now calculate the checksum and represent it as *base-w*.

$$C = \sum_{i=1}^{L_1} (w - 1 - m_i)$$

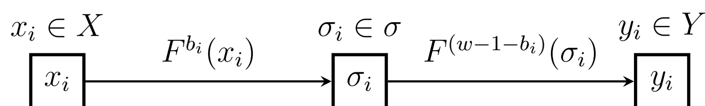
We concatenate both message and checksum $B = M||C = (b_1, b_2, b_3, ..., b_l)$ and can now sign the message by applying our hash function onto our corresponding x_i blocks, b_i times.

$$\sigma = (\sigma_1, \sigma_2, \sigma_3, ..., \sigma_L)$$
$$\sigma = (F^{b_1}(x_1), F^{b_2}(x_2), F^{b_3}(x_3), ..., F^{b_L}(x_L))$$

Signature verification. To verify the signature the user must recreate the base-w B string and compute the remaining hash functions to reach $w-1$ applications, comparing both the public key and the one just generated. If we have calculate the public key through the signature and compare it to the published public key Y we can verify that the signature is valid if those public keys match. Otherwise we know that the signature is false. This operation can be summarized in the following equation:

$$Y = (F^{w-1-b_1}(\sigma_1), F^{w-1-b_2}(\sigma_2)F^{w-1-b_3}(\sigma_3), ..., F^{w-1-b_L}(\sigma_L))$$
$$= (y_1, y_2, y_3, ..., y_L) = Y$$

Figure 4. Winternitz Signature Operations



As a note on key sizes, both the private and public keys can be further reduced. Private keys are often generated as needed by using a pseudo random number generator and storing only the seed. The public key can be concatenated and hashed.

3. EWOTS

We now present EWOTS, a `no_std` Rust crate that provides implementations of WOTS and Gimli that depend only on `core` crates. While debugging was done using specific Cortex-M4 crates, the `core` implementations do not rely on any external library and can theoretically be used on any device able to run Rust programs. The use of *features* to select implementations and constants makes it so that any new algorithm (e.g. a new hash function) can be easily added without interfering in the main functions as long as they have the same signature.

`Core` crate libraries present some problems as many high-level abstractions are only available in the standard crates, creating a more rigid environment. This effect creates a steeper curve to learning and using Rust as a language but it should not deter anyone from trying as there is plenty of material available for embedded Rust, both from the developers and the community. The gain we obtain from using only the `core` crate is seen in the binary size and the portability options.

3.1. Gimli implementation

We keep track of the Gimli state and modify it by using a struct that contains the current state of Gimli. Our functions and transformations do not return the state, but instead, alter it in place. While `Gimli struct` itself is public, the internal *state* is not, so a user can only alter its value by using the intended functions.

3.2. Gimli Permutation

Our Gimli permutation follows the code shown in [Bernstein et al. 2017] along with the reference implementations published by the authors. While every part is implemented in

Figure 5. Gimli Structure

```
pub struct Gimli {
    state: [u32;12]
}
```

the same function, we can see each layer presented in the original work by itself. The non-linear layer seen in listing 6 involves the rotation of bits in our word, luckily, Rust offers the `rotate_left` function that does just that, saving us the trouble of implementing it. The second half of the non-linear layer is a set of bit operations, modifying the original state.

Figure 6. Gimli Non-linear layer

```
    fn gimli(&mut self){
        ...
        // Non-linear layer
        x = self.state[j].rotate_left(24);
        y = self.state[j+4].rotate_left(9);
        z = self.state[j+8];
        self.state[j+8] = x ^ (z << 1) ^ ((y & z) << 2);
        self.state[j+4] = y ^ x ^ ((x | z) << 1);
        self.state[j] = z ^ y ^ ((x & y) << 3);
        ...
```

The second step, the linear layer, can be seen in figure 7. We shuffle the values into an auxiliary array and pass ownership of those values to the corresponding place of the state. Since we use every entry and don't repeat them, we can be sure no value will be lost to ownership. The final layer, constant addition, is done after a small swap.

3.3. Gimli Hash

The implementation of Gimli Hash was shown in the NIST submission for the latest lightweight cryptography competition [NIST 2021]. It follows the usual Keccak structure through the use of *Absorb* and *Squeeze* functions with the underlying *Gimli Permutation*. We chose to implement both the permutation and the hashing process in the same structure since they both rely in the current Gimli state. While the authors do not force a single number of rounds, they strongly recommend 16 rounds, and as such, we fixed the rate number to 16 to keep it as close to the proposal as possible. The first step to our hash seen in listing 8 is to find out how many blocks we will need to absorb and how many bytes will need to be padded in the last block. Afterward, we absorb each block individually except the last one (unless there is no need for padding), since *blocks* is rounded down.

Figure 7. Gimli Linear layer

```
    fn gimli(&mut self){
        ...
        // Small swap
        aux = [self.state[1],self.state[0],self.state[3],
   self.state[2]];
        self.state[0] = aux[0];
        self.state[1] = aux[1];
        self.state[2] = aux[2];
        self.state[3] = aux[3];
        ...
        // Big swap
        aux = [self.state[2],self.state[3],self.state[0],
   self.state[1]];
        self.state[0] = aux[0];
        self.state[1] = aux[1];
        self.state[2] = aux[2];
        self.state[3] = aux[3];
        ...
```

Figure 8. Gimli Hash

```
    pub fn finalize(&mut self, data: &[u8]) -> [u8;32] {
        ...
        let len = data.len();
        let blocks = len/rate;
        let bytes_left = len%rate;

        for i in 0..blocks {
            self.absorb_block(&data[16*i..(16*(i+1))]);
        }
        ...
```

The next step is to handle the last block. Because we are dealing with immutable object `data` we can not directly increase its size to match our rate, therefore we instantiate a new block filled with zeros and transfer the values from data to this auxiliary array and write `0x1` to its last byte to complete the padding and absorb it as normal.

Lastly, we squeeze our Gimli state twice with a Gimli permutation in between to generate the hash output. The squeeze function simply outputs 4 words from the top of the state into a single 16-byte array. These two squeezes are then concatenated to generate the final 32-byte hash.

Figure 9. Gimli hash output

```
pub fn finalize(&mut self, data: &[u8]) -> [u8;32] {
    ...
    let h1 = self.squeeze();
    self.gimli();
    let h2 = self.squeeze();

    for (to, from) in out.iter_mut().zip(h1.iter().
chain(h2.iter())) { *to = *from };
    out
}
```

3.4. Gimli Outputs

These tests were done using the reference code from the second round NIST submission and can be found in [NIST 2021]. As the Gimli permutation is the focal point of the submission, we chose to separate our tests into two distinct sections. These results can be seen in table 1.

To test the permutation algorithm we feed known Gimli states and perform the full 24 round Gimli permutation in both the reference code and EWOTS, comparing each word.

Recently NIST has published a new list of finalists for its lightweight cryptography competition, alas, Gimli was dropped out. However, other candidates stand out as having similar strengths as Gimli had, in particular, Xoodyak seems like an attractive alternative, co-designed by one of the original authors of AES, it seems to have remarkable performance, simplicity, and a small code base.

Table 1. Comparing Gimli Hash outputs

There's plenty for the both of us, may the best Dwarf win.
Reference Output
0xf456f2da6bf5bbded9b6033e63f7564537eec541c78f3c471eca11bffae90f55
EWOTS Output
0xf456f2da6bf5bbded9b6033e63f7564537eec541c78f3c471eca11bffae90f55

4. WOTS Implementation

Our implementations of WOTS introduces three structures, namely, *Wots* that contains the internal states necessary for the WOTS algorithm, *WotsSign* that handles the signature itself and the public key. The `wots_const` seen here are defined in compile time through the use of *features*. By defining these constants in compile time, we can avoid using larger objects such `Box<T>` to instead use a lower level static array, giving us smoother handling of these variables.

The digest algorithms can be set using the appropriate feature, in listing 10 we can see an example of GIMLI being used to generate a digest. Other algorithms can be

set as long as they respect the function signature. The same logic can be applied to any algorithm with the feature flag. Since we can only set one of the digest algorithms without causing compiler errors, by setting a feature we can guarantee not only one algorithm, but the set of algorithms that are necessary to ensure the correctness of our code.

Figure 10. EWOTS Digest

```
#[cfg(feature = "GIMLI")]
fn digest(data: &[u8]) -> [u8; wots_const::N] {
    let mut hasher = Gimli::new().unwrap();
    let digest: [u8; wots_const::N] = hasher.finalize(data
   ).into();
    digest
}
```

Key Pair. We generate secret keys based on the number of blocks through a PRNG function from a seed given by the user. We store both the seed and the private key to speed up the key generation process, but an alternative is to generate each private key on demand while storing the current key state, this, however introduces a few ownership problems that would have to be dealt with during key generation.
Following the WOTS algorithm we generate the public key by chaining the application of our hash function. We store the digest of the public key to save space without losing much efficiency.

Figure 11. EWOTS Public Key

```
fn gen_pkey(&self) -> [u8;wots_const::N] {
        let mut out: [[u8;32];wots_const::L] = [[0;32];
   wots_const::L];
        for i in 0..wots_const::L {
            out[i] = chain_digest(&self.s_key.s_key[i],
   wots_const::W - 1);
        }
        let pk = digest_pkey(&out);
        pk
    }
```

Signing. To sign a message, we first hash it so we can set some variables to a constant value and generate the key pair. We then allocate the necessary space for our signature as a static array, since this array size can be known at compile-time Cargo can optimize its location in memory. We calculate the `base w` representation of the digest and the checksum. The checksum function returns the correct `base w` representation for itself, so we can simply concatenate both arrays. We iterate through both digest and checksum to pass ownership to our auxiliary array `concat`. We then iterate through the secret keys to generate our signature, since

our chain digest assumes at least one application of hash, we need to verify that the number of hash applications is higher than zero. Since we have that $H^0(x) = x$ for a hash function H we simply set the corresponding part of the secret key to the signature.

Figure 12. EWOTS Sign

```
pub fn sign(&self, data: &[u8]) -> Result<WotsSign, &'
   static str> {
    ...
    if *it > 0 {
          sign[i] = chain_digest(&self.s_key.s_key[i], *it
    as usize);
      } else {
          sign[i] = self.s_key.s_key[i];
      }
        }
    Ok( WotsSign { sign, p_key } )
}
```

Verify. The verifying steps are similar to the signature, going through the same steps to generate the final array of `(digest||checksum)`. We then use the same type of hash function to generate the public key through the signature and compress the public keys. This public key is compared to the published public key byte by byte, returning either `true` or `false` when finished. Every object created in this function is deleted since we go out of scope.

Figure 13. EWOTS Verify

```
 pub fn verify(&self, data: &[u8]) -> bool {
        ...
        for (i,it) in concat.iter().enumerate() {
            if *it == 15 {
                verify[i] = self.sign[i];
            } else {
                verify[i] = chain_digest(&self.sign[i], (
   wots_const::W-*it as usize-1 as usize) as usize);
            }
        }
        let pk = digest_pkey(&verify);
        self.p_key.iter().eq(pk.iter())
```

5. Conclusion

We set out to implement and combine two solutions for post-quantum cryptography, a tried-and-true scheme that is key in the modern hash-based signature scheme world,

WOTS, and Gimli, a new lightweight cryptographic hash function from the current NIST's lightweight cryptography competition. As an addendum, we also aimed for this implementation to be portable and to retain as much as possible of Rust's safety guarantees. We have implemented both WOTS and Gimli using Rust. by not using any `unsafe` code blocks, we keep the full certainty of Rust's intrinsic safety assurance, leading to a code free of race conditions and memory concerns. Restricting our external crates to `core` only creates we are able to require the minimum amount of implementations possible from the target architectures. Since `core` only crates are a small sub-set of the default library, in theory, any microprocessor that can rust Rust and has enough memory is able to run EWOTS. The use of `features` made EWOTS flexible enough to be integrated with more implementations of hash functions and constants.

While the implementations of WOTS and Gimli have been successful, some decisions that were made during the development phase of this project can be revisited to shift the focus from one priority to another, in particular, two decisions stand out. The first, from the WOTS implementations, is that we decided to store the secret keys and not only the seed. This improves the speed, simplifies the code, and avoids tricky ownership problems, the alternative would lower the amount of memory required to sign messages. The second decision was similar, as we keep track of the Gimli state we trade a little bit of space for convenience, the alternative would be to pass the Gimli state around when executing each step.

An aspect that was left out of this research is that of energy consumption. Mostly due to our lack of devices to measure. However, energy consumption is a very important metric when dealing with embedded devices and should be considered as a next step in measuring Rust's viability in an embedded scenario. Another worthwhile measurement would be to compare the signature timing when using other primitives such as SHA-2, SHA-3 and other finalists from the lightweight cryptography competition.

References

Bernstein, D. J., Kölbl, S., Lucks, S., Massolino, P. M. C., Mendel, F., Nawaz, K., Schneider, T., Schwabe, P., Standaert, F.-X., Todo, Y., et al. (2017). Gimli: a cross-platform permutation. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 299–320. Springer.

Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. (2007). Sponge functions. In *ECRYPT hash workshop*, volume 2007. Citeseer.

Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. (2013). Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 313–314. Springer.

Bertoni, G., Peeters, M., Van Assche, G., et al. (2011). The keccak reference.

Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654.

Goldreich, O. (2006). *Foundations of Cryptography: Volume 1*. Cambridge University Press, USA.

Manifavas, C., Hatzivasilis, G., Fysarakis, K., and Rantos, K. (2014). Lightweight cryptography for embedded systems – a comparative analysis. In Garcia-Alfaro, J., Li-

oudakis, G., Cuppens-Boulahia, N., Foley, S., and Fitzgerald, W. M., editors, *Data Privacy Management and Autonomous Spontaneous Security*, pages 333–349, Berlin, Heidelberg. Springer Berlin Heidelberg.

Merkle, R. C. (1989). A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer.

NIST (2021). Lightweight cryptography. `https://web.archive.org/web/20210517000141/https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates`. Accessed: 2021-03-22.

Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332.