On the Memory-Hardness of Data-Independent Password-Hashing Functions

Joël Alwen*, Peter Gaži*, Chethan Kamath*, Karen Klein*, Georg Osang*,
Krzysztof Pietrzak*, Leonid Reyzin†, Michal Rolínek*, Michal Rybár*
**IST Austria*
†*Boston University*

***Abstract*—We show attacks on five data-independent *memory-hard functions* (iMHF) that were submitted to the password hashing competition. Informally, an MHF is a function which cannot be evaluated on dedicated hardware, like ASICs, at significantly lower energy and/or hardware cost than evaluating a single instance on a standard single-core architecture. Data-independent means the memory access pattern of the function is independent of the input; this makes iMHFs harder to construct than data-dependent ones, but the latter can be attacked by various side-channel attacks.**

Following [Alwen-Blocki'16], we capture the evaluation of an iMHF as a directed acyclic graph (DAG). The *cumulative parallel pebbling complexity* of this DAG is a good measure for the cost of evaluating the iMHF on an ASIC. If n denotes the number of nodes of a DAG (or equivalently, the number of operations — typically hash function calls — of the underlying iMHF), its pebbling complexity must be close to n^2 for the iMHF to be memory-hard. We show that the following iMHFs are far from this bound: Rig.v2, TwoCats and Gambit can be attacked with complexity $O(n^{1.75})$; the data-independent phase of Pomelo (a finalist of the password hashing competition) and Lyra2 (also a finalist) can be attacked with complexity $O(n^{1.83})$ and $O(n^{1.67})$, respectively.

For our attacks we use and extend the technique developed by [Alwen-Blocki'16], who show that the pebbling complexity of a DAG can be upper bounded in terms of its depth-robustness.

1. Introduction

In cryptographic settings we typically consider functions which are easy to compute for a party having some secret piece of information, but are extremely hard to compute without it; inverting a trapdoor permutation or computing a pseudorandom function are examples.

J. Alwen, P. Gaži, C. Kamath, K. Klein, K. Pietrzak and M. Rybár were supported by the European Research Council consolidator grant (682815-TOCNeT). M. Rolínek was supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement no 616160. L. Reyzin was supported in part by IST Austria and by US NSF grants 1012910, 1012798, and 1422965; this research was performed while he was visiting IST Austria.

1.1. Moderately Hard Functions

Some applications, like password hashing or proofs of work, ask for "moderately hard" functions. Here, we have no secret; instead we want the function to be "somewhat" hard to compute. For example, in the password hashing application, a password server will not simply store login/-password tuples $(\mathsf{login}, \mathsf{pwd})$ in the clear, but instead a tuple $(\mathsf{login}, \mathsf{salt}, f(\mathsf{salt}, \mathsf{pwd}))$ for a random salt and some function $f(\cdot)$. This mitigates the consequences, should the password file get leaked: an adversary given $(\mathsf{login}, \mathsf{salt}, y)$ will have to find a pre-image of y to impersonate login. As passwords often do not have much entropy, this is feasible by a dictionary attack where one evaluates $f(\mathsf{salt}, \cdot)$, starting from the most popular passwords.

To make this dictionary attack expensive, a moderately hard f is used. This puts some extra burden on the password server with every login attempt, but it puts a huge extra cost on an adversary running a dictionary attack, who must evaluate f a large number of times in a dictionary attack. A popular moderately hard function is PBKDF2 (Password-Based Key Derivation Function 2), which essentially iterates a cryptographic hash function h, like SHA-256, n times, i.e., computes

$$h(h(\ldots h(\mathsf{salt}, \mathsf{pwd})\ldots)).$$

However, a password server will have to evaluate this function on its available hardware, like an x86 processor, whereas an attacker can run the dictionary attack on dedicated hardware like ASICs (application specific integrated circuits), where performing this computation costs (in terms of energy but also hardware) several orders of magnitude less than on a standard processor.[1] So we have not gained nearly as much in security by using a moderately hard function as one could have hoped for.

1.2. Memory Hard Functions

Percival [12] suggests *memory*-hard functions (MHF) as a means to get functions whose evaluation cost is more egalitarian across different platforms. The idea is to make

1. According to https://en.bitcoin.it/wiki/Mining_hardware_comparison, the best ASICs make 2^{32} hashes per joule, whereas the most efficient laptops can do 2^{17} hashes per joule, giving a factor of 30'000.

memory — not computation — the main cost of evaluation. In this work we focus on *data-independent* MHF (iMHF), which are MHFs whose memory-access pattern is independent of the input. iMHFs are more restricted — and thus potentially harder to construct — than the more general data-dependent MHFs (dMHF), but dMHFs succumb to various side-channel (in particular timing) attacks, and thus in some contexts iMHFs are preferred.

1.3. Energy and AT Complexity

In this work we analyze the memory-hardness of several iMHFs which were candidates at the password hashing competition[2] in terms of their AT and energy complexity.

The *amortized energy complexity* of an algorithm measures the amount of electricity required by an ASIC implementing the algorithm per evaluation. This is particularly useful when considering the running cost of an ASIC. Indeed, many ASICs sold commercially for mining cryptocurrencies (which in several notable cases such as Litecoin, Dogecoin, and Ethereum boils down to precisely the type of brute-force attack on an MHF considered in this work), are often specified and compared to each other by looking at their rates of electricity consumption per evaluation. Furthermore, to make the results more technology independent, we also equip the model with a *core-memory area* parameter $\bar{R} \in \mathbb{R}^+$ which denotes the ratio between the amount of electricity used to evaluate the hash function and the amount of electricity needed to store the output of the hash function for an equivalent amount of time.[3]

The *AT-complexity* of an algorithm is the product of the area of an implementation of the algorithm in an application specific integrate circuit (ASIC) and the time it takes the circuit to produce output. AT-complexity is often used as an efficiency estimate relating to the financial cost of implementing an algorithm, especially for repeated computations [14]. To upper-bound the cost of a brute-force attack on an MHF, we consider the *amortized* AT-complexity of our attack per instance of the MHF it computes. Similar to the energy complexity case, we make the results more independent of the technology used by parametrizing the complexity using a *core-memory energy* ratio $R \in \mathbb{R}^+$ denoting the ratio of the size of an (on-chip) implementation of the underlying hash function and area required to store one of its outputs.

1.4. iMHFs as DAGs

The computation of an iMHF $f(\cdot)$ can often be captured by a DAG (directed acyclic graph) $G_f = (V, E)$ with one source and one sink. Let $V = \{v_0, \ldots, v_n\}$ be in some topological order (i.e., if $(v_i, v_j) \in E$ then $i < j$). We then assign a label ℓ_i to each vertex v_i, where the label ℓ_0 of the source v_0 is the function input, and the label ℓ_n of the sink v_n is the output. In general, the label ℓ_i of a vertex v_i is some "simple" function of the labels of its parents. For example the DAG underlying PBKDF2 is a path and the simple function mentioned above is the compression function h.

The iMHFs we consider are parameterised by a space parameter σ and a time parameter τ. Here σ specifies how much memory is required to evaluate the iMHF (in blocks which correspond to the output size of the underlying hash-function, so for our $f(\cdot)$ that is 512 bits). Informally, parameter τ specifies how many times the memory is overwritten. Typical parameters are $\sigma = 2^{24}$ (corresponding to $1GB$, block size of 512 bits) and τ a small integer, typically 1.

For the iMHFs considered in this work the underlying DAG will typically be a path $v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_n$ ($n = \sigma \cdot \tau$) with some additional edges, and the simple functions will be cryptographic compression functions (like Blake, or the permutation used in the sponge of Keccak), sometimes mixed with even simpler functions like XORs. For concreteness, for this introduction let us consider an iMHF $f(\cdot)$ where the computation of a label ℓ_i is always done by invoking some compression function $h : \{0,1\}^{1024} \rightarrow \{0,1\}^{512}$. The iMHFs Rig.v2 and TwoCats we attack in this paper are of this form with h being Blake2.

1.5. The Naïve Algorithm

The specification of an iMHF $f(\cdot)$ describes how to evaluate the iMHF. We call this description a "naïve" algorithm $\mathcal{N}$; it typically computes the labels $\ell_1, \ell_2, \ldots, \ell_n$ sequentially, storing the last σ blocks (which means the underlying DAG has no long edges (v_i, v_j) where $j - i > \sigma$). Thus, this algorithm invokes the hash-function $n = \sigma \cdot \tau$ times sequentially, using σ memory for these n steps, which gives an energy complexity of $E_{\bar{R}}(\mathcal{N}) \approx n(\bar{R} + \sigma)$ and similarly, the AT complexity is $AT_R(\mathcal{N}) \approx n(R + \sigma)$. As typically $n \gg \bar{R}, R$ and τ is a small constant,[4] this simply becomes

$$E_{\bar{R}}(\mathcal{N}) \approx n^2 \quad , \quad AT_R(\mathcal{N}) \approx n^2 \ .$$

1.6. Attacks on MHFs

For an MHF to be memory-hard, there should not exist an adversarial algorithm $\mathcal{A}$ that evaluates the function at significantly lower energy or AT cost. This $\mathcal{A}$ is allowed to make parallel queries to the underlying hash-function h (as on an ASIC we can have many cores computing h) and the complexity can be amortized over many invocations (as an attacker will evaluate the function on many inputs). In this paper we analyze five MHFs which were submitted to the password hashing competition, and for all of them show that there exist algorithms which evaluate the function at energy and AT complexity much lower than n^2. Ideally,

2. https://password-hashing.net/

3. More precisely, we consider shortest time (say in clock cycles) from the moment the inputs to the hash function are determined until the output of the hash function has been fully determined.

4. If used as a proof of work in a cryptocurrency like Litecoin, one probably would insist on $\tau = 1$, as the cost of verifying a proof for given σ is linear in τ.

we'd like these complexities to be $\Theta(n^2)$, but [1] showed that no such iMHFs exist, as for every iMHF there exists an attack achieving $O(n^2/\log(n))$.[5] Our attacks are (at least asymptotically) much stronger showing complexities of the form $O(n^c)$ for different constants $c < 2$.

The attacks in this work follow a general framework introduced by Alwen and Blocki [1], who used it to analyse the Argon2i, Catena and Double-Buffer constructions (an earlier attack on Catena was presented in [6]); we first map a given iMHF f to a graph G_f, and the AT and energy quality of the iMHF can then be expressed in terms of the complexity of "pebbling" the graph G_f. In [1], these complexities were upper bounded in terms of the depth-robustness of G_f (Theorem 1 in this paper).

Thus, finding attacks on iMHF boils down to showing that the underlying graph is not depth-robust. We give upper bounds on the depth-robustness of the DAGs underlying the iMHFs, leading to upper bounds on their energy and AT complexity via Theorem 1 as summarized in Figure 1. Our main tools are a new result (Lemma 1) which bounds the depth-robustness of DAGs with not too many "short" edges, and a bound on the depth-robustness on graphs which are close to being "layered" (Lemma 2).

Our bounds (third column of Figure 1) are asymptotically way below the best possible $n^2/\log(n)$. Concretely, we prove an upper bound $O(n^{1.75})$ for all functions, except for Pomelo, where we only get $O(n^{1.83})$ and Lyra2 where we get a better $O(n^{1.67})$. On a high level, the reason is that all constructions except Pomelo have underlying DAGs where most edges are either very short (of constant length) or very long (linear in n). Pomelo additionally has a large fraction of edges of length around $\sqrt{n}$, which makes it more depth-robust (and thus gives a worse attack). In Lyra2 the long edges have a particular distribution (going only in-between "layers") which allows for a better attack than the other constructions.

In light of such asymptotic attacks – which show structural deficiencies in all analysed iMHFs – we strongly believe they must be considered broken. This view is not generally shared. In particular, even after [1] published similar asymptotic attacks on several iMHFs including Argon2i, these were dismissed as not practically relevant by the proponents of the quick standardization of this MHF.

When one simply inputs parameters to the upper bounds we and [1] rigorously prove, the bounds are indeed not impressive (the fourth column in Figure 1 shows the attack quality at 1GB of memory, only for Rig.v2 we get an actual attack, saving a factor 2 in energy). However, one should keep in mind that these proven bounds increase with n and just give a rough upper bound. The actual attacks are certainly much better for several reasons: (1) The *analyses* are not tight (we often have to upper bound some quantities) and moreover only use a single property of the underlying graph, depth-robustness via Theorem 1, but for concrete graphs the attack is potentially much better than what we get

5. The best proven asymptotic lower bound for any construction is $\Omega(n^2/\log^{10}(n))$ from [3].

MHF/section/ref	building block	$E_{\bar{R}}(\mathcal{A})$	$\frac{E_{\bar{R}}(\mathcal{N})}{E_{\bar{R}}(\mathcal{A})}$ at 1GB	$\bar{R}$
Rig.v2 §4 [7]	Blake2	$O(n^{1.75})$	≥ 2	3000
TwoCats §5 [9]	Blake2	$O(n^{1.75})$	> 1	3000
Gambit §6 [13]	Keccak Permutation	$O(n^{1.75})$	$\geq 1/256$	3000
Lyra2 §7 [11]	Blake2 or Keccak Permutation	$O(n^{1.67})$	$\geq 1/6$	3000
Pomelo §8 [15]	Basic Ops.	$O(n^{1.83})$	$\geq 1/512$	100

Figure 1. (3rd column) Bounds on the amortized energy complexity of our attacks. (4rd column) The energy-quality of our attacks (cf. Def. 2) at 1GB of memory using core-memory parameter $\bar{R}$ from the last column (cf. Sec. 1.3). AT complexities behave basically identically.

from this worst-case analysis. (2) The *attacks* can certainly be improved via various heuristics tailored to the underlying graph.

To address the dismissal of their attacks, the authors of [1] simulated their attack on Argon2i, introducing various heuristics to further improve the attack quality [2]. As expected, the derived quantitative bounds were much better than the proven upper bounds in their paper, showing that Argon2i can be broken at 1GB memory even for the most "paranoid" suggested settings (in particular, even at an impractical $\tau = 6$ number of memory passes) using only realistic amounts of bandwidth (we refer the reader to [2] for details). Such a simulation could in principle also be done for the attacks presented in this paper, and we expect a similar gap between the proven and the actual attack quality. We did not implement such simulations as there is not much to be learned from them.

We find the current approach towards constructing MHFs worrisome. Proposed constructions usually come with no meaningful security proofs, and asymptotic attacks are then often dismissed as not practical based on their *proven* performance. Modern cryptographic research should put the burden of proof on the designers of a system, not the attackers. And even if the attacks were at this point just theoretical (which they are not), we believe that asymptotic attacks should be sufficient to disqualify a scheme right away unless it's already widely deployed or we simply lack better alternatives, neither which is the case for iMHFs. In particular, recent results[6] show that iMHFs with asymptotically optimal $\Omega(n^2/\log(n))$ memory-hardness exist, and concrete proposals are being developed.

2. Preliminaries

2.1. The Computational Model

We consider a computational model where an algorithm which can repeatedly make batches of queries to a hash function h (or another simple building block), before producing the final output. Between each call to h, we make explicit (the bit-length of) the state stored by the algorithm at that point in the computation. The complexity measure we ultimately consider is the *cumulative memory complexity*; that is the sum of the bit-lengths of all states stored during

6. Announced at https://calendar.csail.mit.edu/events/171401.

the computation. The desired security property (as motivated already by Percival in [12]) is that no algorithm exists which can compute the MHF with too low cumulative memory complexity per input/output pair computed.

We fix the details of the model bellow following [3]. At its core, the computation consists of repeatedly invoking an algorithm $\mathcal{A}$ making any state maintained between invocations explicit. At invocation $i \in \{1, 2, \ldots\}$ algorithm $\mathcal{A}$ is given the state (bit-string) σ_{i-1} it produced at the end of the previous invocation. Next, $\mathcal{A}$ can make a batch of calls $\mathbf{q}_i = (q_i^1, q_i^2, \ldots)$ to h. Then, $\mathcal{A}$ receives the response from h and can perform some basic computations before finally outputting an updated state σ_i (here basic means much simpler than evaluating h). The initial state σ_0 contains the input to the computation which terminates once a special final state is produced by $\mathcal{A}$. Apart from the explicit states σ, the algorithm may keep no other state between invocations. For an input x and coins r, we denote by $\mathcal{A}(x; r; h)$ the corresponding (deterministic) execution of $\mathcal{A}$. If in all possible executions of $\mathcal{A}$, no batch of queries to h contains more than a single query at a time, then $\mathcal{A}$ is said to be a *sequential* algorithm.

We define the runtime $\mathsf{time}(\mathcal{A})$ to be the maximum running time of $\mathcal{A}$ in any execution (over all choices of x, r and h). Then the *cumulative memory complexity* and *cumulative evaluation complexity* are defined as

$$\mathsf{cmc}(\mathcal{A}) = \max_{x,r} \sum_{i \in [T-1]} |\sigma_i| \qquad \mathsf{cec}(\mathcal{A}) = \max_{x,r} \sum_{i \in [T]} |\mathbf{q}_i| \ ,$$

where $|\sigma|$ is the bit-length of state σ, $|\mathbf{q}|$ is the dimension of the vector $\mathbf{q}$, and $\max_{x,r}$ denotes the maximum over all possible executions of $\mathcal{A}$. Similarly, the *absolute memory complexity* and *absolute evaluation complexity* are defined as

$$\mathsf{amc}(\mathcal{A}) = \max_{x,r} \max_{i \in [T-1]} |\sigma_i| \qquad \mathsf{aec}(\mathcal{A}) = \max_{x,r} \max_{i \in [T]} |\mathbf{q}_i|.$$

We remark that these complexity measures are stricter than it is common, in particular, we treat h as a black-box and do not allow for optimisation like pipelining. But keep in mind that we use them to upper-bound the complexity of our attacks, therefore this strictness can only serve to strengthen our results. Using these tools, we can now define the complexity of an algorithm as follows.

Definition 1. (AT and Energy Complexities) Let $\mathcal{A}$ be an algorithm which computes $\#inst(\mathcal{A})$ instances of an iMHF in parallel. Then for any *core-memory area ratio* $R > 0$ and any *core-memory energy ratio* $\bar{R} > 0$ the *(amortized) AT-complexity* and the *(amortized) energy-complexity* of $\mathcal{A}$ are defined to be

$$AT_R(\mathcal{A}) = [\mathsf{amc}(\mathcal{A}) + R \cdot \mathsf{aec}(\mathcal{A})] \times \frac{\mathsf{time}(\mathcal{A})}{\#inst(\mathcal{A})}$$

$$E_{\bar{R}}(\mathcal{A}) = \frac{\mathsf{cmc}(\mathcal{A}) + \bar{R} \cdot \mathsf{cec}(\mathcal{A})}{\#inst(\mathcal{A})}.$$

Recall that an MHF is specified together with a (sequential) evaluation algorithm $\mathcal{N}$, which we call the *naïve* algorithm. The understanding is that this is the algorithm used by the honest user (e.g. by the login server per login attempt).

With this in mind we consider an evaluation algorithm an "attack", if it has lower (amortized) complexity than the naïve algorithm, as this implies that the adversary has an advantage over the honest user. To this end, we define the following measure for evaluating the quality of an attack.

Definition 2. (Attack Quality) Let f be an MHF with naïve algorithm $\mathcal{N}$ and let $\mathcal{A}$ be an algorithm for evaluating $\#inst(\mathcal{A})$ instance(s) of f. Then for any *core-memory area ratio* $R > 0$ and any *core-memory energy ratio* $\bar{R} > 0$ the *AT-quality* and *energy-quality* of $\mathcal{A}$ is defined to be

$$\mathsf{ATquality}_R(\mathcal{A}) = \frac{AT_R(\mathcal{N})}{AT_R(\mathcal{A})} \ ,$$

$$\mathsf{ENquality}_{\bar{R}}(\mathcal{A}) = \frac{E_{\bar{R}}(\mathcal{N})}{E_{\bar{R}}(\mathcal{A})} \ .$$

In particular, if either quantity is greater than 1, then we call $\mathcal{A}$ an *attack* on f.

2.2. Basic Notation

We denote the set of natural numbers beginning at 0 as $\mathbb{N} = \{0, 1, \ldots\}$. For $n \in \mathbb{N}$ we denote by $[n]$ the set $\{1, 2, \ldots, n\}$. Also, by $[a, b)$ (resp, $(a, b]$), we denote the set $\{a, a+1, \cdots, b-1\}$ (resp., $\{a+1, a+2, \cdots, b\}$). We denote the set of real numbers greater than 0 with $\mathbb{R}^+$. We use the shorthand $[N]_m$ to represent the number $N \mod m$.

We also use the following terminology to talk about properties of a directed acyclic graph (DAG) $G = (V, E)$. The *indegree* of a node $v \in V$ is the number of edges ending in v. In symbols, $\mathsf{indeg}(v) := |\{(u, v) \in E : u \in V\}|$. More generally, the indegree of G is the largest indegree of any node in G. More precisely, $\mathsf{indeg}(G) := \max\{\mathsf{indeg}(v) : v \in V\}$. If a p is a directed path in G, then it's length $\mathsf{len}(p)$ is the number of edges traversed by p. The *depth* of G, denoted $\mathsf{depth}(G)$, is the length of the longest directed path in G. For any set of nodes $S \subseteq V$, we write $G - S$ to denote the DAG obtained from G by removing S (and incident edges).

2.3. Data-Independent MHFs and Graphs

We fix some useful definitions of which we shall make repeated use.

Definition 3. Let $G = (V, E)$ be a DAG and let $e, d \in \mathbb{N}$. We say that G is (e, d)*-depth-robust* if after removing any subset of at most e nodes, there remains a path of length d in G. That is, if

$$\forall S \subset V \qquad |V| \le e \implies \mathsf{depth}(G \setminus S) \le d.$$

If G is not (e, d)-depth-robust, then G is said to be (e, d)-*reducible*.

The attacks in this work follow the general paradigm of determining parameters (and corresponding node set) for which each graph is (e, d)-reducible and then instantiating the generic attack of [1]. The complexity of that attack in terms of the underlying DAG and its reducibility are captured by the following theorem.

Theorem 1 ([1]). Let f be an iMHF where the underlying DAG $G = (V, E)$ has $\mathsf{indeg}(G) = \delta$ and is (e, d)-reducible.[7] Then, for any integer $g \in [d, n]$ and any core-memory area and energy ratios $R > 0$ and $\bar{R} > 0$, there exists evaluation algorithm $\mathcal{A}$ of AT and energy complexity

$$\begin{aligned} AT_R(\mathcal{A}) &\leq 2n\left[\frac{dn(R+1)}{g} + \delta g + e + R\right] \\ E_{\bar{R}}(\mathcal{A}) &\leq n\left[\frac{dn(\bar{R}+1)}{g} + \delta g + e + \bar{R} + 1\right] . \end{aligned}$$

Remark 1. Note that δ can be replaced with $\delta - 1$ if for every node of inedgree δ in G, one of incoming edges is from the immediately preceding (in topological order) node.

3. Attacks on Classes of Graphs

Theorem 1 allows us to attack a graph by showing that it is depth-reducible. In this section we describe two general approaches to demonstrating depth-reducibility of graphs. These approaches are used in most of our attacks below.

3.1. Attacks Based on Edge Lengths

Assume G is a DAG on $V = \{v_1, \ldots, v_n\}$ with edges going from left to right (i.e., $(v_i, v_j) \in E \Rightarrow i < j$). Define the *length* of an edge (v_i, v_j) as $j - i$.

CHAIN GRAPH. Consider first a simple chain graph with all edges of length 1. Pick some γ, divide the graph into contiguous segments of γ nodes, and remove the last node at the end of every segment. Then no path of length γ remains in the graph. In other words, this graph is $(n/\gamma, \gamma)$-reducible for any $\gamma \in [n]$.

ATTACKS ON A CHAIN + SHORT AND LONG EDGES. Now consider any graph G that has edges of length 1 or long edges of length at least β. Again, remove the last node of every γ-node segment. Any path in this graph will have fewer than n/β long edges total, and no more that $\gamma - 2$ short edges in a row before a long edge. Thus, G is $(n/\gamma, \gamma\lceil n/\beta \rceil)$-reducible for any $\gamma \in [n]$.

To see how this helps to attack the graph, consider setting $\gamma = \beta^{1/3}$. We get that G is (e, d)-reducible for $e = n/\beta^{1/3}$ and $d \approx n/\beta^{2/3}$. We can then apply Theorem 1 with $g = e$ to get attack complexity approximately $n^2/\beta^{1/3}$.

This attack can be generalized to a setting where all edges are of length at most α or at least β, for some $\alpha <$

7. Recall that an iMHF f is parameterized by parameters σ, τ, so here G and thus also δ, e and d will depend on σ, τ.

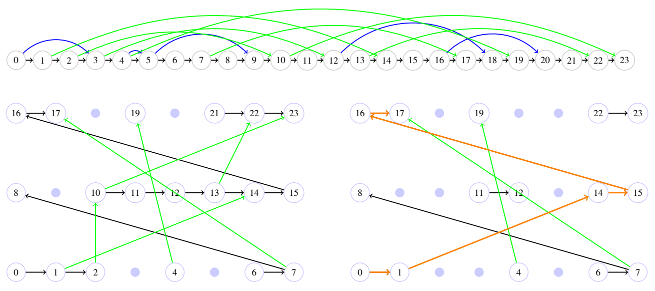

Figure 2. (top) A $(m = 5, \lambda = 3, e = 2, d = 2)$-layered graph G. (bottom left) G after removing a node incident with each short (blue) edge. A path of length 7 is $(e = 2, d = 2)$-reducible (remove the 3rd and 5th node of the path). (bottom left) G after additionally removing 3rd and 5th node of each layer. As claimed in Lemma 2, the graph is $(m+\lambda\cdot e = 11, \lambda\cdot d = 6)$-reducible. For the illustrated graph this is tight, as a path of length 5 (shown in orange) exists.

β: simply remove segments of length α instead of single nodes, to get $(n\alpha/\gamma, \gamma\lceil n/\beta \rceil)$ reducibility. We can then get an attack of complexity approximately $n^2(\alpha/\beta)^{1/3}$.

Finally, if a graph has a few edges of lengths between α and β, we can simply remove their end points. We thus obtain the following lemma.

Lemma 1. If for some η, α, and β, G has at most η edges of length greater than α but less that β, then, for any γ, G is $(\eta + n\alpha/\gamma, \gamma\lceil n/\beta \rceil)$-reducible.

ATTACKS ON A GRAPH WITH UNIFORM EDGE LENGTH. Suppose that the number of edges in G whose length is greater than 1 but less than ℓ is at most ℓ. Such a graph has mostly long edges. Thus, applying the above lemma to $\alpha = 1$, $\eta = \beta = n^{3/4}$, and $\gamma = n^{1/4}$ implies that G is (e, d)-reducible for $e = n^{3/4}$, $d = n^{1/2}$. So, with $g = e$ we can apply Theorem 1 to get attack complexity of approx. $n^{1.75}$.

3.2. Attacks on Layered Graphs

We now show depth-reducibility of a general class of graphs called *layered graphs*.

Definition 4. (Layered Graph) Let integer $\lambda > 0$ and e and d be functions from $[\lambda]$ to $\mathbb{N}$. For $\lambda \in \mathbb{N}$ and functions $e, d : \mathbb{N} \to \mathbb{N}$, a DAG $G = (V, E)$ is a (λ, e, d)-perfectly-layered DAG, if the following holds: There exists a partitioning of V into subset $V_1, V_2, \ldots, V_\lambda$, such that

1) Edges never point to lower layers: $\forall (u, v) \in E$ where $u \in V_i$ and $v \in V_j$ we have $i \leq j$.
2) For all $i \in [\lambda]$ the subgraph of G consisting only of the nodes V_i is $(e(i), d(i))$-reducible.

More generally, for $m \in \mathbb{N}$, a DAG G is an (m, λ, e, d)-layered graph if there exists a node set $S \subseteq V$ of size at most m such that the DAG $G - S$ is a (λ, e, d)-perfectly-layered graph.

The following lemma lower-bounds the reducibility of a layered graph.

Lemma 2. Let G be an (m, λ, e, d)-layered graph. Then G is $(\bar{e}, \bar{d})$-reducible, where

$$\bar{e} = m + \sum_{i \in [\lambda]} e(i) \qquad \bar{d} = \sum_{i \in [\lambda]} d(i) \ .$$

Proof: We construct a set of nodes S to remove and upper-bound the length of the longest path. Initially, S is the set of (at most m) nodes whose removal turns G into an (λ, e, d)-perfectly-layered graph $G' = G - S$. For $i \in [\lambda]$, let G'_i denote the subgraph of G' containing only the nodes in V_i. That means G'_i is $(e(i), d(i))$-reducible. Let $S_i \subseteq V_i$ be a set of nodes of size at most $e(i)$, such that $G'_i - S_i$ contains no path of length $d(i)$. Let $\bar{S} = S \cup (\cup_i S_i)$. Then clearly $|\bar{S}| \leq \bar{e}$. It remains to show that no path in $\bar{G} := G - S'$ has length $\bar{d}$.

Consider any path p in $\bar{G}$. As $S \subseteq \bar{S}$ the DAG $\bar{G}$ is a subgraph of G'. Moreover, for each $i \in [\lambda]$, as $S_i \subseteq \bar{S}$ the path p can not contain a subpath of length $d(i)$ with nodes in V_i. Yet, once p leaves a node set V_i for a node in V_j with $j > i$, no edge ever leads back to V_i. Thus, summing over all layers of $\bar{G}$, we get that $\mathsf{len}(p) \leq d(i) = \bar{d}$. □

4. Rig.v2

Rig was proposed by Chang *et al.* [7]. We analyze its second version and denote it Rig.v2 [8]. It is a password-hashing function strongly influenced by the ideas of the Catena function [10]. The authors provide two variants of Rig.v2 $[H_1, H_2, H_3]$, namely the strictly sequential variant Rig.v2 [Blake2b, BlakeCompress, Blake2b] and a second variant Rig.v2 [BlakeExpand, BlakePerm, Blake2b] (see [8]). For our analysis the choice of hash functions will make no difference. As input, Rig.v2 takes a password pwd, its length ℓ_{pwd}, a salt s of at least 16 bytes, its length ℓ_s, the number λ of iterations, the memory count m_c, the number t of bits to be retained from hash output of the setup phase, the output length ℓ, and the number of rounds r. As output, Rig.v2 gives an ℓ-bit password hash h_r^*.

The naïve algorithm $\mathcal{N}_{\mathsf{R}}$ to compute Rig.v2 from [8] is described by Algorithm 1 in the Reference Material. Using the original analysis from [8], a single password computation using this algorithm can be done in time complexity $O((\lambda+1)mr)$ and space complexity $O(m)$.

4.1. Graph Representation

The corresponding graph G_{R} is parameterized by the number of rounds r, the number of layers λ per round, and the length $m = 2^{m_c}$ of layers. $G_{\mathsf{R}} = (V, E)$ is defined as follows:

$$V = \bigcup_{\substack{1 \leq i \leq r \\ 0 \leq j \leq \lambda}} V_{i,j},$$

where $V_{i,j} := \{v_{i,jm+k} \mid 0 \leq k < m\}$ denotes the j-th layer of the i-th round. Every vertex $v_{i,jm+k}$ represents one

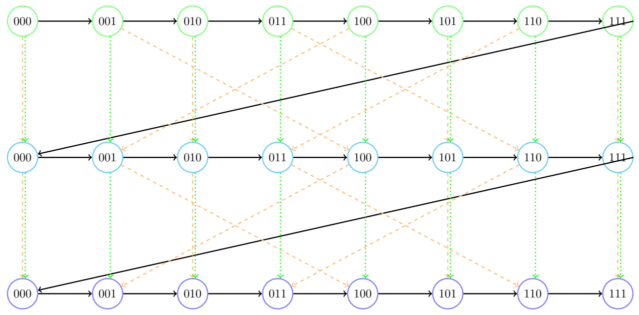

Figure 3. Illustration of Rig.v2 for $m = 8$ memory, $\lambda = 2$ layers and $r = 1$ round.

evaluation of a hash function and two memory cell updates of two temporary arrays. The set of edges is

$$E = \bigcup_{i=1}^{r} \Big(E_{i,1} \cup E_{i,2} \cup E_{i,3} \Big) \ \cup \ E_4,$$

where

$$E_{i,1} := \big\{ (v_{i,k}, v_{i,k+1}) \big) \mid 0 \leq k \leq (\lambda+1)m - 2 \big\}$$

defines a path running through all vertices of the i-th block;

$$E_{i,2} := \big\{ (v_{i,jm+k}, v_{i,(j+1)m+k}) \big) \mid 0 \leq k < m, \ 0 \leq j < \lambda \big\}$$

connects each vertex of a layer to the corresponding vertex of the subsequent layer by a vertical edge; $E_{i,3} :=$

$$\big\{ (v_{i,jm+\mathtt{br}(k)}, v_{i,(j+1)m+k}) \big) \mid 0 \leq k < m, \ 0 \leq j < \lambda \big\}$$

is the set of all edges we obtain by connecting the k-th vertex of the j-th layer to the $\mathtt{br}(k)$-th vertex of the subsequent layer (where $\mathtt{br} : [0, m-1] \to [0, m-1]$ denotes the bit-reversal function, see [8]); and

$$E_4 := \big\{ (v_{i,(\lambda+1)m-1}, v_{i+1,0}) \big) \mid 1 \leq i < r \big\}$$

connects the blocks of the r rounds by single edges.

Note that each of the r blocks of G_{R} looks like the graph corresponding to the Catena password-hashing function [10], except for the additional edges $E_{i,2}$.

4.2. The Attack

Lemma 3. G_{R} has $|V| = r(\lambda+1)m$ vertices, and $\mathsf{indeg}(G_{\mathsf{R}}) = 3$. The subgraphs of G_{R} corresponding to the rounds of the algorithm are $(\lambda+1, \sqrt{m}, \sqrt{m})$-perfectly-layered graphs.

Proof: The number of vertices follows easily through a simple enumeration. All the vertices with incoming edges from the set E_4 have indegree 1. All other vertices have at most three incoming edges, one from each of the three sets $E_{i,1}, E_{i,2}, E_{i,3}$, hence $\mathsf{indeg}(G_{\mathsf{R}}) = 3$. Secondly, from the description of our graph it is clear that all the edges either stay in the same layer or go to the next layer, which satisfies the first condition of Definition 4. As the edges in one layer form a single simple path, by removing every $\sqrt{m}$-th vertex of G_{R}, the longest path in each layer has length at most $\sqrt{m}-2$, so also the second claim is satisfied. $\square$

Corollary 1. For chosen parameters λ and r, there exists an algorithm $\mathcal{A}_{\mathsf{R}}$ for computing Rig.v2 with energy complexity $O(m^{1.75})$.

Proof: Since G_{R} consists of r subgraphs which are $(\lambda+1, \sqrt{m}, \sqrt{m})$-perfectly-layered, by Theorem 1 (setting $n := (\lambda+1)m$ and $g := (\lambda+1)m^{3/4}$), we get an attack complexity $O((\lambda+1)^2 m^{7/4})$ for each subgraph. As two subsequent subgraphs are connected by a single edge, it follows that G_{R} has complexity $O(r(\lambda+1)^2 m^{7/4})$. Since λ and r are relatively small compared to m, we reduce this result to $O(m^{1.75})$. $\square$

Since $\mathcal{N}_{\mathsf{R}}$ has an energy complexity of $O(n^2)$, the (asymptotic) quality of the attack is $O(n^{0.25})$.

4.3. Comparison for Practical Parameters

We now do an analysis of our attack using practical parameters. We set $r = 1$ (there is only a single edge between subsequent rounds, therefore the complexity scales linearly with r), $\lambda = 2$ (as recommended in [8]), $d = e = \sqrt{m}$ (by Lemma 3), $\delta = 2$ (see Remark 1), and apply Theorem 1 to arrive at

$$E_{\bar{R}}(\mathcal{A}_{\mathsf{R}}) \leq 3m\left[\frac{3m^{\frac{3}{2}}(\bar{R}+1)}{g} + 2g + \sqrt{m} + \bar{R} + 1\right],$$

Now, we plug in parameters $m = 2^{33}$ (1 GB) and $g = 2^{24.25}\cdot\sqrt{\bar{R}+1}\cdot\sqrt{3}$ to optimize the upper bound of our attack's complexity, and reach $E_{\bar{R}}(\mathcal{A}_{\mathsf{R}}) \leq 3^{\frac{3}{2}}\cdot 2^{59.25}\cdot\sqrt{\bar{R}}$. In [6] we find a value $R = 3000 \approx 2^{11.6}$ for Blake2. We estimate that the value of $\bar{R}$ should be correlated and therefore set it to the same value and plug it in. On the other hand, the complexity for the naïve algorithm using the same concrete parameters is $E_{\bar{R}}(\mathcal{N}_{\mathsf{R}}) > 3\cdot 2^{67}$. Therefore, if we compare these two values we find that the quality of the attack is roughly $\mathsf{ENquality} = \frac{E_{\bar{R}}(\mathcal{N}_{\mathsf{R}})}{E_{\bar{R}}(\mathcal{A}_{\mathsf{R}})} \geq 2$.

5. TwoCats

The password-hashing scheme TwoCats was proposed by Bill Cox [9]. TwoCats consists of two loops, a data-independent ("resistant") one, followed by a data-dependent ("unpredictable") one. In this paper we will only discuss the data-independent loop. The underlying hash function H is Blake2s or SHA256. As input, TwoCats takes a password, a salt, their sizes, and the desired memory cost.[8] The algorithm for evaluating the data-independent loop of TwoCats from [9] is described in Alg. 2 of the Reference Material. It makes n calls to the underlying hash function (where n is half the memory size divided by the block length); as the naïve algorithm $\mathcal{N}_{\mathsf{TC}}$ maintains all the memory during the entire computation, it has an energy complexity of $E_{\bar{R}}(\mathcal{N}_{\mathsf{TC}}) = O(n^2) = O(m_\ell^2)$.

8. In fact, we are only describing a simple version of TwoCats called SkinnyCat, which the author designed for easy implementation. As the underlying DAGs are the same, we won't go into further details.

5.1. Graph Representation

The corresponding computation graph $G_{\mathsf{TC}} = (V, E)$ is defined as follows. Let $n = 2^{n_c}$ be the number of vertices; let $V = \{v_0, \ldots, v_{n-1}\}$ and $E = E_1 \cup E_2$ where E_1 is the chain $\{(v_i, v_{i+1}) \mid 0 \leq i < n-1]\}$ and

$$E_2 := \{(v_{\mathtt{sbr}(i)}, v_i) \mid 2 \leq i \leq n-1\},$$

where $\mathtt{sbr}$ is the sliding-power-of-two bit reversal function:

$$\mathtt{sbr}(i) := \begin{cases} \bar{\mathtt{br}}(i) & \text{if } i - \bar{\mathtt{br}}(i) \leq 2^{\lfloor \log_2(i) \rfloor} + 1 \\ \bar{\mathtt{br}}(i) + 2^{\lfloor \log_2(i) \rfloor} & \text{if } i - \bar{\mathtt{br}}(i) > 2^{\lfloor \log_2(i) \rfloor} + 1 \end{cases}$$

where $\bar{\mathtt{br}}(i)$ removes the leading 1 bit and reverses the remaining bits. (To understand $\mathtt{sbr}(i)$, consider the following example for $i = 24$, which is 11000 in binary: cut off the first 1 to get 1000, reverse the bits to get $\bar{\mathtt{br}}(i) = 0001$, check if the difference between i and $\bar{\mathtt{br}}(i)$ is larger than $2^4 + 2$, and if so add 2^4 to get $\mathtt{sbr}(24) = 17$.)

Note that the indegree of all but the first two nodes is 2.

5.2. The Attack

Our attack closely follows the ideas described in Sections 3.1 and 3.2: we show that G_{TC} can be seen as a layered graph with exponentially growing layer sizes, where each layer consists of a chain and additional edges of uniformly distributed lengths.

Let the kth layer $\widetilde{G}_k$ of G_{TC} consist of all nodes whose indices are of bit-length k; i.e., $\widetilde{G}_k$ consists of v_i for $i \in [2^k, 2^{k+1})$. First we show that edges of E_2 internal to $\widetilde{G}_k$ have nearly-uniform lengths.

Lemma 4. For $1 \leq k < n_c$, the number of edges E in $\widetilde{G}_k$ that are of length $1 < \mathtt{length}(E) < \beta$ is at most $\beta + 2^{\lfloor k/2 \rfloor}$.

Proof: As $\bar{\mathtt{br}} : [2^k, 2^{k+1}) \to [0, 2^k)$, an edge to node v_i of length > 1 has its origin in $\widetilde{G}_k$ only when $\bar{\mathtt{br}}(i) \neq \mathtt{sbr}(i)$, i.e., when $i - \bar{\mathtt{br}}(i) \geq 2^k + 1$. Let $i = 1acb$ be the bit representation of such a value i, where $a, b \in \{0,1\}^{\lfloor k/2 \rfloor}$ and $c \in \{0,1\}$ if k is odd, or c is empty if k is even. Then (letting $\mathtt{br}$ stand for bit-reversal)

$$\begin{aligned} i - \mathtt{sbr}(i) &= 1acb - 1\mathtt{br}(b)c\mathtt{br}(a) \\ &= (a - \mathtt{br}(b))2^{\lceil k/2 \rceil} + b - \mathtt{br}(a)\,. \end{aligned}$$

Since $2^{\lfloor k/2 \rfloor} < b - \mathtt{br}(a) < 2^{\lfloor k/2 \rfloor}$, the number of edges E in the second half of G_k with $1 < \mathtt{length}(E) < \beta$ is equal to the number of pairs (a, b), such that $(a - \mathtt{br}(b))2^{\lceil k/2 \rceil} < \beta + 2^{\lfloor k/2 \rfloor}$, where the latter is equivalent to $a - \mathtt{br}(b) < \beta/2^{\lceil k/2 \rceil} + 1$. Thus, at most $2^{\lfloor k/2 \rfloor}(\beta/2^{\lceil k/2 \rceil} + 1) < \beta + 2^{\lfloor k/2 \rfloor}$ nodes v_i can serve as endpoints for such edges. $\square$

This lemma implies an attack against TwoCats.

Corollary 2. There is an attack $\mathcal{A}_{\mathsf{TC}}$ on TwoCats of complexity $E_{\bar{R}}(\mathcal{A}_{\mathsf{TC}}) = O(n^{1.75})$.

Proof: We will show that G_{TC} is $(\bar{e}, \bar{d})$-reducible for $\bar{e} \approx n^{3/4}$ and $\bar{d} \approx n^{1/2}$. Let $\widetilde{G}_0$ be the subgraph on the vertices v_0, v_1 and $\widetilde{G}_k$ for $1 \le k < n_c$ be as defined above. Then, by Lemma 1 (applied to $\alpha = 1, \beta(k) \approx 2^{3k/4}$ and $\gamma(k) \approx 2^{\lfloor k/4 \rfloor}$), $\widetilde{G}_k$ is $(e(k), d(k))$-reducible with $e(k) \approx 2^{3k/4}$ and $d(k) \approx 2^{k/2}$. Therefore, by Lemma 2 G_{TC} is $(\bar{e}, \bar{d})$-reducible, where

$$\bar{e} = \sum_{k=0}^{n_c-1} e_k \approx \sum_{k=0}^{n_c-1} (2^{n_c}/(2^{n_c-k}))^{3/4} =$$
$$= 2^{3n_c/4} \sum_{k=1}^{n_c} (1/2)^{3k/4}$$

and by the convergence of the geometric series, it follows $\bar{e} = O(n^{3/4})$, and analogously, $\bar{d} = O(n^{1/2})$. Thus, by Theorem 1, we get an attack complexity of $O(n^{1.75})$. □
Since $\mathcal{N}_{\mathsf{TC}}$ has an energy complexity of $O(n^2)$, the (asymptotic) quality of the attack is $O(n^{0.25})$.

5.3. Comparison for Practical Parameters

If we set $m_c = 30$ as proposed as an upper bound in [9], and $b_l = 2^{12}$, we get $n = 2^{25}$. Optimizing constants in the proof of Corollary 2, we set $\beta(k) := (\bar{R}/a)^{1/4} 2^{3k/4}$ and $\gamma(k) := (\bar{R}/a)^{-1/4} 2^{k/4}$, where $a := \sum_{k=1}^{\infty} (1/2)^{3k/4} \approx 1.5$. We see that G_{TC} is $(\bar{e}, \bar{d})$-reducible, with $\bar{e} \approx 2a(\bar{R}/a)^{1/4} n^{3/4}$ and $\bar{d} \approx a(\bar{R}/a)^{-1/4} n^{1/2}$. By Theorem 1, choosing $g \approx \sqrt{\bar{R}} n^{3/4} 4$, we get as an approximation of the energy-complexity of the attack algorithm $\mathcal{A}_{\mathsf{TC}}$

$$E_{\bar{R}}(\mathcal{A}_{\mathsf{TC}}) \le n[2\sqrt{\bar{d}n(\bar{R}+1)} + \bar{e} + \bar{R} + 1].$$

Using $\bar{R} = 3000$ (see 4.3), we get energy-complexity $E_{\bar{R}}(\mathcal{A}_{\mathsf{TC}}) < 2^{50}$. If $\mathcal{N}_{\mathsf{TC}}$ denotes the naïve algorithm, we obtain:

$$E_{\bar{R}}(\mathcal{N}_{\mathsf{TC}}) = n^2 + n\bar{R} = 2^{50} + 2^{25}\bar{R}.$$

Thus, for such a small value for n, our asymptotic result only gives a very small advantage:

$$\mathsf{ENquality}_{\bar{R}}(\mathcal{A}) = \frac{E_{\bar{R}}(\mathcal{N}_{\mathsf{TC}})}{E_{\bar{R}}(\mathcal{A}_{\mathsf{TC}})} > \frac{2^{50} + 2^{25}\bar{R}}{2^{50}} > 1.$$

6. Gambit

Gambit is a password-hashing function proposed by Pintér [13] and inspired by the sponge paradigm of Bertoni *et al.* [4] It is built on top of a sponge with a state consisting of b blocks; the upper part of the state affected directly by absorbing and squeezing consists of r words. As input, Gambit takes a password *pwd*, salt *salt*, time and memory complexity parameters t and m respectively, and a key identifier *dkid*. The construction itself is parametrized by an *interleave factor* denoted $f \in \{1, \ldots, m-1\}$ satisfying several requirements that we will detail later.

6.1. The Naïve Algorithm

The algorithm for computing Gambit given in [13] is described in Algorithm 4 in the Reference Material.

We evaluate the memory costs of Algorithm 4 (we call it $\mathcal{N}_{\mathsf{g}}$) in terms of words, while the computational costs will be expressed in terms of calls to the sponge permutation. Note that we are being generous to the naïve algorithm by making all other its computation (e.g., updates to Mem and the sponge absorption steps) for free.

$\mathcal{N}_{\mathsf{g}}$ maintains the memory consisting of m words during its entire computation (t steps), hence we have $\mathsf{cmc}(\mathcal{N}_{\mathsf{g}}) = O(mt)$. It performs t calls to the sponge permutation during its operation, hence we get $\mathsf{cec}(\mathcal{N}_{\mathsf{g}}) = O(t)$. Overall, this gives us $E_{\bar{R}}(\mathcal{N}_{\mathsf{g}}) = O(t(m + \bar{R}))$.

6.2. Graph Representation

Below we describe the directed acyclic graph that corresponds to the Gambit function in its simplest form, without considering the optional transformations resulting from `Trans` and ROM.[9] When we refer to i-th absorbing/squeezing/permutation step, we always index from 0 to be consistent with [13].

We define a directed acyclic graph $G_{\mathsf{g}} = (V, E)$ as follows. The set of vertices V consists of three partitions $V := V_1 \cup V_2 \cup V_3$. We let

$$V_1 := \{S_{i,j} : 0 \le i \le t \wedge 0 \le j < b\}$$

where for each fixed i, the tuple $S_{i,0}, \ldots, S_{i,b-1}$ represents the b words of the sponge state after $(i-1)$-th application of the sponge permutation π. Additionally,

$$V_2 := \{T_{i,j} : 0 \le i < t \wedge 0 \le j < r\}$$

where for each fixed i, the tuple $T_{i,0}, \ldots, T_{i,r-1}$ represents the r words of the upper part of the sponge right before the i-th application of the sponge permutation π, but after the i-th absorption. Finally,

$$V_3 := \{M_{i,p} : 0 \le p < m \wedge 0 \le i < t \\ \wedge (\exists j \in \{0, \ldots, r-1\} : p = [ir + j]_m)\}$$

where $M_{i,p}$ represents the state of the memory word $M[p]$ after the updates that occurred as a result of the i-th squeezing step. Note that in each round i we only introduce new M-vertices for memory cells that were updated in this round (as captured by the condition $\exists j \in \{0, \ldots, r-1\} : p = [ir + j]_m$). Let us for every $v \in \{S_{i,j}, T_{i,j}, M_{i,p}\} \subseteq V$ define the *timestamp* of v as $\mathsf{ts}(v) = i$.

We describe the set of edges as a union of six partitions $E := \bigcup_{i=1}^{6} E_i$. The first three sets

$$E_1 := \{(S_{i,j}, T_{i,j}) : 0 \le i < t \wedge 0 \le j < r\}$$
$$E_2 := \{(T_{i-1,j}, S_{i,k}) : 1 \le i \le t \wedge 0 \le j < r \wedge 0 \le k < b\}$$
$$E_3 := \{(S_{i-1,j}, S_{i,k}) : 1 \le i \le t \wedge r \le j < b \wedge 0 \le k < b\}$$

9. Including the operation `Trans` would lead to a similar analysis, while the effects of ROM can be amortized over any number of evaluations of Gambit as the content of ROM remains the same.

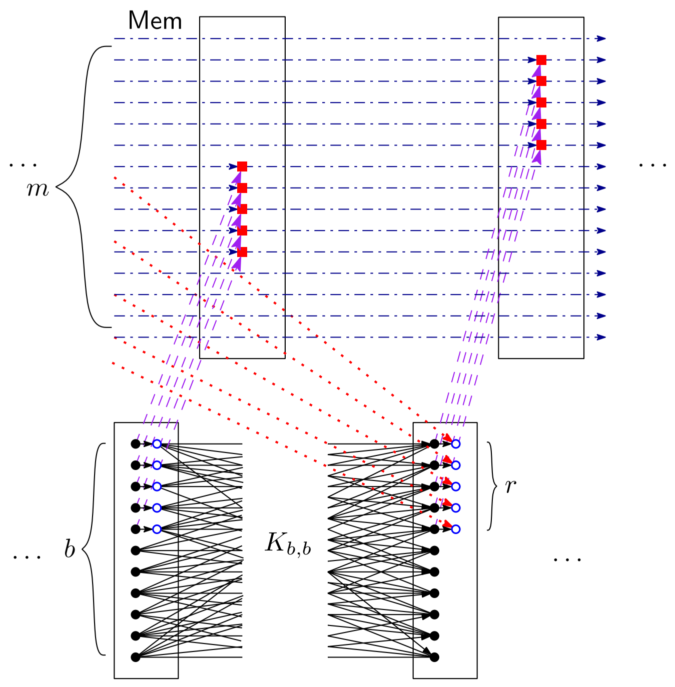

Figure 4. A snippet of the graph G_{g} capturing the i-th permutation step. The sponge nodes $\{S_{i,\bullet}, S_{i+1,\bullet}\} \subseteq V_1$ and $\{T_{i,\bullet}, T_{i+1,\bullet}\} \subseteq V_2$ are denoted by full black nodes and empty blue nodes, respectively. The memory nodes $\{M_{i,\bullet}, M_{i+1,\bullet}\} \subseteq V_3$ correspond to the red squares. The sponge update edges E_1, E_2, E_3 come as solid black, the memory update edges E_4 as purple dashed lines, the memory-to-sponge edges E_5 are the dotted red lines, and finally the dash-dotted blue lines are the memory-dependence edges E_6.

capture the dependence of a sponge state on the state immediately preceding it. Edges in

$$E_4 := \{(S_i, M_{i,ir+j}) : 0 \le i < t \wedge 0 \le j < r\}$$

represent the updates of memory Mem based on the current sponge state in each step. The set

$$E_5 := \left\{(M_{\mathsf{last}(i,j),[f(ir+j)]_m}, T_{i,j}) \colon 0 \le i < t \wedge 0 \le j < r\right\}$$

where

$$\begin{aligned}\mathsf{last}(i,j) := \max\big\{k \in \mathbb{N} : M_{k,[f(ir+j)]_m} \in V \\ \wedge [k < i \vee (k = i \wedge [fj]_r > j)]\big\}\end{aligned}$$

contains edges representing the influence of the memory Mem on the update of the sponge state. Intuitively, the index $\mathsf{last}(i,j)$ represents the last iteration in which the memory at position $[f(ir+j)]_m$ was overwritten before it is read out and used during iteration i. This is guaranteed by the fact that the vertex $M_{\mathsf{last}(i,j),[f(ir+j)]_m}$ exists (i.e., this position was updated in iteration $\mathsf{last}(i,j)$) and either $\mathsf{last}(i,j) < i$, or if the update occurs in the current iteration ($\mathsf{last}(i,j) = i$) then it occurred before the value is read out (this is captured by the condition $[fj]_r > j$). Finally, the edges in

$$\begin{aligned}E_6 := \{(M_{j,p}, M_{i,p}) : M_{j,p}, M_{i,p} \in V \wedge j < i \\ \wedge (\forall k \colon j < k < i \Rightarrow M_{k,p} \notin V)\}\end{aligned}$$

represent that whenever updating a memory cell M_p, the new value also depends on the old one.

6.3. Choice of Parameters

The author of [13] gives some conditions and recommendations on how to choose the parameters. First, m cannot be too big compared to t, otherwise the entire memory is not used, which is captured by the proposed condition $m < tr/4$. Also, the interleave factor f needs to satisfy $\gcd(m, f) = 1$ and $\gcd(m, f-1) = 1$ in order to ensure proper usage of the entire memory and regular behaviour of edge lengths (as we will see shortly), respectively.

For the purposes of our attack we assume that m and t are large compared to b and r, while still evaluating the complexity of the attack in all these parameters. We will restrict to the case $m = \Theta(t)$ since the general paradigm of iMHFs is to use as much memory as possible.

6.4. The Attack

Intuitively, the weakness of Gambit lies in its distribution of edge lengths: the lengths of relevant edges are distributed almost uniformly (i.e., the number of edges of each particular length is roughly the same), which opens door for an attack following the ideas already sketched in Section 3.1.

In our more detailed analysis below, some of our estimates are not completely precise (indicated by $\approx$) for two reasons: (1) the edge lengths considered will be scaled down by a factor of r which might cause rounding errors, and (2) the graph looks slightly differently at its "beginning"/"end" compared to normal operation. However, neither of these effects can have a noticeable impact on our results and so we opt for a simpler presentation in this version of the paper.

First we formalize the statement about edge lengths, assuming $\gcd(m, f) = \gcd(m, f-1) = 1$ as required by [13]. For the purposes of Section 6, we define a slighly different notion of an edge length that we denote by len: for $e = (u, v) \in E$ we define the length of e as $\mathsf{len}(e) := \mathsf{ts}(v) - \mathsf{ts}(u)$. Whenever we talk about edge length in this section, we refer to the function $\mathsf{len}(\cdot)$.

Lemma 5. For any $e \in \bigcup_{i=1}^{4} E_i$ we have $\mathsf{len}(e) \le 1$. For $e \in E_6$ we have $\mathsf{len}(e) \approx m/r$. Finally, for $e \in E_5$ we have $0 \le \mathsf{len}(e) \le m/r$ and the distribution of edge-lengths is approximately uniform, that is

$$|\{e \colon \mathsf{len}(e) = x\}| \approx tr^2/m$$

for each $x \in \{1, \ldots, m/r\}$.

Proof: The claim for $\bigcup_{i=1}^{4} E_i$ is easy to observe directly from the definition. For $e \in E_6$, informally $\mathsf{len}(e)$ corresponds to the difference in timestamps between two consecutive updates of a memory cell $\mathsf{Mem}[p]$. Since the memory cells are updated successively, $m-1$ updates (of other cells) occur before a cell is updated again. However, r consecutive updates always have the same timestamp, hence we have $\mathsf{len}(e) = m/r$ for $e \in E_6$. Finally, for $e \in E_5$ the length $\mathsf{len}(e)$ corresponds to the timestamp difference between updating a memory cell and reading it out. By design of Gambit (due to the requirement $\gcd(m, f-1) = 1$) the number of updates (of other cells) that happen between updating a cell $\mathsf{Mem}[p]$ and reading it out is uniformly distributed for various p. This uniformity is approximately maintained also after downscaling the lengths by the factor r due to r updates happening under each timestamp. □

This allows us to see G_{g} as a layered graph.

Theorem 2. G_{g} is a $(t^{7/4}r^2/m, t^{1/4}, e, d)$-layered graph, where $e(i) = t^{1/2}(b+2r)$ and $d(i) = 3t^{1/4}$ for each $i \in [t^{1/4}]$.

Proof: Let us define the set of nodes to remove as start points of short edges. Namely, we set

$$S = \{v \in V_3 \colon \exists e \in E_5, e = (v, w), \mathsf{len}(e) \le t^{3/4}\}.$$

Due to Lemma 5 we have

$$|S| \approx t^{3/4} \cdot \frac{tr^2}{m} = \frac{t^{7/4}r^2}{m}.$$

Now let us partition V into $V^1, \ldots, V^\lambda$ such that

$$V^i = \{v \in V \colon (i-1)t^{3/4} \le \mathsf{ts}(v) < it^{3/4}\} \setminus S \quad \text{for } i \in [\lambda].$$

Note that $\lambda \approx t^{1/4}$ and that every edge $e = (v, w)$ of $G_{\mathrm{g}} - S$ has $v \in V^i$, $w \in V^j$ for $i \le j$. It remains to discuss the reducibility of the induced subgraphs $G^i = (V^i, E^i)$ of $G_{\mathrm{g}} - S$. First, let us note that E^i is disjoint with both E_5 and E_6. The former holds since the "short edges" were deleted and the "long edges" step outside V^i. For the latter we recall that $\mathsf{len}(e) \approx m/r$ for $e \in E_6$ and the assumed $m = \Theta(t)$.

Now we see that $\mathsf{len}(e) \le 1$ for every $e \in E^i$ and so we will treat G^i as we would treat a chain graph. Note that G^i has $|V^i| \le t^{3/4}(b+2r)$ vertices and $\mathsf{depth}(G^i) \le 3t^{3/4}$. By removing all $t^{1/2}(b+2r)$ vertices v with $\mathsf{ts}(v) \mid t^{1/4}$, we guarantee no path longer than $3t^{1/4}$. This establishes $(t^{1/2}(b+2r), 3t^{1/4})$-reducibility for each G^i. □

After applying Lemma 2, we get that G_{g} is $(e, 3t^{1/2})$-reducible, where $e = t^{3/4}\max(b+2r, r^2t/m)$. Applying Theorem 1 (with $g = e$), this gives us an algorithm $\mathcal{A}_{\mathrm{g}}$ for computing Gambit with the following asymptotic performance.

Corollary 3. For $m = \Theta(t)$, $\mathcal{A}_{\mathrm{g}}$ has energy-complexity $E_{\bar{R}}(\mathcal{A}_{\mathrm{g}}) = O(t^{1.75})$.

This beats the naïve algorithm with its energy complexity of $t(m+\bar{R}) = \Omega(t^2)$.

6.5. Comparison for Practical Parameters

To go beyond the asymptotic analysis, we set $m = tr/4$ as suggested by [13] and apply Theorem 1 setting $n = |V| \approx t(b+2r)$ and $g = e$, $\delta = b$). We obtain

$$E_{\bar{R}}(\mathcal{A}_{\mathrm{g}}) \le t^{7/4}(b+2r)\left[\frac{3(b+2r)(\bar{R}+1)}{e'} + (b+1)e' + \bar{R}\right], \tag{1}$$

where $e' = \max(b+2r, 4r)$, and a similar bound for $AT_R(\mathcal{A}_{\mathrm{g}})$ (see Theorem 1).

Next, we will argue that the value $\bar{R}$ can be replaced with $\bar{R}/b$. The value $\bar{R}$ upper bounds the cost of calling the sponge function and performing a `xor` but now Theorem 1 pays the cost $\bar{R}$ for each node of the b output nodes of the sponge function graph. That is b more times then necessary. The remaining computations are `xor`s and are very cheap compared to calling the sponge function, hence $\bar{R}/b$ still upper-bounds their cost.

[13] proposes to use one of the two variants of the Keccak [5] permutation, taking 1600 bits of input and keeping a capacity of either 256 or 512 bits. This gives us $b = 25$ and $r \in \{12, 17\}$ (recall that these are expressed in words). Using 1GB of memory corresponds to $m = 2^{27}$ words. Plugging these values into (1) gives us approximately $E_{\bar{R}}(\mathcal{A}_{\mathrm{g}}) \approx 2^{62}$ for values of $\bar{R} \le 5000$, not beating the naïve algorithm with $E_{\bar{R}}(\mathcal{A}_{\mathrm{g}}) \approx 2^{54}$.

We stress that the above estimate is overly pessimistic for several reasons related to the difficulty of adapting our model to sponges (namely, representing the state of the sponge as b separate nodes). While this does not affect the asymptotic analysis (where b is considered a constant), it plays a significant role in the concrete evaluation. If the concrete complexity of the attack is a primary goal, we believe that a significant improvement can be obtained by adapting the attack to the sponge structure instead of using the attack of Theorem 1 in a black-box way.

7. Lyra 2

Lyra 2 is a sponge-based password-hashing function proposed in [11]. It is built on top of a sponge with an internal state consisting of w bits, while its upper part affected directly by absorbing and squeezing consists of b bits. Data is handled as words of length W — let $b = \beta \cdot W$ and $w = \alpha \cdot W$. The function is parameterised by R[10]$, C, T \in \mathbb{N}$, and runs in two phases: setup and wandering. The setup phase is data-independent and initializes, in a recursive manner, a memory matrix M of dimension $R \times C$ where each memory cell is of size b bits. The wandering phase, on the other hand, is data-dependent and updates these memory cells in T iterations. Hence, roughly speaking, R and C are the memory parameters, whereas T is the time parameter. Since the paper pertains only to data-independent MHFs, we shall consider only the setup phase, and hence the algorithm corresponding to $T = 0$. The description of this reduced Lyra 2 function, denoted $\mathcal{N}_{\mathsf{L}}$, is given in Algorithm 5 in the Reference Material.

7.1. The Naïve Algorithm

Lyra 2 uses the underlying sponge in two modes of operation: squeeze (which is the normal mode) and duplex (which is squeeze with an input). For sake of simplicity of the analysis, we presume that both modes of the sponge

10. Not to be confused with the core-memory parameter.

have the same time-cost. Also, since other operations — i.e., xor ($\oplus$), wordwise addition ($\boxplus$) and rotation (rot) — are accompanied by a call to the permutation function f, we assume their costs are absorbed by the permutation. As the naïve algorithm $\mathcal{N}_{\mathsf{L}}$ has to save the content of the memory cells M and the state of the sponge in each iteration, its energy complexity is

$$E_{\bar{R}}(\mathcal{N}_{\mathsf{L}}) = (C \cdot R) \cdot (\omega + (R \cdot C) \cdot \beta). \qquad (2)$$

We show that there exists an attack against the set-up phase of the function by giving an evaluation algorithm $\mathcal{A}_{\mathsf{L}}$ that requires energy complexity less than that in (2).

7.2. Graph Representation

For showing the aforementioned attack, we first have to describe the graph G_{L} that corresponds to Lyra 2. We warm up with the description of a simplified graph $G_{\mathsf{L,s}} = G_{\mathsf{L,s}}(R, C)$ in which the state of the sponge and the content of the memory are represented as separate vertices, and then describe how the graph can be attacked. Later, in Section 7.3.1, we show how $G_{\mathsf{L,s}}$ can be transformed to $G_{\mathsf{L}} = G_{\mathsf{L}}(R, C, \beta, \alpha)$ where vertices represent individual words and hence are of the same weights. We also explain a fine-grained attack in terms of β and α.

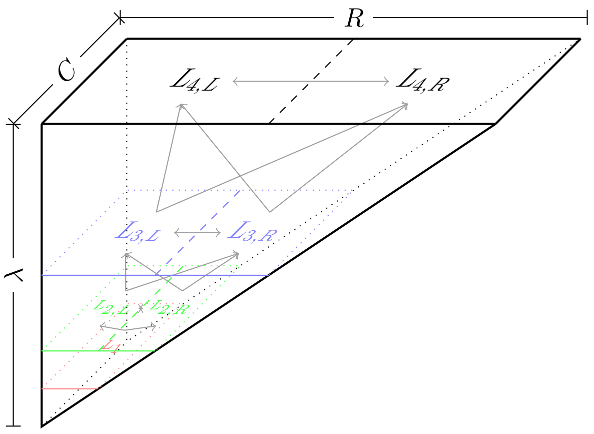

Figure 5. A schematic of the first four layers of $G_{\mathsf{L,s}}$ — the dependence between these layers are indicated by arrows. All the edges coming into a layer L_ℓ belong to the previous layer ($L_{\ell-1}$), whereas all the edges going out of the layer are to the next layer ($L_{\ell+1}$) (except for the extreme layers). Refer to Figure 6 for the structure of a particular layer.

$G_{\mathsf{L,s}}$ has a layered structure (see Figure 5), with each layer (L_ℓ) having a left ($L_{\ell,L}$) and a right half ($L_{\ell,R}$) — there are $\lambda_{\mathsf{L}} = \lceil \log_2 R \rceil - 1$ layers altogether. Roughly speaking, in a particular "window" ℓ, the memory cells in $L_{\ell,R}$ are *filled* by feeding to the sponge function certain values of memory cells from $L_{\ell-1}$. At the same time, these values in $L_{\ell-1}$ are *updated* to $L_{\ell,L}$. And the process is repeated. Thus, G_{L} has three types of vertices, corresponding the *state* of the sponge (V_s), the (*memory*) fills (V_m), and (memory) *updates* (V_u). In particular, V_s and V_m are always on the right half, whereas V_u is on the left half of the layer.[11] A more concrete description follows.

Let $\Lambda(i) = \lfloor \log_2 i \rfloor$ for $i > 4$ and $\Lambda(i) = 1$ for $i \leq 4$. Then, $G_{\mathsf{L,s}} = (V, E)$ where $V = V_s \cup V_m \cup V_u$ and $E = E_1 \cup E_2 \cup E_3 \cup E_4$. Each vertex has a type (s, m or u) and is indexed by its row number, column number and its level, and *in that order*. The vertices are:

1) $V_s := \{s_{i,j,\Lambda(i)} : i \in [0, R), j \in [0, C)\}$
2) $V_m := \{m_{i,j,\Lambda(i)} : i \in [0, R), j \in [0, C)\}$
3) $V_u := \{u_{i,j,\ell} : i \in [0, R), j \in [0, C), \ell \in (\Lambda(i), \lambda_{\mathsf{L}}]\}$

The edges can be classified into three: the edges that correspond to the state of the sponge (E_1, E_2), a fill (E_3) and an update (E_4). A detailed description of the edges follow — henceforth, $E_{x,y}$ denotes the yth part in the edge set E_x; for example, E_1 below equals $E_{1,1} \cup E_{1,2}$.

$$\begin{aligned}
E_1 :=& \{(s_{i,j,\Lambda(i)}, s_{i,j-1,\Lambda(i)})\}_{i\in[0,R), j\in[C]} \\
&\cup \{(s_{i,0,\Lambda(i)}, s_{i+1,C-1,\Lambda(i+1)})\}_{i\in[0,R)} \\
E_2 :=& \{(m_{i-1,j,\Lambda(i-1)}, s_{i,C-1-j,\Lambda(i)}\}_{i\in[0,R), j\in[0,C)} \\
&\cup \{(u_{row^1(i),j,\Lambda(i)-1}, s_{i,C-1-j,\Lambda(i)}\}_{i\in[0,R), j\in[0,C)} \\
&\cup \{(u_{prev^1(i),j,\Lambda(i)}, s_{i,C-1-j,\Lambda(i)}\}_{i\in[0,R), j\in[0,C)} \\
E_3 :=& \{(m_{i-1,j,\Lambda(i-1)}, m_{i,C-1-j,\Lambda(i)})\}_{i\in[0,R), j\in[0,C)} \\
&\cup \{(s_{i,C-1-j,\Lambda(i)}, m_{i,C-1-j,\Lambda(i)})\}_{i\in[0,R), j\in[0,C)} \\
E_4 :=& \{(u_{row^1(i),j,\Lambda(i)-1}, u_{row^1(i),j,\Lambda(i)})\}_{i\in[0,R), j\in[0,C)} \\
&\cup \{(s_{i,C-1-j,\Lambda(i)}, u_{row^1(i),j,\Lambda(i)})\}_{i\in[0,R), j\in[0,C)}
\end{aligned}$$

Intuitively, $row(\cdot)$ and $prev(\cdot)$ can be seen as a set of permutations between the layers. For example, each element in $L_{3,R}$ has a unique predecessor each in $L_{3,L}$ and L_2. Concretely, for a given i, $row^1(i)$ and $prev^1(i)$ are defined as:

$$\begin{aligned}
row^1(i) &= \quad (i+1)(2^{\lfloor 0.5\,\Lambda(i)\rfloor+1} \pm 1) \quad \mod 2^{\Lambda(i)}, \\
prev^1(i) &= \quad i(2^{\lfloor 0.5\,\Lambda(i)\rfloor+1} \pm 1) \quad \mod 2^{\Lambda(i)},
\end{aligned}$$

where we use plus if $\Lambda(i)$ is odd, minus if $\Lambda(i)$ is even. If $i = 3$, then we set $row^1(3) = 1$ and $prev^1(3) = 0$.

7.3. The Attack

Lemma 6. $G_{\mathsf{L,s}} = G_{\mathsf{L,s}}(R, C)$

1) has $|V| < 3(R-2)C$ vertices, $\mathsf{indeg}(G_{\mathsf{L,s}}) = 4$; and
2) is $(\lambda_{\mathsf{L}}, 2^\ell \lfloor C^{1-\chi} - 1\rfloor + C\lfloor 2^{\ell(1-\psi)} - 1\rfloor, 2\lceil 2^{\ell\psi} C^{\chi}\rceil)$-perfectly layered for any real $0 \leq \chi, \psi \leq 1$.

Proof: L_ℓ has at total of $3 \cdot C2^\ell$ vertices, $C2^\ell$ each from V_s, V_m and V_u. Thus, by the sum of a geometric series, the number of vertices in $G_{\mathsf{L,s}}$ is at most $3C \cdot \sum_{\ell=1}^{\lambda_{\mathsf{L}}} 2^\ell = 3C \cdot (R-2)$. The vertex with the highest indegree is in V_s — thus, the indegree of the graph is four. As for *Part 2* of the claim, consider a particular layer L_ℓ. The edges that remain in the layer are (subsets of): E_1, $E_{2,1}$, $E_{2,3}$, E_3 and $E_{4,2}$. Of these, E_1, $E_{2,1}$ and E_3 are within

11. As the first four rows of the memory are filled a differently from the rest, L_1 has a different structure from the rest of the layers. In the description of the edges, however, we treat this layer as the rest for the sake of a clear description. But this does not change affect the attacks.

$L_{\ell,R}$; the edges in $E_{2,3}$ are from $L_{\ell,R}$ to $L_{\ell,L}$, whereas those in $E_{4,2}$ are vice versa from $L_{\ell,L}$ to $L_{\ell,R}$. In particular an edge $(s_{i,j,\ell}, u_{i,j,\ell}) \in V_s \times V_u$ in $E_{4,2}$ has an accompanying edge $(u_{i,j,\ell}, s_{i,j+1,\ell}) \in V_u \times V_s$ in $E_{2,3}$ (or $(u_{i,j,\ell}, s_{i+1,0,\ell})$, if $j = 0$). Thus, L_ℓ is a $2^\ell \times C$ grid as shown in Figure 6. It turns out that the structure is not $(2^\ell \lfloor C^{1-\chi} - 1\rfloor + C\lfloor 2^{\ell(1-\psi)} - 1\rfloor, 2\lceil 2^{\ell\psi} C^\chi\rceil)$-depth robust. This is accomplished by deleting certain "rows" and "columns" *entirely*. The columns (resp., rows) that are removed are subsets of V_m (resp., V_s and V_u) at regular $2^{\ell\psi}$ (resp., C^χ) intervals, where $0 \le \chi, \psi \le 1$. As a result, the whole grid is subdivided into $\lceil 2^{\ell\cdot(1-\psi)} C^{1-\chi}\rceil$ sub-grids with the property that there are no edges *between* any two. The number of vertices removed is $2^\ell \cdot \lfloor C^{1-\chi} - 1\rfloor + C \cdot \lfloor 2^{\ell(1-\psi)}\rfloor$, and the longest path in this sub-grid is $2 \cdot \lceil 2^{\ell\psi} \cdot C^\chi\rceil$ long. □

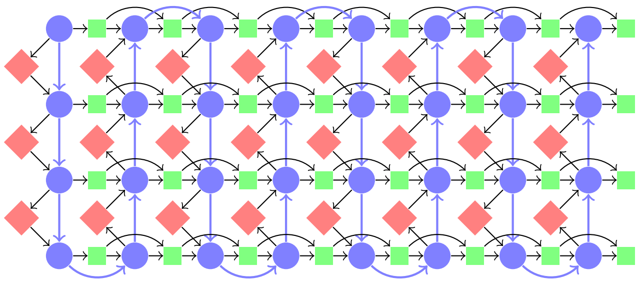
Figure 6. The structure of L_3 for $G_{\mathsf{L,s}}$ with $C = 4$ – the edges going out to the next layer and coming in from the previous layer have been omitted. For the sake of presentation, we have *flipped* the direction of odd columns (i.e., the blue columns $1, 3, 5, 7$ — equivalently, columns $9, 11, 13, 15$ in the function). The blue (circular) nodes represent the state of the sponge ($\in V_s$), the green (square) nodes represents a memory fill ($\in V_m$) whereas a red (diamond) node represents an update ($\in V_u$). Hence, the blue and green nodes belong to $L_{3,R}$, whereas the red nodes belong to $L_{3,L}$. The path with the (thicker) blue edges corresponds to the state of the sponge.

7.3.1. Extending to vertices of arbitrary weights. To construct $G_{\mathsf{L}} = G_{\mathsf{L}}(R, C, \beta, \alpha)$, we replace the vertices and edges in $G_{\mathsf{L,s}}$ in a similar vein to Gambit. Intuitively, we are constructing a graph corresponding to Lyra 2 where all the vertices have the same weight as that of a word. Each vertex $v \in V_m, V_u$ is replaced by a set of vertices $\mathbf{v} := \{v_k : k \in [0, \beta)\}$. On the other hand, each vertex $s \in V_s$ is replaced by a *graph* $G_u = (\mathbf{s}, E_s)$ that represents the sponge function (see Figure 7). The vertices are

$$\begin{aligned}\mathbf{s} &= \mathbf{s}_I \cup \mathbf{s}_O \cup \mathbf{s}_J\\ &= \{s_{I,k}, s_{O,k} : k \in [0, \alpha)\} \cup \{s_{J,k} : k \in [0, \beta)\}\end{aligned}$$

where I, O and J stand for "input", "output" and "junction", respectively. The edges consist of $\{(s_{I,k}, s_{J,k}) : k \in [0, \beta)\}$, along with the complete bipartite graph $K_{\alpha,\alpha}$ between the vertices $\{s_{J,0} \ldots s_{J,\beta-1}, s_{I,\beta} \ldots s_{I,\alpha-1}\}$ and $\mathbf{s}_O$. Therefore, $|\mathbf{s}| = 2 \cdot \alpha + \beta$ and $|E_s| = (\alpha^2 + \beta)$.

An edge $(s_1, s_2) \in V_s^2$ in $G_{\mathsf{L,s}}$ is replaced by α edges $\{(s_{1,O,k}, s_{2,I,k}) : 0 \le k < \alpha\}$, where $\mathbf{s}_i$ is the vertex that corresponds to G_{s_i}, for $i = 1, 2$. An edge $(u, v) \in V_s \times V_k$ is replaced by a set of β edges $\{(s_{O,k}, v_k) : k \in [0, \beta)\}$, whereas an edge $(v, u) \in V_k \times V_s$ is replaced by a set of β

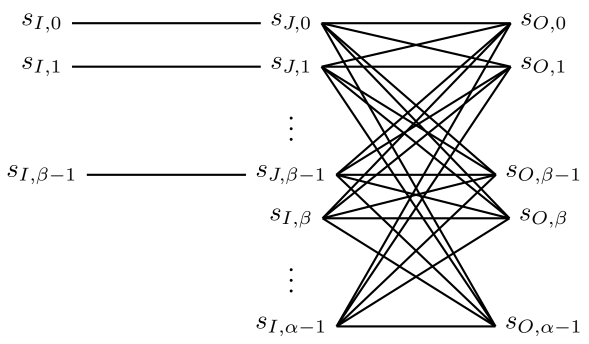

Figure 7. A node representing the sponge computation. The edges are all directed left to right.

edges $\{(v_k, s_{J,k}) : k \in [0, \beta)\}$.[12] Thus, given $G_{\mathsf{L,s}}$, one can construct G_{L}. In addition, it is not difficult to see that G_{L} also captures the computation of the function.

Theorem 3. $G_{\mathsf{L}} = G_{\mathsf{L}}(R, C, \beta, \alpha)$

1) has $|V| < (3\beta + 2\alpha)(R - 2)C$ vertices, $\mathsf{indeg}(G_{\mathsf{L}}) = \max\{4, \alpha\}$; and
2) is $(\lambda_{\mathsf{L}}, (\beta + \alpha)2^\ell\lfloor C^{1-\chi} - 1\rfloor + \beta C\lfloor 2^{\ell(1-\psi)} - 1\rfloor, 4\lceil 2^{\ell\psi} C^\chi\rceil)$-perfectly layered for any real $0 \le \chi, \psi \le 1$

Proof: Note that the number of vertices of each type (i.e., s, m and u) in $G_{\mathsf{L,s}}$ are the same: $C \cdot (R - 2)$. Thus, from the description of the transformation, the number of vertices in G_{L} is $2C \cdot (R-2) \cdot \beta + C \cdot (R-2) \cdot (2\alpha + \beta) = (3\beta + 2\alpha) \cdot C(R-2)$. The node with the maximum indegree in G_{L} is either the output node of $\mathbf{s}$ or the junction node of $\mathbf{s}$ — the former has an indegree α, whereas the latter 4. That establishes *Part 1*. Now, recall the strategy described for a layer L_ℓ in the proof of *Part 2* of Lemma 6. For every vertex of type m (resp., u) that is removed in G_{L}, remove *all* the corresponding vertices $\mathbf{m}$ (resp., $\mathbf{u}$). In addition, for vertex of type s removed, remove the corresponding *output* vertices $\mathbf{s}_O$. A straightforward recalculation shows that the number of vertices removed are

$$(\beta + \alpha) \cdot 2^\ell \lfloor C^{1-\chi} - 1\rfloor + \beta \cdot C\lfloor 2^{\ell(1-\psi)} - 1\rfloor$$

whereas, the resulting depth is $2 \cdot 2\lceil 2^{\ell\psi} C^\chi\rceil$, where the additional factor of two arises due to the introduction of G_u (which has depth 2). □

Corollary 4. Let $\mathcal{A}_{\mathsf{L}}$ denote the algorithm guaranteed by Theorem 1. Then, $E_{\bar{R}}(\mathcal{A}_{\mathsf{L}}) = O(n^{1.67})$.

Proof: Assuming that C[13], β and α are constants independent of R, by *Part 1* of Theorem 3, we have $n = |V| = O(R)$, $\mathsf{indeg}(G_{\mathsf{L}}) = O(1)$. Next, we plug in the

12. To be precise, the edges have to be treated a bit differently to account for the ⊞ and the *rot* operations (presuming that the number of rotations is a product of W, as recommended). However, this changes nothing with regards to the attacks and so we avoid it in favour of a cleaner exposition.
13. The recommended value is $C \ge \rho_{max}/\rho$, where ρ_{max} is the regular number of rounds for the underlying sponge and ρ is the reduced number of rounds. Therefore, $C = 1$ in case the full rounds are used.

values of e and d from *Part 2* into Lemma 2 with $\chi = 1$ and $\psi = 1/3$ to get $\bar{d} = O(R^{1/3})$ and $\bar{e} = O(R^{2/3})$. Finally, applying Theorem 1 with $g = R^{1/3}$ establishes the corollary.[14] □

Since $\mathcal{N}_{\mathsf{L}}$ has an energy complexity of $O(R^2) = O(n^2)$, the (asymptotic) quality of the attack is $O(n^{0.33})$.

7.4. Comparison for Practical Parameters

We evaluate the attack for $1GB$ of RAM on a 64-bit machine — that is, $W = 64$ bits and, thus, the memory consists of 2^{27} words. We consider the two permutation functions that were suggested in the paper:

1) 512-Keccak with an internal state of size $1600b$: i.e., $\alpha_K = 1600/64 = 25$, and $\beta_K = 512/64 = 8$; and
2) 512-Blake2b with an internal state of size $1024b$: i.e., $\alpha_B = 1024/64 = 16$, and $\beta_B = 512/64 = 8$.

Note that the naïve algorithm in has an energy complexity of $\approx (2 \cdot 2^{27}) \cdot 2^3 = 2^{57}$. For establishing concrete bounds, the analysis has to be more fine-grained than in the proof of Corollary 4. The attack performs the best for 512Blake2b with full rounds of computation (i.e., $\rho = \rho_{max}$ and $C = 1$) with $\bar{E} = 3000$. On plugging in the values of e and d from *Part 2* of Theorem 3 with $\chi = 1$ into Lemma 2 we get $\bar{d} < (4 \cdot R^{\psi})/(2^{\psi} - 1)$ and $\bar{e} < ((2\beta_B + \alpha_B)R^{1-\psi})/(2^{1-\psi} - 1)$. Finally, applying Theorem 1 with

$$g = \sqrt{\frac{14\bar{R}}{2^{\psi} - 1}} R^{(1+\psi)/2}$$

and $\psi = 0..23$ to get $E_{\bar{R}}(\mathcal{A}_{\mathsf{L}}) \approx 2^{59.5}$. Thus, the quality of the attack is $\approx 1/6$. The complexity for other settings is strictly worse.

8. POMELO

POMELO is a password-hashing function proposed by [15]. It is parametrized by $m, t \in \mathbb{N}$.

It consists of a data-independent and a data-dependent phase. After an initial setup of a chunk of $\sigma = 2^m$ memory cells, both phases iterate 2^{t-1} times through the chunk of memory. Both phases employ a pseudorandom number generator. The first RNG is data-independent while the second one is data-dependent. Apart from using different RNGs the two phases work the same way. We will only consider the initialization and data-independent phase. During these POMELO iterates $\tau = 1 + 2^{t-1}$ times through the memory chunk for a total of $T = \tau\sigma$ steps.

By the specification, t can be chosen between 0 and 25 while m can be chosen between 8 and 33. [15] Furthermore, a parameter w which will specify the size of a memory window is fixed as 2^{12} in the specification.

14. The attack is asymptotically worse for the case $C = R$. We have $n = O(R^2)$, and by substituting $\chi = \psi = 1/2$, we get $\bar{d} = O(R)$ and $\bar{e} = O(R^{3/2})$. Finally, applying Theorem 1 with $g = \bar{e}$, we get $E_{\bar{R}}(\mathcal{A}_{\mathsf{L}}) = O(R^{7/2}) = O(n^{1.75})$.

15. If $t = 0$, the data-independent phase iterates through the first half of memory and the data-dependent phase over the second half.

Algorithm 6 in the Reference Material outlines the data-independent part of the POMELO Algorithm as described in [15], which we shall refer to as $\mathcal{N}_{\mathsf{P}}$. The naïve algorithm maintains the whole memory of size σ memory cells during the entire computation ($\tau\sigma$ iterations), therefore we have $\mathsf{cmc}(\mathcal{N}_{\mathsf{P}}) = O(\sigma^2\tau)$. If $\bar{R}$ is the core-energy ratio of the bit operations performed during one iteration, we have $\mathsf{cec}(\mathcal{N}_{\mathsf{P}}) = O(\sigma\tau\bar{R})$, for a total of $E_{\bar{R}}(\mathcal{N}_{\mathsf{P}}) = O(\sigma\tau(\sigma + \bar{R}))$.

8.1. Graph Representation

The RNG defines one single deterministic sequence of 64-bit numbers r_i, $\sigma \le i < T$. This sequence is indepedent of the input and the parameters of the algorithm and defined via Algorithm 7 in the Reference Material. We define r_i^{high} as the upper 48 bits of r_i, taken modulo σ, and r_i^{low} as the lowest 13 bits of r_i. These sequences will now define two sequences ℓ_i and g_i of indices to memory cells. We interpret them as "local" and "global" memory indices, and these define the memory cells to be updated at time i. Henceforth all indices to memory cells are implicitly taken modulo σ. For $i \ge \sigma$, set $\ell_i = i - w + r_i^{low}$. Due to the range of r_i^{low} it means that $\ell_i \in [p - w, p + w)$. For $i \ge \sigma$, set $g_i = r_i^{high}$ if i is divisible by 32 and $g_i = g_{i-1}+1$ otherwise. If $i < \sigma$ we don't define ℓ_i and g_i as we don't update "randomly" selected memory cells during the first iteration, i.e. initialization, through the memory matrix.

For a given set of parameters σ, τ we now define a directed acyclic graph $G_{\mathsf{P}} = (V, E)$ as follows.

The vertices $M_{i,p}$ correspond to updates of memory cell at position p at time i. At each time step i, only three memory cells are updated, at locations $p = i \mod \sigma$, ℓ_i and g_i, though the memory cell at p is updated three times. Formally, we define the vertex set of updated cells at time i as $V_i = \{M''_{i,p}, M'_{i,p}, M_{i,p}, M_{i,\ell_i}, M_{i,g_i}\}$ if $i \ge \sigma$ and $V_i = \{M_{i,p}\}$ if $i < \sigma$. In the latter case we let $M'_{i,p}$ and $M''_{i,p}$ be alternative names for $M_{i,p}$. Together we get the vertex set $V = \bigcup_{0 \le i < T} V_i$

As for Gambit, for a vertex $v \in V_i$ we define the timestamp of v as $\mathsf{ts}(v) = i$. Define $\mathsf{last}(i, p)$ to be the last time before step i when memory cell p was updated. More formally, $\mathsf{last}(i, p) := \max\{\mathsf{ts}(M_{j,p}) : j < i, M_{j,p} \in V\}$.

We define the edge set as the union of the following sets:

$$E_S := \bigcup_{0 \le i < T} \{M_{\mathsf{last}(i,i-d),i-d} : d = 2, 3, 7, 13\} \times \{M''_{i,i}\}$$

$$E_Z := \bigcup_{0 \le i < T} \{M''_{i,i}M'_{i,i}, M'_{i,i}M_{i,i}, M'_{i,i}M_{i,\ell_i}, M_{i,i}M_{i,g_i}\}$$

$$E_L := \bigcup_{\sigma \le i < T} \{M_{\mathsf{last}(i,\ell_i),\ell_i}M'_{i,i}, M_{\mathsf{last}(i,\ell_i),\ell_i}M_{i,\ell_i}\}$$

$$E_G := \bigcup_{\sigma \le i < T} \{M_{\mathsf{last}(i,g_i),g_i}M_{i,i}, M_{\mathsf{last}(i,g_i),g_i}M_{i,g_i}\}$$

$$E_M := \{M_{\mathsf{last}(i,i),i}M''_{i,i}, : 0 \le i < T\}$$

Before explaining the intuition behind these edge sets, define the length of an edge $e = (u, v) \in E$ as $\mathsf{len}(e) = \mathsf{ts}(v) - \mathsf{ts}(u)$. Again, thoughout this section by edge length we always refer to the quantity $\mathsf{len}(\cdot)$. For an edge $e = (M_{j,p}, M_{i,p})$ we define the *reach* of e as $\mathsf{reach}(e) = i - p \mod \sigma$, as the edge reaches $\mathsf{reach}(e)$ memory cells back. If $p = i$ and $j < i$ we define $\mathsf{reach}(e) = \sigma$ rather than 0. Note that the length of an edge is at most its reach, as memory cell p was updated $i - p \mod \sigma$ time steps ago.

Then E_S are short edges reaching just a few memory cells back. Their length is at most 13. E_Z contains zero-length edges between memory cells updated during one step. E_L contains local "random" edges that refer to memory cells close to the current memory cell i while E_G contains global "random" edges. Finally, E_M contains the edges from the old to the new state of the main memory cell during an update. Edges from E_M always have reach σ.

We shall also consider the graph G'_{P} derived from G_{P} by contracting the vertices from V_i into a single one for each i and removing loops. Note that this is a chain graph with additional edges, and the length of an edge in this graph reflects the length in the original graph. G'_{P} has T vertices with indegree at most 7, while G_{P} has $\approx 5T$ vertices with indegree at most 5.

8.1.1. Edge Length Distribution. Recall that the RNG provides a sequence r_i of length $(T - \sigma)$ of 64-bit integers. Any such sequence r_i gives rise to a graph in the way outlined in the previous section. We shall consider the set S_{P} of such graphs over all such sequences r_i with uniform distribution.

Lemma 7. Let $\beta_\sigma \in [w, \sigma - w]$ and $\beta_w \in [13, w]$, and consider a graph G' randomly sampled from S_{P}, after contracting the vertex sets V_i. The expected number of edges of length between w and β_σ is at most $(3.5\beta_\sigma/\sigma)(T-\sigma)$. The expected number of edges of length between 13 and β_w is at most $(1.5\beta_w/w + 3\beta_w/\sigma)(T - \sigma)$.

Proof: We first wish to estimate the number of edges of length between w and β_σ. Recall that the length of an edge is always at most the reach of that edge, so we do not need to consider edges of reach less than w.

Case 1: Edges with reach between w and $\beta_\sigma + w$. Notice that local edges $e \in E_L$ either have reach less than w or greater than $\sigma - w$. Edges from E_M have reach σ so only global edges can have reach between w and $\beta_\sigma + w$. Due to uniform distribution of the random numbers, we get that the expected ratio of global edges with reach between w and $\beta_\sigma + w$ is β_σ/σ.

Case 2: Edges with reach greater than $\beta_\sigma + w$. All edges of E_M, expectedly half of the edges of E_L and an expected ratio of $(\sigma - \beta_\sigma - w)/\sigma$ edges from E_G have reach greater than $\beta_\sigma + w$. So for a given vertex in G', the expected number of incoming edges with reach greater than $\beta_\sigma + w$ is $1 + 0.5 + (\sigma - \beta_\sigma - w)/\sigma \leq 2.5$. We want to estimate how many of these edges have short length.

So assume we have a vertex $M_{i,p}$ and let $I := [i - \beta_\sigma, i - w]$. We want to estimate the probability that the set of vertices with timestamp j for $j \in I$ contains a vertex $M_{j,p}$ referring to memory cell p. The probability that p was chosen as a global memory index, i.e. that $g_j = p$ for some $j \in I$, is $(\beta_\sigma - w)/\sigma$. If p was chosen as local memory index for some $j \in I$, then we must have $p \in [(i - \beta_\sigma) - w, (i - w) + w]$, i.e. p is in a local window around I. But this means the reach of $(M_{j,p}, M_{i,p})$ is less than $\beta_\sigma + w$, so we already considered that edge in Case 1.

So taking both cases together, for a given vertex in G' we get an overall expected number of at most $2.5(\beta_\sigma - w)/\sigma + \beta_\sigma/\sigma \leq 3.5\beta_\sigma/\sigma$ incoming edges of length between w and β_σ. As we have T vertices in G' in total and the first σ vertices do not have any local or global incoming edges, we get the first result.

A similar analysis yields the second result. □

While the POMELO specification does not mention any properties expected from the random number generator, we will now assume that the edge length distribution G_{P} is similar to the expected distribution of a randomly sampled graph from S_{P}. In particular we will assume that the properties postulated in Lemma 7 also apply to G'_{P}.

8.2. The Attack

We will now outline the attack on G_{P}. Let $\beta_\sigma \in [w, \sigma]$ and assume we remove all vertices that have incoming edges of length between w and β_σ. Then by Lemma 1, for any γ_σ we can remove wT/γ_σ further vertices such that the depth of the remaining graph is at most $\gamma_\sigma T/\beta_\sigma$. This is achieved by removing chunks of w consecutive vertices, every γ_σ vertices. Thus the factor of γ_σ represents an upper bound for the length of the longest path only using edges of length at most w. We wish to improve upon this upper bound by removing some more vertices.

So consider a chunk of $N := \gamma_\sigma$ consecutive vertices in G'_{P}. Let $\beta_w \in [13, w]$ and assume we remove all vertices that have incoming edges of length between 13 and β_w. Then again, by Lemma 1, for any γ_w we can remove $13N/\gamma_w$ vertices such that the depth of the remaining graph is at most $\gamma_w N/\beta_w$.

Using this new upper bound for the length of the longest path only using edges of length at most w, we obtain an overall bound on the depth of $\gamma_w N/\beta_w(T/\beta_\sigma) = T\frac{\gamma_\sigma}{\beta_\sigma}\frac{\gamma_w}{\beta_w}$. This was achieved by deleting $wT/\gamma_\sigma + 13T/\gamma_w$ vertices, and vertices with incoming edges of certain lengths.

Applying Lemma 7 to estimate the total, we get that in order to reduce the depth to $T\frac{\gamma_\sigma}{\beta_\sigma}\frac{\gamma_w}{\beta_w}$, the number of vertices to be deleted is at most $T(3.5\beta_\sigma/\sigma + 1.5\beta_w/w + 3\beta_w/\sigma + w/\gamma_\sigma + 13/\gamma_w)$.

Notice that if we translate this result back to the original graph G_{P}, the number of vertices deleted is at most 5 times as high, and similarly the longest remaining path is at most 5 times as long, so asymptotically we obtain the same result.

Theorem 4. G_{P} is (e, d)-reducible for $e = T(3.5\beta_\sigma/\sigma + 1.5\beta_w/w + 3\beta_w/\sigma + w/\gamma_\sigma + 13/\gamma_w)$ and $d = T\frac{\gamma_\sigma}{\beta_\sigma}\frac{\gamma_w}{\beta_w}$.

We shall now try to find values for $\beta_\sigma, \gamma_\sigma, \beta_w, \gamma_w$ that allow for an attack on Pomelo. Again, as with Gambit, we focus on the case $\tau = O(1)$ since with iMHFs we want to use as much memory as possible. Let's write $w = \sigma^c$ for some c. If we then set $\beta_\sigma = \sigma^{(2+c)/3}$, $\gamma_\sigma = \sigma^{(1+2c)/3}$, $\beta_w = \sigma^{2c/3}$, $\gamma_w = \sigma^{c/3}$, we get the graph is $(\sigma^e, \sigma^{2/3})$-reducible for $e = \max(\frac{2+c}{3}, 1 - \frac{c}{3})$. Note that this is best for $c \approx 0.5$ and thus $w \approx \sqrt{\sigma}$. If c is close to 0 or 1, the same approach with only one of the two depth reduction steps yields better results.

Recall that G_P has in-degree $O(1)$. Applying Theorem 1 (with $g = \sigma^{5/6}$), this gives us an algorithm $\mathcal{A}_\mathsf{P}$ to compute POMELO with $\mathsf{cmc}(\mathcal{A}_\mathsf{P}) = O(\sigma^{1+e})$. With the core-energy ratio for the operations $\bar{R} = O(1)$ and $w = \Theta(\sqrt{\sigma})$, in particular we get:

Corollary 5. For $\tau = O(1)$ and $w = \Theta(\sqrt{\sigma})$, $\mathcal{A}_\mathsf{P}$ has energy-complexity $E_{\bar{R}}(\mathcal{A}_\mathsf{P}) = O(\sigma^{11/6})$.

Notice that if we treat w like a constant instead (i.e. $c = 0$), we can obtain an algorithm with asymptotic energy complexity of $O(\sigma^{7/4})$ via the remark from Lemma 1.

8.3. Comparison for Practical Parameters

The POMELO specification does not suggest specific choices for the time and memory parameters. We shall choose $\sigma = 2^{25}$, such that the memory usage is 2^{30} bytes, or 1 GB. We set $t = 4$ so that $T = 2^{29}$. With this choice the memory cost is still proportionally high while the graph has a large proportion of local and global random edges (which are missing during the first iteration through the memory matrix). Setting the parameters as in Corollary 5 does not yield a sensible attack as $e > T$, meaning we delete all vertices to reduce the depth of the graph. This is due to the large constant 13 not being taken into account by the asymptotic result. Treating the memory window w like a constant and only doing one of the depth reduction steps is therefore more sensible. We shall set $\beta_\sigma = 2^{19}$ and $\gamma_\sigma = 2^{16}$, estimate the number of edges of length between w and β_σ via Lemma 7 and apply Lemma 1. If we set the core-energy ratio of the bit operations performed to $\bar{R} = 100$ (which is a very conservative estimate and is in practice likely much lower) we then get an estimate of $E_{\bar{R}}(\mathcal{A}_\mathsf{P}) \approx 2^{63}$, compared to $E_{\bar{R}}(\mathcal{N}_\mathsf{P}) \approx 2^{54}$ for the naïve algorithm.

References

[1] Joël Alwen and Jeremiah Blocki. Efficiently computing data-independent memory-hard functions. Cryptology ePrint Archive, Report 2016/115, 2016. http://eprint.iacr.org/2016/115.

[2] Joël Alwen and Jeremiah Blocki. Towards practical attacks on argon2i and balloon hashing. Cryptology ePrint Archive, Report 2016/759, 2016. http://eprint.iacr.org/2016/759.

[3] Joël Alwen and Vladimir Serbinenko. High parallel complexity graphs and memory-hard functions. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 595–603. ACM Press, June 2015.

[4] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 181–197. Springer, Heidelberg, April 2008.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak reference, 2011.

[6] Alex Biryukov and Dmitry Khovratovich. Tradeoff cryptanalysis of memory-hard functions. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 633–657. Springer, Heidelberg, November / December 2015.

[7] Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Rig: A simple, secure and flexible design for password hashing. In *Information Security and Cryptology*, pages 361–381. Springer, 2014.

[8] Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. Rig: A simple, secure and flexible design for password hashing version 2.0. 2014.

[9] Bill Cox. Twocats (and skinnycat): A compute time and sequential memory hard password hashing scheme. *Password Hashing Competition. v0 edn.*, 2014.

[10] Christian Forler, Stefan Lucks, and Jakob Wenzel. The catena password-scrambling framework, 2015.

[11] Marcos A. Simplicio Jr., Leonardo C. Almeida, Ewerton R. Andrade, Paulo C. F. dos Santos, and Paulo S. L. M. Barreto. Lyra2: Password Hashing Scheme with improved security against time-memory trade-offs.

[12] C. Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCan 2009*, 2009.

[13] Krisztián Pintér. Gambit – A sponge based, memory hard key derivation function. Submission to Password Hashing Competition (PHC), 2014.

[14] Clark D. Thompson. Area-time complexity for VLSI. In *Proceedings of the 11h Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1979, Atlanta, Georgia, USA*, pages 81–88, 1979.

[15] Hongjun Wu. POMELO – A Password Hashing Algorithm, 2015.

Reference Material: The Naïve Algorithms

For the convenience of the reviewers, we provide here the descriptions of the naïve algorithms to compute the considered functions. They can also be found in the referenced publications.

9. Rig.v2

Here we give the algorithm to compute Rig.v2 from [8] (Algorithm 1). The function $\pi(|\alpha|)$ initializes h_0 with $|\alpha|$ bits after the decimal point of the constant π. As mentioned previously, the original analysis from [8] gives a time complexity $O((\lambda+1)mr)$ and space complexity $O(m)$ for computing a single password.

Algorithm 1: Rig.v2 $(pwd, s, \lambda, m_c, t, \ell, r)$

```
$x := pwd \,||\, pwd_\ell \,||\, s \,||\, \ell_s \,||\, n \,||\, \ell$
$m := 2^{m_c}$ ; $\alpha := H_1(x)$
for $round := 1 \ldots r$ do
    $h_0 := \pi(|\alpha|)$
    $a[0] := \alpha \oplus h_0$ ; $k[0] := \text{trunc}_t(h_0)$
    for $i := 1 \ldots m$ do
        $h_i := H_2(i \,||\, a[i-1] \,||\, k[i-1])$
        if $i \neq m$ then
            $a[i] := \alpha \oplus h_i$ ; $k[i] := \text{trunc}_t(h_i)$
    for $i := 1 \ldots \lambda$ do
        for $j := 1 \ldots m$ do
            $a[j-1] := a[j-1] \oplus h_{im+j-1}$
            $k_{temp}[j-1] := k[br(j-1)] \oplus \text{trunc}_t(h_{im+j-1})$
            $h_{im+j} := H_2((im+j) \,||\, a[j-1] \,||\, k_{temp}[j-1])$
        $k := k_{temp}$
    $h^*_{round} := (H_3((n+1)m+1) \,||\, h_{(n+1)m} \,||\, s \,||\, m$
    if $round < r$ then $\alpha := h^*_{round}$
return $h^*_r$
```

10. TwoCats

The algorithm for evaluating the data-independent loop of TwoCats as given in [9] is described by Alg. 2. It takes pwd, its size ℓ_{pwd}, salt s, its size ℓ_s, and memory cost m_c. Then the length of a block b_ℓ is 4096 and the memory length m_ℓ is $(1024 * 2^{m_c})/4$. $\tilde{H}$ is a hash where the input are different constants - password pwd, a $salt$, their lengths pwd_ℓ and $salt_\ell$, and the block-length $b\ell$.

The slidingReverse function is a variant of the bit reversal function; its computation given in [9] is described by Alg. 3.

Algorithm 2: TwoCats $(pwd, salt, m_c)$

```
PRK[0...7]:=$\tilde{H}(pwd, salt, pwd_\ell, salt_\ell, b_\ell)$
uint32 state[8] := hashState (PRK, 0)
mem[$0 \ldots b_\ell - 1$] := expand $(H, b_\ell, state)$
preAd := 0, toAddr := $b_\ell$
for $i := 1 \ldots m_\ell/(2 * b_\ell) - 1$ do
    $a$ := state[0]
    fromAd := slidingReverse $(i) * b_\ell$
    for $j := 0 \ldots b_\ell/8 - 1$ do
        for $k = 0 \ldots 7$ do
            state[$k$] :=
            (state[$k$] + mem[preAd++]) $^{\text{mem[fromAd++]}}$
            state[$k$] := rotateLeft (state[$k$], 8)
            mem[toAddr++] := state[$k$]
    state := hashState $(state, a)$
```

Algorithm 3: SlidingReverse(i)

```
$reversePos$ := reverse($i$, numBits($i$) $- 1$)
if $reversePos + (1 \ll (\text{numBits}(i) - 1)) < i + 1$ then
    $reversePos += 1 \ll (\text{numBits}(i) - 1)$
return $reversePos$
```

By [9], the architecture of the "resistant" loop was chosen because it seemed to be most resistant against the author's automated pebbling algorithm. Setting $n := m_\ell/(2b_\ell) - 1$, the naïve algorithm has complexity $O(n^2)$. The naïve algorithm maintains all the memory during the entire computation and thus has space-time complexity of $O(m_\ell^2)$.

11. Gambit

The algorithm for computing Gambit given in [13] is described in Algorithm 4 (note that any indexing into the arrays Mem and ROM is done modulo the size of the respective array).

Algorithm 4: Gambit$(pwd, salt, t, m, dkid)$

```
S.Init()
Mem[$0 \ldots m-1$] := 0
S.Absorb($salt||pwd||pad$)
for $i := 0 \ldots t-1$ do
    $R$ := S.Squeeze()
    for $j := 0 \ldots r-1$ do
        Mem[$i \cdot r + j$] := Mem[$i \cdot r + j$] $\oplus$ Trans($R[j]$)
        $W[j]$ := (Mem[$(i \cdot r + j) \cdot f$] $\oplus$ ROM[$i \cdot r + j$])
    S.Absorb($W$)
S.AbsorbOvr($dkid$)
$key$ := S.Squeeze()
return $key$
```

12. Lyra 2

The naïve algorithm for computing the setup phase of Lyra 2 is given in Algorithm 5.

Algorithm 5: $Lyra2_{H,H_\rho,\omega}(pwd, salt, R, C, \ell)$

```
params ← $len(k)\|len(pwd)\|len(salt)\|R\|C$
$H.absorb(pad(pwd\|salt\|params))$
$gap \leftarrow 1,\ stp \leftarrow 1,\ wnd \leftarrow 2,\ sqrt \leftarrow 2$
$prev^0 \leftarrow 2,\ row^1 \leftarrow 1,\ prev^1 \leftarrow 0$
for $col \leftarrow 0$ to $C-1$ do
    $M[0][C-1-col] \leftarrow H_\rho.squeeze(b)$
for $col \leftarrow 0$ to $C-1$ do
    $M[1][C-1-col] \leftarrow$
    $M[0][col] \oplus H_\rho.duplex(M[0][col], b)$
for $col \leftarrow 0$ to $C-1$ do
    $M[2][C-1-col] \leftarrow$
    $M[1][col] \oplus H_\rho.duplex(M[1][col], b)$
for $row^0 \leftarrow 3$ to $R-2$ do
    for $col^0 \leftarrow 0$ to $C-1$ do
        $rand \leftarrow H_\rho.duplex(M[row^1][col] \boxplus$
        $M[prev^0][col] \boxplus M[prev^1][col], b)$
        $M[row^0][C-1-col] \leftarrow M[prev^0][col] \oplus rand$
        $M[row^1][col] \leftarrow M[row^1][col] \oplus rot(rand)$
    $prev^0 \leftarrow row^0,\ prev^1 \leftarrow row^1,$
    $row^1 \leftarrow (row^1 + stp) \mod wnd$
    if $row^1 = 0$ then
        $wnd \leftarrow 2 \cdot wnd,\ stp \leftarrow sqrt + gap,$
        $gap \leftarrow -gap$
        if $gap = -1$ then
            $sqrt \leftarrow 2 \cdot sqrt$
$H.absorb(M[row^0][0])$
return $H.squeeze(k)$
```

13. Pomelo

Algorithm 6 outlines the data-independent part of the POMELO Algorithm as described in [15] which we shall refer to as $\mathcal{N}_\mathsf{P}$. A memory cell is 256 bits (four 64-bit words). The indices of all accesses to memory are to be taken modulo σ. In these algorithms, $\lll$ denotes bit shift and $\lll$ bit rotations, and $\ll_w$ and $\lll_w$ refer to those operations applied to each word individually.

Algorithm 7 defines the random number generator. The size of its state is 64 bits.

Algorithm 6: Pomelo$(pwd, salt, t, m)$

```
$\tau, \sigma, w := 2^{t-1}, 2^m, 2^{12}$
$\mathsf{Mem}[0 \ldots \sigma-1] := 0$
$params := len(pwd)||len(salt)||output\_len$
$\mathsf{Mem}[0 \ldots 12] := pwd||salt||0 \ldots 0||params||pad$
for $i := 13 \ldots \sigma-1$ do
    $\mathsf{Mem}[i] \mathrel{+}= \mathsf{Mem}[i-2] \oplus \mathsf{Mem}[i-3]$
        $+ \mathsf{Mem}[i-7] \oplus \mathsf{Mem}[i-13]$
    $\mathsf{Mem}[i] = (\mathsf{Mem}[i] \lll_w 17) \lll 64$
RNG.init()
for $j := 0 \ldots \tau-1$ do
    for $i := 0 \ldots \sigma-1$ do
        $random\_number :=$ RNG.get()
        $\ell := i - w + (random\_number \mod 2w)$
        if $i \mod 32 == 0$ then
            $g := random\_number \gg 16$
        $g \mathrel{+}= 1$
        $\mathsf{Mem}[i] \mathrel{+}= \mathsf{Mem}[i-2] \oplus \mathsf{Mem}[i-3]$
            $+ \mathsf{Mem}[i-7] \oplus \mathsf{Mem}[i-13]$
        $\mathsf{Mem}[i] = (\mathsf{Mem}[i] \lll_w 17) \lll 64$
        $\mathsf{Mem}[i] \mathrel{+}= \mathsf{Mem}[\ell] \ll_w 1$
        $\mathsf{Mem}[\ell] \mathrel{+}= \mathsf{Mem}[i] \ll_w 2$
        $\mathsf{Mem}[i] \mathrel{+}= \mathsf{Mem}[g] \ll_w 1$
        $\mathsf{Mem}[g] \mathrel{+}= \mathsf{Mem}[i] \ll_w 3$
```

Algorithm 7: Pomelo Random Number Generator

```
def RNG.init():
    $state = 123456789$
def RNG.get():
    $out = state$
    $state \mathrel{+}= state \ll 2$
    $state = (state \lll 19) \oplus 3141592653589793238$
    return $out$
```