Bauhaus-Universität Weimar
Faculty of Media
Degree Program Computer Science and Media

Implementation of the Catena Password-Scrambling Framework

Master's Thesis

Sascha Schmidt
born 19.04.1988 in Los Alamos

Matriculation number: 80037

1st Reviewer: Prof. Dr. Stefan Lucks
2nd Reviewer: PD Dr. Andreas Jakoby

Submission date: 7th September 2015

Abstract

It has been established that passwords should, instead of being stored in the clear, be accompanied by a random value and then processed by a one-way hash function. The steadily increasing computational capabilities of all attackers has left a severe dent in the security of this approach. Nowadays, passwords should be hashed with a memory- and time-consuming algorithm that penalizes attacks which operate on less than the required memory. This thesis focuses on the implementation of one such algorithm, Catena, which not only resists all known attacks but also provides a convenient set of distinct properties.

The explanation of the preliminaries of password hashing is followed by an introduction of all major competitors in the field of memory- and time-consuming password hashing. The thesis then presents an accentuated summary of the specification of Catena. Finally, the three applications created during this thesis, a reference implementation of Catena and two related tools, are elaborated in detail. This includes rationales for the software design as well as optimized versions of the naive algorithms from the specification. Catena-axungia, the first related tool, is a unique application that eases the deployment of Catena by allowing to search for abstract cost parameters with concrete measurements. The second tool, Catena-Variants, is a modular framework that allows to easily exchange the components of Catena. This significantly simplifies benchmarking and verification of new approaches.

Acknowledgements

Foremost, I would like to thank Prof. Dr. Stefan Lucks and PD Dr. Andreas Jakoby for accepting to review this thesis. Furthermore, I would like to express my sincere gratitude to Jakob Wenzel, Eik List and Dr. Christian Forler for the continuous support and the invaluable advise. I am also very grateful to Katharina Spiel and Kearsley Schieder-Wethy for the final proofreading.

Contents

List of Figures vi

List of Tables vii

List of Listings viii

List of Algorithms ix

List of Acronyms x

1 Introduction 1

1.1 Motivation . . . 1
1.2 Contribution . . . 3
1.3 Outline . . . 4

2 Preliminaries 5

2.1 Hash Functions . . . 5
2.2 Password Hashing . . . 6
2.3 Attacks . . . 10
2.4 Implementation . . . 11
2.5 Galois-Field Multiplication . . . 12

3 Related Work 20

3.1 PHC Finalists . . . 20
3.2 Notable and Historic PHS . . . 22

4 Catena 24

4.1 Specification . . . 25
4.2 Features . . . 27
4.3 Instances . . . 29
4.4 Parameter Recommendation . . . 34

5 Implementation **37**

5.1 CATENA 37

5.2 CATENA-AXUNGIA 45

5.3 CATENA-VARIANTS 48

6 Discussion **52**

6.1 Conclusion 52

6.2 Outlook 53

A Benchmarks **60**

A.1 Galois-Field Multiplication 60

A.2 Compiler Choice & Optimization Options 61

B Additional Information **62**

B.1 clang Optimization Levels 62

List of Figures

2.1 Alignment of the final three Karatsuba summands. 13

2.2 Alignment of the summands contributing to F 15

2.3 Summands of the final reduction result R. 15

4.4 A $(3, 2)$-bit-reversal graph. 33

4.5 A $(3, 1)$-double-butterfly graph consisting of three layers: vertical (solid lines), diagonal (dotted lines), sequential and connecting (dashed lines). . . 34

5.1 A $(3, 2)$-bit-reversal graph altered to match the traversal of the optimization. 41

5.2 The first three rows of a $(3, \lambda)$-double-butterfly graph aligned in $2^g + 2^{g-1}$ blocks of memory. 43

5.3 Illustration of the class structure created by applying the curiously recurring template pattern to the function categories of Catena-Variants, where `DefaultAlgorithm`, `Blake2b1`, `Blake2b`, `Gamma` and `BRG` are examples for concrete functions. 50

List of Tables

4.1 Slightly adapted notational conventions borrowed from [15]. 24

4.2 Overview of the default instances of Catena [15]. 26

4.3 Values for the domain identifier d. 26

4.4 Benchmark comparing BLAKE2b-1 with BLAKE2b on a Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz. 30

4.5 Recommended parameter sets for average systems. All timings are measured on an Intel Core i5-2520M CPU (2.50GHz) system [15]. 36

A.1 Benchmark of several algorithms for 128 bit Galois-Field multiplication . . . 60

A.2 Runtime of Catena-Dragonfly and Catena-Butterfly with different compilers (GCC and clang) and optimization levels (O2, O3, Ofast). 61

B.1 Flags enabled by the optimization levels O2 and O3 of clang 3.5.0. 62

List of Listings

2.1 A logical shift of a 128 bit integer `v` by `count` bits to the left, where `count` < 64. 16
5.1 Usage statement explaining the command-line interface of Catena-axungia. 46

List of Algorithms

1 Right-to-left multiplication in a binary Galois Field 17
2 Precomputation step for optimized right-to-left multiplication 17
3 Multiplication step for optimized right-to-left multiplication 18
4 Repair step for optimized right-to-left multiplication 18
5 CATENA [15] 25
6 Function *flap* of CATENA [15] 26
7 The client-independent-update function [15] 27
8 CATENA-KG [15] 28
9 The main functions of BLAKE2b-1 and BLAKE2b. 30
10 The *compress* functions of BLAKE2b-1 and BLAKE2b. 31
11 The functions SALTMIX and `xorshift1024star` [15]. 32
12 (g, λ)-Bit-Reversal Hashing (BRH^g_λ) [15]. 32
13 (g, λ)-Double-Butterfly Hashing (DBH^g_λ) [15]. 35
14 Optimized (g, λ)-Bit-Reversal Hashing (BRH^g_λ) 42
15 Parameter-Search Algorithm of CATENA-AXUNGIA 47
16 Function *get_garlics* of CATENA-AXUNGIA 48

List of Acronyms

API	Application Programming Interface
BRG	Bit-Reversal Graph
BRH	Bit-Reversal Hashing
COTS	Commercial Off-The-Shelf
CRTP	Curiously Recurring Template Pattern
DBG	Double-Butterfly Graph
DBH	Double-Butterfly Hashing
GCM	Galois/Counter Mode
KDF	Key-Derivation Function
PHC	Password Hashing Competition
PHS	Password-Hashing Schemes
ROM	Read-Only Memory
RtL	Right-to-Left
SSE	Streaming SIMD Extensions
TMTO	Time-Memory Tradeoff

Chapter 1

Introduction

> I'm a little tired of writing about passwords. But like taxes, email, and pinkeye, they're not going away any time soon.
>
> *Jeff Atwood*[I]

1.1 Motivation

Password security is an ongoing concern for everyone using networked or shared computing devices. Applying an irreversible hash function to the password has become the common way to secure stored passwords. The Password Hashing Competition (PHC) [1] was started in 2013 to collect and curate current password-hashing schemes (PHS). The Catena password-scrambling framework [15] was one of the six finalists of this competition and is listed as a special recognition [2].

Security Concerns about Existing Schemes The amount of password leaks[II] shows that relying solely on securing the computer that stores the passwords is not enough. While *salting* and *hashing* may have been enough to secure passwords in the past, currently we face far greater computational power on the side of an attacker. Using modern graphics cards, an attacker can test billions of hashes per second for regular hash functions [19]. This threat gets worse considering attackers whose budgets surpass those of medium-sized companies. Passwords that remain infeasible to crack in such scenarios have to be quite long and vary in character use. Since these passwords are obviously hard to remember, it is unlikely for all users to voluntarily choose them.

[I] `http://blog.codinghorror.com/your-password-is-too-damn-short/`

[II] For example, the credential-monitoring service PwnedList claims to import several thousand leaks per month. `https://pwnedlist.com/stats/landing`

An approach to counter large amounts of cracking power is to increase the computation or the memory required to hash a password. This process is called password scrambling. The widely available crypt[III]-family consists of several time-consuming algorithms [43]. Both, the attacker and the defender, are slowed down by the increase in required computation. It is still possible for an attacker to hash several thousand password candidates in the same time it takes the defender to compute one hash by parallelizing the attack on graphics cards or other dedicated hardware [19].

Since large chunks of memory per core are available only for CPU computations, increasing the memory requirement limits the parallelizing capabilities of an attacker without affecting the defender too much. scrypt was the first password scrambler with variable memory cost [40]. Since its release, two side-channel attacks against scrypt were found [14, Appendix A]. Another downside of scrypt is its sheer complexity. While the design of Catena is much simpler, it still allows to adjust both time and memory cost without being vulnerable to either of the attacks.

Distinctive Properties While the security of an algorithm should always be the greatest concern when selecting a password-hashing scheme, some scenarios may require certain features to increase the practical security or usability. Besides allowing to securely hash passwords under adjustable cost and time requirements, Catena also provides *client-independent updates*, *keyed hashing* and *server relief*.

The static increase of computation power necessitates, that cost parameters of password-hashing schemes are updated regularly in order for the hashes to stay infeasible to crack. While the common way of changing parameters of the authentication process requires to wait for the next login of the user to compute a new hash with adapted cost parameters, client-independent updates can update the old hashes without involvement of the client. Most password-hashing schemes implement client-independent updates by chaining several invocations, which requires an adapted authentication process and additional storage for every update. In contrast, Catena allows to treat all password hashes equally irrespective of whether they have been updated or not. The only additional requirement for these client-independent updates is a single additional field of storage for the initial cost parameter. Without this feature, password hashes will have to be deleted after a certain time without a login from the user. The result is an increased workload for users due to reasons not obvious to them.

For keyed password hashing, the resulting password hash will be encrypted with a secret key. This requires not only the password hashes and *salts*, but also the secret key for any kind of feasible attacks. If they are stored separately, the attacker has to put more effort into obtaining all information required for an attack. This is especially useful in case of backups that can then be stored without great security concern as long as they do not contain the secret key.

[III] often referred to as crypt(3)

The increased time and memory costs of a password-hashing scheme slow down both the attacker and the defender. The result is that setting specific cost parameters will limit a server to a certain number of concurrent logins. Server relief offloads most of the work to the client and therefore increases the throughput of the server. Alternatively, it allows the defender to increase the cost parameters without affecting the throughput. Only the last step(finalization) is computed on the server to ensure that the hash sent via the network is different to the one stored in the database. For Catena, server relief yields the same hashes as regular hashing; therefore, both methods can be used interchangeably.

Related Topics Key generation and proof of work are often associated with password scrambling due to similar requirements regarding randomness of the output and the adaptability of time and memory cost. Forler, Lucks, and Wenzel show in [15] that Catena is suitable for both. The reference implementation of Catena includes a special interface for key generation.

Implementation The complexity of current computational devices and the amount of limiting factors make it impossible to predict the runtime of a memory- and time-consuming algorithm accurately. Therefore, an optimized implementation is a must to be able to measure real-world performance. See [8] and [24] as prime examples for benchmark done in the scope of the PHC.

1.2 Contribution

The first contribution of this thesis was the design, implementation, and optimization of Catena that resulted in an optimized but still easy-to-grasp reference implementation. Part of this process was a novel reduction of BLAKE2b hash function to a single round.

The second contribution of this thesis, Catena-axungia, is a unique search tool for optimal cost parameters. It finds the closest matching cost parameters for provided time and memory limits. In practical applications of password-hashing schemes, these concrete measurements are usually more helpful than the rather abstract cost parameters. Besides allowing the determination of optimal Catena parameters for a system, Catena-axungia could also be used to update recommended parameters in the future.

The design and implementation of Catena-Variants, a flexible and extendable implementation of Catena in C++, is the third contribution of this thesis. Catena-Variants allows to easily exchange several of its components with alternatives, emphasizing the framework aspect of Catena. Benchmarking tools and a wide range of alternative components are already included. The flexibility makes it easier to test and verify variants of Catena. The design and implementation of both Catena and Catena-Variants could serve as a basis for other password-hashing schemes and similar constructions.

1.3 Outline

The remainder of this thesis starts with the explanation of the preliminaries necessary to understand Catena and the implementations in Chapter 2. In Chapter 3, recent and past password-hashing schemes are presented to provide a reference frame. In particular, it includes an overview of all other PHC finalists. Chapter 4 contains a detailed explanation of the Catena password-scrambling framework. The implementation specifics of all three applications that are part of this thesis are presented in Chapter 5. Chapter 6 concludes with the discussion of the work presented in this thesis and provides an outlook on future work.

Chapter 2

Preliminaries

> Stop associating "hashed" with "secure" when it comes to passwords.
>
> *tylerjl*[I]

This chapter contains an introduction to hash functions and password hashing, with an emphasis on the properties of password-hashing schemes as well as attacks against them. The third section explains the technical details required for fast and portable implementations. The rest of the chapter gives an in-depth description of the implementation of Galois-Field multiplications, which can be used as a fast hash function.

2.1 Hash Functions

A hash function $\mathcal{H}$ can be defined as:

$$\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^n;$$

$\mathcal{H}$ is a deterministic one-way function that maps input values of variable length to output values of a fixed length (here: n bit). Invoking a hash function is called hashing and the resulting output values are called hashes. The set of input values, the domain, is theoretically infinite and the range, *i.e.* the space that contains all output values, is limited by the fixed size of the resulting hash. An n-bit hash function can therefore produce 2^n different hashes and the result is a theoretically unlimited set of input values for every output value. Two or more input values that map to the same hash are called a collision.

[I] `http://blog.tjll.net/please-stop-hashing-passwords/`

Cryptographic Hash Functions A cryptographic hash function h is a hash function that fulfills the following additional requirements:

1. **Pre-Image Resistance**: For any specified hash y it is infeasible to find an input x so that $y = h(x)$.

2. **Second Pre-Image Resistance**: Given an input value x_1, it is infeasible to find a distinct input x_2 so that $h(x_1) = h(x_2)$.

3. **Collision Resistance**: It is infeasible to find any two distinct inputs x_1 and x_2 so that $h(x_1) = h(x_2)$.

See [45] for more formal and specific notions of these requirements.

2.2 Password Hashing

Storing passwords only in a hashed form has become the common practice for password-based user authentication. This ensures that an attacker who gains access to the database can not easily retrieve the passwords. It is safe to assume that at least some users tend to reuse passwords or alter them only slightly. A breach that reveals the passwords of these users puts all of their accounts at the risk of getting compromised.

The minimum requirements for a password-hashing scheme are the same as those for cryptographic hash functions. Collision resistance is required in order to have a high probability that a given password yielding the stored hash is the password that was originally submitted. The pre-image resistance and second pre-image resistance guarantee that an attacker can not easily recover the password or a collision of it from the password hash.

It is common practice to add a *salt* to the password before hashing. A salt is a random value of fixed length that is stored alongside the password hash in the database [36]. The additional random value restrains an attacker from precomputing all the hashes of a wordlist or dictionary. While it may still be feasible to hash smaller dictionaries with all possible salts, the size of the result remains a product of the dictionary size and the length of the salt; therefore, hashing wordlists becomes less feasible with increasing salt length. Other *time-memory-tradeoff* attacks are affected in a similar fashion. Salting also increases the number of hashes an attacker has to compute when attempting to crack the whole hash database, because they would have to try every salt with every password candidate. Analysis of leaked password databases shows that it is common for several users to share a password [33]. With a reasonably large salt length, it becomes unlikely for two users to share the same salt. As a result, it becomes improbable that several hashes are cracked at once because the differing salts will result in different hashes, even for users sharing a password. Furthermore, salting reduces the danger of collisions. A second pre-image of

a salted hash only collides with the actual password when using the same salt and hash function whereas a second pre-image for a unsalted hash can be used as a replacement for the actual password everywhere where the same hash function is used.

Key Stretching The strength of a password can be measured in bits of surprisal[II]. An attacker will need to hash 2^μ password candidates to crack a password with μ bit surprisal. Increasing the computation required for a single hash is called key stretching [29]. If the work required for a single hash is 2^σ, the total computation required to crack a password hash becomes $2^{\mu+\sigma}$. This results in σ bits of pseudo-entropy [32]. The most common technique for key stretching, found e.g., in some members of the crypt family[43], is the iteration of a cryptographic hash function. The work of the whole construction can then be expressed as a factor of the work required for a single call to the iterated hash function. While key stretching slows down the attacker and the defender equally, it does not prevent the attacker from parallelizing their attack and thereby reducing the average time per hash significantly.

Since memory and especially fast cache memory are a much more limited resource on graphic cards and hardware solutions like ASICs and FPGAs, increasing the memory requirement of key stretching is an effective way of limiting the parallelization capabilities of the attacker. scrypt was the first implementation of memory-demanding key stretching [40].

Memory-Hardness Memory-Hardness is an additional requirement for memory-demanding key stretching that limits the complexity of a *time-memory-tradeoff*-adjusted algorithm to the overall space and time complexity of the password-scrambling algorithm. Hence, an attack that reduces the memory to $1/n$ must suffer a penalty of at least factor n. The more general notion of λ-memory-hardness was introduced in [14, Chapter 3] and is defined as follows:

Definition 2.1 (λ-Memory-Hard Function)**.** *Let g denote the memory cost factor. For a λ-memory-hard function f, which is computed on a Random Access Machine using $S(g)$ space and $T(g)$ operations with $G = 2^g$, it holds that*

$$T(g) = \Omega\left(\frac{G^{\lambda+1}}{S(g)^\lambda}\right).$$

Using this definition, memory-hardness becomes the special case where $\lambda = 1$.

The stronger notion of sequential memory-hardness was introduced in [40]. A sequential-memory-hard function has the additional property that it can not be sped up at all by parallel computations. This can be achieved by making the access depend on the actual data as seen in scrypt.

[II] often referred to as entropy

Client-Independent Update It is fairly safe to assume that the computational power of the average attacker and the average defender doubles roughly every two years. While this may decrease the load on the defenders side, it also decreases the security, because the attacker requires less time to compute a hash. The security of an authentication process with fixed cost parameters would be halved every two years. Obviously, the defender will have to update the cost parameters regularly.

The common way to change parameters or hash functions for password hashing is to wait for the user's next login. When logging in, the user's given password is verified with the hash from the database and afterwards the password is used to compute a new hash that replaces the old one in the database. Keeping password hashes with outdated cost parameters in the database poses a risk to the user. Therefore, it is common to delete all hashes that could not be updated after a grace period. This usually forces a user to start the same procedure used in case of forgotten passwords. The additional work may annoy users especially because the reason is out of their reach.

Client-independent updates [14] are an alternative to replacing or deleting password hashes. Well-designed password-hashing schemes can update hashes that were generated with outdated cost parameters to new cost parameters without requiring the password. This allows updating all hashes in a database without a single login of a user. Grace periods and code branches to consider password updates on login become unnecessary. It is obvious that this is only possible for schemes that do not require the password for every step.

Server Relief The number of parallel logins that an authentication server, that is using a PHS, can handle is limited by the cost parameters. This can be a huge burden for busy or slow servers. Server relief, as defined in [14], allows to offload the majority of the work to the client. Only a fast finalization step is computed on the server to ensure that the password hash transmitted from the client is different from the hash stored in the database. This is necessary because in this scenario the hash computed by the client becomes the actual password and should therefore never be stored in plaintext. This scenario is virtually the same as the client generating a very strong password.

While it would be possible to create a dedicated server-relief protocol for almost all password hashing schemes, incorporating this property into a password scrambler makes it possible that hashing with server relief and regular hashing yield the same result. Making different modes of hashing interchangeable increases the flexibility. As an example, in a scenario with a wide variety of clients, a server could offload the password scrambling only to fast enough clients and hash the password for all other clients itself.

Keyed Password Hashing For keyed password hashing, the password hashes are encrypted with a secret key [14]. Storing the key separately from the database increases the effort required for an attacker. For a successful attack, they requires not only password hash and salt, but also the secret key. If the key is stored on a hardware security module,

an attacker without hardware access to the server will most likely fail. Without investing in such a module, a defender could also generate the key during the bootstrapping phase and from then on, keep it in the RAM or a register of the CPU. Retrieving keys that are never stored on the hard disk is far more complicated and requires extensive knowledge of the system.

As long as the key is kept separate, databases containing only keyed hashes can be stored without greater concerns about security. This is especially useful for backups as it poses no additional security requirements for backup servers.

Key-Derivation Keys for symmetric-key algorithms are often derived from passwords using a Key-Derivation Function (KDF). It is easy to see that KDFs face similar threats as password-hashing schemes. Since PHS usually have a limited output size, using one for key generation can be a problem when requiring larger keys or when multiple keys should be derived from one password. Hence, only password-hashing schemes that allow an arbitrary output length can be used for key generation. Multiple keys generated from one password should be independent from each other since an attacker should not be able to derive one key from the other.

Proof of Work Proof-of-work systems require a certain amount of work from the client as a measure to limit the use of a resource. This can be used to prevent overuse, e.g., spamming or denial of service [13], or to create an artificial rarity for so-called cryptocurrencies [37]. To prove the execution of the work, the client has to solve a problem that is either known in advance or posed as a challenge by the server.

When using cryptographic hash functions for proof-of-work systems, a challenge could be a hash y and a part x_1 of its random input x [28]. The client then has to find the rest of the input x_2 so that $h(x_1 \mid\mid x_2) = y$. The server can then compare the hash of $x_1 \mid\mid x_2$ from the response to the previously stored y from the challenge.

Hashcash [3] is a common approach for a foreknown problem. The sender has to provide a string s that, when hashed together with the recipient's address a and the timestamp t, results in a hash starting with a fixed number of zero bits. For the verification, the server checks a and t for validity and then verifies that the hash of a, t and s starts with the specified number of zero bits. Requiring a timestamp or nonce is important to prevent the client from reusing the solution. Proof-of-work systems with a foreknown problem are easier to integrate into existing protocols since they affect only a single step.

The two aforementioned approaches to proofs of work can be used with password hashing schemes instead of regular cryptographic hash functions. The clear advantages of using a PHS are the additional performance parameters in form of the cost parameters of the scheme. With the parameter choice, an otherwise time-bound proof-of-work system could be adjusted to require memory and time. It is important to note that, as with all search

tasks, the work required for finding a specific input or output of a hash function or PHS can differ greatly between different clients depending on the precise task or search approach; therefore, the amount of work should be defined for the average case.

2.3 Attacks

Brute-force search is the naive of cracking password hashes. The attacker tries all possible combination of characters from a specified character set. This is obviously only feasible for a limited number of characters. Excluding all characters that are not part of a specific language or keyboard layout can improve this approach drastically. Another improvement can be gained by statistically analyzing passwords and then performing the search starting with the most likely characters at this position or alternatively in this context[III].

As mentioned before, it is common for multiple users to share the same password [33]. An experienced attacker will be able to predict at least some of these poorly chosen passwords. Even a large list of these predictions can be processed faster than all brute-forced combinations of more than a few letters. A regular dictionary could serve as a starting point for such a wordlist. Further improvement is possible by trying to predict the adjustments made to conform to the corresponding password policy.

Time-Memory-Tradeoff Attacks The first kind of time-memory-tradeoff (TMTO) attacks allow a reduction of the runtime in exchange for an increased requirement of storage space or memory. An attacker that plans to conduct several attacks can gain a significant advantage by hashing the password candidates in advance. The actual attack is reduced to a simple comparison between the precomputed hashes and the hashes from the database. The already significant amount of storage required to store all the hashes is drastically increased by salting. Precomputing may stay still be feasible for smaller salt sizes and small dictionaries or limited length brute-force attacks. It is also possible to reduce the amount of storage in exchange for a more time-consuming comparison process [39].

The second kind of TMTO attacks require a password hashing scheme that can be run with a reduced amount of memory. By reducing the required memory the attacker is able to hash more password candidates in parallel. A memory-reduced algorithm could also depend less on cache memory and therefore run faster on devices with a smaller cache. Memory-hardness (see Definition 2.1) quantifies the magnitude of disadvantage suffered from the tradeoff. Biryukov and Khovratovich present a set of TMTO attacks against data-independent password hashing schemes in [5]. For one instance of CATENA, CATENA-DRAGONFLY, the attack uses wisely chosen intervals in every level that can be computed mostly independently. This allows to reduce the memory-hardness of CATENA-DRAGONFLY to significantly less than λ.

[III] see for example hashcat's markov attack: `https://hashcat.net/wiki/doku.php?id=statsprocessor`

Side-Channel Attacks An attack that uses side effects of an implementation to gain additional information is called side-channel attack. Usually, this requires access to the defender's machine. The design decisions of some password-hashing schemes make side-channel-free implementations very slow or even impossible.

When running a spy process on the defender's system, the attacker can log the order in which the implementation of the PHS accesses the memory [14, Appendix A]. The spy process needs to run on the same CPU as the target process and must be able to interrupt it. The cache-timing attack consists of first flushing the cache after the filling step of the PHS by reading from memory. After or during the computation phase the process measures the time it takes to access different parts of the memory. This reveals which parts of the memory were read by the target process. For schemes whose memory accesses depend on the password or a direct derivative thereof, the access pattern could be used to filter password candidates. The attacker can calculate the access pattern of the candidates and compare them to the one measured before. This allows the set of candidates to be drastically narrowed without having to compute full password hashes. It is easy to see that filtering password candidates is less demanding then computing a full password hash and is therefore better parallelizable.

An attacker able to gain access to the memory used by a password-hashing scheme after or during its run can conduct a garbage-collector attack [16]. The attacker can use the internal state received from this attack as a replacement for the final hash. All computations of the PHS during the search for the password can then be stopped at the step at which the state was retrieved. The threat gets worse when the memory contains the password or a direct derivative of it. If the function that is used to derive the password is drastically faster to compute than the password hash, the attack is called weak garbage-collector attack [16].

2.4 Implementation

Intrinsics The most common extension to regular x86-CPU instructions are the Streaming SIMD Extensions (SSE) and their successors SSE2, SSE3 and SSE4. The latest of these extension sets, SSE4, is available on all modern desktop CPUs[IV]. They add support for 128-bit registers and include a broad range of instructions[V]. Some of these instructions simply provide regular operations for the larger registers and others provide access to complex operations. The successors AVX and AVX2 include support for 256-bit registers. Another extension is the set of pclmulqdq instructions. They allow the carry-less multiplication of two 64-bit multiplicands in one operation. These extensions can be called using intrinsic functions or intrinsics for short.

[IV] http://en.wikipedia.org/wiki/SSE4#Supporting_CPUs
[V] https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Endianess Currently, there exist two approaches for storing the bytes of a word: Little-endian systems store the bytes starting in the smallest address ordered after ascending significance [31]. In big endian, the order is reversed. C and C++ code can be written mostly without any concern about the endianess of the system; only the conversion between datatypes and arrays differs between both conventions.

2.5 Galois-Field Multiplication

Galois-Field multiplications can be used as a fast compression function and is an option for hashing in Catena-Variants. Binary Galois Fields, *i.e.* finite fields with an order that is a power of two, are especially suited for fast implementations because their coefficients can be represented as bits and addition in the field is equivalent to XOR. The rest of this section explains two approaches for implementing fast multiplications in the Galois Field $GF(2^{128})$ with the reduction polynomial $f(z) = z^{128} + z^7 + z^2 + z + 1$ from the Galois/Counter Mode (GCM) [35]. The first approach uses pclmulqdq instruction and other extensions specific to modern x86-CPUs to speed up the computation. The second approach is a generic approach that does not require any specific CPU.

The special properties of a binary Galois Field lead to the operations being carry-less. In this context, subtraction is the same as addition and can therefore be replaced with XOR. Multiplication of a polynomial Q with an monomial of degree g is the equivalent to a left shift of Q by g bit.

For this section, let Q denote the integer, *i.e.* the vector of bits, that represents the element $Q(z)$ of the binary field. Furthermore, let Q_0 to Q_{m-1} denote the $m = \lceil \frac{n}{64} \rceil$ 64-bit words of the n-bit integer Q where Q_0 is the lowest word. The n bits of Q are indexed from the least significant bit q_0 to the most significant bit q_{n-1}. These bits represent the coefficients of the element $Q(z)$. A bitwise logical left shift or right shift will be denoted as $\ll$ and $\gg$, respectively.

Karatsuba Multiplication Both approaches presented in this section benefit from splitting up the multiplication into 64-bit multiplications. The general approach restricts the multiplicand size to 64 bits, *i.e.* the largest avaiable datatype, because splitting data into several variables requires costly workarounds for computations that require the data in its entirety. While there are datatypes larger than 64 bit avaiable on computers with extended instruction sets, the pclmulqdq instruction limits the size of its multiplicands to 64 bit.

The Karatsuba algorithm [22, section 2.2.2] allows to compute a 128-bit multiplication with the help of just three 64-bit multiplications and a few additional operations instead of the four multiplications required by a naive algorithm.

Let A and B be the 128-bit multiplicands that can be decomposed into

$$A = (A_1 \ll 64) + A_0$$
$$B = (B_1 \ll 64) + B_0.$$

The multiplication of A and B can then be performed as

$$\begin{aligned} A \cdot B &= ((A_1 \ll 64) + A_0) \cdot ((B_1 \ll 64) + B_0) \\ &= ((A_1 \cdot B_1) \ll 128) + ((A_1 \cdot B_0 + A_0 \cdot B_1) \ll 64) + A_0 \cdot B_0. \end{aligned}$$

The second summand can be rearranged as

$$A_1 \cdot B_0 + A_0 \cdot B_1 = (A_1 + A_0) \cdot (B_1 + B_0) - (A_1 \cdot B_1 + A_0 \cdot B_0).$$

This reduces the total amount of required multiplications to the following three:

$$\begin{aligned} C &= A_1 \cdot B_1, \\ D &= A_0 \cdot B_0, \\ E &= (A_1 + A_0) \cdot (B_1 + B_0). \end{aligned}$$

Substituting these into the middle summand yields

$$A_1 \cdot B_0 + A_0 \cdot B_1 = E - (C + D).$$

The whole multiplication then becomes

$$A \cdot B = (C \ll 128) + ((E - (C + D)) \ll 64) + D.$$

			D_1	D_0	
$+$	C_1	C_0	$\ll 128$		
$+$		$E_1 - C_1 + D_1$	$E_0 - C_0 + D_0$	$\ll 64$	
$=$	X_3	X_2	X_1	X_0	

Figure 2.1: Alignment of the final three Karatsuba summands.

The following computation of the result of the multiplication can be derived from Figure 2.1:

$$\begin{aligned} X_3 &= C_1, \\ X_2 &= E_1 - C_1 + D_1 + C_0, \\ X_1 &= E_0 - C_0 + D_0 + D_1, \\ X_0 &= D_0. \end{aligned}$$

Note that addition and subtraction can both be performed by XOR in binary Galois Fields.

Reduction The 256-bit result X has to be reduced after the multiplication to become an element of the Galois Field. This process can be drastically simplified from polynomial division to a series of shifts and adds.

The order of the field is $m = 128$. The reduction polynomial (modulus) $f(z) = z^{128} + z^7 + z^2 + z + 1$ can then be written as $f(z) = z^m + z^7 + z^2 + z + 1$. Substituting $r(z) = z^7 + z^2 + z + 1$ yields $f(z) = z^m + r(z)$. Due to the nature of the modulo operation, only the coefficients with a degree larger than m are affected by the reduction. For example, the element $q_a \cdot z^a$ with $a \geq m$ is reduced by

$$\begin{aligned} q_a \cdot z^a \bmod f(z) &\equiv q_a \cdot z^{a-m} \cdot r(z) \bmod f(z) \\ &\equiv q_a \cdot z^{a-m+7} + q_a \cdot z^{a-m+2} + q_a \cdot z^{a-m+1} + q_a \cdot z^{a-m+0} \bmod f(z); \end{aligned}$$

consequently, the reduction of $X(z) \bmod f(z)$ becomes:

$$\begin{aligned} & x_{2m-2} \cdot z^{2m-2} + \cdots + x_m \cdot z^m + x_{m-1} \cdot z^{m-1} + \cdots + x_1 \cdot z + x_0 \bmod f(z) \\ & \equiv (x_{2m-2} \cdot z^{m-2} + \cdots + x_m) \cdot r(z) + x_{m-1} \cdot z^{m-1} + \cdots + x_1 \cdot z + x_0 \bmod f(z). \end{aligned}$$

The largest possible degree after this reduction is $(m - 2) + \text{degree}(r(z))$. This would require an additional reduction, but the commutative property allows to evade it. First, only the summands that would have a degree larger than or equal to m after the reduction are reduced and added to X. Then the reduction of the full polynomial is performed. For the GCM modulus $f(z)$, the maximal degree after the first reduction is $126 + 7 = 133$. This means that only the third word X_2 is affected by the first step.

The auxiliary variable F, *i.e.* the updated X_2, can be computed as shown in Figure2.2:

$$F = X_2 + (X_3 \gg 64) + (X_3 \gg (64 - 1)) + (X_3 \gg (64 - 2)) + (X_3 \gg (64 - 7)).$$

Since $(X_3 \gg 64)$ is empty, this can be simplified to:

$$F = X_2 + (X_3 \gg 63) + (X_3 \gg 62) + (X_3 \gg 57).$$

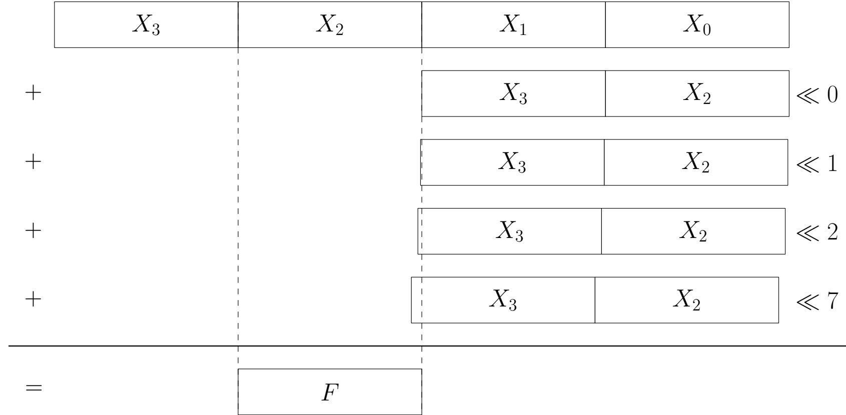

Figure 2.2: Alignment of the summands contributing to F

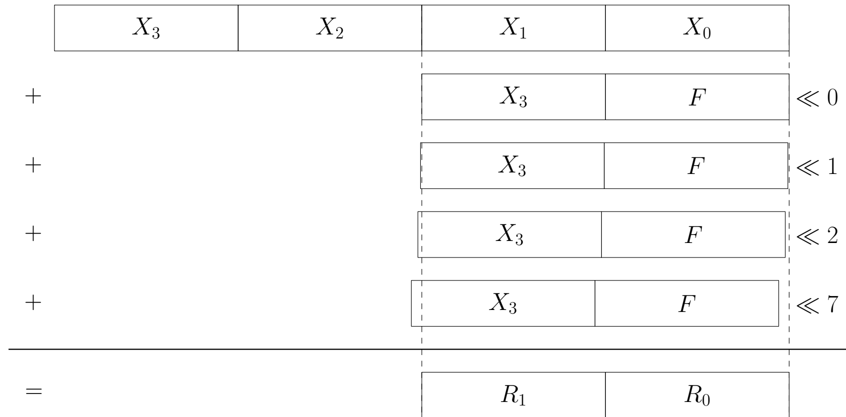

Figure 2.3: Summands of the final reduction result R.

Figure 2.3 shows the final alignment from which the result of the reduction R can then be derived as

$$\begin{aligned} R_1 &= X_1 + X_3 + ((X_3 \ll 1) + (F \gg 63)) + ((X_3 \ll 2) + (F \gg 62)) \\ &\quad + ((X_3 \ll 7) + (F \gg 57)), \\ R_0 &= X_0 + F + (F \ll 1) + (F \ll 2) + (F \ll 7). \end{aligned}$$

Intrinsics Specifics A detailed description of implementing multiplication and reduction using pclmulqdq instruction (see Section 2.4) and other intrinsics can be found in [21]. The Karatsuba algorithm and the reduction are performed as described above. The pclmulqdq instruction is used to compute the 64-bit multiplications. Some steps can be reduced when using 128-bit instructions.

Since all commonly available instruction sets lack the support for bitshifts by values that are not a multiple of 8, the effects of a bitshift have to replicated by a series of operations. To emulate the left shift of a variable V by c bit, both 64-bit words of the variable are

shifted independently to the left:

$$V_1 = V_1 \ll c$$
$$V_0 = V_0 \ll c.$$

The part of the lower word V_0 that is shifted out can be reconstructed by right-shifting a copy of it. This reconstruction is then XORed to the higher word V_1:

$$V_1 = V_1 \oplus (V_0 \gg (64 - c)).$$

Listing 2.1 shows an exemplary implementation. For $c > 64$, the emulation can be simplified to

$$V_1 = V_0 \ll (64 - c)$$
$$V_0 = 0,$$

but this case does not occur when reducing with the GCM modulus.

```
1: __m128i sll128(__m128i v, uint8_t count){
2:     __m128i result, tmp;
3:     result = _mm_slli_epi64(v, count);
4:     tmp = _mm_slli_si128(v, 8);
5:     tmp = _mm_srli_epi64(tmp, 64 - count);
6:     result = _mm_or_si128(result, tmp);
7:     return result;
8: }
```

Listing 2.1: A logical shift of a 128 bit integer v by count bits to the left, where count < 64.

Generic Specifics Support for the pclmulqdq instruction can be found only in x86 processors built after 2010. To support other architectures or x86 processors without this feature, a generic implementation is required. The only requirement for this approach is a compiler with support for 64-bit datatypes. Since those are a part of the C99 standard, they should be widely available [26]. The Karatsuba algorithm and the reduction can be performed exactly as described above. Due to the absence of the pclmulqdq instruction, the carry-less multiplication has to be implemented for this approach.

The main idea behind implementing carry-less multiplications can be derived from the following equation:

$$A(z) \cdot B(z) = a_{m-1} \cdot z^{m-1} \cdot B(z) + \cdots + a_1 \cdot z^1 \cdot B(z) + a_0 \cdot B(z).$$

Any multiplication of $B(z)$ with z^i is the same as left shifting $B(z)$ by this amount. Since any a_i can only be 0 or 1, the value of any summand is either zero or B shifted by the

respective amount. This approach is called right-to-left multiplication [22, section 2.3.2] when processing the bits of A from right to left as displayed in Algorithm 1.

Algorithm 1 Right-to-left multiplication in a binary Galois Field

Input: A, B : Multiplicands, m : Multiplicand Size
Output: R : Result of Multiplication

```
1: R ← 0
2: for i from 0 to m − 1 do
3:     if a_i = 1 then
4:         R ← R ⊕ B
5:     end if
6:     B ≪ 1
7: end for
8: return R
```

For 64-bit multiplications, B is already represented by largest available datatype. Shifting B will increase its size to two words and make additional operations neccessary for every step that involves B. The bitwise processing slows this approach down even more.

A common optimization for Galois-Field multiplications is the precomputation of windows [22, section 2.3.2]. For a window of size w, the results of the multiplication of B with all possible w-bit values is computed. The comparison then becomes a simple lookup of the current w-bit window of A. The right-to-left approach described above would require to shift all precomputation results. Since those consist of two words each, the shifts would become especially expensive.

A faster approach for carry-less multiplication is described in [7]. By limiting the precomputed values to a single word, the disadvantages of right-to-left multiplication for precomputation can be circumvented. The ensuing repair step is less expensive than shifting two words at each step. For the precomputation step shown in Algorithm 2, the possible values of the window are treated as integers and used as the indices of the corresponding result.

Algorithm 2 Precomputation step for optimized right-to-left multiplication

Input: B : Multiplicand, w : Window Size
Output: u : Vector of Precomputed Results

```
1: u[0] ← 0
2: u[1] ← B
3: for i from 2 to 2^w − 1 in steps of 2 do
4:     u[i] ← u[i ≫ 1] ≪ 1          ▷ Restricted to the size of the word
5:     u[i + 1] ← u[i] ⊕ B
6: end for
7: return u
```

Algorithm 3 shows the multiplication step. Extracting the window from A in Line 4 is performed by shifting the current window to the right into the least significant bits and

then applying a bitmask $2^w - 1$ that zeros everything but the last w bit. The resulting integer can then be used for a lookup in u.

Algorithm 3 Multiplication step for optimized right-to-left multiplication

Input: A, B : Multiplicands, w : Window Size, u : Vector of Precomputed Results
Output: R' : Intermediate Result
1: $R'_0 \leftarrow u[a \wedge (2^w - 1)]$
2: $R'_1 \leftarrow 0$
3: **for** i **from** w **to** $64 - 1$ **in steps of** w **do**
4: $tmp \leftarrow u[(A \gg i) \wedge (2^w - 1)]$
5: $R'_0 \leftarrow R'_0 \oplus (tmp \ll i)$
6: $R'_1 \leftarrow R'_1 \oplus (tmp \gg (64 - i))$
7: **end for**
8: **return** R'

The bits omitted from the precomputed values have to be restored in a repair step to get the correct result of the multiplication. Only the highest word of R is affected by this. Furthermore, only the $w - 1$ highest bits of B are omitted during the precomputation and only the 1-bits could affect the result. A value v from the precomputation causes one of these 1-bits b_{64-j} to be shifted out only if v has a 1-bit which is at least as far away from the beginning of the window as the distance j of b_{64-j} from the end of the word. For the repair step, this means that for every 1-bit $a_{j'}$ with $j' \bmod w \geq j$ the bit b_{64-j} has to be added to the result at position $r_{64+j'-j}$.

The bits $a_{j'}$ can easily be extracted from A by using a bitmask. For values $j' \bmod w \geq 1$, the bitmask has to extract all bits except for the first bit of every window. It must therefore consist of a pattern of one 0-bit followed by $(w - 1)$ 1-bits starting with the least significant bit. Instead of adding b_{64-j} to all positions $r_{64+j'-j}$, the vector of extracted bits can be shifted by j bit to the right and XORed to the highest word of the result. It is easy to see that this step can be skipped if b_{64-j} is 0.

Algorithm 4 Repair step for optimized right-to-left multiplication

Input: A, B : Multiplicands, w : Window Size, R' : Intermediate Result
Output: R : Result of the Multiplication
1: $R \leftarrow R'$
2: $k \leftarrow \neg(1 + 2^w + 2^{2 \cdot w} + \ldots + 2^{\lfloor \frac{64-1}{w} \rfloor \cdot w})$ ▷ mask
3: **for** j **from** 1 **to** $w - 1$ **do**
4: **if** b_{64-j} **then**
5: $R_1 \leftarrow R_1 \oplus ((A \wedge k) \gg j)$
6: **end if**
7: $k \leftarrow k \wedge (k \ll 1)$ ▷ exclude all bits $j' : j' \bmod w = j$
8: **end for**
9: **return** R

An improved implementation will have a fixed window size w, which allows to precompute the mask once and embed it into the code. The condition checking if b_{64-j} is 1 can be replaced by a bitmask to prevent branching.

The optimized algorithm (Algorithm 3 and 4) prevents side-channel attacks that gain additional information about the multiplicands by measuring the computation time. The use of precomputation and bitmasks instead of branching makes the computation time of the optimized algorithm independent of the multiplicands, whereas the naive algorithm (Algorithm 1) takes varying amounts of time, because it contains a condition that depends on the 1-bits of one multiplicand.

There are further possible improvements to Galois-Field multiplications. Most of them apply only to multiplications with larger degrees or with one constant multiplicand. Left-to-right multiplication, *i.e.* processing the bits from left to right instead, is useful in combination with precomputation since only the result has to be shifted instead of copying and shifting the precomputed values [22, section 2.3.3]. The advantage of left-to-right multiplication is lost when restricting the precomputed results to a single word since shifting the result alone requires as many operations as the additions in Lines 5 and 6 of Algorithm 3. For multiplication without the Karatsuba algorithm, the precomputed values could be restricted to the size of the multiplicands by directly reducing them [22, section 2.3.3]. Brent et al. suggest to perform two lookups at once in line 4 of Algorithm 3 [7]. gf2x[VI] takes this approach a step further by unrolling the loop altogether. Though, both approaches yielded no improvement in our benchmarks (see Section A.1).

[VI] gf2x is the implementation of [7]: `http://gf2x.gforge.inria.fr/`

Chapter 3

Related Work

So please don't throw away PHC!

Daniel J. Bernstein[I]

Research on password hashing and key derivation has spawned several different schemes over the years. This chapter provides an overview of current and past approaches. The current password hashing schemes are represented by all finalists and some noteworthy non-finalists of the Password Hashing Competition (PHC) [1].

Forler et al. provide an overview of all candidates with a detailed analysis of the weakness to side-channel attacks [16]. Results for several mechanical test as well as various benchmarks of the finalists can be found in [8].

3.1 PHC Finalists

This section covers all nine finalists of the PHC [2] with the exception of Catena, which is described in depth in Chapter 4. From these schemes, the competition panel selected Argon2 as the winner and gave special recognition to Catena, Lyra2, Makwa and yescrypt [2].

The versioning and ordering are borrowed from the official candidate list [2]. Please be aware that this section reflects the state of the candidates during the winner selection stage of the competition. Further tweaks were not considered since the finalization phase of the competition is still ongoing at the time of writing this thesis.

Argon2 In May 2015, Argon2 was accepted as a late addition to the competition, replacing Argon[II]; both are specified in [4]. The two variants of Argon2, Argon2d and Argon2i, differ only in their memory-access pattern. Argon2i features data-independent

[I] http://article.gmane.org/gmane.comp.security.phc/2462
[II] http://article.gmane.org/gmane.comp.security.phc/2963

memory access to be secure against side-channel attacks. Argon2d's memory accesses depend on previously computed data similar to *scrypt*. This allows Argon2d to reach reasonable memory hardness in fewer passes than Argon2i. Both variants rely on BLAKE2b and additionally on a compression function that is based on the round function of BLAKE2b [46].

In addition to allowing to adjust the time and memory costs, Argon2 has a parameter p for the maximum degree of parallelism. It should be noted that the memory cost can be adjusted in fine-grained steps of $4p$ kilobyte. Moreover, Argon2 features keyed hashing, server relief and a variable output that allows it to be used as a KDF. The specification suggests a way of implementing Client-independent updates for time and memory cost, that would require to change the authentication process for every update.

battcrypt battcrypt is a memory-hard PHS that was designed to be easily implementable in PHP [50]. battcrypt relies on SHA-512 and, similar to *bcrypt* [44], on Blowfish. battcrypt extends *bcrypt's* concept of a PHS with adjustable time cost to also include adjustable memory cost. The memory-hardness is achieved by using a data-dependent memory access pattern. battcrypt includes a dedicated interface allowing to create outputs of any size for key derivation.

Lyra2 Lyra2 achieves memory-hardness by applying a sponge construction to the entries of a memory matrix [25]. There are three options for the sponge function: BLAKE2b or one of two modifications of BLAKE2b that include multiplications. The size of the matrix and thus the memory requirement can be adjusted in two dimensions allowing to adjust the memory access for different cache sizes. The required computation time can be controlled by changing the number of iterations. Lyra2_p is a special variant that supports parallelism with shared memory. Since there is no restriction on the output size, Lyra2 is well-suited for key derivation.

Makwa The Makwa password-hashing scheme is based on squarings modulo a Blum integer n [42]. The prime factors of the integer can either be erased or used as additional secrets. If they are known, the computation of the squarings becomes significantly faster; therefore, the security of Makwa depends on the secrecy of the factors. Since squaring is a purely computationally expensive task, Makwa's memory requirements are negligible. The time cost can be adjusted by changing the numbers of squarings required. The novel process awards Makwa with two distinctive properties: escrow and delegation. Escrow refers to the possibility that certain configurations of Makwa allow to efficiently recover the password when the prime factors of n are known. Delegation allows the defender to safely offload some of the required computation to an untrusted third party, e.g., a computational cloud. Makwa can be configured to hash its large output with HMAC-DRBG[III] to obtain

[III] HMAC-DRBG is usually a Deterministic Random Bit Generator, but is used for hashing in Makwa

a hash of smaller size. This makes it possible to use the server relief seen in other PHS instead of delegation. On the other hand, hashing the output of Makwa renders client-independent updates impossible.

Parallel Parallel is a simple scheme for chaining hash-function calls [51]. It allows to adapt the computation cost by setting the number of iterations. Parallel includes a variant for key derivation.

POMELO POMELO is based on three state-update functions [53]. One of these functions uses data-dependent memory access. The size of the state and the number of iterations and therefore the time and memory cost can be adjusted independently. A slight modification allowing POMELO to generate keys of arbitrary size is suggested in the paper. While the paper describes a way of implementing client-independent updates, the approach requires changing the hashing process for every parameter update.

Pufferfish The Pufferfish password-hashing scheme is a modern adaption of *bcrypt* with the addition of data-dependent memory access [20]. In contrast to the similar scheme battcrypt, Pufferfish uses a modified blowfish that works on 64-bit words and relies on HMAC_SHA512. Memory cost, time cost and output length are freely adjustable, which allows to use Pufferfish for key derivation.

yescrypt The yescrypt PHS is a backwards-compatible extension of *scrypt* and inherits its sequential memory-hardness and most of the structure [41]. In comparison to scrypt, yescrypt can optionally use a freely adjustable amount of read-only memory (ROM), that can be shared between different threads. yescrypt extends *scrypt*'s parallelization by also allowing a mode where several threads operate on the same memory.

3.2 Notable and Historic PHS

The first widely used password-hashing scheme for Unix was crypt [43]. The original implementation used the lowest 7 bit of each of the first eight characters of the password as a key for DES and then encrypted a constant string with it. This construction had properties similar to a cryptographic hash function. The obvious limitations of crypt have motivated several successors. Most of those are based on an adjustable number of iterations over a cryptographic hash function.

A noteworthy successor of crypt is bcrypt[44]. It is based on 64 iterations of a modified blowfish cipher. The modifications of the cipher allow to adjust the computational cost of the scheme by altering the key schedule.

The first sequential memory-hard PHS, scrypt, was presented by Percival in 2009 [40]. It was also the first password-hashing scheme that allowed to adjust memory and time costs. scrypt achieves its memory-hardness by deriving the memory access pattern from the data itself. The memory-demanding step is followed by a call to PBKDF2. This allows scrypt to inherit PBKDF2's property of arbitrary output length and its suitability for key derivation. The optional parallelism is achieved by computing several memory-demanding steps in parallel.

BRG-Based Catena-Dragonfly is not the only PHC candidate that relies on a *bit-reversal graph (BRG)* for memory access. Rig also uses a BRG for half of its memory accesses [9]. The sequential memory-hard schemes TwoCats and SkinnyCat combine a data-dependent memory access pattern with a data-independent access pattern based on a BRG to counter cache-timing attacks [11].

Chapter 4

Catena

> Avoiding cache timing attacks is desirable, and Catena shows how to do it.
>
> *Bill Cox*[I]

The CATENA password-scrambling framework is specified in [15]. This chapter summarizes the paper by Forler, Lucks, and Wenzel and accentuates certain details that are important for the implementation (see Chapter 5). The rest of this chapter uses the notation defined in Table 4.1, which is mostly identical to the one used in [15]. Please be aware that the notation for vertices and garlic has been altered slightly.

Identifier	**Description**
pwd	password
F	memory-hard function replaced in a particular instance of CATENA
λ	security parameter of F (depth)
s	salt (public random value)
t	tweak
d	domain (application specifier) of CATENA
V	unique version identifier
γ	public input (e.g., salt)
g_{low}	lowest garlic
g_{high}	highest garlic
H	underlying cryptographic hash function
H'	reduced version of H
n	output length of H and H' in byte
m	output length of CATENA in byte
Γ	function depending on the public input γ
h, y	password hash (or intermediate hash)
v_i^j	i-th vertex of the j-th row/BRG
${}^k v_i^j$	i-th vertex of the j-th row of the k-th DBG
τ	Bit-Reversal Permutation
σ	function determining the index of the diagonal edges (DBG)
AD	associated data
K	secret key
$\|X\|$	size of X in bits or size of a set X

Table 4.1: Slightly adapted notational conventions borrowed from [15].

[I] http://article.gmane.org/gmane.comp.security.phc/489

4.1 Specification

The CATENA framework consists of the main algorithm CATENA, the work function *flap* (see Algorithm 6) and several functions and modes that extend its functionality.

Algorithm 5 CATENA [15]

Input: pwd : Password, t : Tweak, s : Salt, g_{low} : Min. Garlic, g_{high} : Max. Garlic, λ : Depth, m : Output Length, γ :Public Input
Output: x : Hash of the Password
1: $x \leftarrow H(t \mid\mid pwd \mid\mid s)$
2: $x \leftarrow flap(\lceil g_{\text{low}}/2 \rceil, x, \gamma, \lambda)$
3: **for** g **from** g_{low} **to** g_{high} **do**
4: $\quad x \leftarrow flap(g, x \mid\mid 0^*, \gamma, \lambda)$
5: $\quad x \leftarrow H(g \mid\mid x)$
6: $\quad x \leftarrow truncate(x, m)$
7: **end for**
8: **return** x

The CATENA Function The main function shown in Algorithm 5 takes eight inputs: pwd, t, s, g_{low}, g_{high}, λ, m and γ. The password pwd is chosen by the user. Salt s and public input γ are random values that are determined before invoking CATENA. For most applications it is possible to use the same value for γ and s. The cost parameters and the output length m are fixed values of the authentication process, where g_{low} (minimal garlic) and g_{high} (maximal garlic) determine the memory usage, and λ denotes the depth of the underlying graph and therefore the time cost. The tweak t is computed as:

$$t \leftarrow H(V) \mid\mid d \mid\mid \lambda \mid\mid m \mid\mid |s| \mid\mid H(\text{AD}),$$

with V being a unique version identifier and d being a 1-byte domain identifier. The parameters λ, m and $|s|$ are 1-byte values denoting the depth of the graph, the output length in byte and the salt length in byte. The values of V for the default instances (see Section 4.3) are given in Table 4.2. All currently defined values for d can be obtained from Table 4.3. AD is a custom string of associated data that should contain information that is unique to the authentication process. The tweak can be made unique for every user by adding, for example, the user ID. In this case the tweak can function - similar to the salt - as a random public input. The hash function H used to hash AD and V, depends on the instance. While the presented tweak structure is employed by all default instances (see Section 4.3), adapting the tweak for custom variants of CATENA is possible.

The main algorithm (see Algorithm 5) starts by computing the hash of the tweak, the password, and the salt. This allows to erase the password from memory after this step. To provide resistance against weak garbage-collector attacks (see Section 2.3), the hash is then processed once by the work function *flap* with half of the memory cost. After this

Name	F	H'	V
Catena-Dragonfly	BRH_λ^g	BLAKE2b-1	"Dragonfly"
Catena-Dragonfly-Full	BRH_λ^g	BLAKE2b	"Dragonfly-Full"
Catena-Butterfly	DBH_λ^g	BLAKE2b-1	"Butterfly"
Catena-Butterfly-Full	DBH_λ^g	BLAKE2b	"Butterfly-Full"

Table 4.2: Overview of the default instances of Catena [15].

Mode	d
password scrambling	0
key derivation	1
proof of work	2

Table 4.3: Values for the domain identifier d.

step neither the password nor any easy-to-compute derivative of it remain in memory. The main loop executes three steps for every value g from g_{low} to g_{high}. First, the $flap$ function is called with the memory-cost parameter g. If m is less than n, x needs to be padded with 0's to fit the output size n of the underlying hash functions H and H'. The output of $flap$ is then hashed together with g and finally truncated to the m least significant bits. Performing the hashing and truncation every iteration is essential for the support of client-independent updates (see Section 2.2).

Algorithm 6 Function $flap$ of Catena [15]

Input: g : Garlic, x : Value to Hash, γ: Public Input, λ : Depth
Output: x : Intermediate Hash Value

1: $v_{-2} \leftarrow x \oplus 1$
2: $v_{-1} \leftarrow x$
3: $v_0 \leftarrow H(v_{-1} \mid\mid v_{-2})$
4: **for** i **from** 1 **to** $2^g - 1$ **do**
5: $\quad v_i \leftarrow H'(v_{i-1} \mid\mid v_{i-2})$ ▷ initialize the memory
6: **end for**
7: $v \leftarrow \Gamma(g, v, \gamma)$ ▷ one row with γ-based memory accesses
8: $x \leftarrow F(g, v, \lambda)$ ▷ memory-hard function
9: **return** x

The *flap* Function The actual work is managed by the $flap$ function shown in Algorithm 6. Γ and F require $2^g \cdot n$ bits of memory v to operate on, where g denotes the current garlic. The internal state v is initialized in Lines 1 to 6 with a sequential chain of derivatives of the input value x. Afterwards, the Γ function, which uses γ-based memory access, is called. The final step of $flap$ is the application of the memory-hard function F to v.

Actual Functions and Values The actual functions used for H, H', Γ and F as well as the value V (from the tweak) depend on the instance of Catena. The default instances are presented in Section 4.3.

4.2 Features

This sections shows that Catena supports all the desirable features listed in Section 2.2.

Algorithm 7 The client-independent-update function [15]

Input: h : Old Hash, g_{high} : Old Max. Garlic, λ : Depth, m : Output Length, γ :Public Input, g'_{high} : New Max. Garlic
Output: h' : Updated Hash

1: $h' \leftarrow h$
2: **for** g **from** g_{high} **to** g'_{high} **do**
3: $\quad h' \leftarrow flap(g, h' \mid\mid 0^*, \gamma, \lambda)$
4: $\quad h' \leftarrow H(g \mid\mid h')$
5: $\quad h' \leftarrow truncate(h', m)$
6: **end for**
7: **return** h'

The Client-IndependentUpdate Function Any hash computed with Catena can be updated with a client-independent update. The function that updates a hash h with an outdated memory-cost parameter g_{high} to a new hash h' with a larger memory cost parameter g'_{high} can be seen in Algorithm 7.

The Server-Relief Function Server-relief (see Section 2.2) can be implemented by omitting the Lines 5 and 6 from Algorithm 5 in the final iteration of the loop, *i.e.* when $g = g_{\text{high}}$. The client sends the resulting x to the server, which then computes

$$h = truncate(H(g \mid\mid x), m)$$

to get the final hash h. H and *truncate* are far less computationally expensive than *flap* and their memory requirement is negligible.

The Key-Derivation Function Catena can be extended to support generating keys of arbitrary length. This mode is called Catena-KG and presented in Algorithm 8. The key identifier $\mathcal{I}$ allows to generate different and independent keys of length ℓ from the same password. In case that several keys are required at once, the invocation of Catena in Line 1 can be separated from the rest of the computation and x can be kept in memory as long as new keys are required. Please be aware that the domain identifier d of the tweak t' is 1 for key derivation.

Algorithm 8 CATENA-KG [15]

Input: pwd : Password, t' : Tweak, s : Salt, g_{low} : Min. Garlic, g_{high} : Max. Garlic, γ : Public Input, ℓ : Key Size, $\mathcal{I}$: Key Identifier, λ : Depth
Output: k : ℓ-Bit Key Derived from the Password
1: $x \leftarrow \text{CATENA}(pwd, t', s, g_{\text{low}}, g_{\text{high}}, \lambda, n, \gamma)$
2: $k \leftarrow \emptyset$
3: **for** i **from** 1 **to** $\lceil \ell/n \rceil$ **do**
4: $\quad k \leftarrow k \mid\mid H(i \mid\mid \mathcal{I} \mid\mid \ell \mid\mid x)$
5: **end for**
6: **return** $truncate(k, \ell)$ ▷ truncate k to the first ℓ bits

Keyed Password Hashing For keyed password hashing (see Section 2.2) with CATENA, encryption and decryption are the same operation, which is defined as:

$$y = x \oplus H(K \mid\mid userID \mid\mid g \mid\mid K),$$

where x is the value to encrypt or decrypt, K denotes the secret key and $userID$ is a unique and user-specific identification number. For regular CATENA without server relief, the encryption is applied after Line 7 in Algorithm 5. Keyed server relief can be implemented by encrypting h. During client-independent updates, the old hash h has to be decrypted before Line 1 of Algorithm 7. The new hash h' must then be encrypted after Line 6. Note that the decryption during client-independent updates uses the old garlic g and the encryption uses the new garlic g'.

Proof of Work The approaches for proof-of-work systems presented in Section 2.2 can be used with CATENA without any changes to the algorithm. For the challenge approach, the input that the client has to find could be either the salt[II] or a password.

Flexibility It is possible to adapt the CATENA framework to different environments. For a scenario that requires the computations to be parallel, part of the salt could be kept secret[II]. Then, the attacker and defender would have to guess the correct salt. The search for the correct salt can be done in parallel with the restriction that b parallel threads increase the memory requirement by a factor of b. A different scenario that requires the use of a specific hash function, e.g., because it has optimized implementations for all involved system, could be satisfied by a custom instance of CATENA.

Security The fundamental security of the CATENA framework is described by its preimage security (see Section 2.1) and its pseudorandomness. Pseudorandomness denotes the property that an attacker can not gain any significant advantage at distinguishing CATENA from a random function. Chapter 4 of [15] shows that CATENA inherits both

[II]The secret part of the salt is also referred to as pepper

security properties from the underlying hash function H. The pseudorandomness property also holds for Catena-KG, where it is a mandatory property [15, Chapter 8].

Catena provides resistance against the two major classes of side-channel attacks for password-hashing schemes, garbage-collector attacks and cache-timing attacks (see Section 2.3). While security against garbage-collector attacks also depends on the implementation, Catena allows implementations to safely erase the password from memory very early in the algorithm (after Line 1 of Algorithm 5). The measures taken to defend the reference-implementation against garbage-collector attacks are explained in Section 5.1. The resistance against cache-timing attacks comes from the the data-independent memory access pattern.

The memory-hardness (see Definition 2.1) of Catena depends on the actual instance and is therefore discussed in Section 4.3.

4.3 Instances

This section specifies the four default instances of Catena: Catena-Dragonfly, Catena-Butterfly, Catena-Dragonfly-Full and Catena-Butterfly-Full, as well as the functions BLAKE2b-1 and SaltMix, that are both shared between multiple instances.

All instances operate on the memory filled in the $flap$ function. The values in the internal state v are first updated based on the salt and then by using a memory-access pattern based on a given graph, where the memory blocks can be seen as the vertices of the underlying graph. Each block is as long as the output length of the hash function(s). In the concrete instances, this results in a block size of 512 bit.

BLAKE2b-1 BLAKE2b-1 is a modification of BLAKE2b that reduces the number of rounds from 12 to one. It is used by two of the default instances, Catena-Dragonfly and Catena-Butterfly, for H'. For BLAKE2b-1, all invocations of H' need to be augmented with the index of the vertex that is currently computed. The differences between the HashFast function that consitutes BLAKE2b-1 and the main BLAKE2 function of BLAKE2b are shown in Algorithm 9. Please note that the later is abstracted from the pseudocode given in the IETF draft [46]. The functionality is split into `blake2b_update` and `blake2b_final` in the reference implementation of BLAKE2b [38].

As shown in Line 1 of Algorithm 9(b), the state S gets initialized for every execution of BLAKE2, whereas BLAKE2b-1 initializes the state only once every few invocations. For the default instances, the number of invocations that reuse the same Blake2b-1 state is derived from the underlying graph structure. Reusing the state between executions assures that 12 iterations of BLAKE2b-1 are as close as possible to BLAKE2b. Table 4.4 shows that reusing the state also significantly reduces the computation time.

HashFast

Input: I1 : Input1, I2 : Input2, v : Vertex Index
Globals: S : BLAKE2b state
Output: h : Hash

```
1: S.buf ← I1 || I2
2: S.buflen ← 128
3: increment_counter(S, S.buflen)
4: set_last_block(S)
5: r ← v mod 12
6: compress(S, r)
7: h ← S.h
8: return h
```

(a) BLAKE2b-1

BLAKE2

Input: I : Input Array, l : Input Length in Byte, m : Output Length
Output: h : Hash

```
 1: blake2b_init(S, m)                    ▷ init state
 2: for i from 0 to (⌊l/128⌋ − 1) do
 3:     S.buf ← I[i]
 4:     S.buflen ← 128
 5:     increment_counter(S, S.buflen)
 6:     compress(S)
 7: end for
 8: S.buf ← I[⌊l/128⌋] || 0*
 9: S.buflen ← l mod 128
10: increment_counter(S, S.buflen)
11: set_last_block(S)

12: compress(S)
13: h ← S.h
14: return h
```

(b) BLAKE2b

Algorithm 9: The main functions of BLAKE2b-1 and BLAKE2b.

Algorithm	**Median clocks per byte**
BLAKE2b-1	2.44
BLAKE2b-1 without initialization	0.86
BLAKE2b	9.81

Table 4.4: Benchmark comparing BLAKE2b-1 with BLAKE2b on a Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz.

BLAKE2b-1 uses the functions `blake2b_init`, `increment_counter` and `set_last_block` from the BLAKE2b reference implementation without any modification. The `blake2b_init` function creates a parameter block $\mathcal{P}$ with the default settings for sequential hashing and the specified output length. The output length is fixed to 64 byte for BLAKE2b-1. The state S is then zeroed and the chain value $S.h$ is set to $IV \oplus \mathcal{P}$, where IV denotes the initialization vector of BLAKE2b. $S.t$, the 128-bit input-length counter, is incremented by `increment_counter`. The function `set_last_block` sets the the lower word of the finalization flag $S.f$ to $2^{64} - 1$, *i.e.* all bits to 1.

The fixed input length of 128 byte allows BLAKE2b-1 to omit the handling of longer inputs (Line 2 to 7 of Algorithm 9(b)) as well as the padding done in Line 8. Another change from BLAKE2b to BLAKE2b-1 is the additional computation of the round index in Line 5 of Algorithm 9(a).

The compression functions of BLAKE2b and BLAKE2b-1 are shown in Algorithm 10. The only difference is the omission of the loop in BLAKE2b-1. The G function, the initialization vector and the message schedule are the same as in BLAKE2b. For BLAKE2b

and BLAKE2b-1, σ denotes the message schedule and should not be confused with the function σ of CATENA-BUTTERFLY.

```
compress
Input: S : BLAKE2b State, r : Round Index
Globals: IV : Initialization Vector,
    σ : Message Schedule
 1: v[0...7] ← S.h
 2: v[8...15] ← IV
 3: v[12,13] ← v[12,13] ⊕ S.t
 4: v[14,15] ← v[14,15] ⊕ S.f

 5: s[0...15] ← σ[r mod 10][0...15]
 6: v ← G(v,0,4,8,12,S.buf[s[0]],S.buf[s[1]])
 7: v ← G(v,1,5,9,13,S.buf[s[2]],S.buf[s[3]])
 8: v ← G(v,2,6,10,14,S.buf[s[4]],S.buf[s[5]])
 9: v ← G(v,3,7,11,15,S.buf[s[6]],S.buf[s[7]])
10: v ← G(v,0,5,10,15,S.buf[s[8]],S.buf[s[9]])
11: v ← G(v,1,6,11,12,S.buf[s[10]],S.buf[s[11]])
12: v ← G(v,2,7,8,13,S.buf[s[12]],S.buf[s[13]])
13: v ← G(v,3,4,9,14,S.buf[s[14]],S.buf[s[15]])

14: S.h ← S.h ⊕ v[0...7] ⊕ v[8...15]
```

(a) BLAKE2b-1

```
compress
Input: S : BLAKE2b State
Globals: IV : Initialization Vector,
    σ : Message Schedule
 1: v[0...7] ← S.h
 2: v[8...15] ← IV
 3: v[12,13] ← v[12,13] ⊕ S.t
 4: v[14,15] ← v[14,15] ⊕ S.f
 5: for r from 0 to 11 do
 6:     s[0...15] ← σ[r mod 10][0...15]
 7:     v ← G(v,0,4,8,12,S.buf[s[0]],S.buf[s[1]])
 8:     v ← G(v,1,5,9,13,S.buf[s[2]],S.buf[s[3]])
 9:     v ← G(v,2,6,10,14,S.buf[s[4]],S.buf[s[5]])
10:     v ← G(v,3,7,11,15,S.buf[s[6]],S.buf[s[7]])
11:     v ← G(v,0,5,10,15,S.buf[s[8]],S.buf[s[9]])
12:     v ← G(v,1,6,11,12,S.buf[s[10]],S.buf[s[11]])
13:     v ← G(v,2,7,8,13,S.buf[s[12]],S.buf[s[13]])
14:     v ← G(v,3,4,9,14,S.buf[s[14]],S.buf[s[15]])
15: end for
16: S.h ← S.h ⊕ v[0...7] ⊕ v[8...15]
```

(b) BLAKE2b

Algorithm 10: The *compress* functions of BLAKE2b-1 and BLAKE2b.

SALTMIX All default instantiations of CATENA use SALTMIX for Γ (see Line 7 of Algorithm 6). The function SALTMIX, as defined in Algorithm 11, updates the state array v using salt-dependent accesses; hence, γ is set to s. In contrast to the functions used for F, memory access pattern of SALTMIX changes with the salt and is therefore most probably different for every user.

SALTMIX updates the state array v $2^{\lceil 3g/4 \rceil} - 1$ times. For every update, it gets two random values j_1 and j_2 from `xorshift1024star` [52], extracts the g most significant bits from both and then updates the state word v_{j_1} to $H'(v_{j_1} \mid\mid v_{j_2})$.

The algorithm `xorshift1024star` is a statistically sound random-number generator that works on a 1024-bit state of 16 64-bit words. As shown in Line 1 of Algorithm 11(a), the state is initialized with the two 512-bit values $H(s)$ and $H(H(s))$.

4.3.1 CATENA-DRAGONFLY & CATENA-DRAGONFLY-FULL

CATENA-DRAGONFLY and CATENA-DRAGONFLY-FULL use SALTMIX for Γ and (g, λ)-bit-reversal hashing (BRH^g_λ) for F. While CATENA-DRAGONFLY uses BLAKE2b for H and BLAKE2b-1 for H', CATENA-DRAGONFLY-FULL uses BLAKE2b for both.

BRH^g_λ is defined in Algorithm 12 and an example of the underlying (g, λ)-bit-reversal graph (BRG^g_λ), where $g = 3$ and $\lambda = 2$, is shown in Figure 4.4. The edges of the graph and therefore the memory accesses of the algorithm are calculated using the bit-reversal

```
SaltMix
Input: g : Garlic, v : State, s : Salt
Output: v : Updated State
 1: r ← H(s) || H(H(s))
 2: p ← 0
 3: for i from 0 to 2^⌈3g/4⌉ − 1 do
 4:     (j1, r, p) ← xorshift1024star(r, p)
 5:     (j2, r, p) ← xorshift1024star(r, p)
 6:     j1 ← j1 ≫ (64 − g)
 7:     j2 ← j2 ≫ (64 − g)
 8:     v_j1 ← H'(v_j1 || v_j2)
 9: end for
10: return v
```

(a) SaltMix

```
xorshift1024star
Input: r : State, p : Index
Output: idx : Current Index
 1: s0 ← r_p
 2: p ← (p + 1) mod 16
 3: s1 ← r_p
 4: s1 ← s1 ⊕ (s1 ≪ 31)
 5: s1 ← s1 ⊕ (s1 ≫ 11)
 6: s0 ← s0 ⊕ (s0 ≫ 30)
 7: r_p ← s0 ⊕ s1
 8: idx ← r_p · 1181783497276652981
 9: return (idx, r, p)
```

(b) xorshift1024star

Algorithm 11: The functions SaltMix and xorshift1024star [15].

permutation τ. An additional edge connects the last vertex $v^j_{2^g-1}$ of the j-th row with the first vertex v^{j+1}_0 of the $(j+1)$-th row. The function τ is defined as the reversal of the binary representation of a number. The computation of each vertex v^j_i (Line 4) depends on the previous vertex v^j_{i-1} or $v^{j-1}_{2^g-1}$ and $v^{j-1}_{\tau(i)}$.

Since only two rows are required for every computation, the notation for the algorithmic representation can be simplified. In Algorithm 12, r_i denotes the i-th vertex of the current row and is therefore equivalent to v^j_i from the graph representation. The previously computed row is denoted by the vector v without any superscript.

A BRG^g_λ has $\lambda+1$ rows with 2^g vertices each. Since the first row is filled by the initialization step of $flap$ (Lines 1 to 6 of Algorithm 6) and updated by Γ (Line 7 of Algorithm 6), computing BRH^g_λ takes $\lambda \cdot (2^g - 1)$ invocations of H' and λ invocations of H. Because of their 1024-bit input size, BLAKE2b and BLAKE2b-1 need only one compression function call per invocation.

Catena-Dragonfly is proven to be memory-hard (in terms of λ-memory-hardness: 1-memory-hard) with only minor improvements from increasing λ [15] [5].

Algorithm 12 (g, λ)-Bit-Reversal Hashing (BRH^g_λ) [15].

```
Input: g : Garlic, v : State Array, λ : Depth
Output: x : Password Hash
 1: for j from 1 to λ do
 2:     r0 ← H(v_{2^g−1} || v0)
 3:     for i from 1 to 2^g − 1 do
 4:         r_i ← H'(r_{i−1} || v_τ(i))
 5:     end for
 6:     v ← r
 7: end for
 8: return r_{2^g−1}
```

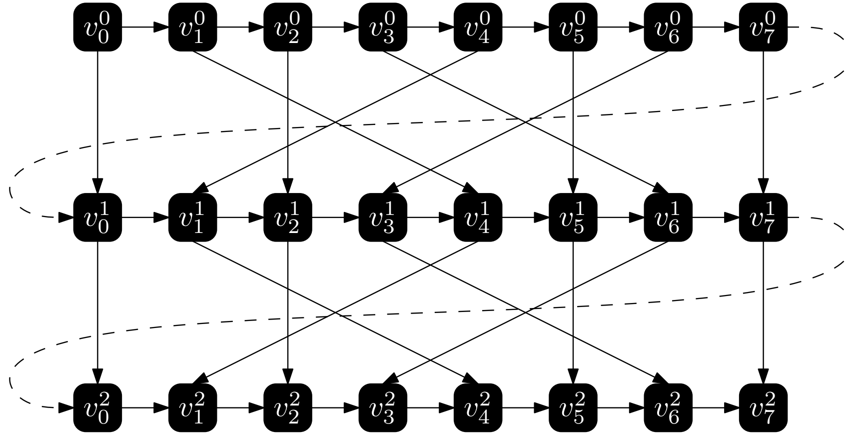

Figure 4.4: A $(3, 2)$-bit-reversal graph.

4.3.2 Catena-Butterfly & Catena-Butterfly-Full

Similarly to the variants of Catena-Dragonfly, both variants of Catena-Butterfly set $\Gamma = $ SaltMix and $H = BLAKE2b$. However, Catena-Butterfly and Catena-Butterfly-Full use (g, λ)-double-butterfly hashing (DBH_λ^g) for F. Catena-Butterfly-Full sets $H' = H$, whereas Catena-Butterfly additionally employs BLAKE2b-1 for H'.

Figure 4.5 shows an example of the graph that double-butterfly hashing is based on with $g = 3$ and $\lambda = 1$. A double-butterfly graph (DBG) is a G-superconcentrator, *i.e.* it has q vertex-disjoint paths that connect any subset of size q of its G input nodes with any subset of size q of its G output nodes with $q \leq G$ [6].

Let DBG_λ^g denote a graph consisting of λ stacked DBGs with 2^g vertices in each row and let ${}^k v_i^j$ be the i-th vertex in the j-th row of the k-th DBG. For double-butterfly hashing, each vertex ${}^k v_i^j$ of the graph is calculated from three different inputs: its vertical predecessor ${}^k v_i^{j-1}$, its sequential predecessor ${}^k v_{i-1}^j$ or ${}^k v_{2^g-1}^{j-1}$ and the vertex given by the diagonal connection ${}^k v_{\sigma(g,i,(j-1))}^{j-1}$. The function σ is defined as

$$\sigma(g, i, j) = \begin{cases} i \oplus 2^{g-1-j} & \text{if } 0 \leq j \leq g-1, \\ i \oplus 2^{j-(g-1)} & \text{otherwise.} \end{cases}$$

Algorithm 13 defines the resulting (g, λ)-double-butterfly hashing. The notation for the algorithmic representation has been simplified similarly to BRH. The vectors r and v denote the j-th and the $(j-1)$-th row, respectively.

A (g, λ)-double-butterfly graph consists of λ double-butterfly graphs with $2 \cdot g$ rows that in turn consist of 2^g vertices each. Since two adjacent graphs share one row, the total number of rows in the graph is $(\lambda \cdot (2 \cdot g - 1)) + 1$. Hence, the computation of DBH_λ^g requires $(\lambda \cdot (2 \cdot g - 1)) \cdot (2^g - 1)$ invocations of H' and $\lambda \cdot (2 \cdot g - 1)$ invocations of H. This does not

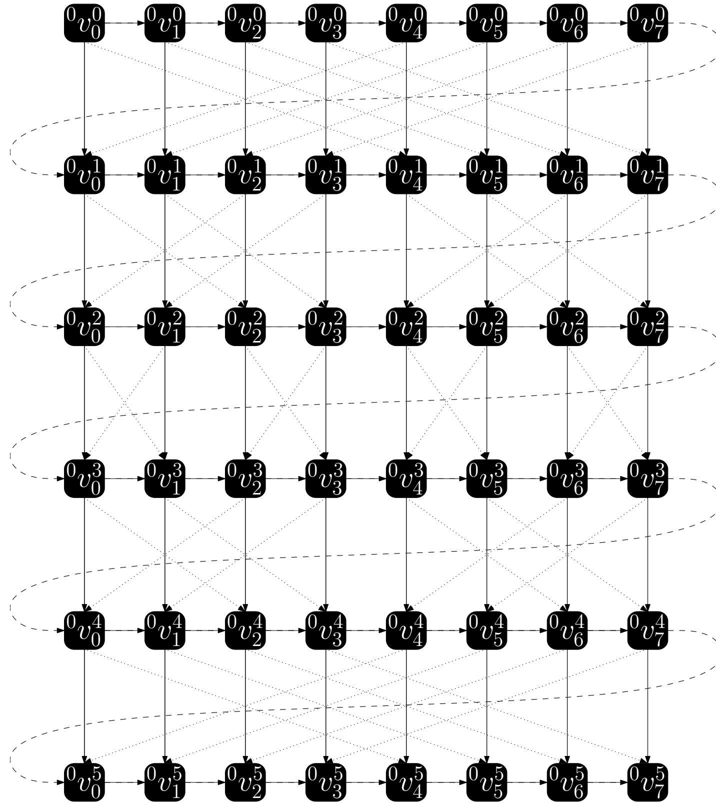

Figure 4.5: A $(3, 1)$-double-butterfly graph consisting of three layers: vertical (solid lines), diagonal (dotted lines), sequential and connecting (dashed lines).

include the computation of the first row that is done by the $flap$ and SaltMix functions for all default instances.

XORing the first two inputs of H or H' (Lines 3 and 5 of Algorithm 13) ensures that the complexity of the calls to H and H' is the same for DBH_λ^g and BRH_λ^g. The recommended hash functions BLAKE2b and BLAKE2b-1 both work with a block size of 1024 bit so that one invocation of the hash function requires only one invocation of the compression function.

Lengauer and Tarjan established time-memory tradeoffs for a stack of λ superconcentrators that conform with λ-memory-hardness [30] [15]. Since a (g, λ)-double-butterfly graph is just a specific G-superconcentrator, their analysis holds for (g, λ)-double-butterfly hashing and therefore, for Catena-Butterfly and Catena-Butterfly-Full.

4.4 Parameter Recommendation

Practical applications of Catena, like authentication processes or key generation, require the choice of one specific instance with fixed cost parameters to ensure interoperability. While it is possible to update the cost parameters retrospectively, the initial cost parameters as well as the updated ones have to be chosen carefully.

Algorithm 13 (g, λ)-Double-Butterfly Hashing (DBH^g_λ) [15].

Input: g : Garlic, v : State Array, λ : Depth
Output: x : Password Hash
1: **for** k **from** 1 **to** λ **do**
2:     **for** j **from** 1 **to** $2g-1$ **do**
3:         $r_0 \leftarrow H(v_{2^g-1} \oplus v_0 \mid\mid v_{\sigma(g,0,(j-1))})$
4:         **for** i **from** 1 **to** 2^g-1 **do**
5:             $r_i \leftarrow H'(r_{i-1} \oplus v_i \mid\mid v_{\sigma(g,i,(j-1))})$
6:         **end for**
7:         $v \leftarrow r$
8:     **end for**
9: **end for**
10: **return** v_{2^g-1}

Instances For scenarios that do not require a λ-memory-hardness with $\lambda > 1$ and scenarios that require a fast memory fill rate, Catena-Dragonfly is preferable to Catena-Butterfly. Please note that while it wont increase the memory-hardness by much, choosing a $\lambda \geq 2$ for Catena-Dragonfly drastically decreases the risk of garbage-collector attacks. Catena-Butterfly has the advantage of achieving λ-memory-hardness. In scenarios that require a high time-memory tradeoff, it is therefore recommended to use Catena-Butterfly. An attacker that reduces the memory by a factor of ten will need 10^λ times more effort to compute DBH^λ_g.

In addition to increasing the time cost by increasing λ, it is also possible to use Catena-Butterfly-Full or Catena-Dragonfly-Full that both take longer to compute than Catena-Butterfly or Catena-Dragonfly, respectively.

Cost Parameters The lower limits for g and λ can be derived semantically from Algorithm 12 and 13 as 1. Setting $g = 0$ or $\lambda = 0$ would result in the omission of loops and therefore alter the hashing process significantly. It can be derived from Algorithm 5 that $g_{\text{low}} \leq g$ holds. Since g_{low} is also used as the garlic for one iteration of the loop in Lines 3 to 7 of Algorithm 5, it has the same restrictions as g and therefore $g_{\text{low}} > 0$ must hold.

While there is no upper limit for both g and λ, implementations might restrict them for performance improvements. For example, support for garlic values larger than 63 on modern 64-bit systems requires a more complicated approach on memory management.

The cost parameters are not only limited by the upper and lower limits but also by the hardware and the time designated for the process. Thus, they should obviously always be chosen for the most restricted or the busiest system involved. It is common practice to select cost parameters that are only just bearable to prevent any unnecessary advantage for an attacker. Forler, Lucks, and Wenzel set the time limit for tolerability to about 0.5 s for authentication and 5 s for key derivation [15]. Table 4.5 shows cost parameters for all default instances that match these restrictions on current commercial off-the-shelf (COTS)

Password Hashing			
Algorithm	$g_{\text{low}}/g_{\text{high}}$	λ	Time
Catena-Dragonfly	21/21	2	0.51 sec
Catena-Dragonfly-Full	18/18	2	0.31 sec
Catena-Butterfly	16/16	4	0.46 sec
Catena-Butterfly-Full	14/14	4	0.30 sec
Key Derivation			
Catena-Dragonfly-Full	22/22	2	3.90 sec
Catena-Butterfly-Full	17/17	4	4.75 sec

Table 4.5: Recommended parameter sets for average systems. All timings are measured on an Intel Core i5-2520M CPU (2.50GHz) system [15].

computers. The memory requirements posed by these recommended parameters are at most about 128 Mb for authentication and about 256 Mb for key-derivation and should therefore be of no concern for modern COTS systems. Note that the continuously rising computational power will lead to these recommendations becoming outdated in just a few years.

Busy servers or systems that significantly differ from COTS computers might require individually adjusted cost parameters. Adapting the cost parameters to technological progress also requires to determine suitable parameters. Catena-axungia (see Section 5.2) allows to search for optimal parameters under given time and memory constrains for the default instances.

Chapter 5

Implementation

> [...] it is also important to use a fast implementation, since the whole concept of slow hashing is a muscle contest between the attacker's machine and the defender's machine.
>
> *Thomas Pornin*[I]

This chapter focuses on the implementation of the three applications that were created for this thesis. The first one is the reference implementation, which includes all default instances of CATENA. CATENA-AXUNGIA, the second application, is a unique search tool for optimal cost parameters for CATENA. The third application is CATENA-VARIANTS, a flexible and extendable implementation of CATENA.

To ensure portability, all implementations presented in this chapter rely on the aliases defined in `stdint.h` [26]. While the types defined by the C99 standard may vary in width as long as they provide a certain minimum precision, all datatypes defined in `stdint.h` have a fixed width on all platforms. Relevant for this thesis are the types `uint8_t`, `uint32_t` and `uint64_t`, which are defined as unsigned integers types of 8 bit, 32 bit and 64 bit, respectively.

5.1 CATENA

My work on the CATENA reference implementation started in September of 2014 and continues the preexisting reference implementation of CATENA V 1[II] by Christian Forler with contributions from Steve Thomas and Bill Cox. The differences between the current

[I] `http://security.stackexchange.com/a/34156`

[II] Final commit of the CATENA V 1 reference implementation: `https://github.com/medsec/catena/tree/3f13e78050547c1e01cf9587da94e0bf6652677b`

reference implementation and the one of CATENA V 1 include, besides various enhancements, the support for H', the extension by CATENA-DRAGONFLY, BLAKE2b-1 and SALTMIX, and the defense against garbage-collector attacks. The reference implementation is released under the MIT license and can be found on

`https://github.com/medsec/catena`.

It is written in the C99 version of the C programming language [26]. The API is exposed in `catena.h` and implemented in `catena.c`. Helper functions shared between multiple instances are defined in `catena-helpers.h` and implemented in `catena-helpers.c`. In addition to SALTMIX, Γ and a function resembling the memory-initialization part of $flap$, these functions also cover *password overwriting* and XOR of two 512-bit values with and without the use of intrinsics.

The header `hash.h` defines the actual functions representing the hash functions H and H'. These are implemented in `catena-blake2b-ref.c` and `catena-blake2b-sse.c`, where the suffix differentiates between the regular and the SSE-optimized version of BLAKE2b and BLAKE2b-1. While BLAKE2b is implemented by simply providing a wrapper for the reference implementation of BLAKE2b [38] found in the folders `blake2-ref` and `blake2-sse`, the aforementioned files also contain the algorithms constituting BLAKE2b-1 (see Section 4.3).

The $flap$ interface for all instances is defined in `catena.h` and implemented in `catena-BRG.h` for CATENA-DRAGONFLY and CATENA-DRAGONFLY-FULL and `catena-DBG.h` for CATENA-BUTTERFLY and CATENA-BUTTERFLY-FULL. The distinction of the -FULL instances is made in `catena-blake2b-ref.c` and `catena-blake2b-sse.c`. While it would be possible to use the same $flap$ function for all instances, the differences in memory allocation (see Section 5.1.2 and 5.1.1) would require a lot of instance-depending code.

Interface The exposed functions that constitute the CATENA API are mostly inherited from the preexisting implementation. During this thesis, functions for keyed server-relief and keyed client-independent updates were added as complements to the already existing functions for hashing, client independent updates, the server and client part of server relief, and key generation. Except for key generation all of these functions are also available as a keyed version. CATENA additionally includes the official `PHS`-interface[III] that is a requirement for all candidates of the Password Hashing Competition. It provides a generalized interface for all candidates of the competition to allow easier testing and benchmarking. Please be aware that, as discussed later, the `PHS`-function is not available when *password overwriting* is enabled.

[III] https://password-hashing.net/call.html

While the choice of datatypes for the parameters is not specified by CATENA, an obvious choice can usually be derived from the type of data represented by the parameter. For the current reference implementation, λ as the parameter `lambda`, g_{low} and g_{high} as `min_garlic` and `garlic`, respectively, are represented as `uint8_t`. This limits all three parameters to a minimum value of 0 and a maximum value of 255, which is sufficient for all scenarios. The garlic parameters have to be restricted further to a maximum value of 63 so that the value $2^{g_{\text{high}}}$ can still be represented in a `uint64_t`. It is also highly unlikely that a defender will be able to provide $2^{64} \cdot n = 2^{70}$ bytes of memory anywhere in the near future. Data, like the salt s or the secret key K, is passed to CATENA as arrays of bytes represented by `uint8_t`. While the password *pwd* and the associated data AD are also arrays of `uint8_t`, they can also be considered as arrays of single-byte characters. It is up to the developer using CATENA, to decide if those parameters are passed as characters or bytes. Note that using the byte representation of larger types requires measures to ensure the same result on big- and little-endian architectures (see Section 2.4). The parameter for the user-specific identification number $userID$ is a `uint64_t`, which should be sufficiently future-proof. The nature of the C programming language requires the lengths of arrays to be provided as an additional parameter. Depending on context, the length parameters are either of type `uint8_t` or `uint32_t`.

The CATENA API provides sanity checks for most parameters. In addition to the aforementioned check for $g_{\text{high}} \leq 63$, the conditions $g_{\text{low}} \leq g_{\text{high}}$ and $\lambda > 0$ as well as $g_{\text{low}} > 0$ must hold. For all functions with the exception of key derivation, an additional check to ensure that the requested output length is less or equal to the output length of the hash function H is required. Since the `PHS`-interface uses different datatypes for the parameters, it requires additional checks to prevent overflows.

Compiling The reference implementation includes a `Makefile` that can be used with make [18] to build CATENA without directly invoking a compiler. While the reference implementation can be compiled with any C compiler that supports the C99 standard, *password overwriting* can only be fully ensured with GCC version 4.4 or newer [49] or clang version 3.5 or newer [47]. The benchmark presented in Appendix A.2 revealed that CATENA is noticeably faster when compiled with clang. The same benchmark also revealed that the optimization level O3 is at least as fast as any other available level. Therefore, clang with the flag `-O3` (see Appendix B.1) is used as the default compilation method. Additional flags include `-std=c99` to enable C99 support and `-march=native` to enable all intrinsic functions and optimizations available on the current system.

It is fairly safe to assume that only one instance of CATENA will be used per authentication process; it is therefore convenient to create executables that include only one instance to reduce the file size. The `Makefile` demonstrates how to build CATENA-DRAGONFLY and CATENA-BUTTERFLY independently and also includes an option to compile CATENA-DRAGONFLY-FULL and CATENA-BUTTERFLY-FULL instead.

Security While the resistance against cache-timing attacks and garbage-collector attacks is already given by the Catena specification, the resistance against weak garbage-collector attacks must be secured by the implementation. The invocation of $flap$ with reduced cost parameters in Line 2 of Algorithm 5 provides security against weak garbage-collector attacks that target the value x. Different attacks could try to retrieve the password from memory; therefore, the implementation must ensure its erasure after Line 1 of Algorithm 5.

Because optimizing compilers might leave out any changes to variables that are not accessed afterwards, measures have to be taken to ensure that the `memset` is actually executed. The reference implementation uses `__attribute__((optnone))` for clang and `__attribute__((optimize("O0")))` for GCC to instruct the compilers to skip the optimization step for the password-overwriting function. These compiler attributes are available from clang version 3.5 [48, *Attributes in Clang*] and GCC version 4.4 [17, Section 6.30] onwards. Furthermore, `free` is used to deallocate the memory that holds the password. This additionally counts as a use of the memory and should provide a reason for all compilers to retain the `memset` that erases the password.

Since it is impossible to impose restrictions on parameters, modifying or deallocating the memory that contains the password might lead to undefined behavior, e.g., when the password is held by a automatic variable or the password was declared constant. Hence, password overwriting is disabled by default in the reference implementation, but the compile process will issue a warning. Enabling password overwriting with the flag `SAFE` silences the warning. Please note that this will disable the `PHS`-interface, because the password has to be declared as `const` to match its function prototype.

Hash Functions Catena-Dragonfly and Catena-Butterfly use BLAKE2b-1 for H' and BLAKE2b for H, while Catena-Dragonfly-Full and Catena-Butterfly-Full use BLAKE2b for both. The reference implementation of Catena contains the regular as well as the SSE-optimized version of the BLAKE2b reference implementation [38]. Similarly, two versions of BLAKE2b-1 are included for systems with and without SSE. The `Makefile` determines if sufficient SSE support[IV] is present and then chooses the versions of BLAKE2b and BLAKE2b-1 accordingly.

Endianess The reference implementation includes two measures to make the output of Catena endianess-independent (see Section 2.4). First, the key length ℓ and the iteration count i for Catena-KG as well as the $userID$ for all keyed functions have to be converted to little endian. If checking the `__BYTE_ORDER` define from `endian.h` reveals that the system is big endian, the byte order of the values is changed using the functions `bswap_64` and `bswap_32` from `byteswap.h`. Second, the seed for `xorshift1024star`, saved as array of 16 64-bit integers, has to be loaded endianess-independently from two arrays of 64

[IV] The SSE-optimized versions of BLAKE2b and BLAKE2b-1 require at least SSE2 and provide further optimizations for later SSE versions and AVX

`uint8_t` each. This is implemented in the function `initXSState` by shifting and ORing each 8-bit word separately into the correct position. While the loading could be sped up on little-endian systems by simply copying the memory area of the two input arrays into the seed array, the impact would be negligible since this functionality is required only once per invocation of $flap$.

5.1.1 Catena-Dragonfly

The reference implementation uses improved algorithms for (g, λ)-bit-reversal hashing and the function τ to reduce the memory usage and increase the speed of Catena-Dragonfly (see Section 4.3.1).

Algorithm 14 shows the optimization of BRH_λ^g implemented by Steve Thomas[V]. In contrast to a naive implementation of Algorithm 12 that would require $2 \cdot 2^g$ 512-bit blocks of memory, the optimization only requires 2^g blocks. By traversing and saving the nodes in bit-reversal order in every odd row, the predecessor of every node can immediately be replaced. Because τ is an involution, the traversal of even rows stays sequential. The traversal and memory layout can be seen in Figure 5.1. Despite the varying memory layout of the rows, no additional consideration of the alignment is required for the return value since the last block of every row always contains the last node v_{2^g-1} of the row.

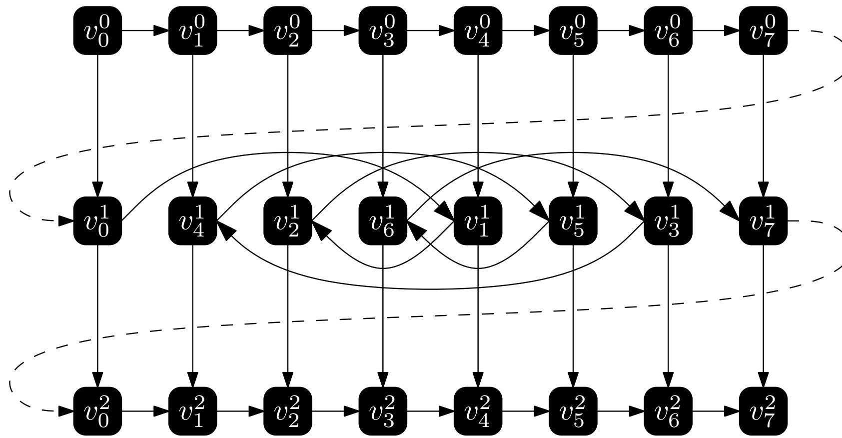

Figure 5.1: A $(3, 2)$-bit-reversal graph altered to match the traversal of the optimization.

Instead of reversing the bits of an integer separately, the function τ as implemented by Christian Forler[VI] starts by swapping the byte order using the `bswap_64` function and then proceeds to swap adjacent groups of four, two and finally single bits. To reduce the number of operations, the groups are extracted using bitmasks and all groups of a specific size are processed at once.

[V] https://github.com/medsec/catena/commit/3266baad8596dec2ed8e68625b8676e63d2150d9
[VI] https://github.com/medsec/catena/commit/89b17d6a4570fbb776ba91c4521bdb94fd647c31

Algorithm 14 Optimized (g, λ)-Bit-Reversal Hashing (BRH^g_λ)

Input: g : Garlic, v : State Array, λ : Depth
Output: x : Password Hash

1: **for** j **from** 1 **to** λ **do**
2: $v_0 \leftarrow H(v_{2^g-1} \mid\mid v_0)$
3: **for** i **from** 1 **to** $2^g - 1$ **do**
4: $v_{\tau(i)} \leftarrow H'(v_{\tau(i-1)} \mid\mid v_{\tau(i)})$
5: **end for**
6: $j \leftarrow j + 1$
7: **if** $j = \lambda$ **then**
8: **break**
9: **end if**
10: $v_0 \leftarrow H(v_{2^g-1} \mid\mid v_0)$
11: **for** i **from** 1 **to** $2^g - 1$ **do**
12: $v_{\tau(i)} \leftarrow H'(v_{\tau(i-1)} \mid\mid v_{\tau(i)})$
13: **end for**
14: **end for**
15: **return** v_{2^g-1}

5.1.2 Catena-Butterfly

This subsection describes two ways of computing (g, λ)-double-butterfly hashing (Section 4.3.2): a fast one which requires $1.5 \cdot 2^g$ 512-bit blocks of memory and a slower one that has a memory requirement of 2^g blocks.

The reference implementation of Catena-Butterfly and Catena-Butterfly-Full uses the fast algorithm since the resulting implementation is easier to understand and it is unfavorable for Catena to be slowed down. The reduced-memory algorithm would clash with the requirement for a simple and readable implementation[VII], because it differs significantly from the naive algorithm for computing DBH^g_λ (see Algorithm 13) and requires a far more complicated traversal. Moreover, the reduced-memory algorithm should be sped up by computing some steps in parallel, which would make the implementation even more complex. In light of Catena being among the slowest password-hashing schemes in the finale of the PHC[8], slowing it down any further is undesirable.

Fast Algorithm Similar to Catena-Dragonfly, the naive algorithm for computing (g, λ)-double-butterfly hashing shown in Algorithm 13 requires $2 \cdot 2^g$ blocks of memory. This memory requirement can be reduced by replacing a vertex as soon as all computations depending on it have been conducted, *i.e.* all vertices connected by the sequential, vertical and diagonal layer of the graph have been computed.

While it is obvious at which point the dependencies from the vertical and sequential edges will be resolved, the length of the diagonal edges differs between rows. The length of a

[VII] https://password-hashing.net/call.html

diagonal edge is given by the difference between the index of the source and the destination vertex and longer diagonal edges require more computations until their dependencies are resolved. The largest index difference which determines the memory requirement can be derived from the function that defines the diagonal edges. This function σ is defined as

$$\sigma(g,i,j) = \begin{cases} i \oplus 2^{g-1-j} & \text{if } 0 \leq j \leq g-1, \\ i \oplus 2^{j-(g-1)} & \text{otherwise,} \end{cases}$$

where g is the garlic, i is the index of the vertex in the current row and j is the index of the previous row. It is easy to see that the index difference of the diagonal edges is maximal when computing the second and the last row, *i.e.* when $j = 0$ and $j = 2 \cdot g - 2$. The index difference between the source and the destination vertex is 2^{g-1} for both cases. Consequently, the memory requirement can be reduced to $2^g + 2^{g-1} = 1.5 \cdot 2^g$. Figure 5.2 shows an example of the resulting alignment. A indexing function idx that translates vertex indices into their memory position can be derived as

$$idx(g,i,j) = \begin{cases} i & \text{if } j \mod 3 \equiv 0, \\ i + 2^g & \text{if } j \mod 3 \equiv 1 \land i < 2^{g-1}, \\ i - 2^{g-1} & \text{if } j \mod 3 \equiv 1 \land i \geq 2^{g-1}, \\ i + 2^{g-1} & \text{otherwise.} \end{cases}$$

Note that if the depth index $k > 1$ and $(2 \cdot g - 1) \bmod 3 \neq 0$, *i.e.* the last row of the previous graph does not start at the first memory block, an offset or carry over has to be added to the row index j. The carry over o can be computed as $o = o + (2 \cdot g - 1)$ where o is set to 0 for $k = 0$.

Reducing the memory requirement of Algorithm 13 requires the following changes: increasing the size of v to $1.5 \cdot 2^g$, removing r and replacing it with v, and wrapping all accesses of v in idx. The resulting algorithm suffers only a negligible increase in computational time from invoking idx. Note that the vertical edges prevent a memory layout optimization similar to the one described for Catena-Dragonfly (see Section 5.1.1), since they would resemble the diagonal edges when the vertices would be aligned in σ order.

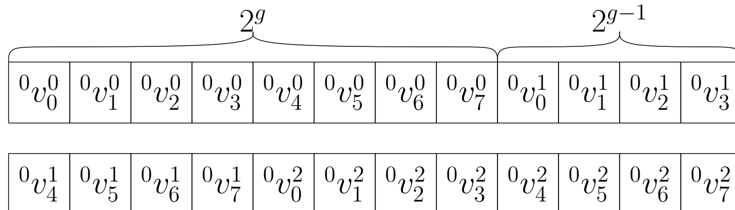

Figure 5.2: The first three rows of a $(3, \lambda)$-double-butterfly graph aligned in $2^g + 2^{g-1}$ blocks of memory.

Reduced-Memory Algorithm It is possible to reduce the memory requirement further down to 2^g blocks of memory at the cost of additional computations per line. Since σ is an involution for the same g and j, the only nodes that are vertically and diagonally connected to ${}^k v_a^{j-1}$ and ${}^k v_{\sigma(g,a,j-1)}^{j-1}$ are ${}^k v_a^j$ and ${}^k v_{\sigma(g,a,j-1)}^j$. Computing the latter allows to replace the former. This requires only one additional memory block to temporary save one of the results before both predecessors can be replaced and another block for the intermediate values that have to be computed to resolve the sequential connection. When computing a pair of nodes ${}^k v_a^{j-1}$ and ${}^k v_{\sigma(g,a,j-1)}^{j-1}$, fulfilling the sequential dependency of the node with the larger index may require additional computations because the nodes before it have not been computed yet. Every row has 2^{j-1} distinct subgroups. Obviously, the sequential dependency is only unresolved for the first node of every subgroup. For $j \leq g$, the number of vertices between ${}^k v_a^j$ and ${}^k v_{\sigma(g,a,j-1)}^j$ can be derived from σ as $2^{g-1-(j-1)} - 1 = 2^{g-j} - 1$. As a result, the total number of additional computations per row is $2^{j-1} \cdot (2^{g-j} - 1)$. Consequently, the first half of the graph, *i.e.* for $j \leq g$, requires

$$\sum_{j=1}^{g} 2^{j-1} \cdot (2^{g-j} - 1)$$

additional computations. With the exception of the middle row, every diagonal layer from the first half also occurs in the second half; hence, the second half requires the same number of additional computations. The diagonal layer of the middle row, *i.e.* $j = g$, occurs only once in the whole graph, but since it requires no additional computations it can be neglected. The total amount of additional computations can then be calculated by doubling the equation for the first half:

$$\begin{aligned} 2 \cdot \sum_{j=1}^{g} 2^{j-1} \cdot (2^{g-j} - 1) &= 2 \cdot \sum_{j=1}^{g} (2^{g-1} - 2^{j-1}) \\ &= 2 \cdot \sum_{j=1}^{g} 2^{g-1} - 2 \cdot \sum_{j=1}^{g} 2^{j-1} \\ &= g \cdot 2^g - 2 \cdot (2^g - 1) \\ &= \underline{(g-2) \cdot 2^g + 2}. \end{aligned}$$

A rough estimate of the relative increase can be calculated by gauging the additional computations as $g \cdot 2^g$ and the total cost of the naive algorithm as $2 \cdot g \cdot 2^g$ (see Section 4.3.2):

$$\frac{g \cdot 2^g}{2 \cdot g \cdot 2^g} = \frac{1}{2}.$$

Please be aware that the actual increase will be lower than that, because of the overhead created by memory initialization and Γ. Furthermore, all of the additional computations are done with H' which are less expensive than the computations of H. But even when computing ${}^k v_a^j$ and ${}^k v_{\sigma(g,a,j-1)}^j$ in parallel, the reduced-memory algorithm will still require the most time.

5.2 Catena-axungia

The memory and time cost are quantified as abstract integers whose effects depend on the password-hashing scheme. Furthermore, the actual runtime of the hashing process varies greatly between different systems and as a result, the time cost can only be used as a comparative value. Catena-axungia addresses this problem by allowing to search for optimal cost parameters under constraints that are supplied in concrete measurements. Finding optimal cost parameters that are still tolerable is a labor-intensive process that has to be executed for every deployment and every update of cost parameters. Catena-axungia makes this process beginner-friendly and allows the automation of most of it. To ensure the applicability of the results, the search for suitable parameters should be conducted on the system with the least computing speed and with the limitations of the system with the least amount of memory in mind. It is also advisable that the search is conducted under realistic conditions, *i.e.* with the same load it would have in a production environment. The implementation of Catena-axungia, which is subject to the MIT license, is published on

`https://github.com/medsec/catena-axungia`.

Figure 5.1 shows the usage statement of Catena-axungia that explains all command-line parameters, their default values and restrictions. The maximum value of `min_mhard` is given by the restrictions of the `uint8_t` that represents λ (see Section 5.1). The time limit for password hashing `max_time` has maximum of a full day, which should cover all scenarios. All other parameters are restricted by the datatypes used to represent them. While the lower limit for `max_memory` corresponds to $g_{\text{high}} = 4$, the search process can actually return lower values if the time limitations are not met. The upper limit for `max_memory` is equivalent to $g_{\text{high}} = 35$, which should be far more than enough for the foreseeable future. While busy system should choose a higher iteration count `iterations` so that the results will be unaffected by load variations, the default of 3 is enough for idle systems. When the `full_hash` option is enabled, the search will be executed for Catena-Dragonfly-Full and Catena-Butterfly-Full instead of Catena-Dragonfly and Catena-Butterfly.

Algorithm 15 defines the parameter search of Catena-axungia for the given constraints λ_{min}, mem_{max} and t_{max} which correspond to the parameters `min_mhard`, `max_memory` and `max_time`. The input i is the equivalent of the command-line parameter `iterations`. The algorithm will be run once for $I =$ Catena-Dragonfly and once for $I =$ Catena-Butterfly or their respective -full counterparts. The parameter-search algorithm starts by setting λ and g to their corresponding start values and determining the maximum garlic g_{max}. To satisfy the lower bound for memory hardness, λ starts at λ_{min} and should never drop below λ_{min}.

The initialization of g as well as the determination of the maximum garlic g_{max} is handled

```
This application searches for the optimal Catena-Butterfly and Catena-Dragonfly parameters for given
     constraints. Make sure to run this under realistic conditions on the most constrained system of the ones
     involved.

Usage: ./catena-axungia --max_time TIME --max_memory MEMORY [--min_mhard
 HARDNESS] [--iterations ITERATIONS] [--full_hash] [--verbose]

-t, --max_time TIME     Upper bound for expected password-hashing time in
                        seconds(floating point). Max: 86400
-m, --max_memory MEMORY Upper bound for memory usage in KiB. Min: 1,
                        Max: 2147483647
-h, --min_mhard HARDNESS        Lower bound for memory-hardness factor.
                                Default: 2, Min: 1, Max: 255
-i, --iterations ITERATIONS     Number of iterations used to determine the
                                runtime. Higher values increase stability.
                                Default: 3, Min: 1, Max: 2147483647
-f, --full_hash Uses a full hash function instead of a reduced one for
                consecutive calls
-v, --verbose   Outputs progress information
```

Listing 5.1: Usage statement explaining the command-line interface of CATENA-AXUNGIA.

by the function *get_garlics*, that is specified in Algorithm 16. The number of memory blocks that fit into mem_{max} can be computed by converting the memory limit mem_{max} to bytes and dividing it by the size of one memory block, *i.e.* the output length n of H and H'. The logarithm to base 2 of the result is the maximum garlic g_{max}. As the garlic for CATENA is represented by integers, the result has to be rounded down. For CATENA-BUTTERFLY and CATENA-BUTTERFLY-FULL, the increased memory consumption of $1.5 \cdot 2^g \cdot n$ has to be taken into account. Since the implications of larger memory restrictions might be ambiguous to the user, they might choose a memory limit that takes too long to compute; therefore, the starting point for g must be set to a value lower than g_{max}. For CATENA-AXUNGIA, this starting point is either a quarter g_{max} or 15 if the result is larger than 15. The absolute maximum of 15 was chosen because it should be computable in conceivable time on most available systems. Moreover, it has to be ensured that g never undercuts the lower limit of 1 (see Section 5.1).

The first step after the initialization is the maximization of the garlic in Line 3. To prevent unnecessary computations, the heuristic $t < (t_{\text{max}}/2)$ is used to determine if it is likely that I can still be computed in less than t_{max} after the garlic is increased. The time it takes to compute I with the parameters g and λ is measured using the function *measure*, which determines the median computation time over i iterations. Since the aforementioned heuristic may fail, incrementing the garlic could lead to the computation time exceeding t_{max}. This case is covered by the additional check in Line 7. The algorithm continues by increasing λ in Line 10. In contrast to the garlic increment, incrementing λ results in a almost linear increase of computation time. To cancel out most of the influence of the overhead created by the memory initialization and Γ, the procedure is repeated twice. Possible exceeding of t_{max} after the increase of λ is handled from Line 19 onwards. The search procedure may fail at finding suitable parameters for some supplied limitations since $t < t_{\text{max}}$ and $g \geq 1$ are strictly adhered to. Please be aware that the search procedure assumes that $g_{\text{low}} = g_{\text{high}}$ since g_{low} is irrelevant for most scenarios and the impact of it can be estimated from the results.

Algorithm 15 Parameter-Search Algorithm of CATENA-AXUNGIA

Input: I : Instance, $\lambda_{\min}$: Lower Bound for Memory Hardness, $\text{mem}_{\max}$: Memory Constraint (in KiB), $t_{\max}$: Time Constraint (in s), i : Iterations
Output: g : Recommended Garlic, λ : Recommended Depth

1: $\lambda \leftarrow \lambda_{\min}$
2: $g, g_{\max} \leftarrow \mathit{get_garlics}(I, \text{mem}_{\max})$
3: **while** $g < g_{\max} \wedge t < (t_{\max}/2)$ **do** ▷ Start by increasing g
4: $g \leftarrow g + 1$
5: $t \leftarrow \mathit{measure}(I, \lambda, g, i)$ ▷ Determine median time over i iterations
6: **end while**
7: **if** $t > t_{\max} \wedge g > 1$ **then** $g \leftarrow g - 1$
8: $t \leftarrow \mathit{measure}(I, \lambda, g, i)$
9: **end if**
10: **for** j **from** 1 **to** 2 **do** ▷ Continue by increasing λ
11: $\lambda_{\text{inc}} \leftarrow \min((t_{\max} - t)/(t/\lambda), 255 - \lambda)$
12: **if** $\lambda_{\text{inc}} > 0$ **then**
13: $\lambda \leftarrow \lambda + \lambda_{\text{inc}}$
14: $t \leftarrow \mathit{measure}(I, \lambda, g, i)$
15: **else**
16: **break**
17: **end if**
18: **end for**
19: **while** $t > t_{\max} \wedge \lambda > \lambda_{\min}$ **do**
20: $\lambda \leftarrow \lambda - 1$
21: $t \leftarrow \mathit{measure}(I, \lambda, g, i)$
22: **end while**
23: **if** $t > t_{\max}$ **then**
24: **return** g, λ
25: **else**
26: **print** No suitable parameters found
27: **end if**

Build Process To eliminate redundancies, CATENA-AXUNGIA relies on the reference implementation of CATENA. The `Makefile` builds all instances of CATENA into separate object files with the interface from `catena.c` being replaced by a simplified interface `wrapper` found in `wrapper.c`. Symbol collisions between the different instances are avoided by renaming all exposed functions and variables via compiler options. The function `wrapper` is also given a unique name per instance. As a result, the instance variable I from Algorithm 15 can be replaced by an identifier that indicates which of the renamed `wrapper` functions should be used.

Algorithm 16 Function *get_garlics* of Catena-axungia

Input: I : Instance, $\text{mem}_{\max}$: Memory Constraint (in KiB)
Output: g : Start Garlic, $g_{\max}$: Maximum Garlic
1: $g_{\max} \leftarrow \lfloor \log_2(\text{mem}_{\max} \cdot 1024/n) \rfloor$
2: **if** $I =$ Catena-Butterfly $\vee\ I =$ Catena-Butterfly-Full **then**
3: **if** $1.5 \cdot 2^{g_{\max}} > (\text{mem}_{\max} \cdot 1024/n)$ **then**
4: $g_{\max} \leftarrow g_{\max} - 1$
5: **end if**
6: **end if**
7: $g \leftarrow \max(\min((g_{\max}/4), 15), 1)$ ▷ Ensure $g \in [1, 15]$
8: **return** $g, g_{\max}$

5.3 Catena-Variants

Catena-Variants allows to easily create and test custom instances of Catena. New functions for $flap$, Γ, F, H and H' can be added and swapped with the default functions without needing to change any of the existing code. The resulting instance can then be used to benchmark the impact of the changes or to compute hashes, e.g. to verify an alternative implementation. The MIT-licensed implementation in C++11 [27] can be found on

`https://github.com/medsec/catena-variants`.

Since Catena-Variants is in ongoing development, this section will only cover the design, features and functions as of the commit on the 4th of August 2015[VIII].

The implementation of Catena-Variants includes two applications that allow to assemble custom instances from command-line parameters. The first application, `catena-measure`, measures the computation time and the memory usage of the instance for a set of supplied parameters. Password hashes of the custom instance can be computed with `catena-scramble`, which is the second application.

Structure While Catena-Variants shares a lot of code with the reference implementation of Catena (see Section 5.1), the desired flexibility requires an object-oriented design that necessitates lot of changes to the structure. Instances of Catena are represented by objects of class `Catena`, which provides an interface that is nearly identical to the reference implementation. To ease the creation of those objects, a factory `CatenaFactory` is provided, which holds references to all available functions and allows to create an instance of `Catena` by supplying identifiers for the desired functions.

The concrete functions of Catena-Variants are represented by classes and split into five categories, `Algorithm`, `Graph`, `HashFast`, `HashFull` and `RandomLayer`, that represent the placeholder functions $flap$, F, H', H and Γ. Every of those categories is implemented

[VIII] `https://github.com/medsec/catena-variants/tree/d67370552172d23c92e1c2cfbc56e62660ed6c64`

as a class that is derived from `Registerable`, which defines an interface for three basic informations about each subclass: a name, a short handle and a description. Concrete functions could be derived from two of the categories, which would lead to the class inheriting the interface of `Registerable` twice. This ambiguity, the so-called deadly diamond of death [34], is avoided by declaring the inheritance between the categories and `Registerable` virtual.

Since C++ does not allow to create objects from the name of a class, the `CatenaFactory` operates on instances of the concrete functions; therefore, the classes that represent the functions must provide the method `clone()` that returns a deep copy of the object. The curiously recurring template pattern (CRTP) [10] is used to provide this method for every of the five category classes and thereby avoids having to implement it in every subclass. For the implementation of the pattern, a layer of abstract classes is introduced and inserted between the categories and `Registerable`. Figure 5.3 shows the resulting structure of classes. It is the responsibility of the abstract classes to provide the interface, whereas its children implement the `clone()` method and all functionality shared between the children's subclasses. To be able for `clone()` to return a copy of the actual subclass, the superclass implementing the function needs to know on which derived class the method is invoked. This is achieved with the CRTP in which each derived class uses itself as a template argument for the superclass. As a result, the classes `Algorithm`, `Graph`, `HashFast`, `HashFull` and `RandomLayer` require a template argument and can no longer be used as a general superclass for their subclasses. The abstract superclasses of the aforementioned classes can still be used as supertypes for all concrete function classes of their respective category. Consequently, all references, e.g. pointers, should use the abstract classes, while subclasses should be derived from the templated classes. To ease the memory management, shared pointers [27, Section 20.6.2] are used for references.

The factory pattern used for `CatenaFactory` is extended by class registration to circumvent the need to change the factory for every function class added to CATENA-VARIANTS. Because class registration must take place before any other code is executed that could access the factory, the function implementations must execute their registration at the beginning of every application that uses CATENA-VARIANTS. While C++ does not provide any explicit mechanism to do this, global variables can be used as a workaround. They are constructed before any other code is executed and can contain arbitrary code in their constructor. This behavior is implemented in the class `Registry`, that takes the class as a template argument and registers it with the factory if the class fits into one of the five categories.

Functions In addition to the functions used by the default instances of CATENA, the implementation of CATENA-VARIANTS includes a few more alternatives. Graphs, instances of F, can be found in the directory `graphs`. As seen in CATENA-BUTTERFLY-FULL and CATENA-DRAGONFLY-FULL, hash functions can be used for both H and H'; therefore, all

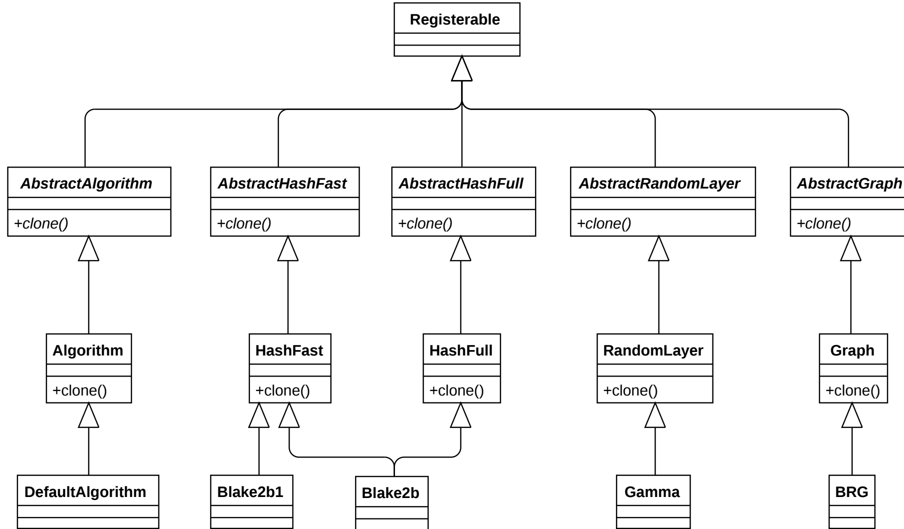

Figure 5.3: Illustration of the class structure created by applying the curiously recurring template pattern to the function categories of CATENA-VARIANTS, where `DefaultAlgorithm`, `Blake2b1`, `Blake2b`, `Gamma` and `BRG` are examples for concrete functions.

classes that are derived from `HashFast`, `HashFull` or both are placed in a shared directory `hashes`. Instances of Γ, random layers, are placed in `random_layers`. While it is possible to swap the function *flap* with an alternative from the directory `algorithms`, only the default (see Algorithm 6) is part of CATENA-VARIANTS as of writing this thesis.

The list of graphs has been extended by a gray-code based modification of bit-reversal hashing as suggested by Ben Harris [23]. It is based on a graph that has a longer period than the regular BRG, which should hamper time-memory-tradeoff attacks.

SHA-512 [12] was added as a hash function and can be used for both, H and H'. While SHA-512 is generally slower than BLAKE2b[IX], some scenarios may require the use of standardized primitives or it could be possible that there are systems for which only optimized implementations of SHA-512 exists. The implementation of CATENA-VARIANTS includes two more functions that can be used to instantiate H': `MulHash` and `GFMul`. The former is an experimental implementation of a multiplication-based hash function by Bill Cox[X]. The latter uses the multiplication in $GF(2^{128})$ with the Galois/Counter Mode reduction polynomial described in Section 2.5. While `MulHash` must be considered unsafe and should rather be seen as an attempt to max out the memory bandwidth, Galois-Field multiplication is a well-known and vetted hash function. Please be note that `GFMul` consists of four independent multiplications; therefore, any instance using it will be vulnerable to advanced TMTO attacks.

[IX] https://blake2.net/

[X] https://github.com/medsec/catena/blob/3a3ce823d4c54f2da33757bf8f6389488c31bd93/src/catena-multhash.c

A dummy random layer has been added that allows to skip this layer altogether. This could be useful to increase the memory fill rate of CATENA or when creating an instance that is as similar as possible to CATENA V 1.

Build Process The `Makefile` of CATENA-VARIANTS was taken from the reference implementation of CATENA and adapted to support C++. It was also extended to compile any source file found in the directories `graphs`, `hashes`, `random_layers` and `algorithms`. CATENA-VARIANTS can then be extended with an alternative function by implementing a class for it, instantiating a variable of type `Registry` with the class as the template argument and placing the source file (and the optional header) in the corresponding directory. Functions that depend on external implementations or libraries will require changes to the `Makefile`.

Chapter 6

Discussion

> Panel members believe that Argon2 and Catena are the top candidates in the category where they're playing.
>
> *Jean-Philippe Aumasson*[I]

The scope of this thesis focused on a fast and comprehensible reference implementation of CATENA. Aside from that, CATENA-AXUNGIA made the concept of a variable-cost PHS more approachable and CATENA-VARIANTS laid the foundation for further development. This chapter will discuss the significance and implications of these contributions as well as suggestions for the future of CATENA.

6.1 Conclusion

CATENA was selected for special recognition by the panel of the Password hashing Competition "for its agile framework approach and side-channel resistance" [2]. Both of these crucial qualities were consolidated and elaborated during this thesis. First of all, CATENA-VARIANTS is not only a logical consequence but also the pinnacle of the mentioned agility. Furthermore, resistance against side-channel attacks must be covered by the specification and the implementation. In contrast to the resistance against cache-timing attacks and garbage-collector attacks that stem solely from the specification of CATENA (see Section 4.2), the defense against weak garbage-collector attacks required additional measures from the implementation. Additionally, the quality and readability of the implementation that brought CATENA into the finale of the PHC was maintained and further improved during this thesis.

[I] http://article.gmane.org/gmane.comp.security.phc/3010

In addition to the emphasis on the agility of the Catena framework, Catena-Variants also paves the way for future development. It is already used in the practical part of an upcoming paper and a master's thesis.

Catena has a unique feature set that distinguishes it from the other schemes that passed the finale of the Password Hashing Competition. Aside from being the only one of those PHS with support for standard hash functions, it is also the only side-channel resistant finalist that has built-in support for client-independent updates. Due to Catena filling these niches, it should find use in several applications and Catena-axungia offers a level of assistance for all these applications seen in no other PHS.

6.2 Outlook

Although the development of Catena in the scope of the PHC was finished during this thesis, further research and development on it should be continued.

First and foremost, explicit support for more application areas could be added to the reference implementation. A set of interfaces for different proof-of-work approaches (see Section 2.2) could help to establish Catena in this area. Furthermore, the current implementation of Catena-Butterfly would allow reducing the overall memory requirement for computing several hashes in parallel by interleaving their computation. This could be useful for any system that might face multiple authentications at once. Some scenarios might also benefit from an implementation of the reduced-memory algorithm for computing Catena-Butterfly as described in Section 5.1.2.

The amount and the variety of candidates in the PHC [2] shows that password hashing is a broad field that requires research beyond the competition. Catena-Variants provides not only an easy way to modify and extend Catena, but also an excellent starting point for any further research. Among the currently developed extensions for Catena-Variants are several hash functions, graphs and instances of Γ. Detailed explanations of these will be given in an imminent scientific paper and an upcoming master's thesis.

All finalists of the Password Hashing Competition use abstract cost parameters and could potentially benefit from an adoption of Catena-axungia. Likewise, an implementation similar to Catena-Variants could improve the flexibility of many PHS, that currently require a lot of code changes for even simple modifications. The solid foundation of Catena, given by the three implementations, establishes it for password hashing and takes over the groundwork to tackle most security concerns regarding passwords.

Bibliography

[1] Jean-Philippe Aumasson. *Password Hashing Competition.* URL: https://password-hashing.net (visited on 2015-05-06).

[2] Jean-Philippe Aumasson. *Password Hashing Competition. Candidates.* (2015-07-20). URL: https://password-hashing.net/candidates.html (visited on 2015-05-06).

[3] Adam Back. *Hashcash - A Denial of Service Counter-Measure.* (2002). URL: http://www.hashcash.org/papers/hashcash.pdf (visited on 2015-05-19).

[4] Alex Biryukov, Daniel Dinu & Dmitry Khovratovich. "Argon and Argon2". Version 2. In: *Password Hashing Competition* (2015-01-31). URL: https://password-hashing.net/submissions/specs/Argon-v2.pdf (visited on 2015-06-08).

[5] Alex Biryukov & Dmitry Khovratovich. "Tradeoff Cryptanalysis of Memory-Hard Functions". In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 227. URL: http://eprint.iacr.org/2015/227.

[6] William F. Bradley. "Superconcentration on a Pair of Butterflies". In: *CoRR* abs/1401.7263 (2014-01-28). URL: http://arxiv.org/abs/1401.7263.

[7] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé & Paul Zimmermann. "Faster Multiplication in GF(2)[x]". In: *Algorithmic Number Theory, 8th International Symposium, ANTS-VIII, Banff, Canada, May 17-22, 2008, Proceedings.* (2008), pp. 153–166. DOI: 10.1007/978-3-540-79456-1_10.

[8] Milan Broz. *Password Hashing Competition second round candidates - tests report.* Version 3. (2015-04-27). URL: https://github.com/mbroz/PHCtest/blob/master/output/phc_round2.pdf (visited on 2015-05-07).

[9] Donghoon Chang, Arpan Jat, Sweta Mishra & Somitra Kumar Sanadhya. "Rig. A simple, secure and flexible design for Password Hashing". Version 2. In: *Password Hashing Competition* (2014-09-23). URL: https://password-hashing.net/submissions/specs/RIG-v2.pdf (visited on 2015-06-15).

[10] James O. Coplien. *Curiously Recurring Template Patterns.* patterns electronic mail discussion group. (1995). URL: http://sites.google.com/a/gertrudandcope.com/info/Publications/InheritedTemplate.pdf (visited on 2015-08-24).

[11] Bill Cox. “TwoCats (and SkinnyCat). A Compute Time and Sequential Memory Hard Password Hashing Scheme”. Version 0. In: *Password Hashing Competition* (2014-04). URL: https://password-hashing.net/submissions/specs/TwoCats-v0.pdf (visited on 2015-06-15).

[12] Quynh H. Dang. *Secure Hash Standard*. Tech. rep. FIPS PUB 180-4. National Institute of Standards and Technology, 2015. DOI: 10.6028/nist.fips.180-4.

[13] Cynthia Dwork & Moni Naor. “Pricing via Processing or Combatting Junk Mail”. In: *Advances in Cryptology - CRYPTO ’92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings.* (1992), pp. 139–147. DOI: 10.1007/3-540-48071-4_10.

[14] Christian Forler, Stefan Lucks & Jakob Wenzel. “Catena: A Memory-Consuming Password Scrambler”. In: *IACR Cryptology ePrint Archive* 2013 (2013), p. 525. URL: http://eprint.iacr.org/2013/525.

[15] Christian Forler, Stefan Lucks & Jakob Wenzel. “The Catena Password-Scrambling Framework”. Version 3. In: *Password Hashing Competition, 2nd round submission* (2015-01-31). URL: https://password-hashing.net/submissions/specs/Catena-v3.pdf (visited on 2015-05-04).

[16] Christian Forler, Eik List, Stefan Lucks & Jakob Wenzel. “Overview of the Candidates for the Password Hashing Competition - And their Resistance against Garbage-Collector Attacks”. In: *IACR Cryptology ePrint Archive* 2014 (2015-07-15), p. 881. URL: http://eprint.iacr.org/2014/881.

[17] Free Software Foundation. *Using the Gnu Compiler Collection (GCC)*. Version 4.9.2. 2014. URL: https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/ (visited on 2015-08-07).

[18] *GNU Make*. URL: https://www.gnu.org/software/make/ (visited on 2015-08-06).

[19] Jeremi M. Gosney. “Password Cracking HPC”. In: Passwordsˆ12 Security Conference. (2012-12-03). URL: http://heim.ifi.uio.no/hennikl/passwords12/www_docs/Jeremi_Gosney_Password_Cracking_HPC_Passwords12.pdf (visited on 2015-05-12).

[20] Jeremi M. Gosney. “The Pufferfish Password Hashing Scheme”. Version 1. In: *Password Hashing Competition* (2015-01-31). URL: https://password-hashing.net/submissions/specs/Pufferfish-v1.pdf (visited on 2015-06-15).

[21] Shay Gueron & Michael E Kounavis. “Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode”. In: *Intel white paper (September 2012)* (2010). URL: https://software.intel.com/en-us/articles/intel-carry-less-multiplication-instruction-and-its-usage-for-computing-the-gcm-mode (visited on 2015-06-02).

[22] Darrel Hankerson, Alfred Menezes & Scott Vanstone. *Guide to Elliptic Curve Cryptography.* 1st ed. Springer-Verlag, 2004. ISBN: 978-0-387-95273-4. DOI: 10.1007/b97644.

[23] Ben Harris. *Replacement index function for data-independent schemes (Catena).* PHC mailing list: discussions@password-hashing.net. (2015-02-19). URL: http://article.gmane.org/gmane.comp.security.phc/2457 (visited on 2015-08-25).

[24] George Hatzivasilis, Ioannis Papaefstathiou & Charalampos Manifavas. "Password Hashing Competition - Survey and Benchmark". In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 265. URL: http://eprint.iacr.org/2015/265.

[25] Marcos A. Simplicio Jr, Leonardo C. Almeida, Paulo C Ewerton R. Andrade, F. dos Santos & Paulo S. L. M. Barreto. "The Lyra2 reference guide". Version 3. In: *Password Hashing Competition* (2015-01-15). URL: https://password-hashing.net/submissions/specs/Lyra2-v3.pdf (visited on 2015-06-10).

[26] JTC1/SC22/WG14. *Rationale for International Standard - Programming Languages-C.* Version 5.10. 2003-04. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/C99RationaleV5.10.pdf (visited on 2015-05-27).

[27] JTC1/SC22/WG21. *Working Draft, Standard for Programming Language C++.* N3337. Please note that a draft is referenced, because the actual Standard (ISO/IEC 14882:2011) is not freely avaiable. ISO/IEC. 2012-01-16. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf (visited on 2015-08-25).

[28] Ari Juels & John G. Brainard. "Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 1999, San Diego, California, USA.* (1999). URL: http://www.isoc.org/isoc/conferences/ndss/99/proceedings/papers/juels.pdf.

[29] John Kelsey, Bruce Schneier, Chris Hall & David Wagner. "Secure Applications of Low-Entropy Keys". In: *Information Security, First International Workshop, ISW '97, Tatsunokuchi, Japan, September 17-19, 1997, Proceedings.* (1997), pp. 121–134. DOI: 10.1007/BFb0030415.

[30] Thomas Lengauer & Robert Endre Tarjan. "Asymptotically Tight Bounds on Time-Space Trade-offs in a Pebble Game". In: *J. ACM* 29.4 (1982-10-04), pp. 1087–1130. DOI: 10.1145/322344.322354.

[31] Charles Lin. *Big and Little Endian.* cmsc311-Class Notes. University of Maryland. 2003-03-10. URL: http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/endian.html (visited on 2015-06-02).

[32] Stefan Lucks. *Maximising Pseudo-Entropy versus resistance to Side-Channel Attacks.* PHC mailing list: discussions@password-hashing.net. (2015-04-30). URL: http://thread.gmane.org/gmane.comp.security.phc/2922 (visited on 2015-05-12).

[33] David Malone & Kevin Maher. "Investigating the Distribution of Password Choices". In: *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012.* (2012), pp. 301–310. DOI: 10.1145/2187836.2187878.

[34] Robert C. Martin. *Java and C++. A critical comparison.* (1997-03-09). URL: http://objectmentor.com/resources/articles/javacpp.pdf (visited on 2015-08-24).

[35] David McGrew & John Viega. "The Galois/counter mode of operation (GCM)". In: *Submission to NIST* (2004). URL: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf (visited on 2015-06-11).

[36] Robert Morris & Ken Thompson. "Password Security - A Case History". In: *Commun. ACM* 22.11 (1979), pp. 594–597. DOI: 10.1145/359168.359172.

[37] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* URL: https://bitcoin.org/bitcoin.pdf (visited on 2015-05-18).

[38] Samuel Neves. *reference source code package of BLAKE2.* 2015-06-11. URL: https://github.com/BLAKE2/BLAKE2/tree/b6525c7fa15dbf6713e0597727827d191a31a26a (visited on 2015-06-29).

[39] Philippe Oechslin. "Making a Faster Cryptanalytic Time-Memory Trade-Off". In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings.* (2003), pp. 617–630. DOI: 10.1007/978-3-540-45146-4_36.

[40] Colin Percival. "Stronger key derivation via sequential memory-hard functions". In: *Self-published* (2009). URL: https://www.tarsnap.com/scrypt/scrypt.pdf (visited on 2015-05-08).

[41] Alexander Peslyak. "yescrypt. a Password Hashing Competition submission". Version 1. In: *Password Hashing Competition* (2015-01-31). URL: https://password-hashing.net/submissions/specs/yescrypt-v1.pdf (visited on 2015-06-15).

[42] Thomas Pornin. "The MAKWA Password Hashing Function. Specifications". Version 1. In: *Password Hashing Competition* (2015-04-22). URL: https://password-hashing.net/submissions/specs/Makwa-v1.pdf (visited on 2015-06-11).

[43] The Linux man-pages project. *CRYPT(3).* Linux Programmer's Manual. (2015-03-02). URL: http://man7.org/linux/man-pages/man3/crypt.3.html (visited on 2015-05-08).

[44] Niels Provos & David Mazières. "A Future-Adaptable Password Scheme". In: *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, June 6-11, 1999, Monterey, California, USA.* (1999), pp. 81–91. URL: https://www.usenix.org/legacy/publications/library/proceedings/usenix99/provos/provos.pdf (visited on 2015-06-10).

[45] Phillip Rogaway & Thomas Shrimpton. "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance". In: *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers.* (2004), pp. 371–388. DOI: 10.1007/978-3-540-25937-4_24.

[46] Markku-Juhani O. Saarinen. *The BLAKE2 Cryptographic Hash and MAC. draft-saarinen-blake2-03.* Version 03. IETF. 2015-04-25. URL: https://tools.ietf.org/html/draft-saarinen-blake2-03 (visited on 2015-06-29). Draft.

[47] The Clang Team. *clang. A C language family frontend for LLVM.* URL: http://clang.llvm.org/ (visited on 2015-08-06).

[48] The Clang Team. *Clang 3.5 documentation.* 2014. URL: http://llvm.org/releases/3.5.0/tools/clang/docs/ (visited on 2015-08-07).

[49] The GCC team. *GCC. The GNU Compiler Collection.* URL: https://gcc.gnu.org/ (visited on 2015-08-06).

[50] Steven Thomas. "battcrypt (Blowfish All The Things)". Version 0. In: *Password Hashing Competition* (2014-04-01). URL: https://password-hashing.net/submissions/specs/battcrypt-v0.pdf (visited on 2015-06-09).

[51] Steven Thomas. "Parallel". Version 1. In: *Password Hashing Competition* (2014-02-01). URL: https://password-hashing.net/submissions/specs/Parallel-v1.pdf (visited on 2015-06-09).

[52] Sebastiano Vigna. "An experimental exploration of Marsaglia's xorshift generators, scrambled". In: *CoRR* abs/1402.6246 (2014-03-23). URL: http://arxiv.org/abs/1402.6246 (visited on 2015-07-01).

[53] Hongjun Wu. "POMELO. A Password Hashing Algorithm". Version 3. In: *Password Hashing Competition* (2015-04-13). URL: https://password-hashing.net/submissions/specs/POMELO-v3.pdf (visited on 2015-06-12).

Affidavit

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Affidavit

I hereby affirm that this master thesis represents my own written work and that I have used no sources or aids other than those indicated. Furthermore, I confirm that this thesis was not submitted in the same or in a substantially similar version, not even partially, to another examination board and was not published elsewhere.

<Sascha Schmidt>

Appendix A

Benchmarks

A.1 Galois-Field Multiplication

The results from Table A.1 were acquired on an Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz using 12 million iterations to ensure stability. The gf2x benchmarks use version 1.1 of gf2x. It was configured using the `configure` script included in the package. The Intrinsics specific approach and the optimized right-to-left (RtL) multiplication are implemented as described in Section 2.5. An implementation of both can be found in the file `src/hashes/gfmul.cpp` of CATENA-VARIANTS. The double-table approach for RtL multiplication is based on the optimized right-to-left algorithm with the changes proposed in Section 1.1 of [7]. All tests were compiled with GCC 4.9.2 and optimization level O3 [49] [17]. Please note that clocks-per-byte measurements vary significantly between different CPUs. The shown benchmark should only be used for relative comparisons.

Algorithm	**Median clocks per byte**
gf2x (compiled for a generic 64-bit cpu)	12.38
gf2x (compiled for a cpu with `pclmulqdq`)	2.50
Intrinsics-specific approach	1.00
RtL optimized $w = 3$	36.19
RtL optimized $w = 3$	25.56
RtL optimized $w = 4$	8.00
RtL optimized $w = 5$	10.88
RtL optimized $w = 6$	22.88
RtL double table $w = 2 \cdot 2$	28.25
RtL double table $w = 2 \cdot 3$	10.31
RtL double table $w = 2 \cdot 4$	10.12
RtL double table $w = 2 \cdot 5$	13.19
RtL double table $w = 2 \cdot 6$	26.12

Table A.1: Benchmark of several algorithms for 128 bit Galois-Field multiplication

A.2 Compiler Choice & Optimization Options

The results from Table A.2 were acquired on an Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz using 20 iterations to ensure stability. Please note that g_{low} was set to g_{high} for this benchmark. The test were compiled with clang 3.5.0 and GCC 4.9.2. Other optimization levels, O1, Os and additionaly Oz for clang, were tested for the sake of completeness, but they yielded no improvement. While GCC lists the flags enabled by the optimization levels in the manual [17, Section 3.10], this information must be retrieved manually for clang (see Section B.1).

Algorithm	λ	g_{high}	Compiler	Median runtime in s		
				O2	O3	Ofast
Catena-Butterfly	4	16	GCC	0.3110	0.2912	0.2891
			clang	0.2685	0.2623	0.2630
Catena-Dragonfly	2	21	GCC	0.3884	0.3838	0.3842
			clang	0.3697	0.3714	0.3717

Table A.2: Runtime of Catena-Dragonfly and Catena-Butterfly with different compilers (GCC and clang) and optimization levels (O2, O3, Ofast).

Appendix B

Additional Information

B.1 clang Optimization Levels

The list of flags enabled by the optimization levels can not be found in the clang manual [48] and must therefore be received from the LLVM optimizer `opt` by using the `-debug-pass=Arguments` flag. Table B.1 contains the output for clang 3.5.0 and the levels O2 and O3. For unknown reasons, Ofast, which was likewise used in the benchmark from Section A.2, can not be analyzed using `opt`. Please note that the documentation of clang is incomplete regarding optimization flags, but due to the large similarities most explanations from the GCC manual [17] should also apply to clang.

Optimization Level	Flags
O2	-no-aa -tbaa -targetlibinfo -basicaa -notti -verify -simplifycfg -domtree -sroa -early-cse -lower-expect -targetlibinfo -no-aa -tbaa -basicaa -notti -verify-di -ipsccp -globalopt -deadargelim -instcombine -simplifycfg -basiccg -prune-eh -inline-cost -inline -functionattrs -sroa -domtree -early-cse -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -instcombine -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -loop-rotate -licm -loop-unswitch -instcombine -scalar-evolution -lcssa -indvars -loop-idiom -loop-deletion -loop-unroll -memdep -mldst-motion -domtree -memdep -gvn -memdep -memcpyopt -sccp -instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree -memdep -dse -loops -scalar-evolution -slp-vectorizer -adce -simplifycfg -instcombine -barrier -domtree -loops -loop-simplify -lcssa -branch-prob -block-freq -scalar-evolution -loop-vectorize -instcombine -simplifycfg -domtree -loops -loop-simplify -lcssa -scalar-evolution -loop-unroll -strip-dead-prototypes -globaldce -constmerge -verify -verify-di
O3	-no-aa -tbaa -targetlibinfo -basicaa -notti -verify -simplifycfg -domtree -sroa -early-cse -lower-expect -targetlibinfo -no-aa -tbaa -basicaa -notti -verify-di -ipsccp -globalopt -deadargelim -instcombine -simplifycfg -basiccg -prune-eh -inline-cost -inline -functionattrs -argpromotion -sroa -domtree -early-cse -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -instcombine -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -loop-rotate -licm -loop-unswitch -instcombine -scalar-evolution -lcssa -indvars -loop-idiom -loop-deletion -loop-unroll -memdep -mldst-motion -domtree -memdep -gvn -memdep -memcpyopt -sccp -instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree -memdep -dse -loops -scalar-evolution -slp-vectorizer -adce -simplifycfg -instcombine -barrier -domtree -loops -loop-simplify -lcssa -branch-prob -block-freq -scalar-evolution -loop-vectorize -instcombine -simplifycfg -domtree -loops -loop-simplify -lcssa -scalar-evolution -loop-unroll -strip-dead-prototypes -globaldce -constmerge -verify -verify-di

Table B.1: Flags enabled by the optimization levels O2 and O3 of clang 3.5.0.