ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Klondike: Finding Gold in SIKE

Master Thesis

Giacomo Fenzi

September 5, 2022

Advisors: Prof. Dr. Kenneth Paterson, Dr. Fernando Virdia

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

This work is an exploration of the practical costs of running van Oorschot and Wiener's (vOW) algorithm [vW94] for golden collision finding in practice. We investigate the cost of memory accesses and propose new models for concrete cost estimation. We also estimate the complexity of vOW's algorithm in a distributed setting, proposing new models and an open-source library for performing large scale golden collision searches, which we refer to as Klondike. In this thesis, we also fill some of the gaps in the theoretical understanding of vOW's algorithm behaviour and its interaction with memory, and propose new state-of-the-art models for estimating both the fill rate of the memory and the number of distinguished triples required to fill the memory partially or in full. We analysize some of the original heuristic parameter selection in [vW94], and find them suboptimal for large instance size. Consequently we propose a new heuristic criterion that yields better practical running times in our experiments. We present the currently available models (from [vW94, TID21]) for computing the running time of vOW's algorithm in terms of function evaluations and present our own model, which relies on our previous results with regards to fill rate. Finally, we present an analysis of the practical costs of attacking the Supersingular Isogeny Path Problem using vOW's algorithm.

Acknowledgments

Dobfar

Papà

Thanking someone is always a delicate effort. On one hand, it is only natural giving thanks to those who contributed to the development of some work, acknowledging that any written piece of work is not birthed from the proverbial cave, but rather is the culmination of many small threads that tangled themselves in precisely the right way. On the other, Giulio Andreotti famously said that '*La riconoscenza è solo speranza di piaceri futuri*', which can loosely be translated to 'Gratitude is only hope of future pleasures'. I hope I can convince the reader that this chapter is fully dedicated to the first aim, and that my intentions are as far from Andreotti's as can be.

My first thanks go to Fernando, which, at the risk of sounding cliché, was the best supervisor that I could have wished for. At the start of this project my knowledge of cryptanalysis and collision finding was minimal, and I have to thank Fernando for, almost by osmosis, helping me gather the understanding which I hope to have conveyed in this work. He was there for my rambles, in discovering what worked and, most importantly, what did not.

Then, I want to thank Kenny for his guidance, his insights, and his mentorship during my time at ETH, and for generously providing a large chunk of the computational resources required for this project.The remaining part was graciously contributed by Martin Albrecht, without whom it would have not been possible to derive some of the key results in this thesis.

I want to also apologize to Kien, Massimiliano and Marc for being subject to my rambles about isogenies during the best part of the last years, and thank them for not once telling me to shut up as they should have righly

done. My thanks also go to Azzurra, Emma, Gianluca and Riccardo, for the *benessere* during this summer in Zürich. Francesco, Giorgio, Mario, Renato, Federico thought me that '*camminare è povertà*', and for that I deeply thank them. Tommaso and Alberto were my guidance and my wisdom in the start of my academic career, and I cannot do justice to how helpful that was.

Finally, I want to thank my family for being my rock. Nothing would have been possible if not for their unconditional trust and support. Mamma, Papà, Caterina and Margherita were with me every step of the way, reminding me that 'dobfar': *dobbiamo farcela*. I want to dedicate this work to Dede, Dino and Mariarosa, who are not here with us, but in a sense never left.

Contents

Chapter 1

Introduction

While the field of cryptography has grown considerably in recent years and now encompasses a variety of different applications, a central focus remains the need of communicating securely (and efficiently) over insecure channels. The integer factorization and the discrete logarithm intractability assumptions have been central in building key exchange mechanisms to satisfy that need, but are being threatened by Shor's algorithm [Sho94] and by the increasing progress towards building quantum computers. As a response to this threat, the field of post-quantum cryptography has developed, which, at its core, aims to develop cryptosystems that can efficiently be run on classical machines and are secure against attackers with a quantum computer at their disposal. In this context of post-quantum cryptography, a number of new computational assumptions have been introduced, and with them follows the need for appropriate cryptanalysis to better understand whether those assumptions are, in fact, justified. This thesis stems from this need, in particular applied to the family of *isogeny-based* assumptions.

Isogenies, as elliptic curves, originate in the context of algebraic geometry and can be used to build public schemes with very compact public keys. Roughly speaking, an isogeny is a 'nice' mapping between elliptic curves, preserving both the algebraic and the geometric characteristics of the curve. We can then consider the undirected graph with elliptic curves as nodes, and where two curves are connected if and only if an isogeny (of a fixed degree) connects them. This graph turns out to be Ramanujan, making so that the distribution of the end node of short walks in the graph is close to the uniform distribution over the graph, a property which makes the graphs very appealing for cryptography. The fundamental hardness assumption on which isogeny-based crypto relies on is the **S**upersingular **I**sogeny **P**ath **P**roblem, that is: given two curves E_0, E_1 connected by a hidden isogeny $\phi : E_0 \to E_1$ of small degree it will be hard to recover such an isogeny. Rephrased in the graph context, SIPP states that given two connected nodes

it is hard to find a path that connects them. From this graph intuition, we can derive a first approach towards solving SIPP: a 'meet-in-the-middle' method. We could compute a list of points $E_{0,1}, E_{0,2}, \ldots$ that are connected to the node E_0, together with the path $\psi_i : E_0 \to E_{0,i}$. Similarly, we can compute a list $E_{1,1}, \ldots$ of those connected to E_1. If we find then a $E_{0,i} = E_{1,j}$ we will have found a connecting path between E_0, E_1. Perhaps suprisingly, this attack is the current state of the art classical algorithm for solving SIPP, in terms of running times. However, in practice it suffers from large memory requirments that make the practical cost of running such an attack unfeasible. Subject to this constraint, then the best classical (and quantum) algorithm is based on the classical van Oorschot and Wiener's (vOW) algorithm for golden collision finding [vW94].

This observation motivated this work, whose focus is on developing models for cryptanalysis using the vOW's algorithm in the context of golden collision search.

Since the attack can be described as texbook vOW on a isogeny-tailored random function, we are able to give a very modular description, in which isogenies only appear in the final instantiation. Thus, this thesis can be considered as an investigation on collision finding algorithms in general.

The problem of collision finding is central to many applications of cryptography, and thus of independent interest. A collision for a function $f : D \to R$ is a pair of distinct elements $(x, y) \in D^2$ that are mapped to the same image by f, i.e. $f(x) = f(y)$. In this work, we consider attacks on 'generic' functions. This is a similar paradigm to the random oracle model, in the sense that we assume we are given query access to this function f, and the description of f is hidden from us (in general, f will be drawn at random from the set of functions, which makes its description unfeasible to even read). The algorithms that we will be presenting can be considered as providing upper bound on the levels of security that any function can have against collision resistance.

The main computational problem that we will be considering in this thesis is that of not only finding a collision, but finding a golden collision (following the terminology introduced in [vW94]). Consider the set of collisions of f, which we denote as $\mathrm{Coll}(f)$. In that set we identify one or few collisions, which we refer to as golden collisions. The question is, given oracle access to f, and oracle access to a 'gold-test', what is the cost for finding such a collision? The aforementioned [vW94] algorithm is considered the best algorithm for solving this problem, and has survived essentially unchanged since its original description. As noted, we are interested in furthering our understanding of the algorithm from a cryptanalytical perspective. Previous cryptanalytical works has considered the practical cost of breaking the SIPP assumption, but this line of work has mainly considered models in which

memory accesses take $O(1)$ time and can be perfectly parallelized.

In order to understand why memory accesses can have a large impact on the cost of golden collision search in practice, a brief description of the algorithm is in order. vOW's algorithm can be run in parallel on a number of processors, each of which performs the same operation. Roughly speaking, each computing unit will mine a trail by iterating f from a random starting point, and then stores the start and end point of the trail in a shared memory unit. Once two trails have the same endpoint, they can be traced back to identify a collision. The mining and backtracking operations can be perfectly parallelized, but the memory is shared between every computing unit. Taking consumer hardware as an example, assuming that the cost of accessing memory can be parallelized is not a fair assumption, and as such can result in a bottleneck when scaling to cryptanalytical sizes. Furthermore, we will see that these memory accesses are essentially uniformly distributed over a potentially very large address space, so we believe that even specialized hardware would not be able to overcome this bottleneck.

A second assumption that was made in the previous analysis is that an attacker determined to break SIPP would utilize its resources to build a large centralized machine, which could then be used to run the attack. In practice, it might instead be convenient to reutilize existing computational resources, and run the attack in a distributed manner. Previous large scale computational efforts such as GIMPS [GIM] and Folding@Home [SP00] have demonstrated the potential of pooling large computational resources across the Internet, and understanding whether this can also be done efficiently for vOW can shed light on the feasibility of such an approach for smaller instances of SIPP. Furthermore, as noted, we suspect that the cost of accessing memory can be noteworthy, and thus, when limited by network latency and bandwidth, we expect this effect to be very significant, warranting an extensive investigation. Running a distributed golden collision search also requires solving a number of practical problems regarding the nodes synchronization that can already cause complications in a local setting and which get exacerbated in the distributed attack.

The primary goal of this thesis was investigating and giving cost models for this memory and distributed setting, with a focus on the SIKE [JAC+20] key encapsulation mechanism, with the promise of running large scale isogeny computations to break the challenge instance of `p217`. SIKE security relied on a problem related to SIPP, namely the **C**omputational **S**uper**s**ingular **I**sogeny problem, which essentially reveals not only the start and end curves of the isogeny $\phi : E \to E'$, but also the image of ϕ on an exponentially large subgroup of the target curve (this information is commonly referred to as torsion points). A spectacular line of work by Castryck, Decru, Martindale, Maino and Robert [CD22, MM22, Rob22] showed polynomial time

attacks on the CSSI assumption, which allowed to break the p217 instance in 9 seconds, and the parameters proposed for standardization in between 22 seconds and a couple of hours[1] on a laptop. The combination of this line of work, our estimates that running such an attack would require computational resources that were outside of our capabilities, and independent confirmation by [Cos] that breaking p217 using vOW's algorithm would have a comparable cost to breaking the RSA-1024 challenge instance[2], lead us to decide not to attempt those record breaking isogeny computations.

We want to still stress that the recent attack on the CSSI assumption crucially relies on the torsion point information, and thus that the 'pure' SIPP problem is still considered hard, with vOW's algorithm representing the best known attack.

Contributions Firstly, we propose two new cost models which fill the previous gap in the literature by accounting for memory accesses. These models are presented in Model 5.3 and Model 5.4, and differ in whether they assume that accesses to memory can be parallelized or if they constitute a serial bottleneck. These models also follow a modular approach, in the sense that they can be instantiated with any model that predicts the number of function evaluations required to run the attack, and improvements in such base model would translate directly into an improvement in the fine grained cost model. We have conducted then a series of experiments to validate these models, on a number of different servers and with different cost metrics. These models were then ported to the distributed setting previously mentioned, resulting in Model 6.1. This model is flexible enough to capture a huge variety of different network topologies, and retains the modular characteristics of the previous models. In order to validate it, we introduce Klondike, a library and a set of binaries for distributed golden collision searching using vOW's algorithm. To our knowledge, Klondike is the first public library of the kind, allowing for great configuration flexibility, modular choice of function f, syncing strategies and more.

Finally, in our investigation we noticed a path towards filling some of theoretical open questions in the analysis of the runtime of vOW's algorithm. The original vOW paper presents a 'flawed' analysis of the runtime, noting estimation of the rate at which memory fills as a complex problem. In this work, we present a series of experimentally validated models, based on classical techniques in probability to solve that problem, and in the process we develop a novel model for estimating the runtime of vOW's algorithm in the golden collision case (dependant on an accurate model for the rate at which memory fills).

[1] Using the optimised implementation of https://github.com/jack4818/Castryck-Decru-SageMath

[2] Current record is RSA-250 (829 bits)

We also show one of the first practical improvements in parameter selection for vOW's algorithm since the original paper, by showing experimentally that the original selection of $\beta = 10$ is suboptimal for large parameter sets, and propose an heuristic selection that yields improved running times.

Roadmap In Chapter 2 we will be fixing notation and terminology that will be used in the rest of this work.

Chapter 3 will introduce and formalize the problem of collision finding, multiple collision finding and golden collision finding (in Problem 3.6, 3.8, 3.9). We present and compare two classic algorithms for collision finding: the table-based approach (Algorithm 1) and Pollard's ρ-method (Algorithm 3). This section also introduces some of the main probabilistic techniques that we will be using troughout this work. Finally, we will be introducing vOW's algorithm (Algorithm 9).

Chapter 4 will introduce the two (dual) theoretical problems that we will partially answer namely: the memory filling problem and the fill rate problem (resp. Question 4.1, 4.9). We will be answering the first question via a coupon-collector inspired method, which will result in the two novel Model 4.4 and Model 4.5, which we experimentally verified to be an improvement over the state-of-the-art. In Remark 4.10 we motivated why answering Question 4.9 is important, and adapt a classical approach to memory filling to answer it, which results in Model 4.12 and Model 4.13. We also experimentally validate these models, and report on their accuracy. Finally, we look at the setting of function versioning, and ask ourselves whether the choice of [vW94] of when to switch function version is optimal, answering the question negatively, and proposing an heuristically justified new parameter selection.

Chapter 5 introduces the pre-existing models of [vW94] and [TID21] for estimating the runtime of the vOW's algorithm, and presents a novel blueprint for constructing a more accurate model for the runtime, based on Remark 4.10 and an answer to Question 4.9. On top of these models, we introduce Model 5.3 and Model 5.4 for fine grained estimation of the practical cost of running the attack, and present our experimental validation.

Chapter 6 introduces the adaptation of the "local" cost models to the distributed setting, resulting in Model 6.1. We also present Klondike, our library for distributed golden collision search, with a description of syncing algorithms and their adaptation to the distributed setting.

Finally Chapter 7 will be dedicated to showing how vOW's algorithm can be adapted to solving the SIPP problem. In doing so, we will be giving a brief introduction to isogeny-based cryptography, formally introducing CSSI (Problem 7.1) and SIPP (Problem 7.2). We report on our implementation of

efficient isogeny evaluations, and apply the models derived in the previous chapters to this setting.

Chapter 2

Preliminaries

General Notation When we are defining a new quantity we use $\triangleq$, when assigning to it (for example in an algorithm) we instead use $:=$. We often do tuple unpacking, i.e. if $r \in S_1 \times \cdots \times S_k$ then the notation $(r_1, \ldots, r_k) \triangleq r$ will define r_i to be equal to the corresponding i-th value of the tuple r.

Sets. We denote by $A \sqcup B$ the **disjoint union** of A and B. A **relation** on S_1, S_2 is a subset $R \subseteq S_1 \times S_2$. For any relation R we write $a \sim_R b \iff (a,b) \in R$, and omit the subscript whenever the relation is clear from context. For a relation $\sim$ we define:

$$a \sim_? b \triangleq \begin{cases} 1 \text{ if } a \sim b \\ 0 \text{ otherwise} \end{cases}$$

An **equivalence relation** on S is a relation $R \subseteq S^2$ that is reflexive, symmetric and transitive (i.e. for every $x \in S$, $x \sim x$, $x \sim y \Rightarrow y \sim x$, $x \sim y \wedge y \sim z \Rightarrow x \sim z$). For a set S and an equivalence relation $\sim$ we denote by $[x]_\sim \triangleq \{y \in S \mid x \sim y\}$ the **equivalence class of** x and by $S/\sim \triangleq \{[x]_\sim \mid x \in S\}$ the **quotient of** S **by** $\sim$.

Notable Sets. We let $\mathbb{N}, \mathbb{Z}, \mathbb{R}$ denote, respectively, the **natural numbers**, the **integers** and the **real numbers**. We take the dogmatic view that $0 \in \mathbb{N}$. For $n \in \mathbb{N}$, we let $[n] \triangleq \{1, \ldots, n\}$. For any integer $n > 0$, we let $\mathbb{Z}_n \triangleq \mathbb{Z}/n\mathbb{Z}$ denote the **integers modulo** n. If $q = p^n$ for p prime we write $\mathbb{F}_q$ for the (unique) **finite field** of size q. For a totally ordered set S, $a, b \in S$, we let $(a,b) = \{x \in S : a < x < b\}$, $[a,b) = \{x \in S : a \leq x < b\}$, $(a,b] = \{x \in S : a < x \leq b\}$. For a set S we let $\binom{S}{k} = \{(x_1, \ldots, x_k) \in S^k : \forall i \neq j : x_i \neq x_j\}/\sim$ where $\sim$ is the equivalence relation on S^k given by $(x_i)_{i \in [k]} \sim (y_i)_{i \in [k]}$ iff there exists a permutation π on $[k]$ such that $x_{\pi(i)} = y_i$ for every $i \in [k]$.

Notable Functions. We use $\log(\cdot)$ to denote the logarithm in base 2, and $\ln(\cdot)$ for the natural logarithm. For the exponential function, we interchangeably use $e^{(\cdot)}$ and $\exp(\cdot)$. We also will need the **Harmonic numbers**, which we denote as H_n, with $H_0 \triangleq 0$ and $H_n \triangleq \sum_{k=1}^{n} \frac{1}{k}$ for $n > 0$. We denote by $\lfloor \cdot \rfloor$ the **floor** of a number, by $\lceil \cdot \rceil$ the **ceil** of a number and by $\lfloor \cdot \rceil$ the **rounding** of a number. More formally, $\lfloor x \rfloor$ is the greatest integer z such that $z \leq x$, $\lceil x \rceil \triangleq -\lfloor -x \rfloor$ and $\lfloor x \rceil \triangleq \lfloor x + \frac{1}{2} \rfloor$. As usual, we let $\binom{n}{k}$ denote the **binomial coefficient**. Note that $\left|\binom{S}{k}\right| = \binom{|S|}{k}$.

Function Notation. We denote by $\mathsf{Funcs}(D, R)$ the set of functions $f : D \to R$. A function $f : D \to R$ is **injective** if $f(x) = f(y)$ implies that $x = y$. It is **surjective** if for every $y \in R$ there is a x such that $f(x) = y$. A function that is both injective and surjective is a **bijection** (or one-to-one). We denote the **image** of a function by $\operatorname{im} f \triangleq f(D) \triangleq \{f(x) : x \in D\}$. For a function $f : D \to R$, and a subset $S \subseteq D$ we denote the **restriction** of f to S as $f|_S$, that is the unique function $S \to R$ that agrees with f on all points of S. We let $\mathrm{id}_S : S \to S$ denote the **identity** function on S. Two functions $f, g : D \to R$ are equal if and only $f(x) = g(x)$ for every $x \in D$. For convenience we also define $\Delta_D : S \to \mathsf{Funcs}(D, S)$ to map any element $s \in S$ to the constant function $d \mapsto s$ on D. We also define the **graph of a function** $f : S \to S$ to be the directed graph on $|S|$ vertices which has an edge $a \to b$ iff $f(a) = b$.

Probability. We use the notation $x \leftarrow\!\!\$\ D$ to denote sampling a element uniformly at random from the set D. If more than one element is sampled, those samples are independent. More formally, for $n > 0$, any $t_1, \ldots, t_n \in D$

$$\Pr_{x_1, \ldots x_n \leftarrow\!\!\$ D}[\forall i\ x_i = t_i] = \frac{1}{|D|^n}$$

For simplicity, we often write $\Pr[\cdot]$ without specifying over which distribution the randomness is distributed. For a random variable X we let $E[X]$ denote the **expectation** of X, which is defined as $E[X] \triangleq \sum_x x \cdot \Pr[X = x]$, where the sum is taken over the set of xs over which $\Pr[X = x] > 0$, which is the **support** of X.

Big-Oh notation [AB09]. Let $g : \mathbb{N} \to \mathbb{R}^+$ be a function. We define, as usual, notation for asymptotic behaviour:

$$O(g) \triangleq \left\{ f : \mathbb{N} \to \mathbb{R}^+ \middle| \exists c > 0, n_0 \in \mathbb{N} \text{ s.t. } \forall n > n_0\ f(n) \leq c g(n) \right\}$$

We say $f \in \Omega(g)$ iff $g \in O(f)$ and $f \in \Theta(g)$ iff $f \in O(g)$ and $f \in \Omega(g)$. Finally, we also allow the costumary abuses of notation by letting $f = O(g)$ mean that $f \in O(g)$ and write $O(g(n))$ for $O(g)$ (and similarly for Ω, Θ). We also write $f = O(\mathrm{poly}(n))$ to signify that there exists a polynomial $p(\cdot)$, of degree independent of n, such that $f = O(p(n))$.

Security Parameters. For simplicity, we will usually describe functions and sets as concrete instances, but instead often it would be more appropriate to describe families parametrized by a security parameter $\lambda \in \mathbb{N}$. For example if we say that a certain algorithm f runs in time poly $(|S|)$ for some set S what is implicitly meant is that for each set in the family $\{S_\lambda\}_{\lambda\in\mathbb{N}}$ there is a corresponding algorithm f_λ and there exists a polynomial $p(\cdot)$ such that the running time of f_λ is $p(|S_\lambda|)$.

Definition 2.1 *A set S has an **efficient binary encoding** iff there is a integer ℓ_S such that:*

- *There is an efficiently computable injection $bin_S : S \to \{0,1\}^{\ell_S}$*
- *The language $x_\ell \in_?$ im bin_S is efficiently decidable.*
- *The inverse bin_S^{-1} : im $bin_S \to S$ is also efficiently computable.*
- *ℓ_S is $O(\text{poly}\,(\log\,(|S|)))$*

Unless otherwise specified, we assume that all the sets we deal with have an efficient binary encoding, and as such they are finite. Note that this allows us to efficiently sample and check set element for equality.

Bit operation. For any bitstring $\{0,1\}^n$ we define a bijection bintoi_n : $\{0,1\}^n \to \mathbb{Z}_{2^n}$ with inverse itobin_n. Given two bitstring $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$ we denote by $a||b$ the string in $\{0,1\}^{n+m}$ obtained by concatenating the two strings, in that order.

Approximations. We will often need to make approximations in our results. Notation wise, we write $f(x) \approx g(x)$ when $x \ll b$ to mean that the distance $|f(x) - g(x)|$ is small when x is much smaller than the bound b. We deliberatly do not completely formalise this, as it is standard. Instead, we write $f(n) \sim g(n)$ to signify asymptotic equivalence, namely that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$. We can also consider closeness of distributions. We say that two random variables A, B have approximately the same distribution if $\Pr[A = x] \approx \Pr[B = x]$ for every x in the union of the support of the two distributions. We list here the most common ones, so that we can later refer to them.

Approximation 2.2

$$\alpha n^d \sim \alpha n^d + \sum_{i=0}^{d-1} \beta_i n^i$$

Approximation 2.3 *If $|x| \ll 1$, $e^x \approx 1 + x$.*

Approximation 2.4 *For f integrable,*

$$\sum_{i=a}^{\infty} f(i) \approx \int_a^{\infty} f(x)dx \, .$$

Approximation 2.5 (Stirling)

$$n! \sim \sqrt{2n\pi}\left(\frac{n}{e}\right)^n$$

Approximation 2.6 *Let $n \to \infty$. Then*

$$\int_0^2 \exp\left(\frac{-x^2}{2n}\right) dx \approx 2$$

Proof The Laurent series of the integral at $n = \infty$ is $2 - \frac{4}{3n} + \frac{4}{5n^2} - \frac{8}{21n^3} + O(n^{-4})$. □

Approximation 2.7

$$H_n \sim \ln(n) + \gamma + \frac{1}{2n}$$

where $\gamma \approx 0.57721\ldots$ is the Euler-Mascheroni constant.

Chapter 3

Collision Finding Algorithms

In this chapter, we will be introducing the collision specific formalism and results that we will be using trought this thesis. Furthermore, we will be presenting some classical algorithms for collision finding such as Algorithm 1 and Pollard's ρ-method (Algorithm 3), analysing their runtime and their drawbacks. Finally, we will be presenting van Oorschot and Wiener's algorithm in Algorithm 9.

3.1 Collisions

First of all, let us define what a collision is.

Definition 3.1 (Collision) *Let $f : D \to R$ be a function. A pair $(x, x') \in D^2$ is a **collision of** f iff $x \neq x'$ and $f(x) = f(x')$. We define*

$$\mathrm{Coll}\,(f) \triangleq \left\{c \in D^2 |\ c \text{ is a collision of } f\right\} / \sim$$

where $\sim$ is the equivalence relation on D^2 given by $(a, b) \sim (b, a)$.

Let us consider a sufficient condition for the existence of collisions.

Theorem 3.2 *Let $f : D \to R$ be a function. If $|D| > |\mathrm{im}\ f|$ then f has a collision. In particular, if $|D| > |R|$ then f has a collision.*

Proof This follows from the pigeonhole principle. Since $|D| > |\mathrm{im}\ f|$, f cannot be injective, as such has a collision. □

We can also prove a sufficient and necessary condition for a function with equally sized domain and codomain to have a collision, namely:

Theorem 3.3 *Let $f : D \to R$ and $|D| = |R|$. f has a collision if and only if f is not a bijection.*

Proof ($\Longrightarrow$) If f has a collision, it is not injective and as such it cannot be a bijection.
($\Longleftarrow$) Suppose f is not a bijection. It can either fail to be injective or fail to be surjective. In the first case, a counterexample to injectivity directly yields a collision. In the second case, $|\text{im } f| < |R| = |D|$ and as such by our previous theorem it has a collision. □

We also note that a randomly sampled function is a permutation with very low probability, and so a randomly sampled function $S \to S$ will, with high probability, have one or more collisions. More formally:

Lemma 3.4 *Let $f \leftarrow\!\!\$\ \mathsf{Funcs}(S, S)$ with $n \triangleq |S|$. Then*

$$\Pr[f \text{ is a permutation}] = \frac{n!}{n^n}$$

In particular, Approximation 2.5 shows this probability is small as $n \to \infty$.

Proof The number of permutations on n elements is exactly $n!$, and the total number of functions on that domain is n^n. Note that by Approximation 2.5 we have that $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ and as such

$$\Pr[f \text{ is a permutation}] \sim \sqrt{2\pi n}\frac{\left(\frac{n}{e}\right)^n}{n^n} = \sqrt{2\pi n}\exp(-n)$$

which goes to 0 as $n \to \infty$. □

Very often, we will prove theorems for sequences of points uniformly sampled at random from some set, and then translate them to evalutions of some random function f. In that context we will be very interested in repeating values, which we define here.

Definition 3.5 *Let $x_1, \ldots, x_t$ be a sequence. The sequence has k **repeated values** iff $\left|\{x_i\}_{i\in[t]}\right| = t - k$.*

The problem of collision finding can be defined as follows:

Problem 3.6 *Let $\mathcal{F} \subseteq \mathsf{Funcs}(D, R)$ be a family of functions. For any algorithm $\mathcal{A}$, with oracle access to $f \leftarrow\!\!\$\ \mathcal{F}$, we define its **advantage against the collision finding problem** as*

$$\text{Adv}(\mathcal{A}) = \Pr\left[c \in \text{Coll}(f) \middle| c \leftarrow\!\!\$\ \mathcal{A}^{f(\cdot)}\right]$$

where that probability is taken over the choice of f and the random coins of $\mathcal{A}$.

Remark 3.7 *For some classes of functions, it can be remarkably hard (or easy) to find collisions. Fix a set S and consider the class of functions $\mathcal{F} \triangleq \{f_{c,r}\}_{c\in S, r\in S-\{c\}}$, where $f_{c,r}|_{S-\{c\}} = \text{id}_{S-\{c\}}$ and $f_c(c) = r$. A given f_c always has a single collision,*

namely (c, r). A generic algorithm that is given oracle access to $f \leftarrow\$ \mathcal{F}$ will only be able to find such collision by querying f at c, and as such must take $O(|S|)$ function calls on average. For the easy case instead, consider the family of constant functions $\mathcal{G} \triangleq \{g_c\}_{c \in S}$ with $g_c(x) = c$. Then every pair of distinct points in S^2 yields a collision, so in fact an adversary can find a collision with no queries.

In practice however, those kind of functions tend not to be as interesting, and instead we look at algorithm's average performance when dealing with random functions, i.e. $f \leftarrow\$ \mathsf{Funcs}(D, R)$. With this model, we will see algorithms (for example Algorithm 1) with expected running time $O(|R|^{1/2})$.

In general, we are also interested of the case of multiple collisions, which we generalize here.

Problem 3.8 *As before, let $\mathcal{F} \subseteq \mathsf{Funcs}(D, R)$ be a family of functions, and $m > 0$. For any algorithm $\mathcal{A}$, with oracle access to $f \leftarrow\$ \mathcal{F}$, we define its* ***advantage against the m-collision problem*** *as*

$$\mathrm{Adv}(\mathcal{A}) = \Pr\left[\begin{matrix}|\{c_1, \dots c_m\}| = m \\ \forall i,\ c_i \in \mathrm{Coll}(f)\end{matrix}\middle| c_1, \dots, c_m \leftarrow\$ \mathcal{A}^{f(\cdot)}\right]$$

where the probability is taken as before.

We note that the relation between m-collision and single collision finding is not as straighforward as one can imagine. Clearly, an adversary against Problem 3.8 can be adapted to one against Problem 3.6. The most natural strategy for the reverse reduction would be to run an adversary against Problem 3.6 multiple times, but this fails if, for example, the adversary is deterministic. Even if the adversary is randomized, it might have biases towards sampling certain types of collisions (as we will see in practice!) which, especially when m is large, can make the runtime of the reduction algorithm blow up or not finish at all. Furthermore, even if an adversary $\mathcal{A}$ were very nicely behaved (sampling collisions uniformly from $\mathrm{Coll}(f)$ for example) a lower bound on the reduction would require m calls to $\mathcal{A}$, and we will see algorithms to tackle Problem 3.8 that only require a factor $\sqrt{m}$ more work compared to the single collision case.

Problem 3.9 *Let $\mathcal{F} \subseteq \mathsf{Funcs}(D, R)$ be a family of functions and for $f \in \mathcal{F}$ let $\mathcal{G}_f \subseteq \mathsf{Funcs}(\mathrm{Coll}(f), \{0,1\})$ be a collection of associated predicates. For any algorithm $\mathcal{A}$, with oracle access to $f \leftarrow\$ \mathcal{F}$ and gold_f, we define its* ***advantage against the golden-collision problem*** *as*

$$\mathrm{Adv}(\mathcal{A}) = \Pr_{f \leftarrow\$ \mathcal{F}, \mathrm{gold}_f \leftarrow\$ \mathcal{G}_f}\left[\begin{matrix}c \in \mathrm{Coll}(f), \\ \mathrm{gold}_f(c) = 1\end{matrix}\middle| c \leftarrow\$ \mathcal{A}^{f(\cdot), \mathrm{gold}_f(\cdot)}\right]$$

where the probability is taken also over the random coins of $\mathcal{A}$. A $c \in \mathrm{Coll}(f)$ such that $\mathrm{gold}_f(c) = 1$ is a ***golden collision****.*

Algorithm 1: Collision-Finding with a table

Data: $f : D \to R$
Result: A collision (x, x')
$T \triangleq []$;
forall $y \in R$ **do**
  $T[y] \coloneqq \perp$;
end
repeat
  $x \leftarrow_{\$} D$;
  $y \triangleq f(x)$;
  if $T[y] \neq \perp$ *and* $T[y] \neq x$ **then**
    return $(x, T[y])$;
  end
  $T[y] \coloneqq x$;
until *False*;

A few notes are in order. Problem 3.9 is defined with respect to both a family of function and a family of predicates associated to such functions. In general, we will be interested to the case in which $\mathcal{F} = \mathsf{Funcs}(D, R)$ and the predicates selects a random collision from the set of collisions. The adversary is given oracle access and is so able to check whether a collision that it has found is indeed a golden collision.

Remark 3.10 *Problem 3.8 and Problem 3.9 are not only interesting from a theoretical standpoint. A natural way in which they occur is the following. Consider a graph G and two vertices $v_0, v_1 \in V(G)$. Suppose furthermore that it is known that a path between v_0 and v_1 exists, and that it has length d. Then one can define the function $f(i, x) : \{0,1\} \times S \to V(G)$, for some suitable S, to be a function that samples a length $d/2$ path $p(x)$ from v_i. Then, a collision of the form $f(0, x) = f(1, x')$ will allow us to recover a path from v_0 to v_1 (by concatenating $p(x)$ with the reverse of $p(x')$. Defining gold_f to test for that condition exactly brings us to the context of Problem 3.9.*

3.2 Table Based Collision Finding

In Algorithm 1 we describe a first method to find a collision in a general function $f : D \to R$. While simple, the techniques that we introduce here to analyze the running times turn out to be very useful, and we will be using them with minor modifications for the more complex methods as well.

If f indeed has a colliding pair (which by Theorem 3.2 and Lemma 3.4 happens, respectively, always and with high probability), this algorithm will eventually find it.

Let us consider the complexity of Algorithm 1[1]. We will model the function f as being randomly sampled from $\mathsf{Funcs}(D, R)$.

Lemma 3.11 *Let S be a set and let $x_1, \ldots$ be a sequence of uniformly and independently sampled points from S. Let T be a random variable denoting the number of points sampled before a value repeats. More formally, T is the smallest index such that $x_i = x_T$ for some $i < T$. Then, for $k \geq 2$*

$$\Pr[T > k] = \prod_{i=2}^{k} \left(1 - \frac{i-1}{|S|}\right)$$

Proof We show this by induction. Let $n \triangleq |S|$. Let us start with $k = 2$. $T = 2$ if and only if the $x_2 = x_1$, and that happens with probability $\frac{1}{n}$. So, since $T > 0, 1$ by definition of T, we have that $\Pr[T > 2] = \left(1 - \frac{1}{n}\right)$. Now let $k > 2$, then

$$\begin{aligned}
\Pr[T > k] &= \Pr[T > k-1 \wedge T \neq k] && (3.1)\\
&= \Pr[T > k-1] \Pr[T \neq k | T > k-1] && (3.2)\\
&= \prod_{i=2}^{k-1} \left(1 - \frac{i-1}{n}\right) \cdot \Pr[T \neq k | T > k-1] && (3.3)
\end{aligned}$$

where Equation (3.3) follows by the inductive hypothesis on k. Now, consider $\Pr[T \neq k | T > k-1]$. Since $T > k-1$ we are guaranteed the sequence $x_1, \ldots, x_{k-1}$ contains $k-1$ distinct elements. Since we are sampling a new one independently, the probability that this newly sampled point is in this list is $\frac{k-1}{n}$ and as such $\Pr[T \neq k | T > k-1] = \left(1 - \frac{k-1}{n}\right)$ □

Theorem 3.12 *[HPS14] Let T be as in Lemma 3.11. Then $E[T] \approx \sqrt{\frac{\pi |S|}{2}}$, for $|S|$ large.*

Proof From Lemma 3.11:

$$\Pr[T > k] = \prod_{i=2}^{k} \left(1 - \frac{i-1}{|S|}\right)$$

[1]Note that the first line of the algorithm would run in $O(|R|)$. In practice, we can implement the memory lazily so the cost upperbounded by the number of memory accesses, which is itself bounded by evaluations of f, and as such we do not consider it.

Now,

$$\begin{aligned}
E[T] &= \sum_{t=2}^{\infty} t \cdot \Pr[T = t] \\
&= \sum_{t=2}^{\infty} t \cdot \Pr[T > t-1 \wedge T \leq t] \\
&= \sum_{t=2}^{\infty} t \cdot (\Pr[T > t-1] - \Pr[T > t]) \\
&= 2 + \sum_{t=2}^{\infty} \Pr[T > t] && \text{(Telescoping sum)}
\end{aligned}$$

We now use Approximation 2.3 and Approximation 2.2[2] which is valid since $i \ll |S|$ to rewrite $\Pr[T > k]$ and obtain that

$$\begin{aligned}
\Pr[T > k] &\approx \prod_{i=2}^{k} \exp\left(\frac{1-i}{n}\right) \\
&= \exp\left(\frac{1}{n}\sum_{i=2}^{k}(1-i)\right) \\
&= \exp\left(\frac{-k(k-1)}{2n}\right) \\
&\approx \exp\left(\frac{-k^2}{2n}\right) && (k^2 \sim k^2 - k)
\end{aligned}$$

Finally, we approximate the infinite sum with an integral (Approximation 2.4) to obtain

$$\begin{aligned}
E[T] &= 2 + \sum_{t=2}^{\infty} \Pr[T > t] \\
&\approx 2 + \sum_{t=2}^{\infty} \exp\left(\frac{-t^2}{2n}\right) \\
&\approx 2 + \int_{2}^{\infty} \exp\left(\frac{-t^2}{2n}\right) dt && \text{(Approximation 2.4)} \\
&\approx \int_{0}^{\infty} \exp\left(\frac{-t^2}{2n}\right) dt + \left(2 - \int_{0}^{2} \exp\left(\frac{-t^2}{2n}\right) dt\right) \\
&\approx \sqrt{\frac{n\pi}{2}} && \text{(Approximation 2.6)}
\end{aligned}$$

□

[2]Note that in this case we are justified by the fact that $\int_0^\infty \exp\left(\frac{-t^2}{2n}\right) - \exp\left(\frac{-t(t-1)}{2n}\right) dt \sim \frac{1}{2}$ and so this inaccuracy does not effect later estimates.

Since we modeled the function f as a random function, we can assume that each query on a fresh value on D is equivalent to sampling uniformly an element of R (independently from previous evaluations). So, finding a collision requires in expectation $O(|R|^{1/2})$ function evaluations on *distinct* points. Points from D are sampled independently uniformly at random from D, and as such the *same analysis* from Theorem 3.12 applies and we expect an element to repeat after approximately $O(D^{1/2})$ samples. In the typical case of $|D| \gg |R|$ then, no such repetition is expected to occur before finding a collision. Consider instead the case of a self-map $f : R \to R$. To handle this case we introduce the coupon's collector analysis.

Theorem 3.13 *Let $x_1, x_2, \ldots$ be a sequence of elements uniformly and independently sampled from a set S. Let $n \triangleq |S|$ and $k \leq n$. Let T be the smallest index such that $|\{x_i\}|_{i=1}^{T} = k$. Then*

$$E[T] = n\left(H_n - H_{n-k}\right) \ .$$

In particular, if $k = n$ then $E[T] = nH_n$.

Proof Denote by t_i the smallest index such that the partial sequence $x_1, \ldots, x_{U_{i-1}}, \ldots, x_{U_{i-1}+t_i}$ contains i distinct elements, with $U_j \triangleq \sum_{i=1}^{k} t_i$. Note that $T = U_k$. Now consider an individual t_i. The probability of sampling a new coupon once $i-1$ have been sampled is exactly $\frac{n-(i-1)}{n}$, so we can model t_i as a geometric distribution, which then has expected value $\frac{n}{n-i+1}$. Then:

$$\begin{aligned} E[T] = E[U_k] = \sum_{i=1}^{k} E[t_i] &= \sum_{i=1}^{k} \frac{n}{n-i+1} \\ &= n \sum_{i=1}^{k} \frac{1}{n-i+1} \\ &= n\left(H_n - H_{n-k}\right) \end{aligned}$$

Where the first line follows by linearity of expectation and the last by rearranging. □

Remark 3.14 *Note that by Approximation 2.7 we have that $H_k \approx \ln(k) + \gamma$ so, for $n \neq k$*

$$H_n - H_{n-k} \approx \ln\left(\frac{n}{n-k}\right)$$

Otherwise

$$H_n \approx \ln(n) + \gamma$$

Let us then use Theorem 3.13 and Remark 3.14 to tackle the case of $f : R \to R$. To sample the required $\sqrt{\frac{\pi|R|}{2}}$ distinct points we will need approximately

$|R| \ln\left(\frac{|R|}{|R|-\sqrt{\frac{\pi|R|}{2}}}\right)$ samples. In practice, this is $\approx \sqrt{\frac{\pi|R|}{2}}$, so even in this case we are unlikely to need to sample many more points than what our analysis predicts.

For example, to find a collision for a typical cryptographic hash function $h : R \to R$ with $|R| = 2^{256}$ we would need to sample $\sqrt{\frac{|R|\pi}{2}} \approx 1.25 \cdot 2^{128}$ unique domain points, and Theorem 3.13 predicts that in doing so we will need in expectation one *single* extra sample.

With the above observations, we can conclude that in both cases the algorithm requires in expectation $O(|R|^{1/2})$ function evaluations to find a collision, and stores $O(|R|^{1/2})$ elements of D, or $O(\log(|D|) \cdot |R|^{1/2})$ bits.

While the large memory requirement makes Algorithm 1 unfeasible to run in practice for moderately sized instances, it has a few desirable features. First of all, it is a general method that also works when $D \neq R$. Most methods that we will later see limit their interface to self-maps, and the transformation from a general map to a self map[3] can be unyieldy. Secondly, the inner loop can be perfectly parallelized, so, if an attacker has L processing units at his disposal, the time complexity can be reduced by a factor of L.[4] Finally, this algorithm can be adapted straighforwardly to tackle Problem 3.8 and Problem 3.9.

The transformation for Problem 3.8 is as follows[5], and is shown in Algorithm 2.

Again, if f really has m distinct colliding pair, this algorithm will terminate and output the list. Our next result predicts how many collisions we should be expecting for a random function, which we denote as $C_{D,R}$.

Theorem 3.15 *Let $f \leftarrow\!\!\$\ \mathsf{Funcs}(D, R)$. The expected number of distinct collisions of f is*

$$C_{D,R} \triangleq \frac{1}{|R|} \cdot \binom{|D|}{2}$$

In particular, if $n \triangleq |D| = |R|$ we expect $\frac{n-1}{2}$ collisions and denote this as C_R.

Proof For each pair $p \in \binom{D}{2}$ define an indicator random variable I_p as follows:

$$I_p \triangleq \begin{cases} 1 \text{ if } p \in \mathrm{Coll}(f) \\ 0, \text{ otherwise} \end{cases}.$$

[3] By composing with a suitable map $R \to D$.

[4] Funnily enough, in this case the (huge) shared datastructure can be implemented without any locking, since two cells will only be accessed concurrently on a collisions, and that should be exceedingly unlikely.

[5] And the one for Problem 3.9 is very similar.

Algorithm 2: Multiple Collision Finding with a table

Data: $f : D \to R, m \in \mathbb{Z}$
Result: m collisions $\{(x_i, x_i')\}_{i=1}^{m}$
$T \triangleq []$;
coll $\triangleq \emptyset$;
forall $y \in R$ **do**
  $T[y] := \emptyset$;
end
repeat
  $x \leftarrow\!\!\$\ D$;
  $y \triangleq f(x)$;
  forall $x' \in T[y]/\{x\}$ **do**
    coll $:=$ coll $\cup \{(x, x')\}$;
  end
  $T[y] := T[y] \cup \{x\}$;
until $|\text{coll}| = m$;
return coll;

Note that the number of distinct collisions is exactly $\sum_{p \in \binom{D}{2}} I_p$, and as such by linearity of expectation it is expected to be $\sum_{p \in \binom{D}{2}} E[I_p]$. The number of unordered pairs of distinct points of D is exactly $\binom{|D|}{2}$. For a given pair (x, x') with $x \neq x'$, the probability that $f(x) = f(x')$ is $\frac{1}{|R|}$ (where the randomness is over the choice of f), and as such $E[I_p] = \frac{1}{|R|}$. Therefore the expected number of collisions is:

$$\frac{1}{|R|} \cdot \binom{|D|}{2} \qquad \square$$

Now let us estimate the running time of the algorithm. This approach was introduced in [TID21] in their analysis of parallel golden collisions search, but can be adapted almost unchanged for this case. To give some context to this proof, we will have T denote the number of points required to find the m-th collision, given that $m-1$ have been found thus far. In our proof we first will give a similar expression to that in Lemma 3.11 for the probability that $T > k$ for some k, and then use this to estimate recursively the number of points required to find m collisions.

Lemma 3.16 *Let $x_1, x_2, \ldots$ be a sequence of elements of S, uniformly and independently sampled. Suppose that the sequence $x_1, \ldots x_t$ contains $(m-1)$ repeated elements (i.e it contains $t - m + 1$ distinct elements). Denote by T the smallest index such that the sequence $x_1, \ldots, x_t, x_{t+1}, \ldots, x_{t+T}$ contains m repeated elements.*

Then, for $k \geq 1$:

$$\Pr[T > k] = \prod_{i=1}^{k} \left(1 - \frac{t-m+i}{n}\right)$$

Proof Let $n \triangleq |S|$. As in Lemma 3.11, we proceed by induction. The probability that $T = 0$ is 0, so $Pr[T > 1] = Pr[T \neq 1]$. Note that $T = 1$ only if x_{t+1} is one of the $t - m + 1$ distinct elements present in $x_1, \ldots, x_t$. So, $Pr[T = 1] = \frac{t-m+1}{n}$ and $Pr[T > 1] = 1 - \frac{t-m+1}{n}$. Now, by the same reasoning of Lemma 3.11 and the derivation of Equation (3.2) we have that

$$\Pr[T > k] = \prod_{i=1}^{k-1} \left(1 - \frac{t-m+i}{n}\right) \cdot \Pr[T > k | T > k-1]$$

Now, since $T > k - 1$ we know that the sequence $x_1, \ldots, x_t, x_{t+1}, \ldots, x_{t+k-1}$ contains only $m - 1$ repeated elements, so x_k will only create a collisions if it is not one of the $t + k - 1 - (m - 1) = t + k - m$ distinct values in the sequence. This happens with probability $1 - \frac{t-m+k}{n}$. This concludes our proof. □

Theorem 3.17 *Let $x_1, x_2 \ldots$ be a sequence as in Lemma 3.16. Let T_1 be the smallest index such that $x_1, \ldots, x_T$ contains a single repeated value. For $i > 1$, let T_i be the smallest index such that the sequence $x_1, \ldots, x_{U_{i-1}}, x_{U_{i-1}+1}, \ldots, x_{U_{i-1}+T_i}$ contains i repeated values, where $U_i \triangleq \sum_{j=1}^{i} T_j$. Then $U_k \sim \sqrt{2kn}$ as $k \to \infty$.*

Proof By Theorem 3.12 we have that $E[T_1] \approx \sqrt{\frac{\pi|S|}{2}}$. Now, to estimate $E[T_j]$ we can take a similar strategy as that in Theorem 3.12.

$$\begin{aligned} E[T_j] &= \sum_{k=1}^{\infty} k \cdot \Pr[T_j = k] \\ &= \sum_{k=1}^{\infty} \Pr[T_j > k] \end{aligned}$$

Now, by Lemma 3.16 and by applying the usual approximations we have

that

$$\begin{aligned}
\Pr[T_j > k] &= \prod_{i=1}^{k} \left(1 - \frac{U_{j-1} - 1 + i}{n}\right) \\
&\approx \prod_{i=1}^{k} \exp\left(\frac{1 - U_{j-1} - i}{n}\right) && \text{(Approximation 2.3)} \\
&= \exp\left(\frac{1}{n} \sum_{i=1}^{k} 1 - U_{j-1} - i\right) \\
&= \exp\left(\frac{1}{n}\left(k\left(1 - U_{j-1}\right) - \frac{k(k+1)}{2}\right)\right) \\
&\approx \exp\left(\frac{1}{n}\left(-kU_{j-1} - \frac{k^2}{2}\right)\right) && (1 - U_{j-1} \approx -U_{j-1},\ k^2 + k \sim k^2) \\
&= \exp\left(\frac{-2kU_{j-1} - k^2}{2n}\right)
\end{aligned}$$

Note that Approximation 2.3 is valid as long as $U_{j-1} + i \ll n$, which we expect to be the case since each U_i will be $O(\sqrt{n})$. Then we replace the sum with an integral:

$$\begin{aligned}
E[T_j] &= \sum_{t=1}^{\infty} \Pr[T_j > t] \\
&\approx \sum_{t=1}^{\infty} \exp\left(\frac{-2tU_{j-1} - t^2}{2n}\right) \\
&\approx \int_0^{\infty} \exp\left(\frac{-2tU_{j-1} - t^2}{2n}\right) dt && \text{(Approximation 2.4)} \\
&= \exp\left(\frac{U_{j-1}^2}{2n}\right) \int_0^{\infty} \exp\left(\frac{-(U_{j-1} + t)^2}{2n}\right) dt \\
&= \sqrt{2n} \exp\left(\frac{U_{j-1}^2}{2n}\right) \int_{\frac{U_{j-1}}{\sqrt{2n}}}^{\infty} e^{-x^2} dx && \left(x := \frac{U_{j-1} + t}{\sqrt{2n}}\right)
\end{aligned}$$

Now we can use integration by part, using the fact that $e^{-x^2} = \frac{-2xe^{-x^2}}{-2x}$ for

$x > 0$ and so

$$\begin{aligned}
E[T_j] &= \sqrt{2n}\exp\left(\frac{U_{j-1}^2}{2n}\right)\int_{\frac{U_{j-1}}{\sqrt{2n}}}^{\infty} e^{-x^2}dx \\
&= \sqrt{2n}\exp\left(\frac{U_{j-1}^2}{2n}\right)\int_{\frac{U_{j-1}}{\sqrt{2n}}}^{\infty} -2xe^{-x^2}\cdot\frac{1}{-2x}dx \\
&= \sqrt{2n}\exp\left(\frac{U_{j-1}^2}{2n}\right)\left(\left[\frac{-e^{-x^2}}{2x}\right]_{\frac{U_{j-1}}{\sqrt{2n}}}^{\infty} - \int_{\frac{U_{j-1}}{\sqrt{2n}}}^{\infty}\frac{e^{-x^2}}{2x^2}dx\right) \\
&= \sqrt{2n}\exp\left(\frac{U_{j-1}^2}{2n}\right)\left(\exp\left(\frac{-U_{j-1}^2}{2n}\right)\cdot\frac{\sqrt{2n}}{2U_{j-1}} - \int_{\frac{U_{j-1}}{\sqrt{2n}}}^{\infty}\frac{e^{-x^2}}{2x^2}dx\right) \\
&\approx \frac{n}{U_{j-1}},
\end{aligned}$$

where we neglect the last integral since it is very small. From this and $U_k = \sum_{i=1}^{k} T_i$ we also get the recurrence $U_k = U_{k-1} + \frac{n}{U_{k-1}}$. Now we seek to apply the Cesaro-Stoltz theorem [Mur09]. Define $V_k \triangleq \frac{U_k}{\sqrt{n}}$ so that $V_k = V_{k-1} + \frac{1}{V_{k-1}}$. Then $V_k^2 = V_{k-1}^2 + 2 + \frac{1}{V_{k-1}^2}$ so we have that $V_k^2 - V_{k-1}^2 = 2 + \frac{1}{V_{k-1}}$. It follows that

$$\lim_{k\to\infty}\frac{V_k^2 - V_{k-1}^2}{k-(k-1)} = 2\,,$$

and by Cesaro-Stoltz, this implies that $\frac{V_k^2}{k} \to 2$ and as such $V_k \sim \sqrt{2k}$ and $U_k \sim \sqrt{2kn}$. □

Let us use Theorem 3.17 to estimate the running time of Algorithm 2.

Let $f : D \to R$, and suppose we aim to find m distinct collisions. A first rough estimate is that, to find m collisions we will need $U_m \sim \sqrt{2m|R|}$ function evaluations. Note that the naive strategy of simply running the algorithm multiple times, stopping after a collision is found, and restarting from scratch would have had a complexity of $O(m \cdot \sqrt{|R|})$ to find m collisions, so this is a saving of a factor of $\sqrt{m}$ function evaluations. However, to be completely accurate we need to account for the fact that the domain points sampled might not be unique, and that the algorithm might find the same collision multiple times.

Note that Algorithm 2 essentially samples collisions uniformly at random from the set of collisions, which by Theorem 3.15 we expect to have size roughly $C_{D,R} = \frac{1}{|R|}\binom{|D|}{2}$. So, by Theorem 3.13, we expect to have to sample $m' \triangleq C_{D,R}\left(H_{C_{D,R}} - H_{C_{D,R}-m}\right)$ collisions. By Theorem 3.17 this will require $U_{m'} \sim \sqrt{2m'|R|}$ distinct evaluations of f, which, again by Theorem 3.13 will

#Collisions	Distinct	Time
$m \ll C_{D,R}$	✓	$O\left(m^{1/2}\lvert R\rvert^{1/2}\right)$
$m \approx C_{D,R}$	✓	$O\left(m^{1/2}\ln\left(\frac{C_{D,R}}{C_{D,R}-m}\right)^{1/2}\lvert R\rvert^{1/2}\right)$
$m = C_{D,R}$	✓	$O\left(m^{1/2}\ln\left(C_{D,R}\right)^{1/2}\lvert R\rvert^{1/2}\right)$
m	✗	$O\left(m^{1/2}\lvert R\rvert^{1/2}\right)$

Table 3.1: A summary of the time requirements of Algorithm 2 for a function $f : D \to R$, dependent on the number of required collisions. Distinct refers to whether we require the found collisions to be distinct. Time is measured in function evaluations.

take a total of

$$|D|\left(H_{|D|} - H_{|D|-\sqrt{2m'|R|}}\right)$$

function evaluations.

For concreteness, let us again take the case of a cryptographic hash function $f : R \to R$ with $n \triangleq |R| = 2^{128}$. Suppose that we wanted to compute $m := 2^{64}$ collisions. A first analysis based on Theorem 3.17 predicts that this will take $\approx \sqrt{2 \cdot 2^{64} \cdot 2^{128}} = 2^{96}$ function evaluations, and a refined analysis based on Remark 3.14 very closely agrees. Instead, if we set $m \approx C_R = \frac{2^{128}-1}{2}$ there is some discrepancy. The first analysis would expect $\sqrt{2 \cdot 2^{127} \cdot 2^{128}} \approx 2^{128}$ function evaluations. By the second analysis instead we have that $m' \approx 2^{133.5}$ and our total number of function evaluations is then expected to be on the order of 2^{131}, i.e. 2^4 times higher.

To summarize, Algorithm 2 can be used for solving the Problem 3.8, and with $m \ll C_{D,R}$ this takes time $O(m^{1/2}|R|^{1/2})$. If instead $m \approx C_{D,R}$ we can use the refined analysis of Theorem 3.13 and Remark 3.14 to obtain a better estimate.

As for Problem 3.9 let $g \triangleq \Pr\left[\mathrm{gold}_f(c) = 1 \middle| c \leftarrow\$ \mathrm{Coll}(f)\right]$. Then, the expected number of collisions (not necessarily distinct) that will need to be sampled to obtain a golden one will be g^{-1}, and Theorem 3.17 predicts that this will take $O(g^{-1/2}|R|^{1/2})$ function evaluations. We summarize the results in Table 3.1.

As for space requirements, Algorithm 2 stores an element of D for each function evaluation in the main datastructure, and at most $|D|$ elements in total. So the total space requirement will be (roughly) $O\left(\log\left(|D|\right)\min\left(|D|, m^{1/2}|R|^{1/2}\right)\right)$.

3.3 Pollard ρ-method [Pol75]

Pollard's ρ-method improves on the previous method by addressing the large space requirements. To do so, it assumes that the function is a self-map, i.e. it is of the form $f : S \to S$. From now on, we will let $n \triangleq |S|$. The

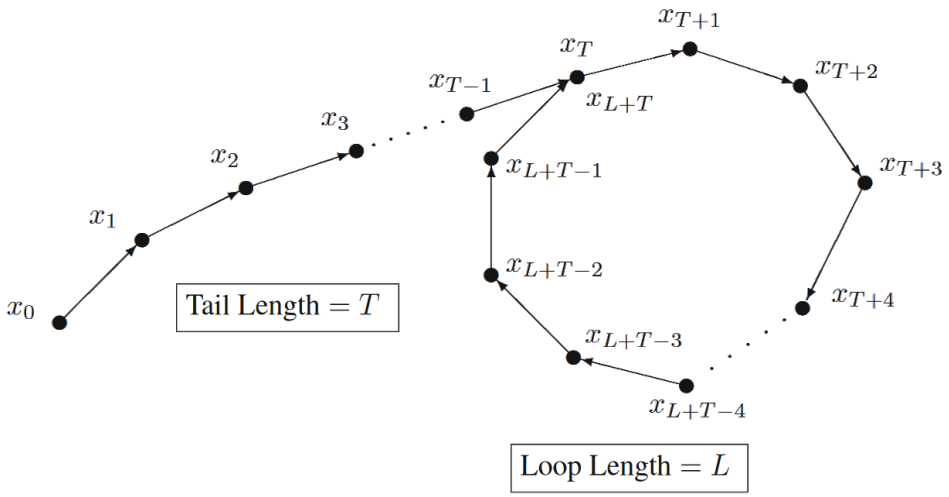


Figure 3.1: Pollard's ρ method. Illustration taken from [Sil09].

Algorithm 3: Pollard's ρ-method

Data: $f : S \to S$
Result: A collision (x, x')
$s \leftarrow\$ S$;
$t, h \triangleq s, s$;
repeat
  $t := f(t)$;
  $h := f(f(h))$;
until $t = h$;
$t := s$;
repeat
  $t', h' \triangleq t, h$;
  $t := f(t)$;
  $h := f(h)$;
until $t = h$;
return (t', h');

intuition behind Pollard's ρ-method is that, when f is a self-map, we can iterate the function from a start point and obtain a sequence $x, f(x), f^2(x), \ldots$ and so on. Since the set S is finite, this sequence will 'trace' a ρ, and the cycle can be detected by a cycle finding algorithm, leading to a collision. Different cycle finding method can be used, but for simplicity our exposition, presented in Algorithm 3, will focus on Floyd's cycle finding algorithm.

With this setup, the memory requirement is $O(1)$. Let us prove correctness.

Lemma 3.18 *Let $x_0 \in S$. Consider the sequence $x_{i+1} \triangleq f(x_i) = f^{i+1}(x_0)$ for $i \geq 0$. Then:*

- *The sequence eventually has a repeating value.*
- *Let y be the first repeating value in the sequence. Let T and C' denote, respectively, the index of the first and second occurrence of y in $x_0, x_1, \ldots$. Then,*
$$x_{T+i} = x_{T+i+kC},\ \forall i, k \in \mathbb{N}$$
where $C = C' - T$
- *There exists an index j such that $x_j = x_{2j}$*

Proof The sequence is infinite and S is finite, so by the pigeonhole principle a value must repeat, and this proves the first point. Now let y be the first appearance of the repeating value in the sequence, say that $y = x_T = x_{T+C}$. Let $i \geq 0$. By definition,

$$x_{T+i} = f^{T+i}(x_0) = f^i(f^T(x)) = f^i(y) = f^i(f^{T+C}(x_0)) = x_{T+C+i}$$

Applying this identity with $i = (k-1)C$ shows that $y = x_{T+kC}$ for $k \geq 0$ and another application yields the second point. Note that this implies that $x_{T+k} = x_{T+(k \mod C)}$. So a solution to $x_j = x_{2j}$ can be found when $j \geq T$ and $j - T \equiv 2j - T \pmod{C}$ so $j \equiv 0 \pmod{C}$ satisfies that, proving existence.□

Theorem 3.19 *Let $x_i \triangleq f^i(x)$, and let T, C be as in Lemma 3.18. Algorithm 3, started at point x_0 always terminates, and if $T \neq 0,\ C \neq 1$ finds a collision.*

Proof First of all, Lemma 3.18 directly show that the first loop of Algorithm 3 terminates. So, at the end of the first loop, we have reached a point of the sequence x_j with $j \geq T$ and $j \equiv 0 \pmod{C}$. Now, since $j \equiv 0 \pmod{C}$, we have that $x_i = x_{i+j}$ for every $i \geq T$. Note also that for $i < T$ we have necessarily that $x_i \neq x_k$ for every k, since otherwise, by definition of T, $i \geq T$. So, the second loop tests this condition and will terminate when $i = T$, at which point, if $T \neq 0$ and $C \neq 1$, the points x_{T-1} and x_{T+j-1} will yield a collision. □

Now that we have shown correctness, we can analyze the runtime of the algorithm. In fact, we will not do so with Floyd's algorithm, since in practice it does more function evaluations than what it is necessary. We first use Theorem 3.12 to estimate the expected value of $T + C$, which will be the most important quantity that cycle finding algorithms depend on.

Theorem 3.20 *Let f be a random function. The expected value of $T + C$ (as defined in Lemma 3.18) is $\sqrt{\frac{\pi n}{2}}$.*

Algorithm	Function Evals	Space
Floyd	$3T + C$	$O(1)$
[Bre80]	$2\max(T + C) + C$	$O(1)$
[SSY82]	$(T + C)\left(1 + \frac{c}{\sqrt{M}}\right)$	M
[Niv04]	$T + (1 + \alpha)C$	$O(\alpha^{-1} \log(T + C))$

Table 3.2: A rough summary of Cycle Detection methods complexity. This is ignoring the cost of finding a collision after the cycle has been detected, which is $T + C$. For [SSY82] M is a parameter regulating how much memory the algorithm can use and c a constant. For [Niv04], α can be arbitrarily reduced at the cost of increased memory

Proof Note that, until the loop closes, since f is a random function, every point on the trail is sampled uniformly at random from S. Then, Theorem 3.12 predicts that we will need $\sqrt{\frac{\pi n}{2}}$ points until duplication occurs. □

Now, there are few methods that we can employ, whose complexity, in terms of $T + C$ and memory usage, are presented in Table 3.2. Detailed analysis can be found in [Jou09].

In fact, a lower bound is know from [Fic81], in that any cycle detecting algorithm that stores at most M element of the sequence must make at least $(T + C) \cdot \left(1 + \frac{1}{M-1}\right)$ function evaluations. Since Theorem 3.20 in any case predicts that $E[T + C]$ will be $O(n^{1/2})$ the runtime of Algorithm 3 will also be $O(n^{1/2})$.

So, Algorithm 3 addresses the main drawback of Algorithm 1, that is the large memory requirement, while having comparable runtime. However, it makes tradeoffs in return. First of all, there is no obvious way to parallelize it. The repeated function evaluation is inherently serial, and as such multiple threads do not seem to bring benefits. Using Pollard to handle Problem 3.8 seems also not to be trivial, since, once a collision is found, the algorithm would continue to traverse the loop and no more useful work would be done. We could run many instances, but then the expected runtime would be $O(m \cdot n^{1/2})$, a factor of $\sqrt{m}$ worse than what Algorithm 2 would achieve (albeit using only constant memory). Also, the previous algorithm sampled collisions uniformly from the set of collisions, while Algorithm 3 might have biases.

Remark 3.21 *When m is small the tradeoff between larger runtime and storage might be desirable. For example, consider $n = 2^{128}$ and say we would like to find $m = 2^{16}$ not necessarily distinct collision. Algorithm 2 is expected to terminate after $2^{72.5}$ function evaluations and need approximately $2^{79.5}$ bits of memory. Instead, Algorithm 3 would require $2^{80.3}$ evaluations but only around 2^9 bits of storage. This might be a desirable tradeoff, altough parallelisation might skew the results towards Algorithm 2. Note that, Algorithm 3 can actually make meaningful use of $L = m$ extra processors (simply by starting a walk on each of the processors*

from points independently sampled). Instead, if $L > m$ the parallelisation will not be perfect and only likely to bring minor benefits compared to the $L = m$ case. So, in the previous example, if $L = 2^{16}$ the balance between the two methods will remain unchanged (with Algorithm 2 taking $2^{56.5}$ parallel evaluations $2^{79.5}$ bits, and Algorithm 3 taking $2^{64.3}$ evaluations and 2^{25} bits of memory). Instead if $L = 2^{24}$ Algorithm 2 gains significant ground requiring $2^{48.5}$ parallel function evaluations compared to the $2^{64.3}$ of Algorithm 3.

Instead, consider the case where we would want to sample $C_R \approx 2^{127}$ collisions. Our previous analysis suggests that the table based method takes time on the order of 2^{128} function evaluations and 2^{135} bits of space. Instead Pollard-ρ would take $2^{191.3}$ function evaluations, with negligible space.

Similarly, using Algorithm 3 to tackle Problem 3.9 has its issues. Finding many collisions is problematic for the same reasons as mentioned before, and the bias in which collisions are found can exacerbate the problem. We do not discuss this in depth, but we will see similar issues with van Oorschot and Wiener's algorithm soon, and a way to partially solve them.

3.4 van Oorschot and Wiener's Algorithm [vW94]

Let us now introduce the main topic of this work, that is the parallel golden collisions finding algorithm introduced by [vW94]. As the name suggests, this is an algorithm to tackle Problem 3.9 for $\mathcal{F} = \mathsf{Funcs}(S, S)$.

As in Algorithm 3, we are interested in the sequence $x_i \triangleq f(x_{i-1})$ obtained by repeatedly applying f to a randomly sampled starting value in S. Before, we were interested in detecting a loop in the infinite sequence, which would then with high probability lead to a collision. In this case instead, we will artificially truncate the sequence once x_i is in a subset $D \subset S$, which we will call the *distinguished points* of S. Now consider two trails $x_1, \ldots, x_d$ and $x'_1, \ldots, x'_{d'}$. If at any point $x_i = x'_j$ then, since the function is deterministic, $x_d = x_{d'}$. So, trails which share a point will end in the *same* distinguished point. This suggests to store such distinguished points, together with the starting point, in a table, indexed by that distinguished point. Then, once two trails ending in the same point are identified, we can traceback and find the shared point, which, with high probability, will yield a collision.

We now give a more formal description of the algorithm, starting by listing the various parameters that we can adjust. Some will be introduced later on, but we list them all in Table 3.3 for completeness.

We will also assume that n, w are powers of two, which makes our description easier.

First of all, we define what we mean by *distinguished points*. We select a subset $D \subset S$ such that $\theta \triangleq |D|/|S|$. We want the membership test $x \in_? D$

Parameter	Type	Description	Typical value	Origin
f	$S \to S$	Input Function	Input	
n	$\mathbb{N}$	$\lvert S \rvert$	Input	
w	$\mathbb{N}$	Memory available[6]	Input	
L	$\mathbb{N}$	Available Processors	Input	
θ	$(0,1]$	Distinguishedness probability	$2.25\sqrt{\frac{w}{n}}$	[vW94]
β	$\mathbb{R}^+$	Function Version Switching	10	[vW94]
maxlen	$\mathbb{N}$	Maximum trail length	$20 \cdot \theta^{-1}$	[vW94]
B	$\mathbb{N}$	Distinguished bound	$\theta \cdot 2^{\log(n)-\log(w)}$	[CLN+20]

Table 3.3: Inputs and Parameters for vOW

to be very efficient and elements of D to be well distributed with respect to f, so that $\Pr_{x \leftarrow \$ S}[f(x) \in D] \approx \theta$. If f is a random function, a natural candidate (and what was proposed in [vW94]) is to set D to be the elements of S whose binary representation starts with $\log \theta^{-1}$ zeros. This also allows to compress distinguished points from needing approximately $\log(n)$ bits to $\log(n) - \log \theta^{-1}$. There are better ways to implement this tough, as described in [CLN+20], but we will defer describing them until the improvement is needed. For now we abstract over this:

Definition 3.22 *Let S be a set, $f : S \to S$, $\theta \in (0,1]$. A **distinguished scheme** Dist for S consists of the following:*

- *A set* $\mathsf{Dist}.D \subseteq S$
- *A polynomial time algorithm that we denote as* $\mathsf{Dist.isDist}(\cdot)$ *for testing membership.*
- $\mathsf{Dist.compress}(\cdot)$*, a deterministic polynomial time algorithm that compresses* $x \in \mathsf{Dist}.D$ *to a binary representation*
- $\mathsf{Dist.decompress}(\cdot)$*, a deterministic polynomial time algorithm that decompresses a binary representation to an* $x \in \mathsf{Dist}.D$

We require that the scheme satisfies the following properties:

- $\mathsf{Dist.isDist}(x)$ *outputs 1 iff* $x \in \mathsf{Dist}.D$*, and 0 otherwise.*
- $\mathsf{Dist.decompress} \circ \mathsf{Dist.compress} = \mathrm{id}_{\mathsf{Dist}.D}$
- *The size* $\lvert \mathsf{Dist}.D \rvert \approx \theta$*.*

From now on, we will assume we have a distinguished scheme Dist, and abbreviated $\mathsf{Dist}.D$ to D. With this, we can define the mining operation, which is shown in Algorithm 4.

The arbitrary bound of maxlen is required in order to avoid exactly the situation that we were hoping to achieve in Algorithm 3. If the algorithm

Algorithm 4: mineDist

Data: $f : S \to S$, distinguished scheme Dist
Result: A triple $\in S \times \mathsf{Dist}.D \times [\mathsf{maxlen}]$ or an error $\perp$
$s \leftarrow_{\$} S$;
$x \triangleq f(s)$;
$c \triangleq 1$;
repeat
  $x := f(x)$;
  $c := c + 1$;
until $\mathsf{Dist.isDist}(x)$ *or* $c = \mathsf{maxlen}$;
if $c = \mathsf{maxlen}$ **then**
  return $\perp$;
end
return (s, x, c);

gets stuck in a cycle which does not contain an element of D the loop would run forever, without doing any useful work. One could consider adding a cycle detection algorithm to detect this and possibly find a collision, but we will see later that the trails are in expectation short enough for it not to be worth it.

Note that we always unconditionally evaluate f on the first sampled point. This ensures that, even if the originally sampled point is distinguished, no trails of length zero are stored, since those will not be useful.

For convenience we make the following definition:

Definition 3.23 *A triple (x_0, x_d, d) is a **vOW triple** iff $x_0 \in S$, $x_d = f^d(x_0) \in D$, $d > 0$ and for every $0 < i < d$, $f^i(x_0) \in S - D$. To any vOW triple we associate a sequence $x_i \triangleq f^i(x_0)$.*

Lemma 3.24 *The outputs of Algorithm 4, is either an error or a vOW triple.*

Proof The starting point s is sampled in S. Then, the loop iterates until the $x \in D$, or the max length is exceeded. In the second case, an error is returned. So, the loop only ends when $x \in D$ and the previous iterations must be in $S - D$. □

We now investigate the distribution of trails sampled by Algorithm 4.

Theorem 3.25 *Let $\frac{1}{\theta} \ll \sqrt{n}$. The error probability of Algorithm 4 is approximately* $\exp(-\mathsf{maxlen} \cdot \theta)$. *Let (x_0, x_d, d) be the (non error) output of Algorithm 4. Then, x_0 is uniformly distributed over S and d approximately follows a geometric distribution.*

Proof We first investigate what is the probability of an error occuring. This happens exactly when $d = \mathsf{maxlen}$. Consider the associated sequence $x_i \triangleq f^i(x_0)$ for $0 \leq i \leq k$. If $k \ll \sqrt{n}$ then, by Theorem 3.12 we expect not to have any repeated elements. The sequence then behaves like sequence of elements randomly sampled from S. Then, the length of the sequence before encountering a distinguished points (denoted as K) follows a geometric distribution, with $\Pr[K = k] = \theta(1-\theta)^{k-1}$. An error occurs exactly when $K \geq \mathsf{maxlen}$ and then:

$$\begin{aligned}
\Pr[\text{mineDist}() = \perp] &= \Pr[K \geq \mathsf{maxlen}] \\
&= \sum_{k \geq \mathsf{maxlen}} \Pr[K = k] \\
&= \left(\sum_{k=1}^{\infty} \theta(1-\theta)^{k-1} - \sum_{k=1}^{\mathsf{maxlen}-1} \theta(1-\theta)^{k-1} \right) \\
&= \theta \left(\sum_{k=0}^{\infty} (1-\theta)^{k} - \sum_{k=0}^{\mathsf{maxlen}-2} (1-\theta)^{k} \right) \\
&= \theta \left(\frac{1}{\theta} - \frac{1-(1-\theta)^{\mathsf{maxlen}-1}}{\theta} \right) && \text{Geometric Series} \\
&= (1-\theta)^{\mathsf{maxlen}-1} \\
&\approx e^{-\theta \cdot \mathsf{maxlen}} && \text{Approximation 2.3}
\end{aligned}$$

Now, let us consider the distribution of the output when the algorithm sucessfully completes. The first element is sampled uniformly at random from S. Next, as discussed beforehand, we can model d as following a truncated geometric distribution, where values $d \geq \mathsf{maxlen}$ are set to have zero probability mass. In a sense, this is also like doing replacement sampling, but discarding trails longer than maxlen. As such, the $\Pr[d = k] = \frac{\theta(1-\theta)^k}{1-\Pr[K \geq \mathsf{maxlen}]}$ which, by the previous result is approximately $\frac{(1-\theta)^k}{1-\exp(-\theta \cdot \mathsf{maxlen})}$. So, if $e^{-\theta \cdot \mathsf{maxlen}}$ is small, d is approximately geometrically distributed. □

Note that in the proof of Theorem 3.25 we rely heavily on the quantity $\exp(-\theta \cdot \mathsf{maxlen})$ to be small. We note that if $\mathsf{maxlen} = \frac{c}{\theta}$ then that value is e^{-c} and so c can be chosen to make this arbitrarily small. In [vW94] c is set to be 20, which then makes $e^{-20} \approx 2 \cdot 10^{-9}$, and the wasted work negligible.

In Algorithm 5 we describe the backtracking algorithm in the classical version described in [vW94]. [CLN+20] proposed a different backtracking algorithm, that has improved performance at the cost of storing some intermediate values.

We now prove correctness of this backtracking algorithm.

Algorithm 5: backtrack

Data: $f : S \to S$, two vOW triples $(x_0, x_d, d), (y_0, y_{d'}, d')$ with $x_d = y'_{d'}$
Result: A collision or an error
if $d > d'$ **then**
  return backtrack$(f, (y_0, y_{d'}, d'), (x_0, x_d, d))$;
end
$\delta \triangleq d' - d$;
$\ell \triangleq f^{\delta}(y_0)$;
$s \triangleq x_0$;
if $s = \ell$ **then**
  return $\perp$; `// Robinhood case`
end
for $i = 1 \ldots d$ **do**
  $\ell', s' \triangleq \ell, s$;
  $\ell := f(\ell)$;
  $s := f(s)$;
  if $\ell = s$ **then**
    return (ℓ', s');
  end
end

Lemma 3.26 *Let (x_0, x_d, d) and (y_0, y'_d, d') be two vOW triples. If $x_i = y_j$ for any i, j, then $x_{i+k} = y_{j+k}$ for $k \geq 0$. If $x_d = y'_d$ there must be a $k \geq 0$ such that $x_{d-k} = y_{d'-k}$.*

Proof For the first statement note that, for $k \geq 0$:

$$x_{i+k} = f^{i+k}(x_0) = f^k(x_i) = f^k(y_j) = f^{j+k}(y_0) = y_{j+k} \,.$$

For the second part, note that $k = 0$ works already (but might not be the largest one). □

Lemma 3.27 *Let (x_0, x_d, d) and $(y_0, y_{d'}, d')$ be two vOW triples with $x_d = y_{d'}$, and $d \leq d'$. Then Algorithm 5 running on that input outputs a collision or an error.*

Proof By Lemma 3.26 the set $\{k \geq 0 \mid x_{d-k} = y_{d'-k}\}$ is nonempty. Let k denote the maximal such k, which by definition must be $\leq d$. As in the algorithm let $\delta \triangleq d' - d \geq 0$. Then $d' - k = \delta + d - k$ so define a new sequence $z_i \triangleq y_{i+\delta}$. At $j \leq k$, then $x_{d-j} = z_{d-j}$, and so it suffices stepping x_i and z_i at unison to find the maximal k. Since k is maximal, for $k+1$ we will have that $x_{d-k-1} \neq y_{d'-k-1}$ and that will yield a collision, unless $d - k - 1 \leq -1$, which implies then that $d \leq k$ and, since $d \geq k$, this implies that $d = k$. This

then implies that $x_0 = y_{d'-k} = y_{\delta+d-k} = z_0$ and that is what we check for and return an error. □

Lemma 3.26 and Lemma 3.27 prove the correctness of Algorithm 5. What is left is then analyzing the performance of the algorithm.

Note that the number of steps is at most $\delta + 2 \cdot d = d' + d$, and since d, d' are geometrically distributed by Theorem 3.25 the expected number of steps will be at most $\frac{2}{\theta}$.

Next we define the interface to our memory.

Definition 3.28 *A **memory scheme*** Mem *consists of:*

- *A initalization algorithm* $\mathsf{Mem.Init}(w)$ *which takes a memory size w and returns an handle to memory* mem,
- *An address space* $\mathsf{mem}.A$ *and a value space* $\mathsf{mem}.T$,
- *An algorithm* $\mathsf{Mem.sendPoint}(\mathsf{mem}, a, t)$ *that takes an handle, an address $a \in \mathsf{mem}.A$ and a value* $\mathsf{mem}.T$*. It returs a new updated handle and either a value or an error $\perp$.*

Definition 3.29 *We define an addressing scheme* Addr *for a memory handle* mem *consists of:*

- *An index set* $\mathsf{Addr}.I$,
- *A deterministic algorithm* Addr.addr *that converts a value of* $\mathsf{Addr}.I$ *to an address in* $\mathsf{mem}.A$.

We defer the implementation of Definition 3.29 to later on (only mentioning that $\mathsf{Addr}.I = S$ will be sufficient for the main description), but can already present the memory scheme we will be using. Memory is modeled as a large array, as in [vW94] and [CLN+20]. Recent work in [TID21] has proposed a new datastructure that might bring performance benefits, but we present this simpler one for now, since it translates better to the distributed setting we will be seeing later on. The initalization is shown in Algorithm 6 and the point sending routine in Algorithm 7

With our main building routines in place, we can now describe the parallel golden collision search algorithm, which we do in Algorithm 8.

Algorithm 8 tries to mine a vOW triple, retrying as often as needed. Once the triple has been mined it is sent to memory, which then returns either the occupant of the cell or a distinguished error symbol. If an occupant is present then we check whether the distinguished portion of the triples match, and if so we can invoke the backtracking algorithm, which will either return a collision or an error (in the Robinhood case). Finally, we check whether this collision is in fact the golden one, and if so return.

Algorithm 6: Mem.Init

Data: How many triples to allocate $w \in \mathbb{N}$
Result: An handle to memory mem
$\mathsf{mem}.A \triangleq [w]$;
$\mathsf{mem}.T \triangleq S \times \{0,1\}^{\mathsf{comp}} \times [\mathsf{maxlen}]$;
$\mathsf{mem}.\mathsf{arr} \triangleq []$;
for $i = 0, \ldots, w$ **do**
  $\mathsf{mem}.\mathsf{arr}[i] = \perp$;
end
return mem;

Algorithm 7: Mem.sendPoint

Data: A memory handle mem, an address $a \in \mathsf{mem}.A$, a value $t \in \mathsf{mem}.T$
Result: Either the existing value $\in \mathsf{mem}.T$ or an error
$t' \triangleq \mathsf{mem}.\mathsf{arr}[a]$;
$\mathsf{mem}.\mathsf{arr}[a] := t$;
return t'

We now have enough context to introduce the missing puzzle piece. Similarly to Algorithm 3, and differently from Algorithm 2, Algorithm 8 has a bias in which collision it finds. For example, collisions after distinguished points (i.e. those of the form $a \to x \leftarrow b$ for either a or b distinguished) will be harder to find compared to those which are 'far' from a distinguished point. In order to overcome this we will need to protect against this case, as was already noted by [vW94]. This can be done using the concept of **function versioning**. In fact, this technique also allows us to adapt Algorithm 8 to functions $f : S \to R$ with $S \neq R$.

Definition 3.30 *Let $f : S \to R$ be a function. Let $g_0, g_1, \cdots \in \mathsf{Funcs}(R, S)$. We define the k-**th version of** f to be*

$$f_k \triangleq g_k \circ f$$

Lemma 3.31 *Let $f : S \to R$, Then $\mathrm{Coll}(f) \subseteq \mathrm{Coll}(f_k)$ for $k \in \mathbb{N}$*

Proof Let $(x, y) \in \mathrm{Coll}(f)$. Note that $x \neq y$ and $f(x) = f(y)$. Then $f_k(x) = g_k \circ f(x) = g_k \circ f(y) = f_k(y)$ and so $(x, y) \in \mathrm{Coll}(f_k)$. □

Lemma 3.32 *Let $f : S \to S$, $k \in \mathbb{N}$, g_k a permutation on S, and f_k as in Definition 3.30. Then $\mathrm{Coll}(f) = \mathrm{Coll}(f_k)$*

Proof $\mathrm{Coll}(f) \subseteq \mathrm{Coll}(f_k)$ by Lemma 3.31. Let $(x, y) \in \mathrm{Coll}(f_k)$. Then $g_k \circ f(x) = f_k(x) = f_k(y) = g_k \circ f(y)$ and composing with the inverse shows that $(x, y) \in \mathrm{Coll}(f)$. □

Algorithm 8: vOWVersion

Data: $f : S \to S$, golden test gold_f memory handle mem, distnguished scheme Dist, addressing scheme Addr, memory scheme Mem
Result: A golden collision or an error

```
dist ≜ 0;
while dist ≤ βw do
    r ←$ mineDist(f, Dist) ;                          // Mine points
    if r = ⊥ then
        continue;
    end
    dist := dist + 1;
    (x_0, x_d, d) ≜ r;
    a ≜ Addr.addr(x_d);
    c ≜ Dist.compress(x_d);
    (t, mem) ≜ Mem.sendPoint(mem, a, (x_0, c, d)) ;          // Store
    if t = ⊥ then
        continue;
    end
    (y_0, c', d') ≜ t;
    y_{d'} := Dist.decompress(c');
    if x_d ≠ y_{d'} then
        continue;
    end
    b ≜ backtrack(f, (x_0, x_d, d), (y_0, y_{d'}, d'));
    if b ≠ ⊥ and gold_f(b) = 1 then
        return b;
    end
end
return ⊥;
```

For Lemma 3.32, note that, in fact, composing with a permutation is useful as the resulting graph is not necessarily isomorphic to the starting graph, which follows from the fact that not all permutations are graph isomorphisms.

Remark 3.33 *While this concept of versioning is very useful there are some subtleties that are important to highlight. In particular, for* $f \leftarrow\$ \mathsf{Funcs}(S, R), g \leftarrow\$ \mathsf{Funcs}(R, S)$ *the composition* $g \circ f : S \to S$ *is not in general a random function. A easy counterexample is letting* $R = \{1\}$*, which makes the composition become a constant function. In general, the composition is only a random function when* f *is an injection. In all of our setting we will have* $|R| > |S|$ *and* f *close to an injection*

Algorithm 9: vOW

Data: $f : S \to R$
Result: A golden collision
$\mathsf{mem} \triangleq \mathsf{Mem.Init}(w)$;
for $k = 0, \ldots$ **do**
  $g_k \leftarrow\!\!\$\ \mathsf{Funcs}(R, S)$;
  $f_k := g_k \circ f$;
  $r = \text{vOWVersion}(f_k, \text{gold}_{f_k}, \mathsf{mem}, \mathsf{Dist}_k, \mathsf{Addr}_k, \mathsf{Mem})$;
  if $r \neq \perp$ **then**
    return r;
  end
end

(i.e. with only a single collision), which makes the composition approximately a random function. Because of this, we consider acceptable modelling f_k as a randomly sampled function for each k.

We can easily translate a golden collision test for f to one for f_k.

Definition 3.34 *Let $f : S \to R$ be a function, gold_f be as in Problem 3.9, and f_k as in Definition 3.30. Define $\text{gold}_{f_k} : \text{Coll}(f_k) \to \{0, 1\}$ as*

$$(x, y) \mapsto (f(x) =_? f(y)) \wedge \text{gold}_f(x, y)$$

Note $\text{gold}_{f_k}(c) = 1$ iff $\text{gold}_f(c) = 1$.

Using this, we can finally show the final algorithm, which is presented in Algorithm 9.

We also show how to implement Definition 3.22 and Definition 3.29 in a function-version aware manner, following the ideas from [CLN+20]. We will assume that f_k is close to a randomly sampled function in $\mathsf{Funcs}(S, S)$, so we can consider elements of S obtained in our iteration as random bitstrings of length $\log n$. We will take this $\log n$ bits substring and subdivide it into sections. The first will be $\log w$ bits and will be used for deciding the address, and the second will take the remaining $\log n - \log w$ bits and be used in order to check distinguishedness. We start with addressing scheme, which is shown in Algorithm 10. As mentioned before, we let $\mathsf{Addr}.I = S$.

The reason for the shift is that, between function iterations, memory is reused, and Lemma 3.27 only finds collisions w.h.p. when two trails for the same function versions are considered. Therefore, we would like to minimize the number of times when two vOW triples (x_1, x_d, d) and $(x'_1, x'_{d'}, d')$ with $x_d = x_{d'}$ but obtained with different function versions are mapped to the same address. With our addressing scheme, if the first triple is obtained

Algorithm 10: $\mathsf{Addr}_k.\mathsf{addr}$

Data: A point $x \in \mathsf{Addr}.I = S$
Result: An address $a \in \mathsf{mem}.T$
$b \triangleq \mathrm{bin}_S(x)$;
$a \triangleq \mathrm{bintoi}_{\log n}(b) + k \pmod w$;
return a;

by version k and the second by version k' the address will be the same if and only if $k \equiv k' \pmod w$, which only occurs every w function versions. If w is large enough, and functions are switched only after mining a considerable number of distinguished points, the probability that a point in version k is not overwritten after w version will be very little, as the next lemma shows.

Lemma 3.35 *Fix $a \in [w]$. Let $a_1, \ldots, a_k, \ldots$ be a sequence of integers, sampled uniformly and independently from $[w]$. The probability that the sequence $a_1, \ldots, a_k$ does not contain a is*

$$\left(1 - \frac{1}{w}\right)^k \approx \exp\left(-\frac{k}{w}\right)$$

In fact, when $k = w \cdot \beta w$ this probability is $\exp(-\beta w)$.

Proof The probability that $a_i = a$ is exactly $1 - \frac{1}{w}$ for every i, and each a_i is sampled independently and uniformly. Then we use Approximation 2.3. □

If we model our address function as a random function (which in general it will approximate, since the $\log w$ lower bits of a distinguished point are random and independently from the distinguished property[7]) then the remark at the end of Lemma 3.35 estimates the probability of the unwanted case to occur, which is negligible if $\beta \cdot w$ is large.

With this addressing scheme, we are essentially guaranteed that points in memory from previous functions version will not lead Algorithm 8 to wasteful backtracking and thus allows us to reuse the same memory across all function versions without having to clear it explicitly. When w is large, or in the distributed setting, this can be a significant advantage since the cost of clearing memory can be considerable.

Next we look at an implementation for Definition 3.22. We let $B \triangleq \theta \frac{n}{w}$, and $\mathsf{comp} \triangleq \log w + \lfloor \log B \rfloor + 1$. Our specification are shown in Algorithm 11, Algorithm 12, Algorithm 13.

As mentioned before, this amounts to taking the highest $\log n - \log w$ bits of the bit representation of an element of S, and then testing whether that

[7] The addresses in fact will not be independent, which will be explored more in Section 4.3, but this rough approximation suffices here

Algorithm 11: Dist_k.isDist

Data: A point $x \in S$
Result: $x \in_? \mathsf{Dist}_k.D$
$b \triangleq \mathrm{bin}_S(x)$;
$i \triangleq \lfloor \mathrm{bintoi}_{\log n}(b)/w \rfloor$;
return $i + k \cdot B \leq B \pmod{\frac{n}{w}}$;

Algorithm 12: Dist_k.compress

Data: $x \in \mathsf{Dist}_k.D$
Result: A string $\{0,1\}^{\mathsf{comp}}$
$s \triangleq \mathsf{comp} - \log w$;
$b \triangleq \mathrm{bin}_S(x)$;
$i \triangleq \mathrm{bintoi}_{\log n}(b)$;
$l \triangleq i \pmod{w}$;
$u \triangleq \lfloor i/w \rfloor + k \cdot B$;
$u' = u \pmod{2^s}$;
return $\mathrm{itobin}_s(u') || \mathrm{itobin}_{\log w}(l)$;

Algorithm 13: Dist_k.decompress

Data: A string $\{0,1\}^{\mathsf{comp}}$
Result: $x \in \mathsf{Dist}_k.D$
$s \triangleq \mathsf{comp} - \log w$;
$u' || l' \triangleq c$;
$u \triangleq \mathrm{bintoi}_s(u') - k \cdot B \pmod{\frac{n}{w}}$;
$l \triangleq \mathrm{bintoi}_{\log w}(l')$;
$x \triangleq \mathrm{bin}_S^{-1}(u \cdot 2^w + l)$;
return x;

value, incremented by $k \cdot B$, where k is the function version, is less than the specifed bound. As noted in [CLN+20], adding $k \cdot B$ makes any point in S distinguished every B function versions, and heuristically reduces every instance to the average case.

As for compression, if a point is distinguished then, by definition, $i + k \cdot B \leq B \pmod{\frac{n}{w}}$ then $i + kB$ can be stored in at most the same number of bits needed to store B, namely $\lfloor \log B \rfloor + 1$. Since we also need to store the $\log w$ address bits, a compressed point will have exactly comp points.

Remark 3.36 *The previous algorithms have assumed that the elements of S are uniform bitstrings. If S and f have some more structure the addressing and distin-*

guishedness will need to be adjusted accordingly in order to keep the nice properties we seek.

Chapter 4

Memory Filling & Function Versions

In this chapter we set out to answer some theoretical questions on the behaviour of memory in a run of vOW's algorithm. We define formally the question, and propose a number of models to answer them, verifying the models experimentally in the process. Furthermore, we also investigate whether the choice of β in [vW94] is optimal and conclude negatively, proposing a new heuristic selection that yields better concrete efficiency.

4.1 Experimental Setup

In order to validates our models and observations, we performed a number of simulations, via a script provided in `python/new_advanced_model.py`. Given input n, w, θ and a number of repetitions, the simulation will iteratively sample distinguished points using Algorithm 4, using SHAKE128 [Dwo15] as f. For each sampled distinguished point, we kept track of the current number of distinguished points in memory, and exported this as our experimental data.

4.2 Memory filling

In this section we aim to answer the following question:

Question 4.1 *How many distinguished points does Algorithm 8 need to compute before the entire memory is full?*

We start by giving a motivation on why we are interested in the problem. As mentioned in the description of Algorithm 9 and Algorithm 10 each function version effectively starts with an empty memory, and progressively applies Algorithm 4 to produce distinguished points, which fill the memory gradually. Obtaining a more careful estimate of the number of distinguished points needed to fill the memory is helpful in both figuring out the optimal

value for β (i.e. when to switch function versions), and as an intermediate step in the computation of the runtime in [TID21]. Historically, difficulty in estimating this quantity was also source of the 'flawed' analysis in [vW94]. We show a number of different models, starting with most unrealistic one, which is implicitly assumed in [TID21].

Model 4.2 (Linear) *Letting w, θ be as in Table 3.3, the number of distinguished points required to fill the memory is w.*

This model is a clear lower bound, since it is impossible to fill a memory of size w without storing at least w points. However, it completely ignores memory collisions, which in practice have a rather large effect. In fact, if we model each distinguished point as a randomly sampled point in D, and the address Addr.addr as a random function $D \to [w]$ for $|D| \gg w$ we can use Theorem 3.13 to obtain a more accurate model.

Model 4.3 (Coupon Collector) *Letting w, θ be as in Table 3.3, the number of distinguished points required to fill the memory is $wH_w \approx w \ln(w) + \gamma w$.*

While, as we will see later, Model 4.3 is already an improvement compared to Model 4.2 in our experiments, there is still room for improvement. Consider a random function f. Before any query is made, each value $f(x)$ for $x \in S$ can be seen as being distributed uniformly and independently in S. Once a query is made tough, the value is fixed for any following query. In particular, consider a sequence $x_1 \to f(x_1) \to \cdots \to f^d(x_1) = x_d$. Each following query of f at $x_1, \ldots, x_{d-1}$ will in fact lead back to x_d and as such lead to a memory collision.

Motivated from this observation, let us develop a more refined model to account for this fact. In fact, the following discussion will result in three further models, each harder to compute but more accurate.

In all of the three cases we follow the approach of Theorem 3.13. For $i > 1$ we let T_i be the numbers of distinguished points required to fill i memory cells after $i-1$ cells had been filled. T_1 instead is the number of distinguished points required to fill the first cell, which is exactly 1. We then let $U_k = \sum_{i=1}^{k} T_i$. The value that we are seeking is then U_w. By linearity of expectation, $E[U_w] = \sum_{i=1}^{w} E[T_i]$. So, by estimating $E[T_i]$ for every $1 \leq i \leq w$ we can get an answer to our problem. What makes the problem challenging is that the T_i do not only depend on the number of points in memory (as was the case in Theorem 3.13) but also on the number of points sampled so far i.e. U_{i-1} and in fact on the current sampling trial that we are on. This is because each new point sampled will determine the functions f and Addr.addr completely on the trail and the resulting distinguished point, and any new sampled trails that touches one of these points will yield a memory collision. Our three models will progressively 'book-keep' more, in practice improving in accuracy while becoming harder to compute.

First of all, let us fix some terminology. At each point in time, the **state** of the system consists of the set of points on which f or $\mathsf{Addr.addr}$ have been evaluated. Any point that is not in the state will be **fresh**. Each time Algorithm 8 does one iteration, a new trail is sampled by Algorithm 4, and $\mathsf{Addr.addr}$ is then evaluated on the final point, so if (x_0, x_d, d) is the (non-error) output of Algorithm 4 the state is augmented by the elements $\{x_0, \ldots, x_d\}$. Suppose the state S consists of P points, and let us sample a vOW triple (x_0, x_d, d). What is the probability that x_d is fresh?

$$\begin{aligned}
\Pr[x_d \notin \mathsf{S}] &= \underbrace{\Pr[x_d \notin \mathsf{S} | x_{d-1} \in \mathsf{S}]}_{=0} \Pr[x_{d-1} \in \mathsf{S}] + \Pr[x_d \notin \mathsf{S} | x_{d-1} \notin \mathsf{S}] \Pr[x_{d-1} \notin \mathsf{S}] \\
&\approx \left(1 - \frac{P}{n}\right) \Pr[x_{d-1} \notin S] \\
&= \left(1 - \frac{P}{n}\right)^{d+1} \\
&\approx \exp\left(-\frac{Pd}{n}\right)
\end{aligned}$$

Where in the first line we have used the fact that a non fresh non distinguished point will lead to a non fresh point by Lemma 3.26[1]. By Theorem 3.25 we also expect $d \approx \frac{1}{\theta}$, and, if p distinguished points have been collected so far, the number of points in the state is expected to be $P \approx \frac{p}{\theta}$. By this, the probability that a new trail is fresh when p distinguished points have been sampled is approximately $\exp\left(-\frac{p}{n\theta^2}\right)$. Now, what we are really interested in is whether the address of the newly sampled distinguished point is fresh. Suppose that $i-1$ distinct addresses have been sampled so far, and let us denote this set as A. By an entirely similar reasoning as before

$$\Pr[\mathsf{Addr.addr}(x_d) \notin \mathsf{A}] = \left(1 - \frac{i-1}{w}\right) \Pr[x_d \notin \mathsf{S}]$$

With this observation, let us develop our models. For the first model, we will assume that, when $i-1$ points are stored in memory, only the trails from which those points are originated are counted in the state, i.e. $p = i-1$. So, $p_i = \Pr[\mathsf{Addr.addr}(x_d) \notin \mathsf{A}] = \left(1 - \frac{i-1}{w}\right) \exp\left(-\frac{i-1}{n\theta^2}\right)$. As in the proof of Theorem 3.13 we can model then the probability of an address being new as a geometric distribution with expectation $1/p_i$. So with this $E[T_i] = \frac{1}{p_i}$ and we get the following model.

[1] In fact, here we also assumed that, of the P state points, approximately θP will be distinguished

Model 4.4 (Sum Model) *Letting n, w, θ be as in Table 3.3, the number of distinguished points expected to fill the memory is approximately*

$$\sum_{k=1}^{w} \left(\exp\left(-\frac{k-1}{n\theta^2} \right) \left(1 - \frac{k-1}{w} \right) \right)^{-1}$$

For the next refinement, we instead make the more realistic observation that when $i-1$ points are in memory the state will consists of the trails for the previous U_{i-1} points, and so $P \approx U_{i-1}/\theta$. As such, the probability that an address is new will be

$$p_i = \exp\left(-\frac{U_{i-1}}{n\theta^2} \right) \cdot \left(1 - \frac{k-1}{w} \right) .$$

In this model the probability is independent from trials, and as such we can use the same geometric reasoning to conclude that the $E[T_i] = \frac{1}{p_i}$. Note however that p_i depends on U_{i-1}. To estimate this, we can compute $E[U_{i-1}]$ and recurse until the base case $E[U_1] = E[T_1] = 1$. This leads us to the next model.

Model 4.5 (Recurrence Model) *Letting n, w, θ be as in Table 3.3, we let $\mu_{T_1} = 1$, and*

$$\mu_{T_k} \triangleq \left(\exp\left(-\frac{\mu_{U_{k-1}}}{n\theta^2} \right) \cdot \left(1 - \frac{k-1}{w} \right) \right)^{-1} \text{ for } k > 1$$

with $\mu_{U_k} \triangleq \sum_{i=1}^{k} \mu_{T_i}$. The expected number of distinguished points to fill the memory is μ_{U_w}.

A further refinement can be obtained by addressing the last sampling inaccuracy that we have allowed. In Model 4.5 we have a probability of an address being fresh p_i that depends only on U_{i-1}. So, we could model the distribution as geometric. However in fact the probability will not be the same for each of the samples, since the previous samples will also go and effect the state. A more accurate model would let $p_{i,j}$ be the probability that the j-th sample is fresh after $i-1$ points have been stored in memory. Using our estimate with $P \approx \frac{U_{i-1}+(j-1)}{\theta}$ we can then conclude that

$$p_{i,j} = \exp\left(-\frac{U_{i-1} + (j-1)}{n\theta^2} \right) \left(1 - \frac{i-1}{w} \right)$$

Due to this dependence we cannot model the distribution as geometric anymore. What can be done is rewriting $E[T_k] = \sum_{i=1}^{\infty} \Pr[T_k > i]$ and noting that $\Pr[T_k > i] = \prod_{j=1}^{i} (1 - p_{k,j})$ to obtain Model 4.6.

Model 4.6 *Letting n, w, θ be as in Table 3.3, we let $\mu_{T_1} = 1$, and*

$$\mu_{T_k} \triangleq \sum_{i=1}^{\infty} \prod_{j=1}^{i} \left(1 - \exp\left(-\frac{\mu_{U_{k-1}} + j - 1}{n\theta^2}\right)\left(1 - \frac{k-1}{w}\right)\right)$$

with $\mu_{U_k} \triangleq \sum_{i=1}^{k} \mu_{T_i}$. The expected number of distinguished points to fill the memory is μ_{U_w}.

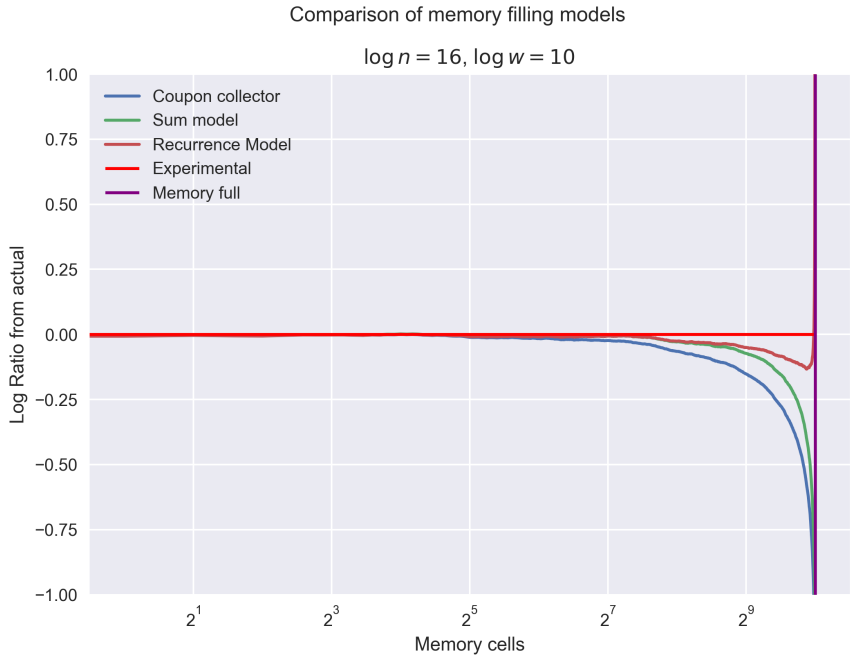
Remark 4.7 *Model 4.2 and Model 4.3 are very easy to compute, as they take $O(1)$ floating point operations. Instead, Model 4.4 and Model 4.5 are much more computationally expensive, taking $O(w)$ operations to compute naively. Model 4.6 is even more complex, and finding a closed formula or approximation is left as future work.*

In order to validate our models, we have run a number of experiments. Note, first and foremost, that our models do not only make a prediction on the number of points required to completely fill the memory, but also can be adapted to predict the number of distinguished points sampled before a certain number of cells have been filled. This adaptation, in the case of Model 4.3, can be done by using Remark 3.14, and in the case of Model 4.4 and Model 4.5 by simply taking μ_{U_k} with k is the number of memory cells to be filled. In order to understand where our models have predictive power and where they fail, we have plotted that estimate against the experimentally measured average. In this analysis, and in general in most of our figures we have plotted the ratio between model prediction and actual measurement, so we would like our graphs to be as close to a ratio of 1 as possible.

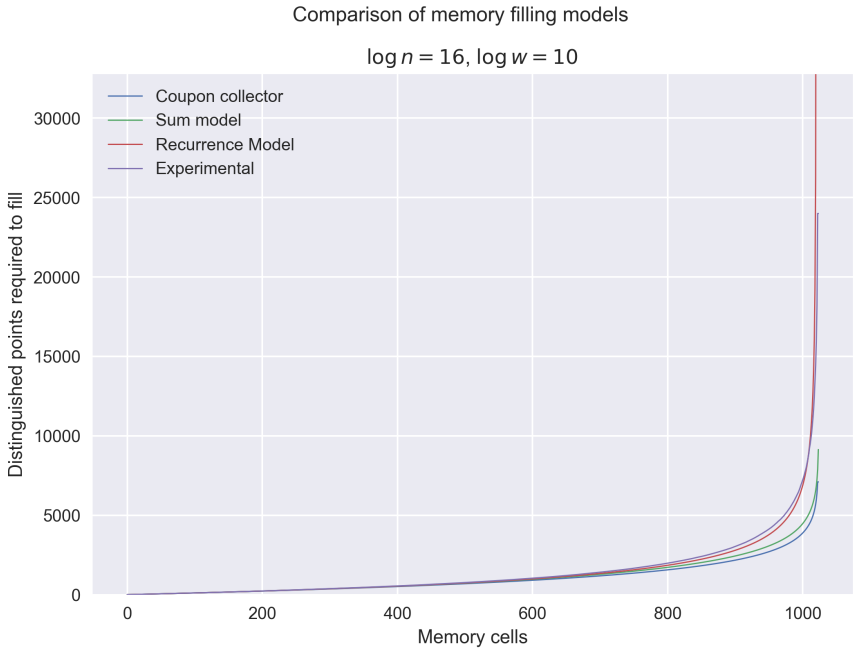
As Figures 4.1 to 4.3 show, Model 4.3 is the worst performing model, systematically underestimating the number of points required to fill the corresponding memory cells. Model 4.4 is a significant improvement, but the best model is really Model 4.5, which agrees with the experiment extremely closely. Note, however, that all three models systematically fail when the number of filled cells $k \approx w$. In such case Model 4.3 and Model 4.4 undershoot and Model 4.5 overshoots the experimental values.

Remark 4.8 *To the best of our knowledge, Model 4.4, Model 4.5 and Model 4.6 are new and an improvemnt over the state of the art. However, there are still clear areas of improvement, that we list here for future reference.*

- *The length of the trails is not always $\frac{1}{\theta}$. This can have some effect since, for example, longer trails in the function graph will be sampled with an higher probability and thus lead to more memory collisions than expected.*
- *In Model 4.4 we were helped by the fact that the trails for the $i-1$ points in memory must have been distinct by Lemma 3.26. In the following models however we neglected the fact that the U_{i-1} sampled trails will definitely be intersecting, and as such the number of points will be most likely less than the $\frac{U_{i-1}}{\theta}$ that we would be expecting otherwise.*

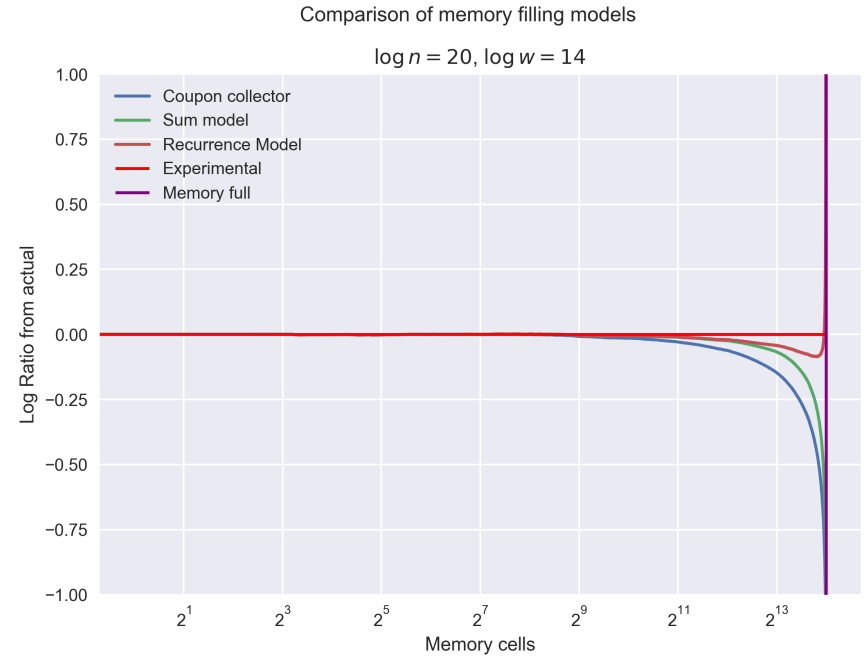


(a) Ratio to expected value (log-log)

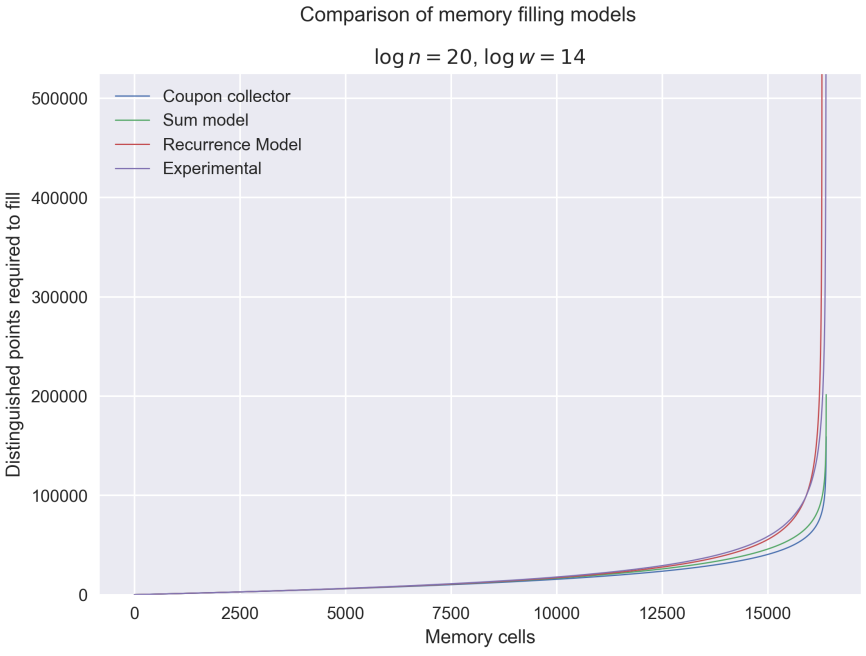


(b) Raw comparison

Figure 4.1: 'Coupon Collector' refers to Model 4.3 and 'Sum model' to Model 4.4 and 'Recurrence Model' to Model 4.5. Experiment averaged over 100 runs. Parameters: $n = 2^{16}, w = 2^{10}, \theta = 0.2813$.

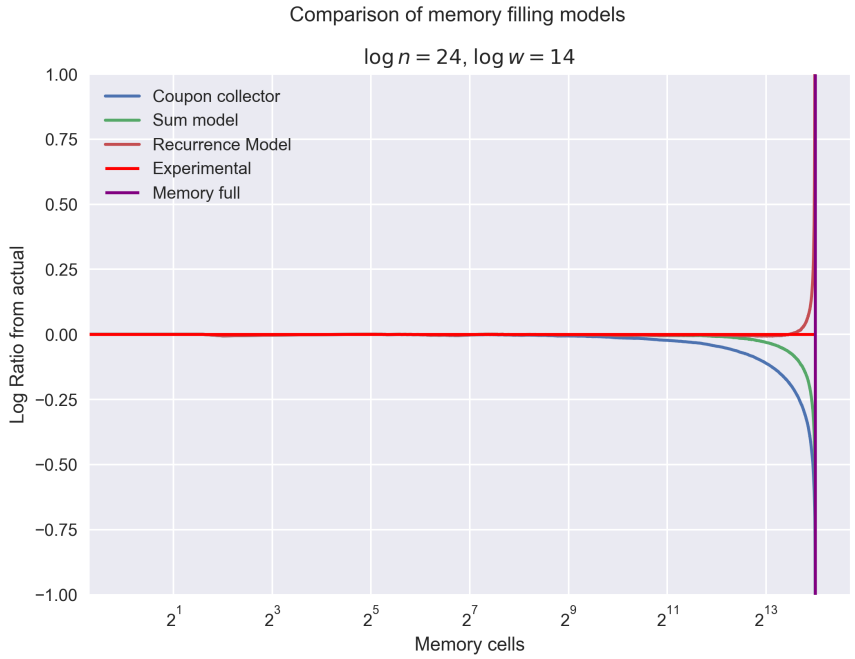


(a) Ratio to expected value (log-log)

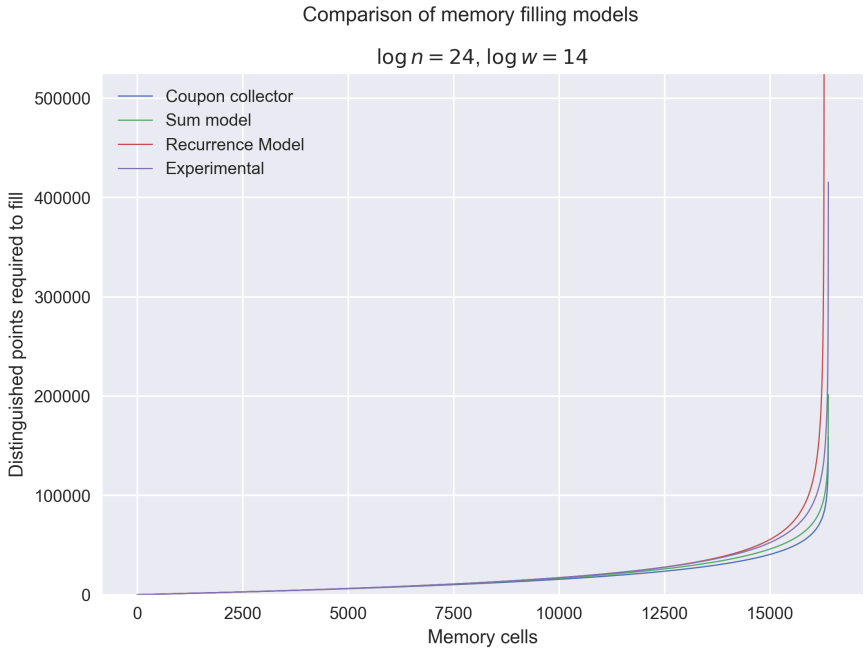


(b) Raw comparison

Figure 4.2: 'Coupon Collector' refers to Model 4.3 and 'Sum model' to Model 4.4 and 'Recurrence Model' to Model 4.5. Experiment averaged over 50 runs. Parameters: $n = 2^{20}, w = 2^{14}, \theta = 0.2813$.



(a) Ratio to expected value (log-log)



(b) Raw comparison

Figure 4.3: 'Coupon Collector' refers to Model 4.3 and 'Sum model' to Model 4.4 and 'Recurrence Model' to Model 4.5. Experiment averaged over 50 runs. Parameters: $n = 2^{24}, w = 2^{14}, \theta = 0.0703$.

4.3 Fill rate

In this section we aim to answer the dual to Question 4.1.

Question 4.9 *How full will the memory be when Algorithm 8 has computed k distinguished points?*

Remark 4.10 *Why is this question of interest? Well consider one run of Algorithm 8, in which a total of k distinguished points are mined. As we will see in Section 5.3, a rather important question to answer is how many collisions will be detected in those k points. An answer to Question 4.9 directly translates to an answer to this arguably more important question. Let us assume that U_i is a model that predicts the number of filled memory cells when i distinguished points have been mined. Define the indicator variable I_i as follows:*

$$I_i \triangleq \begin{cases} 1, \textit{if a collision is detected when the i-th distinguished point is mined} \\ 0, \textit{otherwise} \end{cases}$$

Note that the expected number of detected collisions (C say) is exactly $C = \sum_{i=1}^{k} I_i$. As such:

$$E[C] = \sum_{i=1}^{k} E[I_i]$$

And since I_i is binary $E[I_i] = \Pr[I_i = 1]$. Now, note that when the i-th distinguished point is sampled, the memory is expected to contain U_{i-1} distinguished points, each with an associated trail of expected length $1/\theta$. The only way for I_i to equal 0 is if all the $1/\theta$ points in the trail sampled are not in the set of these associated trails, which happens with probability $\left(1 - \frac{U_{i-1}}{\theta n}\right)^{\frac{1}{\theta}}$. As such, the total we get that

$$\begin{aligned} E[C] &= \sum_{i=1}^{k} 1 - \left(1 - \frac{U_{i-1}}{\theta n}\right)^{\frac{1}{\theta}} \\ &\approx \sum_{i=1}^{k} 1 - \left(1 - \frac{U_{i-1}}{\theta^2 n}\right) \\ &= \sum_{i=1}^{k} \frac{U_{i-1}}{\theta^2 n} \\ &= \frac{1}{\theta^2 n} \sum_{i=1}^{k} U_{i-1} \end{aligned}$$

Note here that the only simplification in this model is that the length of the trails is θ^{-1}, since they are guaranteed to be disjoint since they have different memory addresses. The approximation is only valid when $U_i \ll \theta n$ which is what we expect

with typical parameters since by definition $U_i \leq w$. As such, we do expect that a good model U_i that answers the fill rate question would directly yield a good model for predicting the number of detected collisions by Algorithm 8.

As before, we will build up to our final model incrementally, starting from the equivalent of Model 4.3.

Theorem 4.11 *Let $a_1, \ldots, a_k$ be a sequence of values, sampled uniformly and independently from $[w]$. The expected size of $\{a_i\}_{i=1}^k$ is*

$$w \cdot \left(1 - \left(1 - \frac{1}{w}\right)^k\right) \approx w \cdot \left(1 - \exp\left(\frac{-k}{w}\right)\right)$$

Proof Define a random variable I_i, for $1 \leq i \leq w$ as:

$$I_i = \begin{cases} 1 \text{ if, for some } j,\ a_j = i \\ 0 \text{ otherwise} \end{cases}$$

Note that the expected size of the set $\{a_j\}_{j=1}^k$, which we denote as U, is exactly $\sum_{i=1}^w I_i$, and so by linearity of expectation $E[U] = \sum_{i=1}^w E[I_i] = wE[I_1]$, where the last equality comes from the fact the distribution is uniform over $[w]$. Now, since I_1 is binary, we have that $E[I_1] = 1 - \Pr[I_1 = 0]$ and the probability that $I_1 = 0$ is $\left(1 - \frac{1}{w}\right)^k$ by Lemma 3.35. □

From Theorem 4.11 we get then the following model

Model 4.12 *Let w be as in Table 3.3. After k distinguished points have been sampled the fill rate of the memory is*

$$1 - \left(1 - \frac{1}{w}\right)^k$$

To approach the vOW case we follow most of the setup of Theorem 4.11, except for the very last step. As in Model 4.6 the state of the system is effected by the previous trials, so, for example, if for the first $x_1, \ldots, x_{k-1}$ distinguished points a certain address has not been hit the probability that the x_k point hits it will be lower than what predicted by Theorem 4.11 since there will be $k-1$ trails pointing to other memory addresses. We formalise this intuition as follows.

We let U, I_1 be as before, and again look at $\Pr[I_1 = 0]$. Let $x_1, \ldots, x_k$ be the sequence distinguished points sampled so far, and let S_j be the set of points in the trails leading to $x_1, \ldots, x_j$. In the interest of space, we let $a \triangleq \mathsf{Addr.addr}$ and also write E_j for the event that, for $i = 1, \ldots j$, it holds that $a(x_i) \neq 1$.

Effectively, E_j is the event that none of the distinguished points $x_1, \ldots, x_j$ is mapped to the memory address 1. Note that

$$\begin{aligned}\Pr[I_1 = 0] &= \Pr[E_k] \\ &= \Pr[a(x_k) \neq 1 | E_{k-1}] \Pr[E_{k-1}] \\ &= p_k \cdot \Pr[E_{k-1}] \\ &= \prod_{j=1}^{k} p_j\end{aligned}$$

where $p_j \triangleq \Pr[a(x_j) \neq 1 | E_{j-1}]$. Note that

$$\begin{aligned}p_j &= \Pr[a(x_j) \neq 1 | E_{j-1}] \\ &= \Pr[a(x_j) \neq 1 | E_{j-1}, x_j \in \mathsf{S}_{j-1}] \Pr[x_j \in \mathsf{S}_{j-1}] + \Pr[a(x_j) \neq 1 | E_{j-1}, x_j \notin \mathsf{S}_{j-1}] \Pr[x_j \notin \mathsf{S}_{j-1}] \\ &= \Pr[x_j \in \mathsf{S}_{j-1}] + \left(1 - \frac{1}{w}\right) \Pr[x_j \notin \mathsf{S}_{j-1}]\end{aligned}$$

Where the last simplication is given by the fact that if a distinguished point is not fresh, and all the previous distinguished point did not map to 1, then that point will not map to 1 as well. Now, if we assume that the set of trails induced by the first j distinguished points do not overlap and have expected length $1/\theta$ as predicted by Theorem 3.25, we can say that $|\mathsf{S}_j| \approx \frac{j}{\theta}$. By an observation entirely analogous to that in the derivation of Model 4.4 we can then say that $\Pr[x_j \notin \mathsf{S}_{j-1}] \approx (1 - \frac{j-1}{\theta n})^{1/\theta} \approx 1 - \frac{j-1}{\theta^2 n}$. This tells us that

$$\begin{aligned}p_j &\approx \frac{j-1}{\theta^2 n} + \left(1 - \frac{1}{w}\right)\left(1 - \frac{j-1}{\theta^2 n}\right) \\ &= 1 - \left(\frac{\theta^2 n - (j-1)}{\theta^2 n w}\right) \\ &\approx \exp\left(-\frac{\theta^2 n - (j-1)}{\theta^2 n w}\right)\end{aligned}$$

This leads to conclude that

$$\begin{aligned}\Pr[I_1 = 0] &= \prod_{j=1}^{k} p_j \\ &\approx \prod_{j=1}^{k} \exp\left(-\frac{\theta^2 n - (j-1)}{nw\theta^2}\right) \\ &= \exp\left(\sum_{j=1}^{k} -\frac{\theta^2 n - (j-1)}{nw\theta^2}\right) \\ &= \exp\left(\frac{-1}{\theta^2 nw}\sum_{j=1}^{k}\left(\theta^2 n - (j-1)\right)\right) \\ &\approx \exp\left(\frac{k^2 - 2kn\theta^2}{2nw\theta^2}\right)\end{aligned}$$

And as such we can conclude that

$$E[U] \approx w\left(1 - \exp\left(\frac{k^2 - 2kn\theta^2}{2nw\theta^2}\right)\right)$$

Note however, that the quadratic polynomial inside the exponential has a minima at $k = n\theta^2$, so after that point the model would predict that the number of points stored in memory will start to decrease, which of course is nonsensical. This leads us to formulate Model 4.13, with that restriction on its predictive power.

Model 4.13 *Let n, w, θ be as in Table 3.3. After $k < n\theta^2$ distinguished points have been sampled the fill rate of the memory will be*

$$1 - \exp\left(\frac{k^2 - 2kn\theta^2}{2nw\theta^2}\right)$$

We have run a number of experiments to verify the accurateness of the models. In Figure 4.4 we report our measurements.

Note that, for $k \ll n\theta^2$ Model 4.13 predicts the fill rate very accurately, always within 10% of the experimental value. When k increases, the simpler Model 4.12 catches up and eventually outperforms the new model. When far away from the critical point, not only is the average experimental value very close to the predicted one, but, as the box plots in Figure 4.5 show, is very consistently close to it.

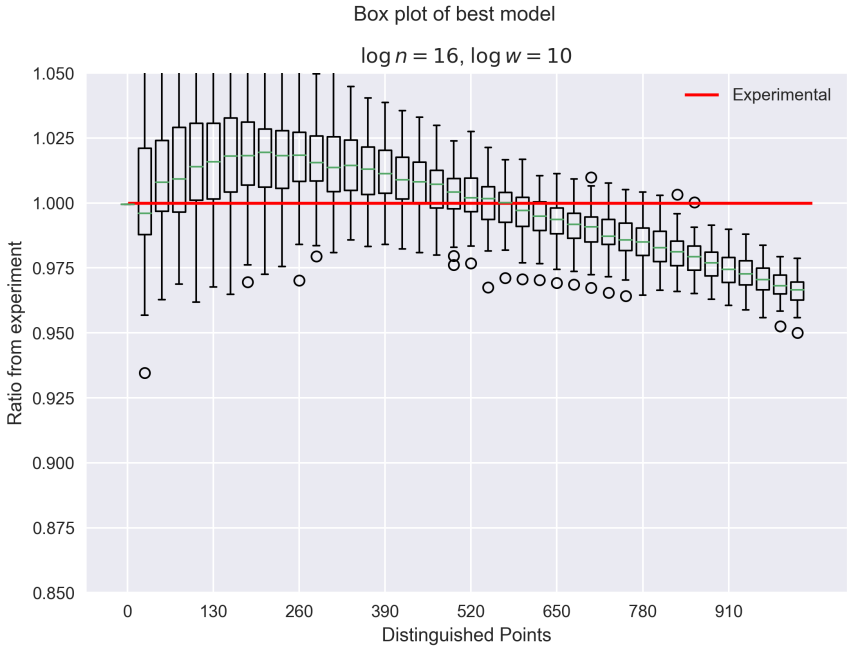
Remark 4.14 *The same points raised in Remark 4.8 apply without fail to this section. An open path towards improving the modelling can be either accounting for the fact that not all trails have in fact length $1/\theta$ and/or by considering the fact that the aforementioned trails will be likely to intersect when many distinguished points have been sampled.*

(a) $n = 2^{16}, w = 2^{10}, \theta = 0.28125$

(b) $n = 2^{20}, w = 2^{14}, \theta = 0.28125$

(c) $n = 2^{24}, w = 2^{10}, \theta = 0.0703$

Figure 4.4: Ratio of prediction of Model 4.12 and Model 4.4 to actual fill rate. 'Ignoring collision' refers to Model 4.12 and 'Accounting for collisions' to Model 4.13. Actual is averaged over 100 runs. x-axis cut off at $n\theta^2$. Log-Log scale.



(a) $n = 2^{16}, w = 2^{10}, \theta = 0.28125$

(b) $n = 2^{20}, w = 2^{14}, \theta = 0.28125$

(c) $n = 2^{24}, w = 2^{10}, \theta = 0.0703$

Figure 4.5: Boxed ratio of prediction of Model 4.4 to actual fill rate. Measurements are taken over 50 runs. Note the changed scale. Parameters: $n = 2^{16}, w = 2^{10}, \theta = 0.28125$. x-axis cut off at $n\theta^2$

4.4 Updating Function Versions

The next step in this investigation can be seen as motivated from Model 4.3. The model predicts that, in order to completely fill a memory of size w, we will be requiring around $w \ln(w)$ distinguished points. As Figures 4.1 to 4.3 show, the model predicts substantially fewer points than what is truly required, and as such can be considered a lower bound. In Algorithm 8, memory is shared between function version, but our definition of Addr.addr allows us to consider the simpler case in which memory is reset on every function version switch, which happens after every βw distinguished points have been mined.

In [vW94], it was determined, by testing with $\log(n) = 32$ and $\log(w) = 16$, that the optimal value of β is 10. This value of β was then used in subsequent work, such as [CLN+20]. The observation of Model 4.3 suggests that this might not be the optimal setting of β. For any w such that $\ln(w) > 10$ (note the changed base of the logarithm), the model predicts that a single function version will not fill the memory completely. While it is not immediately obvious that this is suboptimal, intuitively this would mean at least a portion of the resources used to run Algorithm 9 would not be utilised fully, and as such that there could be room for improvement. This suggest the conjecture that choosing $\beta \approx \ln(w)$ would translate to better practical runtimes, by ensuring that the memory has a chance to fill completely.

We have performed experiments to validate this conjecture. Our methodology was a following. We selected two instance sizes, namely $n = 2^{24}, 2^{32}$. For each of these sizes, we tested a number of different memory sizes w. For each pair of n, w, we then selected a range of β, including the original $\beta = 10$ from [vW94] and the values $\lfloor \ln(w) \rceil - 3, \ldots, \lfloor \ln(w) \rceil + 3$. For each triple of parameters (n, w, β) we run Algorithm 9 to completion, using the generic implementation from [CLN+20]. Each attack is run 50 times, and the results are then averaged. The results are shown in Table 4.1.

Note that Table 4.1 does not directly show that β_{opt} is always $\ln(w)$, but does suggest that the two quantities are closely related, and that choosing a β closer to that value can yield significantly better practical runtimes than what [vW94]'s method yielded. That suggests that additional effort towards identifying what the optimum value of β can be worthwhile in order to reduce the cost of running the attack.

Question 4.15 *Given n, w as in Table 3.3, what is the optimal β that minimizes the runtime of Algorithm 9?*

We do not fully answer this question, and leave it for future work, but we mention a few factors that we believe are important. The main reason that switching function versions is required at all is that Algorithm 9 does

$\log n$	$\log w$	$\ln(w)$	β_{opt}	$\frac{\beta=10}{\beta_{opt}}$
24	12	8.3	9	1.40
24	13	9.0	8	1.02
24	14	9.7	10	1.00
24	16	11.0	10	1.00
24	17	11.7	11	1.50
24	19	13.1	15	1.82
24	20	13.8	14	1.52
32	22	15.2	13	1.29
32	23	15.9	16	1.05
32	26	18.0	16	1.26

Table 4.1: β_{opt} is the optimal β according to our experiments (by number of function evaluations). The ratio is the number of steps required to find the golden collision when $\beta = 10$ compared to when $\beta = \beta_{opt}$. Averaged over 50 runs.

not sample collisions uniformly at random as Algorithm 1 does, but instead has biases towards certain collisions. Since, for a given function version, the golden collision might be very unlikely to be found, we then apply the versioning techniques, which effectively randomize the graph of the function. A smaller β allows us to perform this randomization step more often. However, randomizing too often is counter-productive, since (as mentioned in the description of Algorithm 10) switching version effectively clears the memory, and Algorithm 8 tends to find more collisions when the memory is close to full. Quantifying this tradeoff would be a possible path towards determining the optimal β. In particular, in Section 5.3 we propose a blueprint to estimate the cost of running Algorithm 9 in terms of β, w, and once such a model is developed this could allows us to answer Question 4.15.

Chapter 5

Fine-grained cost analysis vOW

In our exposition of Algorithm 9, we purposefully did not present an analysis of the running cost of the attack. The purpose of this chapter is to present the two models to estimate the theoretical cost of the attack, and to provide a more fine-grained analysis to note the practical cost of running the algorithm. This fine-grained analysis has then been verified via a number of experiments.

In the sequel, we will use $n, w, \theta, \beta, S, f$ as in Table 3.3. We will also consider the setting of Problem 3.9 in which there is a single golden collision in $\mathrm{Coll}(f)$.

5.1 [vW94] Model

Our first analysis is the one that was presented in the original [vW94] paper.

Their analysis (as they point out), is simple, but flawed. They start by assuming that the memory is full with w distinguished points. Each of these distinguished points will lie in a trail, which, by Theorem 3.25, is expected to have length approximately θ^{-1}. When sampling a new trail by Algorithm 4, a collision will be detected exactly when this newly sampled trail contains any of the $\frac{w}{\theta}$ points. Consider each invocation of f as a sampling a new point in S. Then, the probability that a newly point sampled uniformly at random generates a collision is the probability that this point is one of the $\frac{w}{\theta}$ trail points, i.e. it is $\frac{w}{\theta n}$. As such, modelling as a geometric distribution, each collision will require $\frac{n\theta}{w}$ function evaluations to be detected. Then, the cost for backtracking (Algorithm 5) is $\frac{2}{\theta}$ function evaluations and as such the total per collision cost is $\frac{n\theta}{w} + \frac{2}{\theta}$. This quantity is minimized if $\theta = \sqrt{\frac{2w}{n}}$, and this, applied to the $C_{S,S} \approx \frac{n}{2}$ collisions that we are expected to have to find by Theorem 3.15, yields a final estimate of $\sqrt{\frac{2n^3}{w}}$ function iterations.

The first, and most glaring, flaw is that the memory in a run of Algorithm 8 starts as empty and not as full. In fact, as our discussion in Chapter 4 mentioned, with $\beta = 10$ and a large enough w the memory will almost never be completely full. More subtle issues also play a role: collisions are not all equally likely to be found (as discussed with Algorithm 3 and in Definition 3.30), and as such simply sampling $C_{S,S}$ collisions might not in fact find the golden collision.

In order to examine the true performance of the algorithm, [vW94] run several experiment, using function versioning, and developed Model 5.1.

Model 5.1 ([vW94]) *Let n, w be as in Table 3.3. Let $\theta = 2.25\sqrt{\frac{w}{n}}$ and $\beta = 10$. Then, the number of function iterations required to find the golden collision can be slightly overestimated as*

$$2.5\sqrt{\frac{n^3}{w}}$$

One of the drawback of this model is the fact that it is heuristic only, even tough in practice it is quite accurate. It also is not as granular as we would like it to be, since for example it does not mentions how much of the runtime will be used in computing points and how much in locating collisions. Heuristic measurements (mentioned in [vW94] and [CLN+20]) show that around 80% of the running time is spent mining points and 20% in locating collisions.

5.2 [TID21] Model

Recently, [TID21] proposed a new model that aims to remove some of the heuristics in Model 5.1. The core of their model is an analysis entirely similar[1] to that in Theorem 3.17. In our derivation, we let U_k be the number of function evaluations required to find k collisions, and determined that $U_k \sim \sqrt{2kn}$. In their analysis, they instead let S_k the number of distinguished points required to find k collisions in Algorithm 8, while not being limited by memory, and in their analysis find that $S_k \sim \theta\sqrt{2kn}$. From that, they set $S_k = w$ and solve to find how many collisions will have been found when the memory fills up, and find this to be $\frac{w^2}{2\theta^2 n}$. As such, when we are trying to compute m collision, $C_{S,S} - \frac{w^2}{2\theta^2 n}$ will have to be found when the memory is full, and for each of these the cost will $\frac{n\theta}{w}$ as in [vW94]'s analysis. To actually find those m collisions then, we will have to bactkrack which costs $\frac{2m}{\theta}$ function evaluations. This results in model Model 5.2.

[1] Since our analysis is effectively based on theirs

Model 5.2 ([TID21]) *Let n, w, θ be as in Table 3.3. The number of function evaluations that are required to find m collisions is*

$$\frac{w}{\theta} + \left(m - \frac{w^2}{2\theta^2 n}\right)\frac{n\theta}{w} + \frac{2m}{\theta}$$

In particular, in the context of Problem 3.9, the number of function evaluations to find the golden collisions will be

$$\frac{w}{\theta} + \frac{n\theta^2 - w^2}{2\theta w} + \frac{n}{\theta}$$

And choosing $\theta = \frac{\sqrt{w^2+2nw}}{n}$ the final number of function evaluations is

$$\frac{n(4n+3w-1)}{2\sqrt{w(2n+w)}}$$

While this model is in many way an improvement compared to the almost entirely heuristic analysis of Model 5.1, it has a few shortcomings of note. First of all, setting $S_k = w$ to estimate the number of collisions found when the memory full is effectively akin to using Model 4.2 and, as seen in Chapter 4, in practice filling the memory requires many more distinguished points than that. In fact, Model 4.12 predicts that at that point only a $1 - e^{-1} \approx 0.63$ fraction of the memory will be occupied, and Model 4.13 (for typical choices of θ) reduces this to ≈ 0.59.

We also found a small algebra mistake in their derivation. In particular, they derive the recurrence $U_k = U_{k-1} + \frac{\theta^2 n}{LU_{k-1}}$ and then substitute $V_k = \frac{LU_k}{\sqrt{n\theta}}$ to obtain the recurrence $V_k = V_{k-1} + \frac{1}{V_{k-1}}$. However, in fact with that substition the result would be $V_k = V_{k-1} + \frac{L}{V_{k-1}}$, and the correct substition to obtain the recurrence would be $V_k = \frac{\sqrt{L}U_k}{\sqrt{n\theta}}$. As such, following their line of reasoning the corrected value for U_k would then be $\frac{\theta\sqrt{2kn}}{\sqrt{L}}$, which leads to an apparent contradiction, since then the number of distinguished points for m collisions would become $\theta\sqrt{2mnL}$ and this should not depend on L (which is the number of threads).

5.3 Towards a complete model

What both Model 5.1 and Model 5.2 do not capture fully is the effect of function versioning. Let us take Model 5.2 as an example for this discussion. Their model essentially divides the execution of Algorithm 8 into two parts: before w distinguished points have been gathered and after. So, in that analysis, once the memory has been filled, it remains full until all m required collisions have been found. In fact, as the pseudocode of Algorithm 9

shows, in the full attack we will switch function version after every βw distinguished points have been mined, and (as our remark in Algorithm 10 mentioned) this effectively resets the memory to be empty. What we propose instead is the following approach.

We make the reasonable assumption (justified by our implementation of Algorithms 10 and 11 and by Definition 3.30) that each run of Algorithm 8 will be independent from the other runs. In each of these runs, the algorithm will mine βw points, and in the process detects a number of collisions. If we can derive a model to predict the number of *detected* (not necessarily distinct) collisions from sampling βw distinguished points (denote these as C), then the number of required function versions to detect the golden collision will be $V \triangleq \frac{n}{2C}$. The number of function evaluations required for each of these function versions will be $\frac{\beta w}{\theta}$ to mine the required points, and $\frac{2C}{\theta}$ for the backtracking required to locate the collisions, and as such the final cost of the attack, in terms of function evaluations, would be

$$\frac{n}{2C} \cdot \left(\frac{\beta w}{\theta} + \frac{2C}{\theta} \right) = \frac{n\beta w}{2\theta C} + \frac{n}{\theta}$$

Note that, by Remark 4.10 we expect $E[C] \approx \frac{1}{n\theta^2} \sum_{i=1}^{\beta w} U_i$ for some unspecified model U_i that answers Question 4.9. Letting $S_{\theta,\beta} = \sum_{i=1}^{\beta w} U_i$, this yields a final model of

$$\frac{n^2 w \beta \theta}{2 S_{\theta,\beta}} + \frac{n}{\theta}$$

Optimizing then β, θ should allow us to find the optimal cost of Algorithm 9 in terms of function evaluations.

Unfortunately, due to time requirements, we have not been able to plug Model 4.12 and Model 4.13 into this framework to confirm the accuracy of the model, but we leave this for future work. Ideally, we envision that when starting with an appropriate memory rate model this section would provide a blueprint to estimate runtimes, and when instantiated with $\beta = 10$ recover the asymptotics and the leading constant of 2.5 that was observed heuristically in [vW94].

5.4 Modelling Practical Costs

In practice, when running Algorithm 9, function evaluations do not tell the whole story. While they are an important metric, and in practice often a very considerable portion of the runtime, practical implementations have to deal

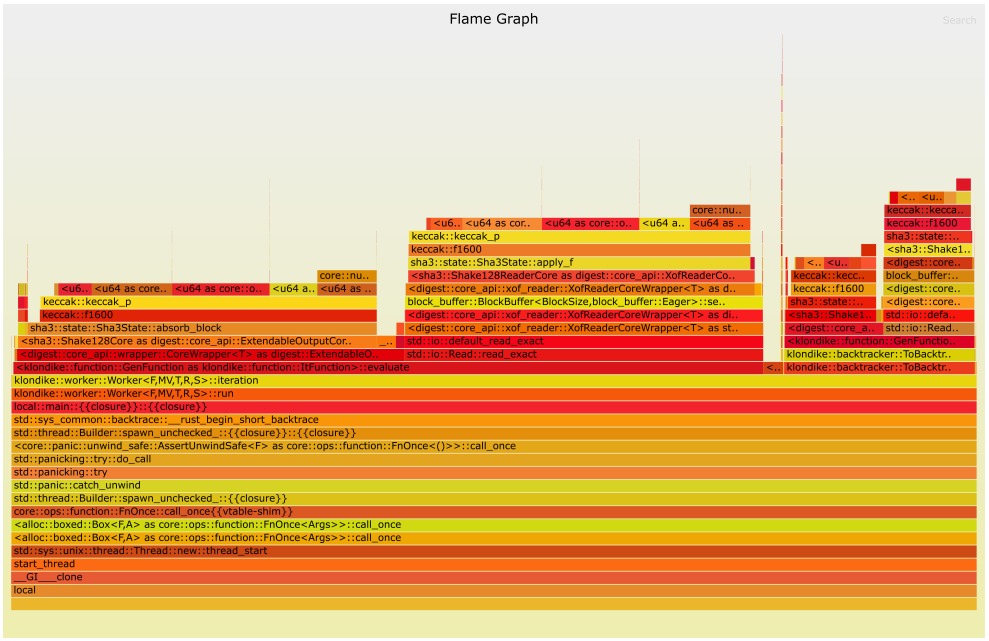


Figure 5.1: Flamegraph of a local execution of Klondike. Parameters $n = 2^{32}, w = 2^{24}$. Run with 40 cores. Algorithm 4 is responsible of 77.6% of the runtime, Algorithm 5 of 19.7%, and Algorithm 7 of 1.96%

with a much wider set of possibly costly operations, and that can elevate the cost of the attack considerably. When scaling the attack to cryptographically sized instances these costs can also increase and come to dominate the overall cost of Algorithm 9.

Instrumentation of our practical implementation (shown in Figure 5.1), run in a local setting and with small parameters, show that most of the algorithm runtime can be explained as part of three main parts.

The first, and bigger portion, which in our measurements account for around 75% of the runtime, is Algorithm 4. As Theorem 3.25 shows, this is expected to require θ^{-1} function evaluations, and[2] each evalution will mine a distinguished point.

The second fragment comes for the backtracking in Algorithm 5, which accounts for approximately 20% of the runtime (as was also noted in [vW94]). The runtime of the backtracking algorithm is then expected to be $2\theta^{-1}$ function evaluations.

The previous two parts have been already considered vastly in the literature, what we aim to focus on is the third and smallest chunk, which is caused by Algorithm 7. In our experiments, this accounts for a relatively

[2]Modulo the small error probability

Quantity	Model 5.1	Model 5.2
P_c	$2\sqrt{\frac{n^3}{w}}$	$\frac{w}{\theta} + \left(m - \frac{w^2}{2\theta^2 n}\right)\frac{n\theta}{w}$
P_s	$5.625n$	θP_c
P_b	$0.5\sqrt{\frac{n^3}{w}}$	$\frac{n}{\theta}$

Figure 5.2: Values of P_c, P_s, P_b predicted by Model 5.1 and Model 5.2

small portion of the runtime, approximately 2%, which raises the question of why do we exactly care at all? The cost of Algorithm 7 is caused almost completely by the cost of memory accesses, and while this is almost negligible in the case of a small local attack, when the scale of the instance scales to cryptanalytic sizes this cost can come to dominate the attack. The situation is only exacerbated in the setting of a distributed attack, but we will be looking at that later in Chapter 6.

Our aim is also to make this fine grained analysis to be modular and independent from the underlying model that predicts the asymptotic cost of the attack in terms of function evaluations. To that end, we will denote by P_c the number of function evaluations during a run of Algorithm 9, by P_s the number of vOW triples stored, and by P_b the number of function evaluations during backtracking. Some values predicted from Model 5.1Model 5.2 are shown in Figure 5.2.

Note that $P_s = \theta P_c$, since each point computed from mining in Algorithm 8 will then be stored immediately.

With this, we can develop a fine grained model for the cost of running Algorithm 9. Until now, we have deliberately not discussed the impact of multithreading in detail, but one of the main advantages of Algorithm 9 over Algorithm 3 is that the function evaluations coming from P_c, P_b can be perfectly parallelized. There are some details in practice on how this parallelization should be implemented, but we refer to Section 6.2.1 for a summary. Whether the cost of P_s can be effectively parallelized is instead a very delicate point.

Most commonly used consumer hardware follows the von Neumann architecture, in which (roughly) central processing units access a shared large memory via a bus. The speed at which data can travel trough this bus is a bottleneck, which cannot be circumvented by just increasing the number of cores. If the computing part of the system produces enough triples to saturate the bandwidth, then the cost of storing P_s of them will become a mostly serial operation. If instead the computing end produces relatively few triples per unit time, the write to memory will be effectively parallel. As such, we propose two models, for each of these two situations. Choosing

which of the two models will be more accurate is complicated, as very much reliant on the available hardware.

Model 5.3 (Parallel Storing) *Let t_c be the time for a single evalution of f, and t_s be the time required for storing a triple in memory. Let P_c, P_s, P_b be as in Figure 5.2 and L be the number of processing unit. The total time to run an instance of Algorithm 9 in practice will be*

$$\frac{1}{L}\left((P_c + P_b)t_c + P_s t_s\right)$$

Model 5.4 (Serial Storing) *Let t_c be the time for a single evalution of f, and t_s be the time for storing a triple in memory. Let P_c, P_s, P_b be as in Figure 5.2 and L be the number of processing unit. The total time to run an instance of Algorithm 9 in practice will be*

$$(P_c + P_b)\frac{t_c}{L} + P_s t_s$$

Remark 5.5 *In our later estimates for Model 5.3 we will use the latency of writing to main memory as our main metric for t_s, and bandwidth for Model 5.4. In both cases, we do not believe that this strategy captures the full cost of the attack, but should be seen as providing rough lower bounds.*

Remark 5.6 *In modelling the cost of memory access, we only consider the cost of writing to main memory. This choice is motivated by two factors. First of all, memory addresses in $[w]$ are accessed uniformly at random, which is essentially the worst case scenario for microarchitectural optimizations such as caches. In typical consumer hardware, this pattern of memory accesses will cause a large number of cache faults and essentially make the overall operation bottlenecked on the cost of accessing main memory. An other option would be to consider disk storage, as that would allow to store a larger number of triples. While we have not investigated this fully, we remark that latency for SSDs is three orders of magnitude higher than that for writing to RAM, and SSDs are mostly optimized for sequential operations, which is far from our use case. Furthermore, hard storage tend to be specialized for a small number of input-output operations of large size, rather than our use-case in which we have many operations, each reading and writing a small amount of information.*

In both the case of Model 5.3 and Model 5.4, the choice of θ should be informed by the the actual values of t_c, t_s. In particular, using Model 5.4 as an illustration, we note that the total time cost of the attack is

$$C(\theta) = (P_c(\theta) + P_b(\theta))t_c + P_s(\theta)t_s$$

and, as such, minimizing this quantity is equivalent to finding the zeros of $C'(\theta)$, which are dependent on the actual costs. In practice, what this amounts to is choosing smaller θs when the cost of writing to memory are large compared to the costs of computation.

Server Name	Processor	Cores	Memory
Choripan	Intel(R) Core(TM) i5-9500 CPU @ 3.00GHz	6	8 GB
Iwo	Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz	8	32 GB
Daisen	Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz	112	377 GB
Alishan	AMD EPYC 7742 64-Core Processor @ 2.25GHz	128	504 GB
Holzfusion	Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz	96	755 GB
Euler	Varied	$\gg 256$	$\gg 1$ TB

Figure 5.3: Servers we utilized for the experiments. Cores refer to physical and virtual cores. Euler is ETH's supercomputing cluster, whose exact number of cores and available memory is ever-changing.

5.5 Experiments

In order to validate Model 5.3 and Model 5.4, we performed a series of experiments. The methodology was the following. We took as a starting point the C++ `vOW4SIKE` library developed in [CLN+20], and modified the interface to memory in order to conform to the interface of Algorithm 19. Then, we introduced arbitrary slowdowns in either the evaluation of f or in Mem.sendPoint to control t_c, t_s. We then run the modified code with a range of parameters and function versions. Finally, we compare the wall time and/or the cyles to what the model predicts.

5.5.1 Servers

We tested our models on a number of different servers. Their capabilities, in terms of cores and main memory sizes, are summarized in Section 5.5.1. Access to Choripan, Iwo, Daisen, Alishan and Euler was obtained thanks to ETH, while Holzfusion by gentle concession of Royal Holloway.

5.5.2 Busy Waiting vs Sleeping

In order to obtain a fine grained control towards the cost of various operations, we had to develop a reliable way to introduce arbitrary delays in a computation. There are two general ways to obtain this: sleep delays and busy waiting. Sleep delays, or just "sleeps" are system calls that suspend the execution of the current thread and cede control to the operating system. Busy waiting is instead the process of doing arbitrary wasteful computation in order to waste CPU cycles. Sleep delays are the natural first tool to reach for, since they allow for fine grained temporal control, to the level of hundreds of nanoseconds, and can be compared directly to the wall time of process. However, the standard POSIX implementations (`nanosleep` and `clock_nanosleep`) do only give guarantees that the process will be suspended for at least as long as it is specified, and the scheduler can arbitrarily

```c
#include <time.h>

#define ITERS 100000
#define MICROSECONDS 10

int main() {
    struct timespec ts;
    ts.tv_sec = 0;
    ts.tv_nsec = MICROSECONDS * 1000;

    for (int i = 0; i < ITERS; i++) {
        nanosleep(&ts, NULL);
    }
}
```

Figure 5.4: A sleep program that sleeps more than what it should

```c
void busy_wait(uint64_t iterations) {
    for (uint64_t i = 0; i < iterations; i++) {
        __asm__ volatile("" : "+g" (i) : :);
    }
}
```

Figure 5.5: An useless program which the compiler thinks is useful

decide how much extra time to allocate. For short intervals the effect is exacerbated since the scheduler will always suspend the thread for a minimal time (in the order of magnitude of 60μs).

As an example, consider the program in Figure 5.4. Compiled with GCC 12.1.0 with `-O3` this program takes approximately 6s to execute, while we would expect it to take $10 \cdot 10^5 \mu s = 1s$. Increasing the microseconds by a factor of 10 and decreasing the iterations accordingly reduces the elapsed time to 1.5s, and higher delays also close the gap. This creates a conflict, since we would like to have the highest precision possible, but also have as short delays as possible to make sure that the experiments do not take excessive time.

Busy waiting instead faces different challenges. First of all, compilers tend to be very good at optimising away useless code, and our code is utterly useless. This can be circumvented by appropriate uses of inline assembly instructions. The function we used for busy-waiting is shown in Figure 5.5 There are some problems with this approach. First of all, on different architectures and machines the number of cycles that the loop takes to execute

can vary wildly. In order to measure that, we can run benchmarks at the start of the execution, which we can then base our analysis on. Unfortunately, that is not the end of the story, since modern processors can under and overclock depending on the machine load. That can also skew the measurements, since the load the process is under at the start of the run (when the benchmarks are being run) and in the middle of the execution can be very different and as such our initial readings be inaccurate.

On machines that we fully control (over/under)-clocking can be disabled which results in very accurate measurements. Yet on others servers (such as the Euler cluster) it is very hard to reach the level of control needed to get accurate readings. On the contrary, sleeping techniques tend to be uneffected from machine load, and as such for shared servers they are ideal. To obviate the sleep timing inconsistency we can measure the time spent sleeping and tally that up at the end of the execution, using this tally *a posteriori* to deduce the time spent per sleep call. In order to validate our models then, we decided to use both approches, busy waiting on the Holzfusion server and sleeping on Euler.

5.5.3 Results

We have run two sets of experiments, one on Holzfusion and one on Euler, using the techniques detailed in the previous section. We have collected the results using the busy-wait strategy in Table 5.1 and the ones using the sleep techniques in Table 5.2.

$\log(n)$	$\log(w)$	Ratio
24	17	1.07
28	20	1.01
30	20	1.01

(a) Evaluation slowed by $1.2 \cdot 10^6$ cycles.

$\log(n)$	$\log(w)$	Ratio
24	17	1.10
28	20	1.05
30	20	1.04

(b) Evaluation slowed by $1.2 \cdot 10^6$ cycles for half of the cores, by $3.2 \cdot 10^6$ for the other half.

$\log(n)$	$\log(w)$	Ratio
24	17	1.08
28	20	1.05
30	20	1.04

(c) Evaluation slowed by $1.5 \cdot 10^5$ cycles for half of the cores, $3 \cdot 10^5$ for the other half. Memory accesses slowed down by $1.5 \cdot 10^5$ cycles.

Table 5.1: Ratio between the prediction of Model 5.3 applied to Algorithm 9 with busy-waiting and the measured cycles. Averaged over 100 function versions. Experiment on Holzfusion.

$\log(n)$	$\log(w)$	Ratio
20	10	1.00
22	14	1.00
26	16	1.00
28	18	1.01
30	20	1.01

(a) Evaluation slowed by $100\mu s$.

$\log(n)$	$\log(w)$	Ratio
20	10	1.00
22	14	1.00
26	16	1.00
28	18	1.01
30	20	1.01

(b) Evaluation slowed by $100\mu s$ for half of the cores, by $300\mu s$ for other half.

Table 5.2: Ratio between the predictions of Model 5.3 applied to Algorithm 9 with sleeps and to measured wall-time. Averaged over 100 function versions. Experiment on Euler.

The full data collected and the scripts used to run these experiments are available in our fork of `vOW4SIKE` at `github.com/WizardOfMenlo/vOW4SIKE`. As the table shows, the models that we have developed accurately predict the experimental data.

Chapter 6

Distributing vOW & Klondike

In the previous chapters, we have considered Algorithm 9 in a mostly local setting, and developed some cost models accordingly. In a real instance of the attack, especially one of considerable size, we believe that an attacker might be inclined in 'pooling together' computation from different pre-existing machines, rather than spending the considerable resources to build a new single large machine. In this section, we will be investigating what would be additional costs, difficulties and tradeoffs that would appear when running Algorithm 9 in a distributed setting.

6.1 Revised Model

Let us start by revising our models Model 5.3 and Model 5.4 for this distributed setting. In the local setting, we have a single machine with L computing units and which was responsible for the entire memory space w. For that machine, the cost of computing a single iteration of f was t_c and the cost of storing a point to memory t_s. In the distributed setting, we will have k nodes, each of which with L_i computing units (that we assume equally powerful) and memory for w_i triples. As such, $w = \sum_{i=1}^{k} w_i$. We let $t_{c,i}$ be the time that a computing unit on node i requires to compute f, and $t_{s,i\to j}$ the time required to store a triple generated on node i in node j's memory. Note in particular that $t_{s,i\to i}$ is the cost to store a triple in local memory on node i.

We let $L \triangleq \sum_{i=1}^{k} \frac{L_i}{t_{c,i}}$. Note that $\frac{1}{t_{c,i}}$ is the number of function evaluations computed by an unit of node i per unit time, so L is exactly the number of points computed per unit time by the ensemble of the nodes.

As such, computing the P_c required function evaluations will take time P_c/L, and this will be cost of function evaluations for the system.

Now, if f is a random function, a node i will be tasked to store a share of $\frac{w_i}{w}$ of the computed points, and an according share of the backtracking points. As such, each node i will be computing $\frac{w_i P_b}{w}$ backtracking evaluations, each of which at cost $t_{c,i}$ shared across L_i threads. As such, the cost of the backtracking will be exactly $\sum_{i=1}^{k} \frac{P_b w_i t_{c,i}}{w L_i}$.

Finally, for the cost of storing. Each node i will produce points to store in accordance to their performance, so write $S_i \triangleq \frac{L_i}{t_{c,i} L}$ for the share of points node i will compute. The number of points each node will have to store will be $S_i P_s$, and of these a w_j/w fraction will be stored in node j (including $j = 1$). As such, the total number of points that will have to be stored from node i to j will be $P_s S_i \frac{w_i}{w}$. What will be the cost of this? As before, it depends on whether storing to memory (in this case in a distributed setting!) can be parallelized. Compared to before, the situation is even more complicated, since for some connections $i \to j$ it might be that there is enough bandwidth for the storing to appear parallel, while for others it might be sequential, or it might be something in between that. As such, we will introduce a parallelization factor $1 \leq \pi_{i \to j} \leq L_i$ to account for this. For example, the local connection $i \to i$ is unlikely to be bandwidth limited, and as such we expect $\pi_{i \to i}$ to be closer to L_i. Instead, a remote network connection might quickly reach its capacity, and as such the parallelization factor will be closer to 1. In practice, determining the correct value for the parallelization factor is complicated, since it will both depend on the cost of function evaluations, the value θ, the state of the network and many other factors. As such, it should be carefully estimated before the start of the attack.

Model 6.1 *Let L_i, w_i be, respectively, the number of computing units and memory size of node i, $w = \sum_{i=1}^{k} w_i$ where k is the number of nodes. Let $t_{c,i}$ be the time for a single evalution of f on node i, $t_{s,i \to j}$ be the time for storing a triple from node i to node j. Let $\pi_{i \to j}$ be the parallelization factor of the connection from $i \to j$.*

Let P_c, P_s, P_b be as in Figure 5.2.

Let $L \triangleq \sum_{i=1}^{k} \frac{L_i}{t_{c,i}}$, $S_i \triangleq \frac{L_i}{t_{c,i} L}$. The total time to run a distributed instance of Algorithm 9 in practice will be

$$\frac{P_c}{L} + \sum_{i=1}^{k} \sum_{j=1}^{k} \frac{S_i P_s w_j t_{s,i \to j}}{w \pi_{i \to j}} + \sum_{i=1}^{k} \frac{P_b w_i t_{c,i}}{w L_i}$$

6.2 Klondike

In order to validate our model, and as part of our contributions, we developed **Klondike**, a library and a set of binaries for executing Algorithm 9 efficiently and in a distributed setting. This section will be focusing on describing Klondike, and on a more theoretical side the issue of syncronization

between workers in Algorithm 9, which is exacerbated in a distributed setting.

6.2.1 Synchronization

One of the most important improvements of Algorithm 9 over Algorithm 3 is the fact that it claims to be perfectly parallelizable. Looking at the body of the loop in Algorithm 8, we can see that the mining routine (Algorithm 4) and the backtracking routine (Algorithm 5) can be executed in parallel by many workers threads with no dependencies on each other. While memory would need to be shared between workers, if its size is large enough compared to the number of workers, as is expected to be the case in practice, the probability that two writes to the same memory cell occur concurrently becomes very small. The remaining shared data between workers is given by the distinguished point counter, which is used to switch function version.

As a reminder, when a certain number of points have been mined (namely $\beta \cdot w$) a new function version should be selected. What is crucial is that all the workers always are *synced*, i.e. they should all be computing on the same function version. That is because points mined using different function version, as mentioned in Algorithm 10, are not useful. Consider, for example, two threads minining with function version $k \neq k'$. Each distinguished point generated by version k and stored in memory is seen as an empty memory cell by the worker on version k', which will then overwrite it.

As such, our synchronization strategy should, as a priority, ensure that different workers switch at unison. Switching when precisely βw points have been mined is desirable, but not fundamental. For Klondike, we have implemented two syncing strategies from [CLN+20], namely 'No Biggie' and 'Stakhanov'. Our local implementation is general and allows to specify the desired syncing strategy.

In both cases, each worker keeps track of the number of distinguished point it has mined using the current function versions. The worker counter is periodically synced with a global counter, using atomic instructions to avoid locking and at the same time ensuring consistency between threads. Each worker has a local copy of the function version, which is what is required to be updated as function versions are changed. We also designate one of the local workers as the leader, which will be responsible for keeping the global state consistent.

No Biggie The general strategy of 'No Biggie Sync' is the following. After every worker mines a new distinguished point, it syncs the local statistics to the global one, and checks whether that value is higher than the threshold. If so, it waits (using a barrier) until every threads also realizes that. Then, the

Algorithm 14: NoBiggieSync

Data: nbState, localDist the local number of distinguished points mined since last sync.
Result: $\perp$ or a new function version
nbState.dist = nbState.dist + localDist ; `// Atomically`
if nbState.dist $\leq \beta \cdot w$ **then**
  return $\perp$;
end
B_1: Wait for every thread;
if *Worker is leader* **then**
  nbState.ver = nbState.ver + 1;
end
B_2: Wait for every thread;
return nbState.ver;

leader updates the function version in a global location, which then every worker checks and moves their local function version to this new version. A more accurate specification is shown in Algorithm 14.

A real implementation of this algorithm is slightly more complicated, as for example it needs to check if any worker has found the golden collision, and can be found in `sync/nobiggie.rs`.

Algorithm 14 can be called either at the end of every loop of Algorithm 8 or every few iterations, and a choice should be informed by benchmarks and the cost of atomic operations on the architecture of choice (See [SBH20] for an overview). In our case, we note that each iteration takes θ^{-1} function evaluations and a write to memory (plus possibly a backtracking step), and that should be much more expensive than the single atomic operation that is executed unconditionally on a call to Algorithm 14. Thus, in our implementation we make the call to syncing after every iteration. In general, this syncing strategy gives very strong guarantees. Supposing that the algorithm is called every s iterations, and that there are L workers in the system, Algorithm 14 guarantees that no more than $\beta w + sL$ distinguished points are mined, and that no two workers will be mining points with different function versions. The drawback is that waiting workers do not do any useful work, so, especially when some of the workers are slow, this might waste valuable time.

Stakhanov 'Stakhanov sync' addresses the drawback of Algorithm 14 by letting workers work harder than what they have to. Each worker is given a share of the βw points to mine, in accordance to their performance. Every worker mines its share, and then signal that it is done by writing to a shared

Algorithm 15: StakhanovSync

Data: svState, i thread identifier, localDist the local number of distinguished points mined since last version update
Result: $\perp$ or a new function version
if svState.share$[i] \leq$ localDist **then**
  return $\perp$;
end
if svState.sync$[i] = \perp$ **then**
  svState.sync$[i] :=$ started;
end
everyoneDone $\triangleq \forall j$: svState.sync$[j] \neq \perp$;
shouldSync $\triangleq$ everyoneDone $\wedge$ svState.sync$[i] \neq$ done;
if *worker i is leader* **then**
  everyoneMovedOn $\triangleq \forall j \neq i$: svState.sync$[i] =$ done;
  shouldSync $:=$ everyoneMovedOn $\wedge$ everyoneDone;
end
if $\neg$shouldSync **then**
  return $\perp$;
end
ver $\triangleq$ svState.ver $+ 1$;
if *worker i is leader* **then**
  for *worker j* **do**
    svState.sync$[j] :=$ done;
  end
  svState.ver $:=$ ver;
else
  svState.sync$[i] :=$ done;
end
return ver;

array, and keeps mining. Once all the workers have signaled that they are done (by looking at the shared array) every non-leader switches the function versions and signals again. Once every non-leader worker is done, the leader updates the global version and switches itself. Algorithm 15 gives a formal description of this strategy.

As before, a fully featured implementation of Algorithm 15 can be found in `sync/stakhanov.rs`.

Remark 6.2 *In deciding the share of points that each worker should mine, we make the strong assumption that the performance of a worker is stable over time. In practice, due to frequency boosting, optimizations in processor's design, varying load of the server and other factors, this is not the case. Some measures can be*

taken in order to minimize these factors, and they should be taken to ensure that the balance between worker remains accurate during the run of Algorithm 9. In our local implementation, we simply assume that each core of the server's processor has the same performance, and simply assign each worker (one per core) a share of $\beta w/L$, where L is the number of cores.

As for Algorithm 14, we can choose how often to call the sync function, but note that for Algorithm 15 the cost is very much reduced. The unconditional check is only an integer comparison (without atomic instructions) and as such the cost is essentially negligible. We make concessions on both of the strong guarantees of Algorithm 14. First of all, some workers will concurrently be mining points on different function versions, leading to a small amount of wasted work. Secondly, especially if our share allocation is fallacious, it can happen that the function version is switched after considerably more than βw points have been mined. While these are valid concerns, experimental work in [CLN+20] has shown that Algorithm 9 implementations using Algorithm 15 outperform in practice those using Algorithm 14, and as such we have elected to use Algorithm 15 as our syncing algorithm for the networked implementation.

Networked Stakhanov The networked setting presents new challenges, but luckily Algorithm 15 needs only minor adjustments to work in that setting. We start by introducing some terminology. A **worker node** is a collection of local workers. An **orchestrator node** will instead be a central server whose only responsability is to keep track of the syncing status of the worker nodes. We stress that the orchestrator node does not run Algorithm 9 at all, and as such can be very low powered, with the only requirements being able to handle a connection from each of the worker nodes.

The networked adaptation of Stakhanov Sync works as follows. Each worker node will locally run a version of Algorithm 15, but once everyone has mined their share locally the leader of the cluster will signal the orchestrator node. Once the orchestrator node has received confirmation that every worker node is done, it signals to each of them to switch to the next function version. A more formal specification is shown in Algorithm 16 and Algorithm 17.

An implementation of Algorithm 16 can be found in `sync/networked_stakhanov.rs` and one of Algorithm 17 in `networking/orchestrator/manager.rs`. In our implementation, communication between the orchestrator node and the worker nodes is implemented as a TCP connection, since the syncing operation is only done occasionally (in the case of $n = 2^{24}, w = 2^{16}, \beta = 10$ for example, a function version is switched only every 0.6 seconds) and reliability is very important for the overall status of the attack. Using Algorithm 15 compared to Algorithm 14 as a base for the networked syncing algorithm

Algorithm 16: LocalStakhanov

Data: Sync state svState, node state node, i thread identifier, localDist the local number of distinguished points mined since last version update
Result: $\perp$ or a new function version

```
if svState.share[i] ≤ localDist then
    return ⊥;
end
if svState.sync[i] = ⊥ then
    svState.sync[i] := started;
end
everyoneDone ≜ ∀j : svState.sync[j] ≠ ⊥;
shouldSync ≜ everyoneDone ∧ node.gDone ∧ svState.sync[i] ≠ done;
if worker i is leader then
    if everyoneDone then
        Signal to orchestrator node is done;
    end
    everyoneMovedOn ≜ ∀j ≠ i : svState.sync[i] = done;
    shouldSync := everyoneMovedOn ∧ everyoneDone ∧ node.gDone;
end
if ¬shouldSync then
    return ⊥;
end
ver ≜ svState.ver + 1;
if worker i is leader then
    for worker j do
        svState.sync[j] := ⊥;
    end
    svState.ver := ver;
    node.gDone := false;
else
    svState.sync[i] := done;
end
return ver;
```

also allows the worker node to do useful work while the network syncs, which is why we chose it for our implementation.

6.2.2 Memory

Distributing memory in the networked setting also presents challenges. Previous work such as [BBB+09] mainly focused on the case of a few central

Algorithm 17: Stakhanov Manager

```
Data: Number of nodes n
nodeStatus ≜ [];
for i=1...n do
    nodeStatus[i] := ⊥;
end
repeat
    Wait for message from node i;
    nodeStatus[i] := done;
    if ∀i : nodeStatus[i] = done then
        for i = 1...n do
            Set gDone to true in node i;
            nodeStatus[i] := ⊥;
        end
    end
until False;
```

Algorithm 18: Memory Init

```
Data: Total size of memory w, local memory size w_i
Result: An handle to memory mem
mem.A ≜ [w];
mem.T ≜ S × S × [maxlen];
mem.local = Mem.Init(w_i);
return mem;
```

servers at a single site whose only responsability was to receive distinguished points. This work enables a more flexible model, in which any node can double as a memory node, and both send and receive points. We assume that each node n_i has associated a memory w_i and let $w \triangleq \sum w_i$ be the total memory. We associate to each node a partition of the address space $[w]$. In particular, we define $O_1 \triangleq 0$ and $O_i = O_{i-1} + w_i$, and let node n_i be responsible for the address range $[O_i, O_{i-1})$. To actually construct this new shared memory, we can simply abstract over the original scheme described in Algorithm 6 and Algorithm 7. The resulting memory scheme is described in Algorithm 18 and Algorithm 19. In a nutshell, depending on the address, a mined distinguished point is either written to local memory, or sent to the node whose address space contains the address. On the receiving end, the node will receive the point, write it to its local memory and do backtracking if necessary.

Implementations of this memory scheme can be found in the `memory` mod-

Algorithm 19: Send Point for node ℓ

Data: A memory handle mem, and address $a \in [w]$, a value $t \in \mathsf{mem}.T$
Result: Either the existing value $\in \mathsf{mem}.T$ or an error $\perp$
if $a \in [O_\ell, O_{\ell+1})$ **then**
  return $\mathsf{mem.local.sendPoint}(a - O_\ell, t)$;
end
for $i \neq \ell$ **do**
  if $a \in [O_i, O_{i+1})$ **then**
    Send point to n_i;
  end
end
return $\perp$;

ule, while the receiver implementation is in `networking/receiver.rs`. In our implementation, we have decided to use UDP for sending distinguished points. The reason for this is that, in a distributed attack, the cost of sending packets is the bottleneck of the system, especially if evaluating f is not too computationally expensive. TCP has an expensive handshake protocol, and offers strong guarantees of sequential ordering and reliability that our application does not need. Ordering of distinguished points can be important (consider for example the case in which two triples are sent to the same memory address, the first one, in conjunction with the current occupant, leading to a golden collisions, and the second leading to a memory collision; then ordering would be crucial to find the golden collision), but it seems hard a priori imposing an optimal order. A possible strategy would be buffering triples with the same address and checking all possible orderings, but we do not have explored this solution further, and do not know whether it could bring some performance improvements in practice. Also, in the case of UDP, even if a fraction of our mined points are not delivered successfully we expect the attack to still succeed with a possibly longer runtime.

6.3 Measurements and Observations

In this section, we report our findings in running Klondike. First of all, we compared the diffent syncing strategies and heuristically observe that while Stakhanov sync (Algorithm 15) often mines more distinguished points than what it is required, in a local setting this number of points is only marginally higher than βw (by at most 1%). Since Stakhanov ensures that processing units are always active, we recommend using it over NoBiggie sync (Algorithm 14) in most settings. When running in a distributed setting instead we observe that the number of points mined per function version is

Figure 6.1: Flamegraph of a distributed execution of Klondike. Parameters $n = 2^{32}, w = 2^{24}$. Run with 20 cores on Alishan, 20 on Daisen, orchestrator on Iwo. Algorithm 4 is responsible of 55.87% of the runtime, Algorithm 5 of 4.96%, and Algorithm 7 of 31.44%. Recorded on Alishan.

often much higher than what desired, from 50% to 700%, with this number increasing when bandwidth is decreased.

We have profiled the run of the application in a few settings, and shown the resulting flamegraphs in Figures 6.1 to 6.3.

The relevant network topology is that Daisen and Alishan are on the same local area network, Iwo is also on the university network but not in the same local network, finally Choripan is a personal server not directly connected with the other two. As such, we expect the network connection to Choripan to be the most congested. What the flamegraphs show is that in this new distributed setting the cost of writing point to memory becomes very significant. Compare Figure 5.1 to Figures 6.1 to 6.3. The cost of writing to memory in the local case accounted for less than 2% of the runtime, while in the distributed case it is between 31% to 37%. In fact, note that in every distributed instance, despite the bandwidth being very different, the share of computation devoted to sending point remains quite similar. For

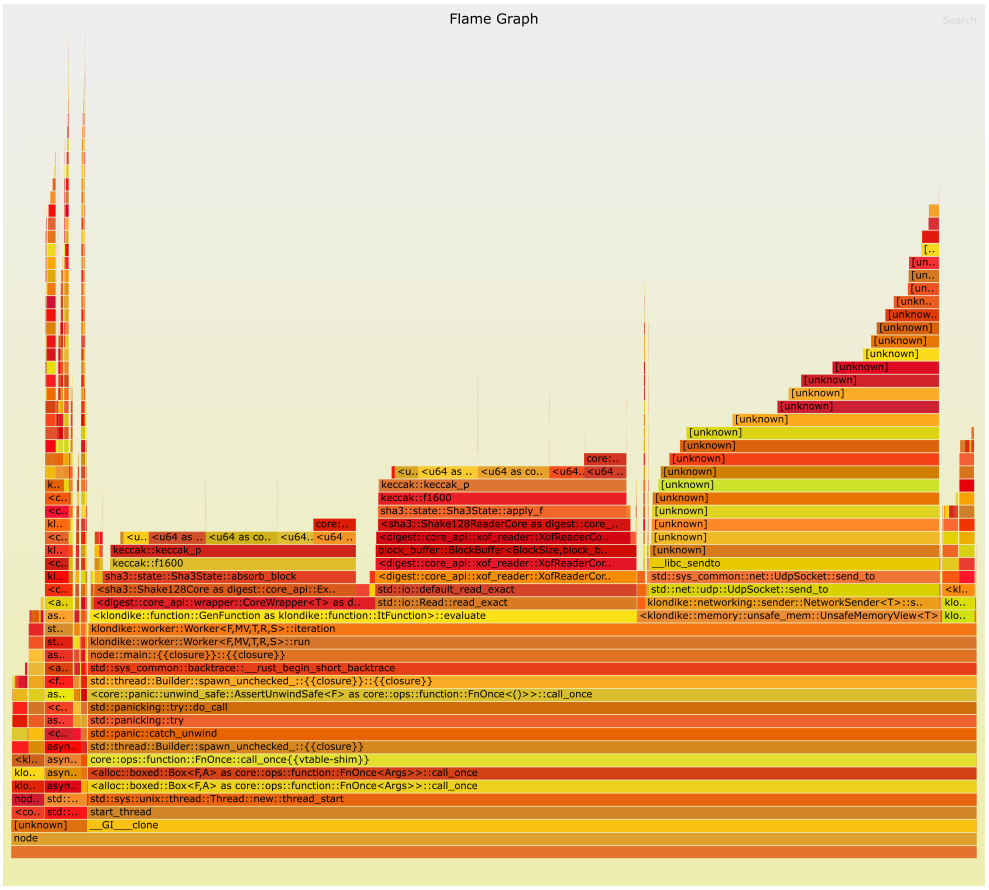


Figure 6.2: Flamegraph of a distributed execution of Klondike. Parameters $n = 2^{32}, w = 2^{24}$. Run with 20 cores on Alishan, 4 on Iwo, orchestrator on Daisen. Algorithm 4 is responsible of 56.6% of the runtime, Algorithm 5 of 3.8%, and Algorithm 7 of 31.35%. Recorded on Alishan.

reference, the bandwidth between Alishan and Daisen is 200 MB/s, while that between Alishan and Iwo is 95 MB/s, and that between Alishan and Choripan is 30 MB/s. The similar share is explained by the fact that, in a bandwidth constrained setting, triples are dropped. In our estimates, in a run between Alishan and Choripan, only 1 in 4 triples sent from Alishan actually is delivered to Choripan. Instead, almost every triple sent from Choripan to Alishan arrives to destination.

6.4 Improvements

In this section, we sketch some of the possible improvements and/or modifications to Klondike that we did not have time to test in detail, and that we leave for future development. First of all, in our implementation a UDP packet is sent for each distinguished point that has to be sent. Each packet

Figure 6.3: Flamegraph of a distributed execution of Klondike. Parameters $n = 2^{32}, w = 2^{24}$. Run with 64 cores on Alishan, 2 on Choripan, orchestrator on Daisen. Algorithm 4 is responsible of 56.76% of the runtime, Algorithm 5 of 2.84%, and Algorithm 7 of 37.45%. Recorded on Alishan

has 28 bytes of header information, so for small triples most of the packet consists of header information. For $n = 2^{32}$, for example, a triple is on the order of 12 bytes so only a third of the packet is used for the payload. An obvious solution to this issue is buffering the triples, sending more than one per packet. As future work, we aim to investigate the performance benefits of this optimization. Secondly, we have chosen to use UDP for our implementation, and we do believe this is the optimal choice. However, experiments show that, under heavy network load, a majority of the packets are silently dropped by the network, leading to wasted computation. It could be worthwhile establishing whether TCP's reliability would offer benefits, and whether this outweighs its additional costs.

Chapter 7

Case Study: SIKE

The **S**upersingular **I**sogeny **K**ey **E**xchange [JAC+20] was a key encapsulation scheme, currently in 4th round of the National Institute of Standards and Technology (NIST) competition for Post Quantum Cryptographyic primitives. SIKE security relied on the difficulty of the Computational Supersingular Isogeny (CSSI) problem, and until very recently the best known attack (classical or quantum) on this assumption was Algorithm 9 ([CLN+20] [ACC+19] [JS19]). Recently, [CD22] showed a classical polynomial time attack on CSSI when the endomorphism ring of the starting curve is known (as in SIKE). The limitation of the attack was then rapidly lifted firstly the subexponential attack of [MM22] and then by a polynomial time attack in [Rob22]. This line of work showed that the CSSI problem is in fact solvable in polynomial time, and as such SIKE is insecure.

In this section, we will be introducing CSSI and the related Supersingular Isogeny Path Problem (SIPP). SIPP is known from [EHL+18] to be equivalent to the Supersingular Smooth Endomorphism Problem on which the soundness of SQISign [DKL+20] relies. Currently, the best attack on SIPP is still Algorithm 9, which makes it a viable candidate for our investigation.

7.1 Preliminaries

This section is inspired by [GV17] and general references can be found in [Sil09]. If G is a group and H is a subgroup of G we write $H \leq G$. Let $p > 3$ be a prime, $q = p^n$ for $n > 0$. Recall that there exists a unique field $\mathbb{F}_q$ with q elements, and we denote by $\overline{\mathbb{F}}_q$ its **algebraic closure**. An **elliptic curve over** $\mathbb{F}_q$ is given by two coefficients $A, B \in \mathbb{F}_q$ and is defined as set of points

$$E(\mathbb{F}_q) = \{(x, y) : y^2 = x^3 + Ax + B\} \sqcup \{\infty\}$$

where ∞ is the point at infinity. To any elliptic curve we associate the **discriminant** and the j-**invariant** which are defined as

$$\Delta = -16(4A^3 + 27B^2), \quad j(E) = -1728\frac{(4A)^3}{\Delta}$$

If $\Delta = 0$, the curve is **singular**. From now on, all considered curves will be non-singular. Any elliptic curve has an associated abelian group structure with identity ∞, which we write additively. In particular, we denote addition of two points $P, Q \in E(\mathbb{F}_q)$ as $P + Q$, the inverse of a point as $-P$ and scalar multiplication by $k \in \mathbb{Z}$ as $[k]P$. An elliptic curve E over $\mathbb{F}_q$ is **supersingular** if and only if $|E(\mathbb{F}_q)| \equiv 1 \pmod{p}$. For $k \in \mathbb{N}$, we let $E[k] = \{P \in E(\overline{\mathbb{F}}_q) : [k]P = \infty\}$ be the k-**torsion subgroup** of E. If $p \nmid k$ then $|E[k]| = k^2$ and $E[k] \cong \mathbb{Z}_k \times \mathbb{Z}_k$ as a group.

A **morphism** of elliptic curves is $f : E \to E'$ is a function that map points of E to points of E' and can be described as ratios of polynomials. An isomorphism $f : E \to E'$ is a morphism which also maps $f(\infty) = \infty$ and whose inverse is also a morphism. $j(E) = j(E')$ if and only if there exists an isomorphisms $f : E \to E'$.

Any supersingular curve over $\overline{\mathbb{F}}_p$ is isomorphic to one over $\mathbb{F}_{p^2}$ and as such all the supersingular j-invariants are in $\mathbb{F}_{p^2}$. There are approximately $p/12$ supersingular j-invariants.

An **isogeny** between E and E' is a morphism $\phi : E \to E'$ that satisfies $\phi(\infty) = \infty$, in this case we say that E and E' are **isogenous**. It can be shown that an isogeny is also a group homomorphism. We also say that an isogeny is **defined over** $\mathbb{F}_q$ if the isogeny is given as a rational function of polynomials in $\mathbb{F}_q[x, y]$. Tate's isogeny theorem proves that two elliptic curves defined over $\mathbb{F}_q$ are isogenous over $\mathbb{F}_q$ if and only if $|E(\mathbb{F}_q)| = |E'(\mathbb{F}_q)|$. The **degree** of an isogeny is the size (as a set) of its kernel[1], and is multiplicative over composition, in the sense that $\deg \phi \circ \psi = \deg \phi \cdot \deg \psi$. For any $k \in \mathbb{N}$, the map $[k] : E \to E$ defined as $P \mapsto [k]P$ is an isogeny. For any isogeny $\phi : E \to E'$ there exists a unique **dual isogeny** $\hat{\phi} : E' \to E$ such that $\phi \circ \hat{\phi} = [\deg(\phi)]$.

Every isogeny $\phi : E \to E'$ has a finite kernel $G = \ker(\phi) \leq E(\overline{\mathbb{F}}_q)$. Conversely, every finite subgroup G of $E(\overline{\mathbb{F}}_q)$ determines a unique (up to composition with isomorphisms) isogeny $\phi : E \to E/G$ with $\ker(\phi) = G$. If $G \subseteq E(\mathbb{F}_q)$ then ϕ is defined over $\mathbb{F}_q$. Vélu's formulas allow us to compute ϕ and E/G in time linear in $|G|$. In practice, we would like to compute isogenies of large degree, and as such we will not be able to apply these formulas directly. However, we can decompose long isogenies into shorter

[1]This is true for separable isogenies, and in this exposition we will only consider that kind.

ones. A separable isogeny $\phi : E \to E'$ can be written as $\phi = \phi_1 \circ \cdots \circ \phi_k \circ [n]$ and, by multiplicativity of the degree, $\deg(\phi) = n^2 \prod_{i=1}^{k} \deg(\phi_i)$. In practice, this means that we can compute an isogeny of degree ℓ^e as a composition of e ℓ-isogenies, at cost $O(e \cdot \ell)$ compared to $O(\ell^e)$. In fact, [BDLS20] recently showed how to improve the cost of computing an isogeny of prime degree ℓ in $O(\sqrt{\ell})$, reducing the cost of the isogeny to $O(e \cdot \sqrt{\ell})$.

7.2 Isogeny Problems

We now can introduce a (now) easy problem on isogenies and an hard one.

Problem 7.1 *Let p be a prime, with $p = 2^{e_2}3^{e_3} - 1$, $\ell \in \{2, 3\}$, E and E' two supersingular elliptic curves defined over $\mathbb{F}_{p^2}$ which are ℓ^{e_ℓ}-isogenous via an isogeny $\phi : E \to E'$. Let P, Q be two generators for $E[\ell^{e_\ell}]$. The **computational supersingular isogeny problem** is the following: given $p, \ell, E, E', \phi(P), \phi(Q)$, compute ϕ.*

Note in particular that in CSSI we are given the image of ϕ on the generators of $E[\ell^{e_\ell}]$, via the points $\phi(P), \phi(Q) \in E'$. Those points are referred to as **torsion points**, and this information is crucial in the attacks of [CD22], [MM22] and [Rob22].

A related problem that does not reveal those torsion point is given next, slightly generalized for the setting of [DKL+20].

Problem 7.2 *Let p be a prime, E and E' two supersingular elliptic curves defined over $\mathbb{F}_{p^2}$ which are isogenous via an isogeny $\phi : E \to E'$ of order d with known factorization. The **supersingular isogeny path problem** is the following: given p, d, E, E', compute ϕ.*

To our knowledge, the best known attack on Problem 7.2 is still based on Algorithm 9, and we will be showing this attack in the next section, and analyzing its runtime. As previously mentioned, this problem is not only of theoretical interest: soundness of the SQISign signature scheme [DKL+20] relies on an equivalent problem to a version of Problem 7.2[2].

7.3 An attack on SIPP

Remark 7.3 *The literature of attacks on Problem 7.1, 7.2 starts in [DJP11], which phrased the problem as a claw finding problem, that is, given two function $f : A \to B$, $g : B \to C$, computing a pair $(a, b) \in A \times B$ such that $f(a) = g(b)$. In the classical case, the optimal black-box asymptotics for solving such problem are $O(|A| + |B|)$, while in the quantum case an algorithm due to [Tan09] improves this to $O(\sqrt[3]{|A||B|})$. However, successive work by [ACC+19] showed that, while*

[2]Only changing the prime

classical claw finding attacks had the best runtime, the storage costs that it imposed were large enough as to make the attack unfeasible in any reasonable cost model. They proposed instead a new cryptanalytical model, in which the attacker only has access to a large (but limited) amount of memory, in the order of 2^{80} units of storage, and analyzed concrete attacks on SIKE in that setting, concluding that Algorithm 9 would be the best classical algorithm for tackling Problem 7.2[3]. On the quantum side, [JS19] showed that, under a hypothetically more accurate modelling of quantum memory, known quantum algorithms do not have significant advantages over Algorithm 9 in solving Problem 7.2.

Motivated by Remark 7.3, we here present how to leverage Algorithm 9 to solve Problem 7.2. This general attack was first described by [ACC+19], and later refined by [CLN+20], on whose version we base our description.

We assume that we are given a prime p, and two supersingular elliptic curves E_0, E_1 that are d-isogenous. We assume that the factorization of d is known so that $d = \prod_{i=1}^{k} p_i^{e_i}$ for p_i (small) known primes. Assuming that $p \nmid d$, we have that $E[d] \cong \mathbb{Z}_d \times \mathbb{Z}_d$. Note that $\mathbb{Z}_d \cong \bigoplus_i \mathbb{Z}_{p_i^{e_i}}$ and so the number of subgroups of $E[d]$ of order d is exactly $\prod_{i=1}(p_i+1)p_i^{e_i-1}$. Now, let us select d_0, d_1 such that[4] $d_0 \approx d_1$ and $d_0 d_1 = d$. The main idea of the attack is the following. Consider the d-isogeny $\phi : E_0 \to E_1$ that we are hoping to find. We can write $\varphi = \hat{\psi} \circ \theta$ for a d_0-isogeny $\theta : E_0 \to E'$ and a d_1-isogeny $\psi : E_1 \to E'$, with E' an unknown curve. Since $\hat{\psi}$ can be efficiently computed from ψ, finding θ, ψ leads directly to finding the secret φ. Note now that E' can be reached from E_i by an isogeny of degree d_i, so what this suggests is computing the j-invariants of curves reached in this manner until an intersection of this form is found. Let us make this more formal.

Let us write $d_i = \prod_k p_k^{e_{i,k}}$ for $i = 0, 1$ so that $e_{0,k} + e_{1,k} = e_k$.

Define $S_i = \{K : K \leq E[d_i] \text{ is cyclic, of order } d_i\}$. Then consider the set $R_i = \{j(E_i/K) : K \in S_i\}$. By a reasoning as before this set has size $|R_i| = \prod_k (p_k+1)p_k^{e_{i,k}-1}$. In our choice of d_0, d_1 we also want to ensure that $|R_1| \approx |R_2|$. Now, since we can decompose $\varphi = \hat{\psi} \circ \theta$, we know that $|R_0 \cap R_1| > 0$, and, since $|R_i| \ll p/12$ which is the size of the set of the j-invariants of all supersingular curves, we can assume that $|R_0 \cap R_1| = 1$, i.e. there is a unique curve E' that is reachable from both E_0, E_1 with a small degree isogeny. Finding this curve can be done via claw-finding (for example, let $f_i : S_i \to \mathbb{F}_{p^2}$ be the map $K \mapsto j(E_i/K)$ and look for a claw) but we would like to reduce our memory footprint.

Without loss of generality[5], let $|S_0| \geq |S_1|$ and let $N \triangleq |S_0|$. We assume

[3] In fact, at the time this was best attack against Problem 7.1 as well

[4] We will also require a further condition, but more on this later

[5] And without much efficiency problem, since the assumptions that $|R_0| \approx |R_1|$ implies that $|S_0| \approx |S_1|$ by the correspondence of isogenies and subgroups.

that we have maps $\iota_i : [N] \to S_i$ (ideally, ι_0 is an injection and ι_1 is an injection composed with the modulo operation $[N] \to [|S_1|]$).

We build a set $S : \{0,1\} \times [N]$, and define the map $f : S \to \mathbb{F}_{p^2}$ as mapping $(i,x) \mapsto j(E_i/\iota_i(x))$. We also define our golden collision test $\text{gold}_f : \text{Coll}(f) \to \{0,1\}$ as mapping $((i,x),(i',x')) \mapsto i \neq_? i'$, i.e. testing whether the f-collision originated from two different curves.

By our definition of versioning with Algorithm 9, it is easy to see that a golden collision for f is a pair $(0,x),(1,x')$ with $j(E_0/\iota_0(x)) = j(E_1/\iota_1(x'))$ and from that we can then compute the isogenies ψ, θ and thus the secret isogeny.

Remark 7.4 *Note that, as mentioned in Remark 3.36, since S has some special structure, we should adapt Algorithm 10 and Algorithm 11 accordingly.*

As such, we can apply Algorithm 9 to solve Problem 7.2. What would be the complexity of this? Recalling our models from Chapter 5 we know that asymptotically the cost of the attack when using w memory cells is $O(|S|^3 w^{-1/2})$. $|S| = 2N$ and depends crucially on the choice of d, d_0, d_1. For example, in the case of SIKE we have that $d = 2^e \approx p^{1/2}$, and that $d_0 \approx d_1 = 2^{e/2}$ and as such $N = 3 \cdot 2^{e/2} \approx p^{1/4}$ so that the final asymptotic cost of the attack is $O(p^{3/4} w^{-1/2})$ evaluations of f. In general if we can select d_0, d_1 close in value and as thus approximately equal to $\sqrt{d}$ we can mount the attack with $O(d^{3/2} w^{-1/2})$ functions evaluations.

The cost of evaluating f is dominated by the cost of d_i-isogeny computations, which is $O(\sum_k e_{i,k}\sqrt{p_k})$ field operations using the techniques of [BDLS20][6]. In order for this to be efficient (which is also desirable constructively) we would like for $E[d_i]$ to be defined over a small field extensions of $\mathbb{F}_p$, and we will assume this for the remainder of this chapter.

7.4 Efficient Isogeny Implementation

The asymptotic analyses mentioned in Chapter 5 give an estimate of the costs of running Algorithm 9 in terms of function evaluations. As such, in order to give a pratical estimate of the cost of breaking Problem 7.2, it is important to understand the cost of computing isogenies of a given degree. Our starting point in doing so is the Microsoft's [PQC] library, which provides an optimised constant time implementation of both SIDH and SIKE, for the parameter sets specified in the SIKE spec, namely `p434`, `p503`, `p610`, and `p751`.

[6] In fact, for $p_i < 100$, [DKL+20] suggest that using naïve technique might be faster.

As part of our contributions, we have also provided support for p217, and optimised such implementation for x64 platforms. The repository will be available as supplementary material to this work.

7.4.1 Efficient Field Operations

The distinct form of SIKE primes allows for a number of optimisations in implementing arithmetic over both $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$. The current state of the art is due to Longa [Lon22], in this section we will be sketching the core of their results and how it is applied in the implementation of p217. Recall that in SIKE the characteristic of the base field has a special form, namely $p = 2^{e_2}3^{e_3} - 1$. Computation in SIKE is over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, and as such central to the efficiency of the scheme is efficient arithmetic in that setting.

For consistency with [Lon22], in this section we will let w be the word size of the processor, which in our case will be 64 bits. We will let $l \triangleq \lceil \log(p) \rceil$, $n \triangleq \lceil l/w \rceil$ and $N \triangleq nw$.

The core of the arithmetic is Montgomery multiplication [Mon85]. Let p be a prime, and suppose that we wanted to compute in $\mathbb{F}_p$. We define $R \triangleq 2^N$ and $p' = -p \mod R$. For a value $a \in \mathbb{Z}_p$ we define its Montgomery representation as $\tilde{a} = aR \mod p$. Note that for $a, b \in \mathbb{Z}_p$ we have that $\widetilde{a+b} = \tilde{a} + \tilde{b}$. In the case of multiplication, let us assume that $\tilde{a}\tilde{b} < pR$. Then $\tilde{a}\tilde{b} = abR^2 \mod p \neq abR \mod p = \tilde{ab}$. Instead, we would like to compute the Montgomery residue $c = \tilde{a}\tilde{b}R^{-1} \mod p = abR \mod p$. This can be done by computing

$$c = \left(\tilde{a}\tilde{b} + (\tilde{a}\tilde{b}p' \bmod R) \cdot p\right) / R$$

Note that since $R = 2^N$ the division and the modulo can be computed very efficiently.

To compute this residue, an interleaved approach can be taken. We set $r = 2^w$ as the radix[7], and compute c in an iterative manner. We represent $a = (a_{n-1}, \ldots, a_0)_r$ in its base r representation, and we set $c := 0$, in each iteration computing

$$c := \left(c + \tilde{a}_i\tilde{b} + (c + \tilde{a}_i\tilde{b}p' \bmod r) \cdot p\right) / r$$

for i going from 0 to $n - 1$.

Finally, the main contribution of [Lon22] is the fact that this interleaved computation can be used to compute efficiently sums $\sum_{i=0}^{t-1} a_i b_i$ (under some conditions). The full algorithm is presented in Algorithm 20

[7] [Lon22] used a slightly different formulation in which $r = 2^{Bw}$, but in our setting $B = 1$ is sufficient

Algorithm 20: [Lon22] Algorithm 2

Data: A prime p. w, R, l, r, p' as before. A list of integers $(a_0, \ldots, a_{t-1}), (b_0, \ldots b_{t-1})$ with $a_i, b_i \in [0, 2p)$ and $0 \leq \sum_i a_i b_i < pR$. Integers are in radix r, so $a_{i,j}$ is the j-th r-word of a_i.

Result: The Montgomery residue $c = \sum_i a_i b_i \cdot R^{i-1} \mod p$ with $c \in [0, 2p)$

$u := 0$;
for $j = 0 \ldots n-1$ **do**
  $u := \sum_{i=1}^{t-1} a_{i,j} b_i$;
  $q := up' \mod r$;
  $u := (u + qp)/r$;
end
return u

Algorithm 21: [Lon22] Algorithm 4

Data: A prime $p = 2^{e_2}3^{e_3} - 1$. w, R, l, r, p' as before. $z \triangleq \lfloor e_2/w \rfloor$, $\hat{p} \triangleq (p+1)/2^{zw}$. A list of integers $(a_0, \ldots, a_{t-1}), (b_0, \ldots b_{t-1})$ with $a_i, b_i \in [0, 2p)$ and $0 \leq \sum_i a_i b_i < pR$. Integers are in radix r, so $a_{i,j}$ is the j-th r-word of a_i.

Result: The Montgomery residue $c = \sum_i a_i b_i \cdot R^{i-1} \mod p$ with $c \in [0, 2p)$

$u := 0$;
for $j = 0 \ldots n-1$ **do**
  $u := \sum_{i=1}^{t-1} a_{i,j} b_i$;
  $q := up' \mod r$;
  $u := \lfloor u/r \rfloor + 2^{(z-1)w} q \cdot \hat{p}$;
end
return u

Remark 7.5 *Note that integers are represented as being in the range $[0, 2p)$ instead of the more traditional $[0, p)$. This allows to elide a conditional subtraction after multiplication to bring the result back in that range.*

Since the prime p for SIKE has a special form, we can enable some optimizations. First, $p' \equiv 1 \pmod r$. Second, the prime p will start with some number z of zeros, so we can replace multiplication by p with multiplication by the much smaller value $\hat{p} \triangleq (p+1)/2zw$. In brief, the final algorithm is presented in Algorithm 21

Finally note that if we take $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ for $i^2 - 1 = 0$ then we can write multiplication in $\mathbb{F}_{p^2}$ as $(a + bi)(c + di) = ac - db + (ad + bc)i$, and so can

Operation	p217	p434
$\mathbb{F}_p$ Addition	30	30
$\mathbb{F}_p$ Subtraction	30	30
$\mathbb{F}_p$ Multiplication	54	107
$\mathbb{F}_p$ Inversion	9637	48388
$\mathbb{F}_{p^2}$ Addition	31	40
$\mathbb{F}_{p^2}$ Subtraction	30	33
$\mathbb{F}_{p^2}$ Multiplication	137	307
$\mathbb{F}_{p^2}$ Inversion	9927	49705

Table 7.1: Number of cycles for common operations over $\mathbb{F}_p, \mathbb{F}_{p^2}$. Measured on AMD Ryzen 3950X 16-Core Processor. Averaged over a million iterations

Operation	p217	p434
Point Doubling	693	1661
Point Tripling	1373	3329
4-isogeny computation	437	900
4-isogeny evaluation	923	2233
3-isogeny computation	628	1507
3-isogeny evaluation	668	1728

Table 7.2: Number of cycles for common operations in SIKE. Measured on AMD Ryzen 3950X 16-Core Processor. Averaged over a million iterations

be computed as applying Algorithm 21 to the sums $ac - db$ and $ad + bc$.

We have implemented this for p217 and have collected (and compared with SIKEp434) the cost of operations in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ in Table 7.1.

On top of these, we also have measured the cost of computing isogenies and common elliptic curves operation using our implementation. The results are shown in Table 7.2

7.5 Attack Models

Before presenting our estimates, it is crucial to fix our attack model. In this section we propose a few different models for attacking Problem 7.2 (as such without torsion point information) using Algorithm 9. In each case, we will be looking at instantiations of SIPP in which p, d are of the form of SIKE, since that is where we have investigated the cost of evaluating the attack function f. Some of the models will be mainly local, and as such can be compared to other models in the literature, while others will be distributed, and, as far as we know, are novel.

Name	Target	L	w	Cost of f	Bandwith
Academic	p217	$[2^{16}, 2^{28}]$	$[2^{34}, 2^{47}]$	65 μs	20 GB/s
Nation State	p434	$[2^{54}, 2^{64}]$	$[2^{72}, 2^{80}]$	3 ms	1 TB/s

Table 7.3: Local models for attackers on Problem 7.2. Target reflects whether the target instance has p, d as in p434 or p217. Memory size in units where a unit is approximately $2 \log n$ bits for $n \approx p^{1/4}$

Name	Target	Nodes	$\sum_i L_i$	w	Cost of f	L. Bandwidth	R. Bandwidth
Even Share	p217	8	$[2^{19}, 2^{31}]$	$[2^{37}, 2^{50}]$	65 μs	20 GB/s	20 MB/s
2 Compute	p217	3	$[2^{19}, 2^{31}]$	$[2^{37}, 2^{50}]$	65 μs	20 GB/s	20 MB/s
8 Compute	p217	9	$[2^{19}, 2^{31}]$	$[2^{37}, 2^{50}]$	65 μs	20 GB/s	20 MB/s

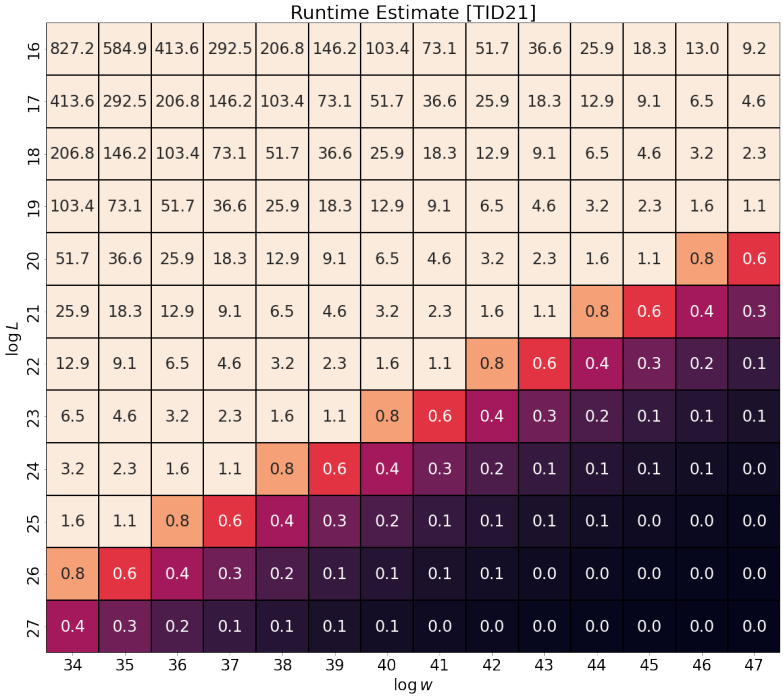
Table 7.4: Distributed models for attackers on Problem 7.2. Target reflects whether the target instance has p, d as in p434 or p217. L_i refers to computing units per node. Memory size in units where a unit is approximately $2 \log n$ bits for $n \approx p^{1/4}$

The local models are specified in Table 7.3. Some notes on why we chose those numbers. For the academic model the ranges of processors start from approximately the total number of cores on the Euler [Eul] cluster to the number of cores on the Sugaku Supercomputer [Sug]. Memory similarly ranges from very little (around 32 GB) to the 4.85 PB of Sugaku. The cost of an evaluation is what we measured on a local machine, and the memory bandwidth is standard for DDR4 RAM modules. For the nation state model we used the same range as [ACC+19] for processors and memory units, and assumed that the attacker can access the whole of its memory at L1 memory bandwidth. For the cost of function evaluations, we have taken the estimate of [LWS21] for ASIC implementations of `SIKEp434`.

As for the distributed setting we propose a number of models, shown in Table 7.4. For simplicity, in the distributed case we have focused on the p217. We assume the same time costs for function evaluation as in the local case, and fix the bandwidth between two nodes to 20 MB/s, as suggested by our measurements. The three models differ in the following way. The Even Share model shares computing units and memory evenly between the 8 nodes. The 2 Compute model share computing units evenly between the three nodes, but places all of the memory in a single node. The 8 Compute is similar to the 2 Compute node but 8 nodes are used instead for computation. In both cases, we will be using Model 5.2 as our underlying model.

7.6 SIKEp217

First of all, let us see what the cost of breaking Problem 7.2 would be in a local setting using our academic attack model. We have collected the estimates in Figure 7.1, showing the estimates in years to break p217 under our

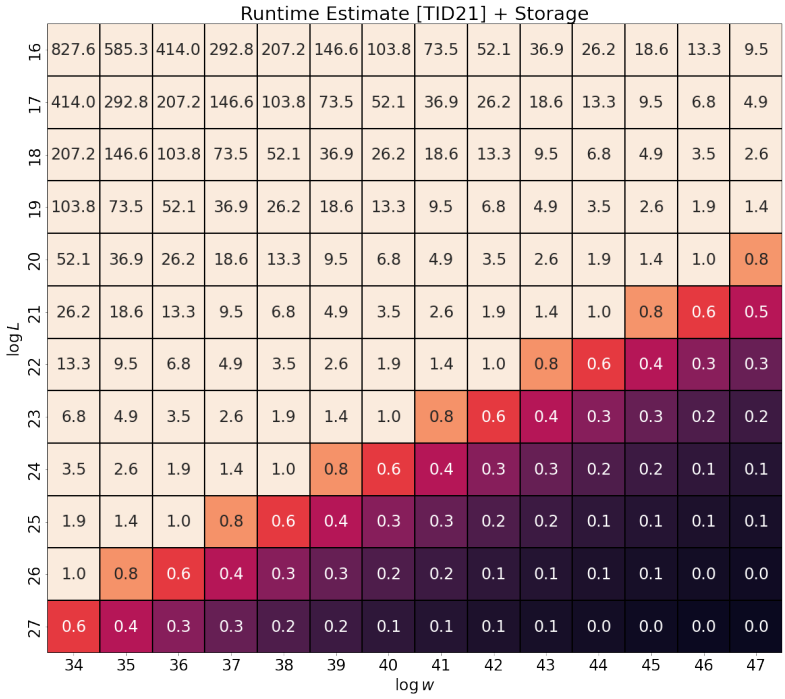


(a) Without storage cost

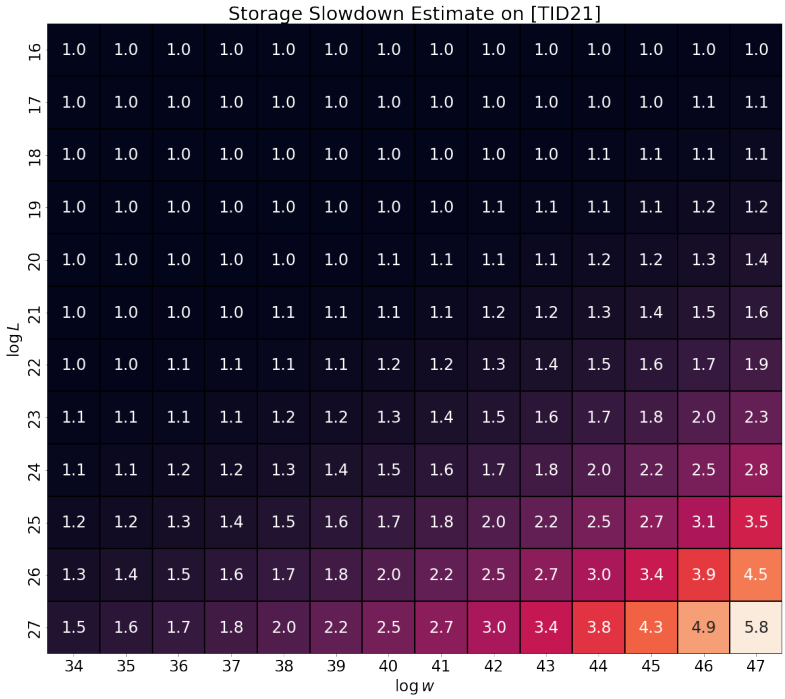
academic models, and plotting the slowdown when accounting for storage. As it is evident from Figure 7.1c, as the capabilities increase, the 'slow' local memory becomes a bottleneck.

Finally, in Figure 7.2 we have shown the cost of breaking p217 in a distributed attack within the models of Table 7.4. Note that even if these models allow 8 times the computing resources of the original local model, the costs imposed by the slow networked connections are very noticeable. Compared to equivalent resources, in Figure 7.2a, this is anywhere between a 10 to a 83 time slowdown, which increases when L, w increase. As such, if the network's bandwidth does not scale accordingly it rapidly becomes a very noticeable bottleneck. Comparing with Figures 7.2b and 7.2c, moving all the memory to a single node seems to have a noticeable improvement in the running time of the attack, and increasing the number of nodes (while keeping total number of processors and memory equal) increases still the cost of the attack[8]. As such, our reccomendation for running vOW's algorithm in a distributed setting is to try to have a few large memory nodes, and few

[8]This is caused mostly by the fact that comparatively fewer computational resources are available for backtracking. In fact, these particular models likely undershoot the costs that would originate by using more nodes, since they assume that pairwise bandwidth between peers remains constant. Most likely in a real world setting adding more nodes would reduce the available bandwidth at a node and as such reduce the pairwise bandwidth accordingly.

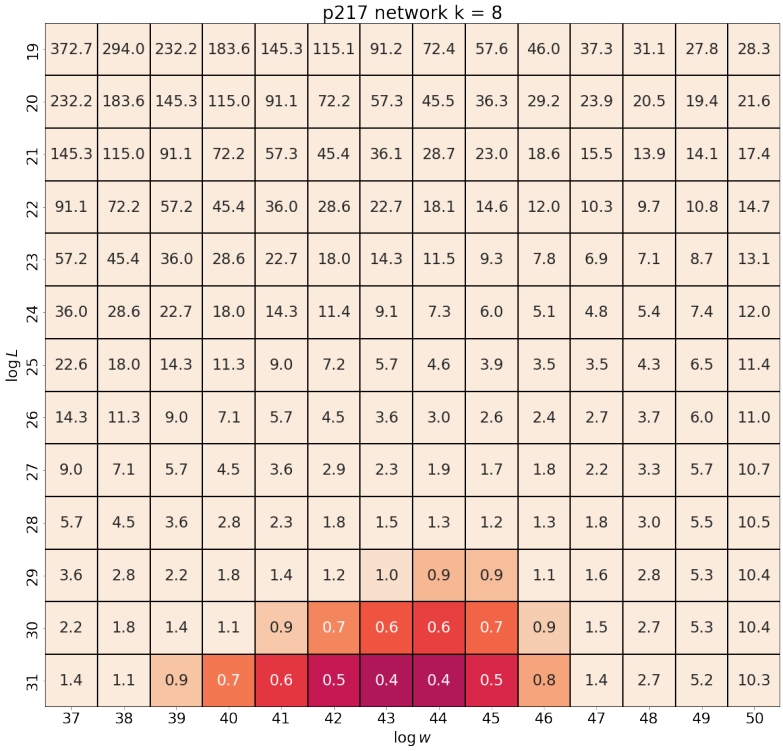


(b) With storage cost



(c) Relative Slowdown

Figure 7.1: Cost estimates for breaking Problem 7.2 with p217 parameters using the Academic model of Table 7.3. Estimates in years

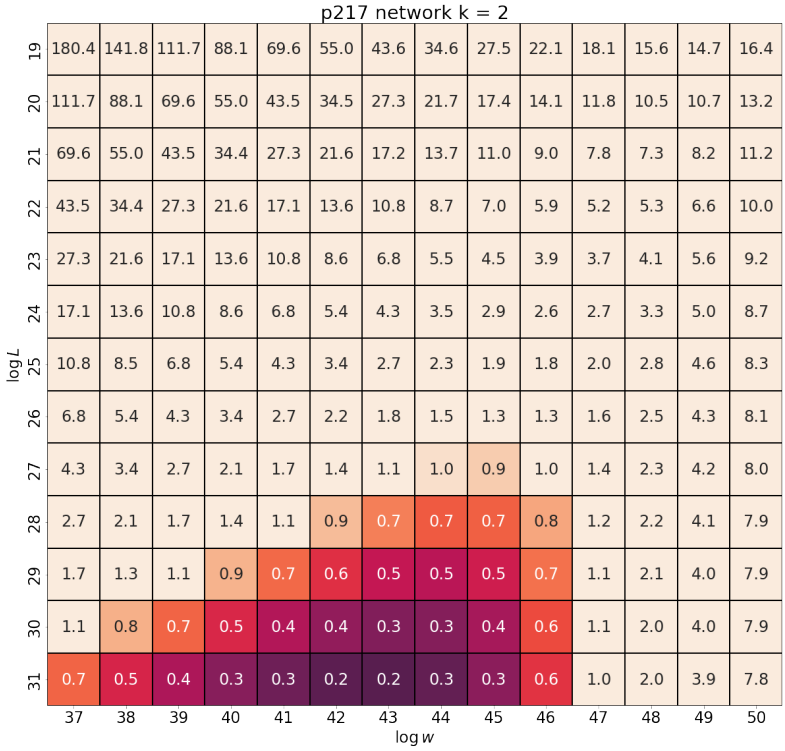


(a) Even Share

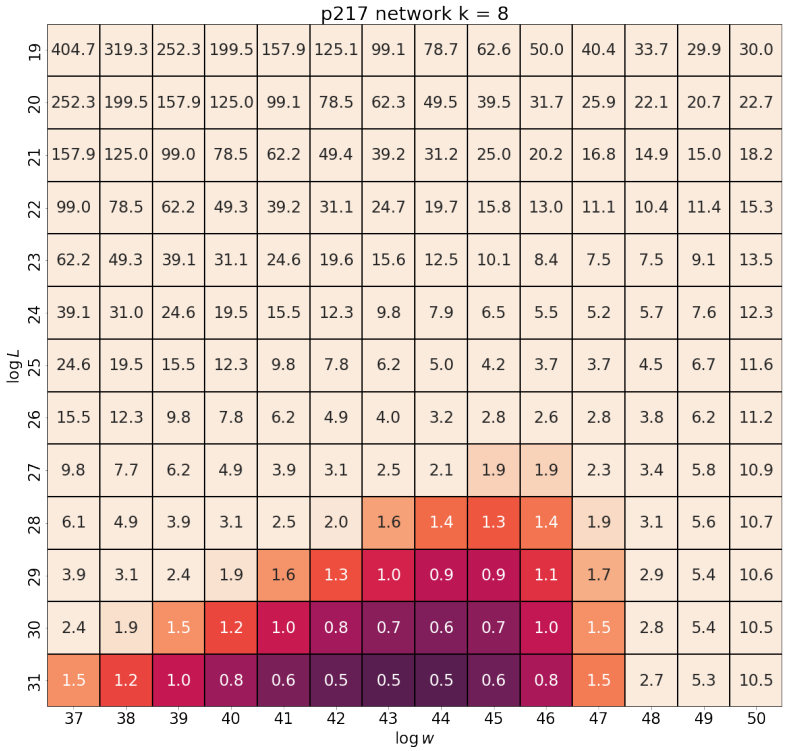
powerful compute nodes.

7.7 SIKEp434

We have also applied our cost model estimates to p434 in the context of the Nation State Attacker of Table 7.3, and we report our findings in Figure 7.3. What the models suggest is that, especially when the attacker capabilities increase, memory becomes the bottleneck, and this increases the cost of attacking the SIPP problem noticeably. In particular, Figure 7.3c shows that breaking SIPP with p434 is approximately 8-bits more costly than previously estimated. We also note that the Nation State model that we are considering here makes a strong assumption, namely that an attacker is able to access an *extremely* large memory at the same bandwidth as that guaranteed by a L1-cache which can generally only store on the order of hundreds of KBs of data. We stress that hardware design for achieving such bandwidth is out of the scope of the work, and out of our area of expertise, but it does seem plausible that practical attacks will be much more memory limited, and as such attacking SIPP might be even harder than what this analysis predict.

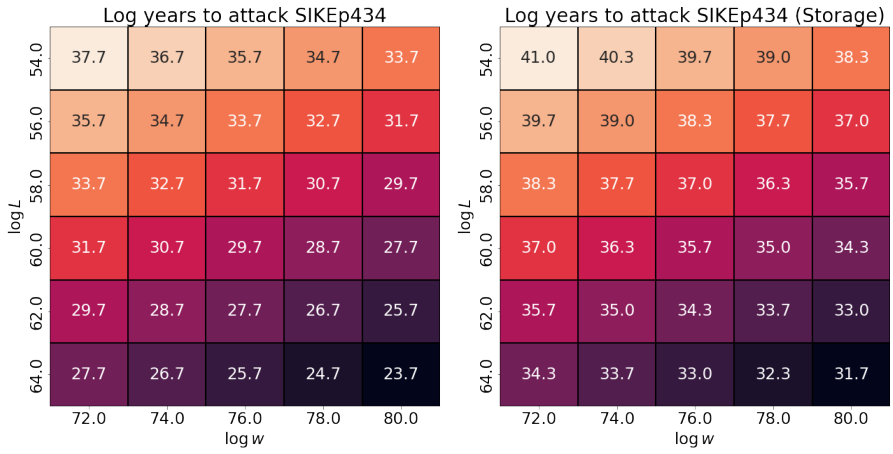


(b) 2 Compute



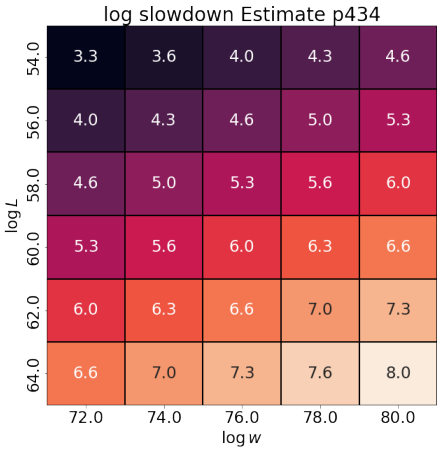
(c) 8 Compute

Figure 7.2: Cost estimates for breaking Problem 7.2 with p217 parameters using the Distributed models of Table 7.4. Estimates in years



(a) Without storage cost

(b) With storage cost



(c) Relative Slowdown

Figure 7.3: Cost estimates for breaking Problem 7.2 with p434 parameters using the Nation State model of Table 7.3. Estimates in (log)-years

Appendix A

Appendix

A.1 Failed Approaches in Saving Memory on Function Switches

Recall that, in Algorithm 9, the function version is switched after βw distinguished points have been collected, which essentially entails replacing f_k to f_{k+1} and (explicitly or implicity) clearing the memory. As we have seen in our analysis, and as mentioned in [vW94]'s analysis, collisions are detected most efficiently when the memory is full, so this operation essentially discards the useful work done so far and requires some time for the algorithm to start finding collisions again. A natural question that we can ask then is the following:

Question A.1 *Is it possible to save some of the distinguished points that are stored in memory between function versions?*

At the start of this work we had investigated a few ways to achieve this aim, and discarded most of them for various reasons. In this appendix we sketch the two most promising approaches and their pitfalls, in the hope to inspire future work in this direction.

First of all, the correctness of Algorithm 5 relies crucially on the fact that the two trials are computed with the same version. So, if (x_0, x_d, d) is mined in function version k and $(x'_0, x_{d'}, d')$ is mined with version k' and $x_d = x_{d'}$, no collision can be found if $k \neq k'$. So, if we want to successfully reuse points mined with the previous function version, it is important that $f_k|_U = f_{k+1}|_U$ for some $U \subseteq f$. Not only that, but since backtracking is only done when a distinguished point is found, also the distinguished check Dist.isDist should agree for at least a subset of Dist.D (ideally the subset of points kept in memory).

Recall from Definition 3.30 that the main reason for switching function versions is to prevent 'unlucky' functions in which the golden collision is

hard to find. The reason for which a golden collision can be hard to find are essentially two. Suppose that (in a given function version) the collision is of the form $x \to i \leftarrow x'$, i.e. $f(x) = i = f(x')$ for $x \neq x'$. Algorithm 8 will find the collision if it samples two trails, one containing x and one containing x'. That is increasing likely if x, x' are on a long 'path' of function evalutions. These long paths exist exactly when x, x' are far from distinguished points and *extremal points*. Extremal points are points in the function graph that have no incoming edges, i.e. a point p is extremal iff $\forall x \in S : f(x) \neq p$.

Our first method is to modify the distinguished test Dist.isDist. When switching function version, we do not in fact change the function (at least, not on every switch), but modify the test. Let $\mathsf{Dist.isDist}_k$ be the test for the current function version and $\widetilde{\mathsf{Dist.isDist}}_{k+1}$ be the distinguished tests described in Algorithm 11. We define

$$\mathsf{Dist.isDist}_{k+1}(x) = \begin{cases} \widetilde{\mathsf{Dist.isDist}}_{k+1}(x), \text{ if } \mathrm{bin}_S(x) \text{ starts with } 1 \\ \mathsf{Dist.isDist}_k(x), \text{ otherwise} \end{cases} .$$

In essence, for a half of S the distinguished test agrees with the old test, while for the other half a new fresh test is used. The effect of this is to partially rerandomize where the distinguished nodes in the function graph, which should make sure that if a golden collision is 'close' to a distinguished points, it will not be with high probability in the next function version. The positives of this technique is that it can be implemented very efficiently, with minimal changes. The negative is that the function graph is not randomized at all, but only where some of these distinguished point lie. As such, extremal points remain the same, and golden collisions near an extremal point remain a problem. A quick estimate also shows that in fact the probability that a random point is extremal is very high ($= \left(1 - \frac{1}{n}\right)^n \approx e^{-1}$), and so we expect that extremal points account for a large part of the 'bad cases'.

The second approach would be to modify partially the function itself. A strategy that we investigated was using a Bloom filter [Blo70]. A Bloom filter is a probabilistic datastructure that allows to test efficiently whether an element was inserted into it, trading space cost with the possibility of false positives. The idea that we sketched was to build incrementally a Bloom filter during the mining operations, and use that to fix the value of f_{k+1} to equal f_k on a chosen subset of S. What makes this challenging is that the value of f_k has to be fixed not only on the distinguished points in memory, but also on the trails computed so far, which makes the number of points needed to be stored very high. For example, some back of the envelope calculation estimated that with p217-like parameters a Bloom filter as large as the memory itself would be needed just to reuse $1/255$ of the points stored in memory, which is clearly not a worthy tradeoff.

We hope that these ideas, here deliberately left vague, will enable future research in this direction, and yield practical improvements in vOW's runtime.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. 8

[ACC+19] Gora Adj, Daniel Cervantes-Vázquez, Jesús-Javier Chi-Domínguez, Alfred Menezes, and Francisco Rodríguez-Henríquez. On the cost of computing isogenies between supersingular elliptic curves. In Carlos Cid and Michael J. Jacobson Jr:, editors, *SAC 2018*, volume 11349 of *LNCS*, pages 322–343. Springer, Heidelberg, August 2019. 81, 83, 84, 89

[BBB+09] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. https://eprint.iacr.org/2009/541. 75

[BDLS20] Daniel J. Bernstein, Luca De Feo, Antonin Leroux, and Benjamin Smith. Faster computation of isogenies of large prime degree. Cryptology ePrint Archive, Report 2020/341, 2020. https://eprint.iacr.org/2020/341. 83, 85

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422426, jul 1970. 96

[Bre80] Richard P. Brent. An improved monte carlo factorization algorithm. *BIT*, 20(2):176–184, June 1980. 26

[CD22] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH (preliminary version). Cryptology ePrint Archive, Report 2022/975, 2022. https://eprint.iacr.org/2022/975. 3, 81, 83

[CLN+20] Craig Costello, Patrick Longa, Michael Naehrig, Joost Renes, and Fernando Virdia. Improved classical cryptanalysis of SIKE in practice. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part II*, volume 12111 of *LNCS*, pages 505–534. Springer, Heidelberg, May 2020. 28, 30, 32, 35, 37, 54, 58, 64, 71, 74, 81, 84

[Cos] Craig Costello. personal communication. 4

[DJP11] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. Cryptology ePrint Archive, Report 2011/506, 2011. https://eprint.iacr.org/2011/506. 83

[DKL+20] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 64–93. Springer, Heidelberg, December 2020. 81, 83, 85

[Dwo15] Morris J. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report, July 2015. 39

[EHL+18] Kirsten Eisenträger, Sean Hallgren, Kristin E. Lauter, Travis Morrison, and Christophe Petit. Supersingular isogeny graphs and endomorphism rings: Reductions and solutions. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 329–368. Springer, Heidelberg, April / May 2018. 81

[Eul] Euler. https://scicomp.ethz.ch/wiki/Euler. Accessed: 2022-08-26. 89

[Fic81] Faith E. Fich. Lower bounds for the cycle detection problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing - STOC '81*. ACM Press, 1981. 26

[GIM] Great internet mersenne prime search. https://www.mersenne.org/. Accessed: 2022-08-26. 3

[GV17] Steven D. Galbraith and Frederik Vercauteren. Computational problems in supersingular elliptic curve isogenies. Cryptology ePrint Archive, Report 2017/774, 2017. https://eprint.iacr.org/2017/774. 81

[HPS14] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *An Introduction to Mathematical Cryptography*. Springer New York, 2014. 15

[JAC+20] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. SIKE. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions. 3, 81

[Jou09] Antoine Joux. *Algorithmic Cryptanalysis*. Chapman and Hall/CRC, June 2009. 26

[JS19] Samuel Jaques and John M. Schanck. Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 32–61. Springer, Heidelberg, August 2019. 81, 84

[Lon22] Patrick Longa. Efficient algorithms for large prime characteristic fields and their application to bilinear pairings and supersingular isogeny-based protocols. Cryptology ePrint Archive, Report 2022/367, 2022. https://eprint.iacr.org/2022/367. 86, 87

[LWS21] Patrick Longa, Wen Wang, and Jakub Szefer. The cost to break SIKE: A comparative hardware-based analysis with AES and SHA-3. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 402–431, Virtual Event, August 2021. Springer, Heidelberg. 89

[MM22] Luciano Maino and Chloe Martindale. An attack on SIDH with arbitrary starting curve. Cryptology ePrint Archive, Report 2022/1026, 2022. https://eprint.iacr.org/2022/1026. 3, 81, 83

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985. 86

[Mur09] Marian Mureşan. *A Concrete Approach to Classical Analysis*. Springer New York, 2009. 22

[Niv04] Gabriel Nivasch. Cycle detection using a stack. *Information Processing Letters*, 90(3):135–140, May 2004. 26

[Pol75] J. M. Pollard. A monte carlo method for factorization. *BIT*, 15(3):331–334, September 1975. v, 23, 25

[PQC] SIDH v3.5.1 (c edition). https://github.com/microsoft/PQCrypto-SIDH. Accessed: 2022-08-26. 85

[Rob22] Damien Robert. Breaking SIDH in polynomial time. Cryptology ePrint Archive, Report 2022/1038, 2022. https://eprint.iacr.org/2022/1038. 3, 81, 83

[SBH20] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. 2020. 72

[Sho94] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. *SIAM Journal of Computing 26*, pages 124–134, 12 1994. 1

[Sil09] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. Springer New York, 2009. 24, 81

[SP00] Michael Shirts and Vijay S. Pande. Screen savers of the world unite! *Science*, 290(5498):1903–1904, 2000. 3

[SSY82] Robert Sedgewick, Thomas G. Szymanski, and Andrew C. Yao. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing*, 11(2):376–390, May 1982. 26

[Sug] Sugaku. https://www.fujitsu.com/global/about/innovation/fugaku/specifications/. Accessed: 2022-08-26. 89

[Tan09] Seiichiro Tani. Claw finding algorithms using quantum walk. *Theoretical Computer Science*, 410(50):5285–5297, nov 2009. 83

[TID21] Monika Trimoska, Sorina Ionica, and Gilles Dequen. Time-memory analysis of parallel collision search algorithms. *IACR TCHES*, 2021(2):254–274, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8794. i, v, 5, 19, 32, 40, 58, 59

[vW94] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with application to hash functions and discrete logarithms. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, and Ravi S. Sandhu, editors, *ACM CCS 94*, pages 210–218. ACM Press, November 1994. i, v, 2, 5, 27, 28, 29, 30, 31, 32, 33, 35, 37, 39, 40, 54, 57, 58, 60, 61, 95