# TwoCats (and SkinnyCat):
# A Compute Time and Sequential Memory Hard Password Hashing Scheme


Bill Cox

waywardgeek@gmail.com

For Consideration in the Password Hashing Competition

Chapel Hill, NC, March 2014

Abstract. Scrypt demonstrated a new way to thwart off-line brute-force password guessing attacks, as the first successful sequential-memory-hard password hashing scheme (PHS). While Scrypt forces an attacker to use significant amounts of memory, custom ASICs can speed up Scrypt's inner loop substantially. Attackers also gain up to a 4X benefit in time*memory cost through a time-memory trade-off (TMTO). With cache timing information, an attacker can attack Scrypt in parallel on GPUs. TwoCats is a new "hybrid" sequential-memory-hard PHS which is compute time hardened through sequential multiply operations, limiting a custom ASIC's speed advantage to about 2X. TwoCats greatly increases the time*memory cost per guess of custom ASIC attacks, while greatly reducing benefits from time-memory trade-offs. TwoCats's "hybrid" design also provides good defense against cache timing attacks, without sacrificing time*memory performance.

# 1 Introduction

TwoCats is a sequential-memory and compute-time hard password hashing scheme (PHS) that maximizes an attackers time*memory cost for guessing passwords. TwoCats:

- Fills and hashes memory rapidly – ½ as fast as memmove
- Hardens runtime through sequential multiplications
- Provides strong defense against GPU, FPGA, and ASIC attacks
- Reduces time-memory trade-off options
- Defends against cache-timing attacks
- Uses parallelism for improved protection (both multithreading and SIMD)
- Offers protection in case of leaked memory
- Is suitable for desktop PCs, web servers, mobile, and embedded applications
- Supports client independent updates
- Supports server relief
- Performs well hashing many GiB of DRAM, or a few KiB of cache

SkinnyCat is a compatible subset of TwoCats designed as a KISS password hashing scheme streamlined for ease of implementation. SkinnyCat takes only a memory cost, hash function, password, and salt, while TwoCats has three levels of APIs from basic to advanced.

In this paper, a memory-hard PHS which does no password derived memory addressing is called "resistant", while those that do so early in hashing are called "unpredictable", and combinations of the two are called "hybrid". A "resistant" memory-hard PHS should be immune to cache-timing attacks, while an "unpredictable" PHS can be attacked on a massively parallel scale once an attacker knows what memory addresses are accessed when hashing a password. A "hybrid" PHS can have some resistance against cache-timing attacks while being as effective against brute-force attacks as an "unpredictable" PHS. Scrypt[2] is a "hybrid" PHS, and has better cache-timing attack resistance than if it were "unpredictable".

Like Scrypt, TwoCats uses a 2-loop architecture, where the first loop is “resistant”, and the second loop is “unpredictable”. Unlike Scrypt, both loops read two prior blocks, hash them together, and append the result to memory, limiting an attacker's TMTO options. Like Script, external memory is written once, and read on average once. A new fast memory hashing function is employed to maximize performance. Compared to single thread SSE optimized Script, single thread SSE optimized TwoCats achieves about a 6X speed increase when hashing 2 GiB of memory. I believe this will encourage users to use more memory per password hash, increasing security.

All else being equal, TwoCats has about a 5X lower time*peak memory cost against cache-timing attacks than a resistant PHS, and about a 7X lower time*average memory cost. An unpredictable PHS may have nearly zero time*memory cost in this case. At the same time, TwoCats will have about a 3.5X higher time*peak memory cost and a 2.3X higher time*average memory cost against brute force guessing attacks. This is discussed more in section 4.

The importance of time*memory cost can be thought of as how much memory an attacker needs to buy per guessing core times how long that memory will be in use per guess. Without compute-time hardening, an ASIC attack would likely be limited in speed by memory bandwidth. A reasonably high end desktop PC might have 25GiB/s of memory bandwidth, while recent graphics cards have up to about 250GiB/s of bandwidth. A custom combined DRAM and hashing chip could greatly increase memory bandwidth per hashing core giving attackers a large advantage.

To counter this, TwoCats employs multiplication based compute-time hardening, forcing ASIC attackers to run no more than about 2X faster than a recent PC, regardless of memory bandwidth. This multiplication chain will force FPGA attackers to run even slower. Modern GPUs are another matter, requiring rapid small unpredictable memory accesses for good defense. TwoCats does in the second loop. GPUs are discussed more in Section 3.

A second line of compute-time hardening defense is L1 cache bandwidth. With high “repetitions”, TwoCats can achieve close to the maximum L1 cache bandwidth supported by modern CPUs. It is difficult for an ASIC attacker to speed up memory access to higher than modern cache times.

TwoCats uses a pluggable hash function architecture, and currently offers Blake2s, Blake2b, SHA256, and SHA512. It runs well on both 32 and 64-bit architectures, with and without SIMD units.

TwoCats is free software. The source code for the reference implementation can be downloaded from:

https://github.com/waywardgeek/TwoCats

## 1.1 Credits

While I deserve credit for this document, my code, and the inspiration for naming a password hashing scheme after my cat[1], most of the good ideas found here come from the authors of Escrypt and Catena[1], as well as several other members of the PHC email list. The cache timing defense strategy is motivated by excellent work and generosity of ideas from Christian Forler, author of Catena, while many of the other good ideas came from Alexander Peslyak, aka Solar Designer, author of Escrypt. I consider them both to be the unofficial primary authors of this work and invite them to be listed in the top spots officially at any time in the future. I hope that in some ways I have helped the eventual winning entry to be better. In particular, any PHC author should feel free to borrow any ideas in TwoCats to improve their own entry. I will feel honored rather than offended by entries copying TwoCats's multiplication chain hardening, or even significant portions of TwoCats's algorithm directly.

TwoCats is a substantial improvement on my prior algorithm, NoelKDF. All of these improvements were suggested by Solar Designer, and include:

- Better GPU defense through randomized sub-block hashing from the “prev” block
- Cryptographically strong hashing between SIMD “lanes” after each block hash. As SolarDesigner put it, there is no excuse for chaining a billion non-cryptographic hashes together.
- Better performance through SSE/AVX2 optimized memory hashing, in parallel with scalar multiplication compute-time hardening

Solar Designer deserves both credit and blame for goading me into this massive rewrite of NoelKDF. His suggestions simultaneously described how to dramatically improve performance, better defend against GPU attacks, take advantage of modern SIMD instructions, and how to fix my billion long non-cryptographic hash chains.

He also deserves credit and blame for goading me to rewrite my original “keystretch” algorithm as NoelKDF. We discussed multiplication based compute hardening on the PHC discussion forum, and NoelKDF was the result. TwoCats also gains it's TMTO resistance while supporting multiple threads from an idea Solar Designer posted, and his posts on multiplication techniques as well as SSE/AVX2 heavily impacted TwoCats, not to mention his generosity in providing access to a Core i7 Haswell machine and high memory bandwidth Sandy Bridge server.

I was motivated by Christian Forler's Catena paper to add several features, including cache-timing defense, garlic, client-independent update, a client/server relief API, and having a pluggable cryptographic hash function. I learned about pebbling algorithms initially from the Catena paper as well. I used his bit-reversal function and combined it with Solar

[1] My wife informs me that naming a PHS after our cat was actually her idea.

Designer's sliding power-of-two window to create the memory access pattern used in the "resistant" first loop.

I also would like to thank the Blake2 authors. Their efficiently optimized SSE code made a good example for me to follow while writing SSE code for TwoCats, and Blake2 significantly improved the performance of TwoCats. Colin Percival, who did such excellent work in Scrypt, deserves credit for the whole approach. Steve Thomas deserves credit for pointing out some bugs and a severe weakness in the first version of NoelKDF, with a modulo-4 attack against it, forcing me to revise and resubmit it. Gary Hvizdak was kind enough to review of the NoelKDF version of this paper, which readers will appreciate. Others on the forum have also been very generous.

Given my inexperience in this field, this submission is only possible because of the generosity of those on the PHC email list.

## 2 Algorithm Specification

SkinnyCat is a subset of TwoCats, and is described first. It differs from the default simple API in TwoCats only by what default parameters are used in hashing, in order to disable many of TwoCat's more advanced features. As a result, compliant SkinnyCat implementation will be far easier to implement than a compliant TwoCats implementation.

### 2.1 SkinnyCat

There is a need for a memory-hard password hashing scheme that is easy to implement, simple, and secure. SkinnyCat attempts to provide this, while being compatible with the full TwoCats algorithm. Many features of TwoCats are disabled in SkinnyCat, but all of these features can be disabled in TwoCats through the extended API to compute the same result. SkinnyCat is offered both as an additional API in TwoCats, and as a stand-alone reference implementation in the twocats/skinnycat directory.

SkinnyCat has only one C API:

```
bool SkinnyCat_HashPassword( TwoCats_HashType hashType, uint8_t hash[32],
                             uint8_t *password,     uint32_t passwordSize,
                             const uint8_t *salt,   uint32_t saltSize,
                             uint8_t memCost,       bool clearPassword)
```

hashType is Blake2s or SHA256. A 32-byte hash is returned. Adding hash types is simple, for any hash function that can generate 256 bit results. The memory hashed is $1024*2^{memCost}$. If clearPassword is set, the password is cleared once the pseudo-random key is derived, and before memory hashing begins.

*SkinnyCat Algorithm:*

```
# Password and salt have length < 256, and memCost <= 30
Inputs: H, uint8 password[], uint8 salt[], uint8 memCost
Output: uint8 hash[32]

# Choose smaller blocklen for smaller memCost values
blocklen = 4096
while blocklen > 8 && memlen/blocklen < 256:
        blocklen >>= 1

# Extract PRK.  Constants are for compatibility w/ TwoCats
PRK[0 .. 7] = decodeLittleEndian(H(passwordSize || saltSize || 0u || blocklen*4 || blocklen*4 ||
                memCost || 0b || 0b || 8b || 1b || 0b || password || salt))

# Allocate memory
memlen = (1024*2^memCost)/4
uint32 mem[0 .. memlen-1]

# Initialize state, separated from PRK by H
uint32 state[8] = hashState(PRK, 0)

# Initialize first block
mem[0 .. blocklen-1] = expand(H, blocklen, state)

# Hash without password dependent addressing
prevAddr = 0
toAddr = blocklen
for i = 1 .. memlen/(2*blocklen)-1:
        a = state[0] # For compatibility w/ TwoCats
        fromAddr = slidingReverse(i)*blocklen
        for j = 0 .. blocklen/8-1:
                for k = 0 .. 7:
                        state[k] = (state[k] + mem[prevAddr++]) ^ mem[fromAddr++]
                        state[k] = ROTATE_RIGHT(state[k], 8)
                        mem[toAddr++] = state[k]
        state = hashState(state, a)

# Hash with password dependent addressing
for i = memlen/(2*blocklen) .. memlen/blocklen-1:
        a = state[0] # For compatibility w/ TwoCats
        fromAddr = (i - 1 - distanceCubed(i, state[0]))*blocklen
        for j = 0 .. blocklen/8-1:
                for k = 0 .. 7:
                        state[k] = (state[k] + mem[prevAddr++]) ^ mem[fromAddr++]
                        state[k] = ROTATE_RIGHT(state[k], 8)
                        mem[toAddr++] = state[k]
        state = hashState(state, a)
addIntoHash(PRK, state)
# One extra hash for compatibility with TwoCat's server relief
output H(H(encodeLittleEndian(state)))
```

Expand is a loop around hashState to fill memory with pseudo-random data derived from state in counter mode.

hashState is just decodeLittleEndian(H(encodeLittleEndian(state || a)). The slidingReverse function is:

```
slidingReverse(i):
        reversePos = reverse(i, numBits-1)
        if reversePos + (1 << (numBits(i)-1)) < i:
                reversePos += 1 << (numBits-1)
        return reversePos
```

The function reverse is just the bit-reversal function. For example reverse(b1011000, 6) is b000110.

distanceCubed simply chooses a random distance back in memory of size $i*rand(0..1)^3$. The integer based aproximation is:

```
v = state[0]
v2 = v*v >> 32
v3 = v*v2 >> 32
distance = (i-1)*v3 >> 32
```

AddIntoHash simply adds the 8 values of state to the 8 values of PRK. After that, two cryptographic hashes are performed in sequence to generate the result. The reason for two rather than just one is that in server relief mode, a TwoCats client will transmit a hashed password to the server without the final hash being applied.

A simple SkinnyCat reference implementation can be found in the twocats/skinnycat directory. SkinnyCat mode can be used with the twocats executable using the -a skinnycat parameter.

## 2.2 TwoCats

Three levels of flexibility are supported in the API: the default TwoCats_HashPassword, a fuller interface called TwoCats_HashPasswordFull, and a bare metal API called TwoCats_HashPasswordExtended. A SkinnyCat API is also provided.

Some users may be confused if there are multiple work parameters, such as memCost and timeCost. To help them succeed, the default API takes only a memCost plus the usual password and salt, and returns a 32-byte hash value. This API differs from SkinnyCat in that several additional security features are enabled, including multiplication compute-time hardening, multithreading, improved GPU defense, support for 512-bit hash functions, and overwriting early memory to provide some protection in case of memory leaks to attackers. Users who are more comfortable specifying time, memory, and parallelism can use the full API, while expert users can control every low level parameter,through the extended API.

These C APIs are:

```
bool TwoCats_HashPassword(          TwoCats_HashType hashType, uint8_t *hash,
                                    uint8_t *password,     uint32_t passwordSize,
                                    const uint8_t *salt,   uint32_t saltSize,
                                    uint8_t memCost,       bool clearPassword)

bool TwoCats_HashPasswordFull(      TwoCats_HashType hashType, uint8_t *hash,
                                    uint8_t *password,     uint32_t passwordSize,
                                    const uint8_t *salt,   uint32_t saltSize,
                                    uint8_t memCost,       uint8_t timeCost,
                                    uint8_t parallelism,   bool clearPassword)

bool TwoCats_HashPasswordExtended( TwoCats_HashType hashType, uint8_t *hash,
                                    uint8_t *password,     uint32_t passwordSize,
                                    const uint8_t *salt,   uint32_t saltSize,
                                    uint8_t *data,         uint32_t dataSize,
                                    uint8_t startMemCost, uint8_t stopMemCost,
                                    uint8_t timeCost,      uint8_t multiplies,
                                    uint8_t lanes,         uint8_t parallelism,
                                    uint32_t blockSize,    uint32_t subBlockSize,
                                    uint8_t overwriteCost,
                                    bool clearPassword,    bool clearData)
```

For all of these functions, these are the restrictions on sizes:

hash is 32 or 64 bytes, depending on the selected hashType
memCost <= 30
timeCost <= 30
multiplies <= 8
1 <= prallelism <= 255
startMemCost <= stopMemCost <= 30
oldMemCost < newMemCost <= 30
32 <= subBlockSize <= blockSize <= 2^20 -- both must be powers of 2
1 <= lanes <= hashType size/4 (for example, 8 for SHA256)

NULL values and 0 lengths are legal for all variable sized inputs. Lengths for NULL values must be 0.

Initially supported hash types are:

- TWOCATS_BLAKE2S
- TWOCATS_BLAKE2B
- TWOCATS_SHA256
- TWOCATS_SHA512

New hash primitives can easily be added to TwoCats.

Preferably, passwords and any other secret data are passed in fixed sized buffers. This insures that the hash primitive can not leak any length information.

Preferably clearPassword is set to true so that the password buffer can be overwritten with 0's at the beginning of hashing rather than by the user afterward.

All of these functions return true on success, and false if there is a memory allocation error.

Each increment of memCost doubles difficulty. The memory hashed = 2^memCost KiB. For the full and extended APSs, the inner loop of hash blocks is repeated 2^timeCost times, and the resulting hash block is written in the last iteration. This supports L1 cache bandwidth runtime hardening, since an attacker will have difficulty building significantly faster L1 caches, even on a custom ASIC.

### 2.3 The Extended API

The extended API is for those who know what they are doing. To help, an API for guessing good parameters is provided to help choose memCost, timeCost, multiplies, and lanes for the user's machine.

The data parameter can be any application specific data, such as a secondary key or application name, or a concatenation of various data. It is treated as sensitive data, just like the password.

startMemCost is normally equal to stopMemCost, unless a password hash has been strengthened using TwoCats_UpdatePassword (client independent update).

stopMemCost is the main memory hashing difficulty parameter, which causes 2^stopMemCost KiB of memory to be hashed. Each increment doubles memory hashed.

timeCost causes the inner loop to repeat 2^timeCost times, repeatedly hashing blocks that most likely fit in on-chip cache. It can be used to increase runtime for a given memory size, and to reduce DRAM bandwidth while increasing cache bandwidth. For memory sizes large enough to require external DRAM, it is ideally set as high as possible without increasing runtime significantly. For memory sizes that fit in on-chip cache, timeCost needs to be set high enough to provide the desired runtime security.

multiplies is used to force attackers to run each guess as almost as long as you do. It should be set as high as possible without increasing runtime significantly. 2 is a reasonable default for hashing memory sizes larger than the CPU cache size, 1 is reasonable for L2/L3 cache sizes, and 0 may be required for L1 cache sizes. For CPUs without hardware multiplication or on-chip data cache, multiplies should be set to 0 to aid in memory bandwidth defense.

lanes is used to make use of SIMD parallelism available on the CPU, such as SSE2 and AVX2. Older CPUs without any SIMD unit should set lanes to 1. Sandy Bridge and Ivy Bridge Intel processors run best with lanes set to 4. Haswell runs best with lanes set to 8.

Parallelism is the number of threads used in parallel to hash the password. A reasonable number is half the CPU cores you expect to have idle at any time, but it must be at least 1. Each thread hashes memory so fast, memory bandwidth will likely max out with only 2 threads. Higher values can be used on multi-CPU servers with more than two memory banks to increase password security.

OverwriteCost determines how much early memory to overwrite. In case memory is flushed to disk for some reason, unless this happens during the very first part of hashing an attacker will have to still compute the early memory for each guess. 1 means hash as much memory as used for hashing, and each increment reduces this by half. The default of 6 leads to about a 3% increase in runtime, and the first 3% of memory is overwritten.

If clearPassword is true, the password is set to 0's early in hashing, and if clearData is set, the data input is set to 0's early in hashing.

### 2.4 Additional APIs

A system administrator can update an existing password hash to a more difficult level of memCost using TwoCats_UpdatePassword:

```
bool TwoCats_UpdatePassword(TwoCats_HashType hashType, uint8_t *hash,
        uint8_t oldMemCost, uint8_t newMemCost, uint8_t timeCost,
        uint8_t multiplies, uint8_t lanes, uint8_t parallelism, uint32_t blockSize,
        uint32_t subBlockSize)
```

Server relief is supported through an extended password hashing API that splits the hash into a client-side compute intensive hash called TwoCats_ClientHashPassword, and a server-side low-CPU effort hash called TwoCats_ServerHashPassword:

```
bool TwoCats_ClientHashPassword(TwoCats_HashType hashType, uint8_t *hash,
        uint8_t *password, uint32_t passwordSize, const uint8_t *salt,
        uint32_t saltSize, uint8_t *data, uint32_t dataSize, uint8_t startMemCost,
         uint8_t stopMemCost, uint8_t timeCost, uint8_t multiplies, uint8_t lanes,
        uint8_t parallelism, uint32_t blockSize, uint32_t subBlockSize,
        uint8_t overwriteCost, bool clearPassword, bool clearData)

bool TwoCats_ServerHashPassword(TwoCats_HashType hashType, uint8_t *hash,
        uint8_t hashSize)
```

Users can find decent parameter settings for their machine for a given desired runtime and maximum memory using the TwoCats_FindCostParameters API:

```
void TwoCats_FindCostParameters(TwoCats_HashType hashType,
        uint32_t milliSeconds, uint32_t maxMem, uint8_t *memCost,
```

uint8_t *timeCost, uint8_t *multplies, uint8_t *lanes)

## 2.5 TwoCats Upgrades

TwoCats provides several new features over SkinnyCat (with credits for idea sources):

1. Introduce a sequential multiplication chain to compute-time harden the algorithm against ASIC attacks (Solar Designer & Bill Cox)
2. Introduce small unpredictable reads to thwart GPU attacks (Solar Designer)
3. Introduce repetitions to increase compute time and L1 cache bandwidth for memory-limited systems (Solar Designer & Bill Cox)
4. Introduce "lanes" parameter to best take advantage of SIMD unit parallelism (Solar Designer)
5. Introduce thread level parallelism to take advantage of multi-core CPUs (Scrypt)
6. Introduce inter-thread memory hashing to thwart TMTOs (Solar Designer)
7. Introduce client-independent update (Catena)
8. Protect against memory leaking to attackers by overwriting early memory (Bill Cox)

Every one of these enhancements either increases speed or improves security. Since the security of a memory-hard PHS can be in part measured by it's time*memory cost, increasing speed also increases security, since it allows us to hash more memory. Similarly, increasing CPU time through repetitions also increases security.

## 2.6 Multiplication-Hard Hash Function

Speeding up multiplication on custom ASICs versus modern CPUs is difficult, and unlikely to result in a significant speed-up. In comparison, the Salsa20/8 hash function used in Scrypt[2] is likely to run at about 1 or 2 nanoseconds per Salsa20/8 round per core on a 28nm custom ASIC. Solar Designer's SSE optimized version of scrypt on my development machine[2] hashes 2GiB twice in 2.71 seconds, which is about 30X slower. Salsa20/8 has only 16 levels of ADD/XOR logic per 32-bit register, of which there are 16, making about as complex as 16 Booth-encoded 32x32 → 64 multipliers running in parallel.

In contrast, TwoCats is hardened against ASIC attacks through the use of a multiplication-hard hash function, which cannot be substantially sped up, even on a carefully optimized custom ASIC. Each iteration depends on the previous, and must be computed with one sequential multiply followed by one sequential XOR. A lower bound on the runtime is:

$$T(memlen) \geq memlen \times multTime \times multsPerByte$$

On my development machine, a 32x32 → 64 multiply operation has latency multTime = 0.88ns. Hashing 2GiB of data this way with 8 serial multiplications per 32-bytes would require a minimum of 0.88ns*231*8/32 = 0.47 seconds. TwoCats performs this calculation in 0.83 seconds, which corresponds to being 57% multiplication compute-time hardened.

**Definition: Multiplication-Hard Hash Function**

[2] My "development machine" is my son's 3.4 GHz quad-core i7-3770 Manjaro Linux MineCraft server.

*A multiplication-hard hash function is a hash function that sequentially computes values using no more than a 1-to-3 ratio of sequential multiplication operations to simple operations, where simple operations are the usual single-cycle ALU operations such as: add, sub, XOR, AND, OR, complement, increment, decrement, and shift/rotate.*

The reason for the choice of 1-to-3 is that current advanced Intel, AMD, and ARM processors have either 3 or 4 clock cycle latencies to compute a 32x32->64 multiplication, while all of the simple operations are computed in 1. This means in a multiplication-hard hash function, it should be possible to spend at least 50% of the compute time on multiplications, assuming other operations can execute in parallel. TwoCats has a multiplication to simple operation ratio of 1-to-1. It does one multiply and one XOR sequentially, twice in every loop.

The TwoCats hash function below implements upgrades 1-4 mentioned above:

```
hashBlocks(H, uint32 state[], uint32 mem[], uint32 blocklen, uint32 subBlocklen,
            uint64 fromAddr, uint64 prevAddr, uint64 toAddr,
            uint8 multiplies, uint32 repetitions, uint8 lanes):
    numSubBlocks = blocklen/subBlocklen
    a = state[0]
    b = state[1]
    c = state[2]
    d = state[3]
    for r = 0 .. repetitions-1:
        for i = 0 .. numSubBlocks-1:
            randVal = mem[fromAddr]
            p = prevAddr + subBlocklen*(randVal & (numSubBlocks – 1))
            for j = 0 .. subBlocklen/lanes - 1:

                # Compute the multiplication chain, preferably in CPU registers
                for k = 0 .. multiplies:
                    a ^= (uint64_t)b*c >> 32
                    b += c
                    c ^= (uint64_t)a*d >> 32
                    d += a

                # Hash lanes of memory, preferably in the SIMD unit
                for k = 0 .. lanes-1:
                    state[k] = (state[k] + mem[p++]) ^ mem[fromAddr++]
                    state[k] = (state[k] >> 24) | (state[k] << 8)
                    mem[toAddr++] = state[k]

    H.HashState(state, v)
```

On machines supporting SSE or AVX2, the multiplication computations occur in CPU registers, and run in parallel with the memory hashing loop, which is done in the SSE/AVX2

unit. This hash function can compute from 0 to 16 32x32->64 multiplies per inner loop, in parallel with hashing 4*lanes bytes of memory. A block is 4096 32-bit values by default. When this function is called in the first “resistant” loop, subBlocklen is set to blocklen, since predicable sub-block hashing gains us little. In the second “unpredictable” loop, subBlocklen is 16 which causes modern GPUs some difficulty. After the new hashed block is written to memory, H is used to mix bits between the state registers.

### 2.7 The “Resistant” loop

To provide stronger resistance against offline brute-force guessing attacks, TwoCats runs an “unpredictable” second loop called “hashWithPassword”.

```
hashWithoutPassword(H, uint32 state[], uint32 mem[], uint32 p, uint64 blocklen,
        uint32 blocksPerThread, uint32 multiplies, uint32 repetitions,
        uint8 lanes, uint32 parallelism, uint32 completedBlocks):

    uint64 start = blocklen*blocksPerThread*p
    uint32 firstBlock = completedBlocks
    if completedBlocks == 0:
        # Initialize the first block of memory
        mem[start .. start + blocklen-1] = H.ExpandUint32(blocklen, state)
        firstBlock = 1

    # Hash one "slice" worth of memory hashing
    numBits = 1
    for i = firstBlock .. completedBlocks + blocksPerThread/SLICES - 1:
        while 1 << numBits <=i:
            numBits++

        # Compute the "sliding reverse" block position
        reversePos = reverse(i, numBits-1)
        if reversePos + (1 << (numBits-1)) < i:
            reversePos += 1 << (numBits-1)
        uint64 fromAddr = blocklen*reversePos

        # Compute which thread's memory to read from
        if fromAddr < completedBlocks*blocklen:
            fromAddr += blocklen*blocksPerThread*(i % parallelism)
        else:
            fromAddr += start

        uint64 toAddr = start + i*blocklen
        uint64 prevAddr = toAddr - blocklen
        hashBlocks(H, state, mem, blocklen, blocklen, fromAddr, prevAddr, toAddr,
            multiplies, repetitions, lanes)
```

This function is called SLICE/2 number of times, where SLICE is 4. This is because we execute this function from “parallelism” threads (when pthreads are enabled). It is desirable to allow one thread to hash another thread's memory into it's hash block in order to force attackers to keep all the thread memory loaded at once, rather than running them sequentially with only one thread's memory loaded at a time. Since threads do not run at the same speed, memory for each thread is broken into SLICE slices, and we run “parallelism” threads in parallel to generate one slice of hashed memory, and then join the threads.

This function finds the prior block to hash using the “sliding power of 2” window suggested by Solar Designer, with a bit-reversal addressing pattern from Catena. This tested most resistant to cache timing attacks in my benchmarks. This is discussed more in the Efficiency section below.

### 2.8 The “Unpredictable” Loop

The unpredictable loop follows execution of the resistant loop. This loop provides TwoCats it's “sequential-memory-hardness”. Its primary purpose is providing solid defense against offline brute-force guessing attacks. Because an attacker cannot know what address will be accessed in the immediate future, he risks considerable recomputation if he tries to implement a TMTO attack with ¼ memory or less. Also, the unpredictability of this loop keeps him from using parallel cores to recompute values before they are needed. A “resistant” PHS has a simple “free” ½ memory TMTO which results in zero recomputations, and typically ¼ memory TMTO attacks have few recomputations. This loop gives an attacker no such free ride.

The function is:

```
hashWithPassword(H, uint32 state[], uint32 mem[], uint32 p, uint64 blocklen,
                 uint32 subBlocklen, uint32 blocksPerThread, uint32 multiplies,
                 uint32 repetitions, uint8 lanes, uint32 parallelism, uint32 completedBlocks):

    uint64 start = blocklen*blocksPerThread*p;

    # Hash one "slice" worth of memory hashing
    for  i = completedBlocks .. completedBlocks + blocksPerThread/SLICES - 1:

        # Compute rand()^3 distance distribution
        uint64 v = state[0]
        uint64 v2 = v*v >> 32
        uint64 v3 = v*v2 >> 32
        uint32 distance = (i-1)*v3 >> 32

        # Hash the prior block and the block at 'distance' blocks in the past
        uint64 fromAddr = (i - 1 - distance)*blocklen

        # Compute which thread's memory to read from
        if fromAddr < completedBlocks*blocklen:
```

```
            fromAddr += blocklen*(state[1] % parallelism)*blocksPerThread
        else:
            fromAddr += start

        uint64 toAddr = start + i*blocklen
        uint64 prevAddr = toAddr - blocklen
        hashBlocks(H, state, mem, blocklen, subBlocklen, fromAddr,
            prevAddr, toAddr, multiplies, repetitions, lanes)
```

This function differs from the simplified version of TwoCats in that instead of selecting a prior block to hash uniformly, it biases selection to more recently generated blocks. If uniform were used, the average distance would be i/2. With this cubed distribution, the average is i/4, which is still very good, and at the same time the number of very short edges is greatly increased. Short edges cause an attacker more recomputations than long edges if he has dropped blocks from memory. This is discussed more in the Efficiency section.

The average edge length from node *i* is computed as:

$$l = i \int_0^1 x^3 \, dx = \frac{1}{4} i$$

### 2.9 Thread Management

The function hashMemory initializes thread states, and launches the threads once per slice, first on hashWithoutPassword, and then hashWithPassword. In comparison, pepper can take advantage of multiple CPU cores, but not without giving an attacker a simple TMTO. The function is:

```
hashMemory(H, uint32 hash32[], uint32 mem[], uint8 memCost, uint8 timeCost,
        uint8 multiplies, uint8 lanes, uint8 parallelism, uint32 blockSize,
        uint32 subBlockSize, uint32 resistantSlices):

    memlen = (1024/4) << memCost
    blocklen = blockSize/4
    subBlocklen = subBlockSize/4
    blocksPerThread = TWOCATS_SLICES*(memlen/
        (TWOCATS_SLICES * parallelism * blocklen))
    repetitions = 1 << timeCost

    # Initialize thread states
    uint32 states[H.len*parallelism] = H.Expand32(H.len*parallelism, hash32)

    for slice = 0 .. SLICES - 1:
        for p = 0 .. parallelism-1:
            if slice < resistantSlices:
                state = states[p*H.len .. (p+1)*H.len-1]
                hashWithoutPassword(H, state, mem, p, blocklen,
                    blocksPerThread, multiplies, repetitions, lanes,
```

```
                    parallelism, slice*blocksPerThread/SLICES)
        else:
            hashWithPassword(H, state, mem, p, blocklen,
                    subBlocklen, blocksPerThread, multiplies, repetitions,
                    lanes, parallelism, slice*blocksPerThread/SLICES)

    # Apply a crypto-strength hash
    addIntoHash(H, hash32, parallelism, states)
    hash32 = H(hash32)
```

This heuristic hides the complexity of choosing reasonable values for blocklen and blocksPerThread, from the user, simplifying the API.

The function addIntoHash simply adds the state vectors computed during hashing hash32. Expand then computes the output hash from hash32 using a cryptographically secure hash function.

### 2.10 Garlic (aka memCost)

The final upgrade is adding Catena style “garlic” and protecting against memory leaks. Catena's “garlic” which I implemented in TwoCats, enables a feature called “client independent update”. What this does is make it possible for a system administrator to increase the hashing difficult of any password in his database at any time, without having to know the password. Basically, it's just a wrapper around hashMemory which calls it over and over with increasing memCost. If an administrator wants to increase difficulty, he can simply call TwoCats with startMemCost == oldMemCost, and stopMemCost == desiredMemCost. The output from hashMemory is the input to the next call to hashMemory with 1 higher memCost.

This loop also is a good place to deal with a major security issue in memory-hard PHSs: attackers have a much higher chance of gaining access to leaked memory, enabling them to abort incorrect password guesses early. Because memory-hard PHSs are so memory hungry and run for so long, they can segv, get swapped to disk during hibernation or if memory is full, the process might core dump, or a compromised peripheral might have enough time to detect the password hashing in progress and somehow gain access to RAM data, possibly through DMA.

To reduce the chance of leaking the most critical earliest memory to attackers, TwoCats overwrites 1/32nd of memory, in increasing levels of memCost. This causes the first 1KiB to be hashed, then overwritten, then 2 KiB, then 4 KiB, and so on until 1/64th of memory is filled. At this point, TwoCats jumps ahead to the user's specified startMemCost and applies levels of garlic the same way Catena does. This slows down the algorithm about 3%, but the extra protection warrants this performance hit.

The TwoCats main hash function is:

```
bool TwoCats(H, uint32 hash32[], uint8 startMemCost, uint8 stopMemCost,
             uint8 timeCost, uint8 multiplies, uint8 lanes, uint8 parallelism,
             uint32 blockSize, uint32 subBlockSize, uint8 overwriteCost):

    uint32 mem[(1024 << stopMemCost)/4]

    # Iterate through the levels of garlic.  Throw away some early memory to reduce the
    # danger from leaking memory to an attacker.
    for i = 0 .. stopMemCost-1:
        if i >= startMemCost || i < overwriteCost:
            if (1024 << i)/(parallelism*blockSize) >= TWOCATS_SLICES:
                resistantSlices = SLICES/2
                if i < startMemCost:
                    resistantSlices =  SLICES
                hashMemory(H, hash32, mem, i, timeCost, multiplies, lanes,
                    parallelism, blockSize, subBlocksSize, resistantSlices)
            if i != stopMemCost:
                # Not doing the last hash is for server relief support
                hash32 = H(hash32)

    # The light is green, the trap is clean
    return true
```

This simple outer wrapper provides support for client independent updates, server relief, and memory leak protection.

## 3 Security Analysis

Except for TwoCat's complexity, there are reasons to believe its design secure. Analysis of SkinnyCat may benefit the more difficult analysis of TwoCats. The analysis of TwoCats security will be covered in detail in this section.

There are many security measures by which to judge an algorithm. Breaking them down, the first category is “critical” features, which if circumvented would be disastrous for any passwords still protected by the algorithm. Since this is an analysis of a memory-hard PHS, critical features also include memory-hard specific features that if compromised would render the algorithm susceptible to massively parallel attacks without the memory cost. Then comes “important” features, which would somehow limit the effectiveness of an algorithm if circumvented.

Critical features include:

- The derived key cannot be reversed to reveal the password with significantly fewer guesses than brute-force
- “The hash should behave randomly with respect to any of its input” - PHC FAQ

- The time*memory cost cannot be significantly reduced, forcing an attacker to devote a comparable amount of resources as the defender for each password guess

In this context, “significantly reduced” and “significantly fewer” are fuzzy, but striving for no more than a 10X cost advantage per guess for an attacker is a good goal for a memory-hard PHS. Any algorithm showing 10,000X or more cost advantage for an attacker should be avoided, as better algorithms exist. This includes defense against attackers using GPUs, FPGAs, and custom ASICs.

Important features include:

- Low flexibility in TMTO attacks
- Cache timing attack resistance, through password independent memory addressing
- Defense against modern GPU architectures
- Simplicity, enabling better cryptanalysis and more secure implementations
- Resistance to memory leak attacks such as when memory is written to swap

TwoCats addresses all of these needs other than simplicity, while SkinnyCat loses some resistance to GPU attacks and memory leaks, and has lower compute-time hardness, but it gains simplicity.

### 3.1 Key Reversal and Derived Key Randomization

TwoCats and SkinnyCat use an extract function motivated from HKDF[9] at the start of the algorithm to derive a cryptographic pseudorandom key from the input data. Every single input, assuming startMemCost == stopMemCost, is passed to extract, including sizes of all variable sized data, making the initial key derivation in TwoCats “strongly secure”, according to Yao and Yin's definition[10].

```
hash32 = extract(hashType, saltSize || passwordSize || password || dataSize || data ||
          startMemCost || timeCost || multiplies || parallelism || blockSize ||
         subBlockSize)
```

When startMemCost < stopMemCost, the first iteration of memory cost is strongly secure, and the rest are simply client-independent updates made without knowledge of the password, making them more computationally difficult. Attacks such as the chosen-c attack on PBKDF2 are not possible when all the inputs are hashed upfront. Also, input collisions such as PBKDF2 has when adding additional 0's to short passwords, or when hashing long passwords, cannot happen in TwoCats because the password length is hashed upfront

The initially supported hash functions are Blake2s, Blake2b, SHA256, and SHA512. There is no known way of reversing the key for any of these which is easier than guessing the input to the secure hash function which generated the output. In addition, H is called again at

the end of the algorithm with only the final hash as input. In between, memory hashing and multiplication-intensive hashing strive to lose little entropy. The result is isolated from the initial key by an application of the cryptographic hash, and at the end is added into it, after which the result is hashed one more time to insure good isolation, and one more time to support server relief.

### 3.2 Memory Hardness

Scrypt appears to have a hardened time*memory cost of about ¼ for an attacker vs defenders. To achieve this, an attacker reduces memory by a high factor, and increases his compute effort by a bit more than ¼ of this factor. No better TMTO attack is currently known against Scrypt.

TwoCats starts with this level of time*memory cost hardening and enhances it by limiting an attacker's TMTO options. Each hashed memory block depends on the previous as in Scrypt, and also on a randomized prior block, frustrating the DAG-cut attack used against Scrypt by increasing the data in any DAG cut to a number comparable to the graph size – the mid-cut is about 18% of the number of nodes in the computation DAG. A feasibleTMTO attack has been identified, which reduce time*cost by about 10%, though the complexity of memory tracking in this attack limits it's usefulness to an attacker. In this attack, the fact that memory locations near the end of each loop have low probability of being accessed is used to save some memory with a low recomputation cost. The TMTO hardness of TwoCats is further discussed in the Efficiency Analysis section.

### 3.3 Secure Memory Hash Function

It may be simpler to focus on two separate functions performed in hashBlocks: memory hashing, and compute-time hardening. Memory hashing is not impacted by the multiplication chain computation, and when this is removed, we get a simpler memory-hashing only loop that writes the same data to memory:

```
for i = 0 ..  numSubBlocks-1:
        randVal = mem[f]
        p = prevAddr + subBlocklen*(randVal & mask)
        for j = 0 .. subBlocklen/8 – 1:
                for k = 0 .. 7:
                        state[k] = (state[k] + mem[p++]) ^ mem[fromAddr++]
                        state[k] = ROTATE_LEFT(state[k], 8)
                        mem[toAddr++] = state[k]
```

The memory hash function is designed to be fast, SIMD friendly, lose negligible entropy, and to force an attacker to compute mem[0 .. i-1] before he can compute mem[i]. It uses addition, rotation, and XOR (ARX)[7]. These are the primitives behind several popular fast cryptographically strong hash functions, including SHA-3 competitors Skein, Blake, CubeHash, and Salsa20[7]. These operations are not combined in a manner to insure hashing cryptographically secure, but they still have to be computed, which is what counts.

The memory being hashed into the state is separated by at least one application of the cryptographically strong hash function H, and many rounds of memory block hashing are applied. It is important that the hash function lose little entropy, which is assured by the reversible ARX operations. The data written to memory passes the dieharder tests, and the number of 32-bit collisions in the data written to memory is as expected.

It is critical to force an attacker to compute each memory location's value, and to have no short-cuts that allow mem[i] to be computed without first computing mem[0..i-1]. Confidence that this is the case comes from the similarity of this hash function to other ARX hash functions.

### 3.4 Hardened Compute Time Cost

TwoCats introduces the concept of multiplication based compute-time hardening of a PHS. By sequentially calling a hash function dominated by the runtime of serial multiplications, an attacker is forced to spend a comparable time computing the TwoCats hash function as the defender, even on a custom ASIC.

What matters in the multiplication hardened hash function is that the attacker needs to be forced to compute the multiplications sequentially, and there should be as little non-multiplication computation as possible. The interesting portion of the hashing loop for multiplication hashing:

```
a = state[0]
b = state[1]
c = state[2]
d = state[3]
for i = 0 ..  numSubBlocks-1:
        for j = 0 .. subBlocklen/8 – 1:
                for k = 0 .. 7:
                        a ^= (uint64_t)b*c >> 32
                        b += c
                        c ^= (uint64_t)a*d >> 32
                        d += a
hashState(state, v)
```

The variables a through d are allocated in regular CPU registers, while the state variables are loaded into SSE/AVX2 registers. The memory hashing function runs very well on SSE and AVX2 SIMD units in Sandy Bridge, Ivy Bridge, and Haswell CPUs. At the end of the loop, the resulting value of a and the state is hashed with Blake2s.

If addition were used rather than bitwise XOR, then the multiplications could be distributed, and parallel computation could be used to accelerate computation. Since XOR does not distribute with multiplication, the multiplications must be computed sequentially.

### 3.5 Cache Bandwidth Hardening

In addition to compute-time hardening through sequential multiplies, TwoCats has a timeCost parameter, which causes it's 16 KiB block hashes to be repeated 2^timeCost times, but results are only written on the last iteration, which works well with write-through caches. With high timeCost values, bandwidth to L1 cache can be very high. As benchmarks show, about 79 GiB/s can be sustained to L1 cache using this parameter with one thread on Intel Haswell CPUs, or about 36% of the theoretical maximum, and 57 GiB/s on Ivy Bridge, or 52% of the maximum possible. This represents yet another barrier, in addition to multiplication compute-time hardening, which will make it difficult for ASIC attackers to significantly speed up attacks on TwoCats.

### 3.6 GPU, FPGA, and ASIC Attack Resistance

GPUs naturally run with long instruction latency, and the multiplication chain will perform only a bit worse than an addition chain. There are two approaches to defeating graphics card attacks. First, use a lot of memory. Scrypt runs at near parity with GPUs at around 4MiB, and at 1GiB, Scrypt is not practical to attack with GPUs. However, Bcrypt runs at parity with GPUs in only 4KiB of memory. The difference is the way Bcrypt does many small unpredictable reads in series, which GPUs do poorly. TwoCats includes similar small random reads in the second loop (but not the first), which by default is 64 bytes compared to Bcrypt's 16. Exactly where TwoCats achieves memory parity with GPUs has not yet been benchmarked, but I expect it to be in between that of Scrypt and Bcrypt.

Against ASICs, TwoCats deploys a multiplication computation time hardened inner loop. A custom ASIC in the same technology node as a modern CPU should not have any advantage in the speed of multiplication. TwoCats's inner loop is typically about 50% dominated by multiplication, meaning an ASIC attacker has to spend about 50% as long as the defender computing the hash. This is a dramatic improvement over prior algorithms that only use addition, rotation, and XOR in their hashing loops.

FPGAs will fare even worse from the multiplication computation time hardening, due to slower clock cycles, and multiple clocks per 32-bit multiplication.

### 3.7 Memory Leak Attacks

The single greatest weakness in memory-hard PHSs is likely what happens when an attacker gains accessed to hashed memory. Writing password derived data to large amounts of memory greatly increases the odds that password derived data will be leaked to an attacker, through swap, hibernation, memory recycling without reinitialization, core dumps, DMA transfers to a compromised device, etc. With this data, an attacker can abort incorrect password guesses early, and mount a massively parallel attack with little time and memory per guess.

TwoCats provides some resistance against memory leaks by overwriting early hashed memory. It repeatedly hashes memory with the “resistant” hash algorithm, starting with just

256 blocks, and increases the size by a factor of two until it reaches at least 1/64th of memory. After that, it starts from scratch, doing the “resistant” first loop on the first half of memory, followed by the “unpredictable” second loop on the second half of memory. The compute overhead is roughly 1/32 or about 3%.

This algorithm initially presents a minimal attack surface to an attacker. If memory leaks after this phase has completed, an attacker will potentially gain a 2048X time*cost reduction, or about 11 bits of strength. With a TMTO additional attack, memory can be cut another 4X, leading to a 13 bit combined loss in password strength. I feel this is a reasonable trade-off for a 3% increase in runtime.

### 3.8 Cache-Timing-Attack Resistance

The first loop does no memory lookups that in any way depend on the password. If a cache-timing signature is leaked to an attacker, it could be used to abort an incorrect password guess at the start of the second loop. The first loop takes about half of the runtime, so an attacker gains a 2X improvement in guessing speed. Like all the cache-timing resistant algorithms I tested, the first loop has a low recomputation penalty for attackers using about ¼ memory compared to the DAG size. The “sliding reverse” algorithm used showed a 3X recomputation penalty for an attacker using only ¼ memory, which was the best defense of all algorithms tested. Assuming there are no recomputations required at ¼ memory gives an attacker another 4X memory reduction, in addition to the 2X memory reduction for not having to compute the second loop. Combining these factors, I estimate an attacker gains a 16X reduction in time*memory cost given cache timing information.

A significant problem with cache-timing resistant algorithms is that more CPUs can be used to recompute missing values from memory faster. The defense is to force an attacker to pay a very high recomputation penalty. An attacker with 1% node coverage of a sliding-reverse DAG will find while pebbling the second half, that every 2% he covers requires that he re-pebble 25% of the graph. This is because the bit-reverse pattern causes N/50 evenly spaced nodes to be accessed, and only half of them could have been pebbled by N/100 nodes, requiring the whole interval before it to be completely repebbled. This will cause an attacker to repebble 25% of the graph 25 times, or 2,500 times. In reality, because the intervals that get repebbled also depend on other unpebbled nodes and so on, a significant portion of the entire graph will need to be recomputed for every move in the last half, leading to a considerably worse penalty. Finding a better lower bound remains an open problem.

### 3.9 Weaknesses

I recommend that users of TwoCats use fixed-length password buffers initialized to 0's, and to reject passwords with non-printable characters, when that makes sense for their application, or which do not fit in the buffer. This insures no branching or memory addressing specific to the password length ever happens. Otherwise, as with HKDF and PBKDF2, there might be a

chance an attacker could gain password length knowledge do to calling memcpy on the passoword.

It is possible for an attacker to make better use of TwoCats leaked memory than if cryptographically secure data were written to memory. In particular, an attacker might more easily determine which data block is the first block, minimizing his time to abort an incorrect password guess. He might also be able to search swap or data on an SSD specifically for TwoCats hash data, which would be more difficult if the data written to memory were cryptographically indistinguishable from random. The decision to accept this weakness in TwoCats is deliberate: the alternative is to slow down memory hashing considerably. Slowing down the algorithm by even 2X would give an attacker with a leaked password database a potential 4X faster cracking time, since he only needs ½ the memory per core, and ½ the time per core. Because time*memory security with a memory-hard PHS increases as the square of the hashing speed, this was an easy decision to make. However, TwoCats overwrites memory early on, providing some defense even if memory is leaked. If the first 1/64th of memory is overwritten before an attacker gains access, he will gain only about an 11-13 bit advantage in cracking the password.

On older CPUs, and some modern embedded CPUs, multiplication does not run in constant time, leading to the possibility of a side-channel timing attack. On such CPUs, it is possible to set mulitiplies to 0, both to avoid this timing leak, and because it is unlikely the CPU can execute the multiplication in parallel with memory hashing, thus lowering security for a given runtime. If multiplications are used, the runtime will reflect average multiplication times, which do not strongly depend on the password. If an attacker can count the exact number of clock cycles required to hash a password, he may still gain an advantage. In that case, however, any algorithm that calls memcpy on the password, such as HKDF and PBKDF2, already has a severe weakness.

My lack of understanding of GPU attacks may have contributed to weaknesses against current GPU architectures relative to algorithms such as Bcrypt. More benchmarking will be required to determine TwoCats's strength against GPUs. However, small (64-byte) unpredictable reads are done in the inner hashing loop in the second half of the algorithm, which I believe makes it harder for GPUs to be effective.

With cache timing information, an attacker gains an estimated 16X lower peak time*memory cost (though only 5X worse than a resistant PHS). However, this relies on the assumption that the sliding-reverse window DAGs cannot be efficiently pebbled with less than about ¼ memory versus the first-loop DAG size. The TwoCats first loop uses the DAG architecture that tested most resistant in this respect, but only upper bounds on recomputation penalties have been established. There is some risk that better pebbling algorithms can be found that make a $1/8^{th}$ memory attack practical, but this would only result in one bit lost in resistance.

Against simple brute-force attacks, TwoCats is resistant to TMTOs. The best TMTO found so far reduces the time*memory cost by 10% by keeping every other value in memory near the end of both loop ranges, since this memory is not likely to be read. There is some risk that better TMTOs can be found with more clever memory coverage of computed values. However, the 4X TMTO improvement we see attackers use against Scrypt cannot be used against TwoCats, so the security risk here is low.

While I have considerable experience with ASIC and FPGA architectures, I have little cryptography experience. Extensive expert review of TwoCats will be required before it can be considered secure.

# 4 Efficiency Analysis

TwoCats introduces an efficient compute-time hardened hash function, in addition to being sequential-memory-hard. This hash function on most machines will take from 4 to 8 clock cycles per 32-bit result written to memory, with one thread, because the multiply operation takes 3-4 cycles, the addition takes 1, and the rest (memory read/write, OR, and increments) can often be done in parallel. With multiple threads, throughput can be increased to fill about half of the memory bandwidth before the memory bottleneck begins to heavily impact runtime.

## 4.1 The Cost of Cache Timing Attack Resistance

The “resistant” PHSs proposed so far seem to allow an attacker to use about ¼ of the memory compared to the computed DAG size, with little or no recomputation penalty. There are good reasons for this. First, there is a recomputation-free ½ memory attack against all “resistant” PHSs which have computation DAGs with max fanout degree 2. When pebbling a resistant PHS's DAG, simply pick up one of the pebbles used to compute the next node, or if those pebbles will be needed in future computations, pick a pebble which is not pointed to by any node beyond the node being pebbled. This always works for DAGs with max fan-out degree <= 2.

For DAG pebbling with 1/3 the pebbles compared to the DAG size, we can always pebble to the 2/3rds mark with no recomputation. For every new node pebbled after that, we gain a pebble that is not needed to cover some node that is still pointed to by the remaining unpebbled nodes. This makes it very hard to design a hard-to-pebble DAG where an attacker has 1/3 pebbles. This results in resistant PHSs having a 3-4X lower memory cost than unpredictable or hybrid PHSs.

When CPU limited, run-times should be approximately the same for resistant, hybrid, and unpredictable PHSs when using the same hashing algorithm, though some PHSs do not write to memory corresponding to DAG nodes that have in-degree 1, saving on memory bandwidth. This can lead to a lower performance penalty than my estimated 3-4X, and this is in fact the

case for Catena-2, which has between a 2-3X reduction in the time*memory cost compared to unpredictable PHSs. However, resistant PHSs do not suffer from a memory bandwidth degradation in comparison to hybrid and unpredictable PHSs. Attackers will have to pay a bit more for the extra RAM, but this is likely to be a small factor, typically less than 2X.

I have written a pebbling application that attempts to pebble TwoCats, Catena-3, and what I believe is similar to Escrypt's sliding power-of-two window. In the pebbling algorithm, I assume an attacker knows every detail about the computation DAG ahead of time, and can plan his memory usage strategy carefully. Automated pebbling confirms that all DAG types tested are easily pebbled with ¼ pebbles compared to number of nodes.

In summary, the cost for resistant PHSs versus unpredictable PHSs is typically about a 3-4X penalty in time*memory cost, though in some cases it may be in the 2-3X range.

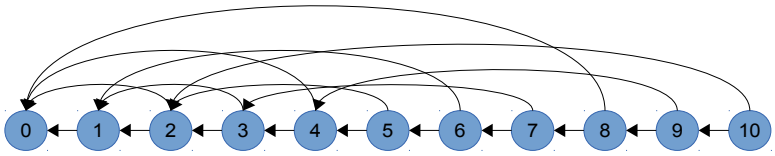
### 4.2 Computation DAGs

The original Script algorithm writes a linear chain of hashed blocks to memory, each hashed block depending sequentially on the data computed for the previous. Its directed acyclic computation graph is a linear chain:

*Scrypt Computation DAG:*


Script is vulnerable to TMTO attacks. An attacker covering nodes 3 and 7 sees an average recomputation of 1.5 nodes + second loop hashing, for a 2.5 computations per node in the second loop. Adding first loop computations gets the total to 3.5 computations per node, compared with 2 computations per node when keeping all nodes in memory. That's a 1.75X computation penalty, but the attacker only covered 1 in 4 nodes, so his time*memory is down to 0.44X of his original cost. As an attacker reduces his memory coverage, his time*memory cost converges to ¼ of the original.

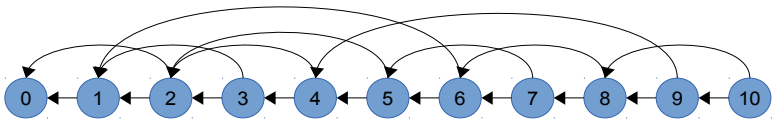
*TwoCats's "resistant" Computation DAG:*




One visual give-away that this is a sliding-power-of-two bit-reversal DAG is that every power of 2 node points to node 0. To compute the destination of an edge, take the binary representation of the source node, remove the leading 1, and reverse the bits. For example, node 6 is 110, which becomes 10 after removing the leading 1, and then 01 with bit-reversal, so node 6 points to node 1. If an edge length is 2 greater than largest power of 2 less than the source node number, then add that power of two. So, for example, if the graph were larger, we'd see node 11000 (node 24) would point to node 0001 (node 1), but since 1 + 16 + 2 = 19 < 24, we add 16. Therefore, node 24 points to node 17. This causes the destination node to always fall within the "sliding" power-of-two window preceding a node (actually it follows 2 pebbles behind to avoid 1-long edges).

This DAG architecture was chosen after it demonstrated the strongest resistance of all DAG architectures tested to my automated pebbling algorithm. An attacker attempting to pebble such graphs with a combination of fixed-spaced pebbles, fixing pebbles on high degree nodes, and fixing pebbles on destinations of short edges will find this graph requires more pebbles and recomputation than the other DAGs tested.

*TwoCats's "unpredictable" Computation DAG:*

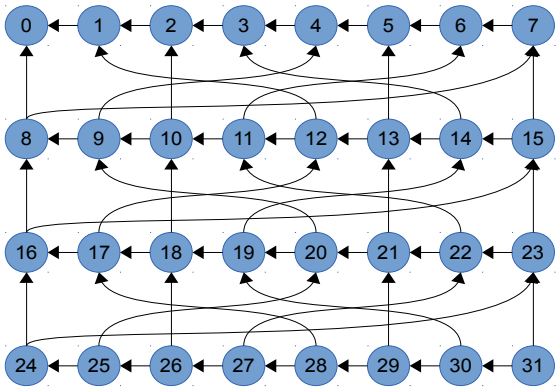



TwoCats "unpredictable" computation DAGs have the same linear chain, but instead of waiting until all memory is written before doing password dependent pseudo-unpredictable reads, it reads and hashes while writing. This creates a unpredictable-ish looking graph where

edges on average point back ¼ from their position, and there are lots of short edges. This keeps the midpoint cut size around 17% of the number of nodes. In this case, the cut size to the right of node 5 is 4.

If an attacker has covered only nodes 3 and 7 only, then to compute node 8 requires recomputation of every single missing node because 8 points to 6, which points to 5 which points to 4, which points to 2 and then onto 0. Similarly, computing 9 and 10 also require full recomputation of every missing node, since they point to 8. In general, a TwoCats graph recomputation penalty grows to a substantial portion of the entire graph for each missing node requiring recomputation by the time an attacker has only 1/8th of the nodes in memory, resulting in a runtime proportional to the square of the graph size. While a ½ memory TMTO with every other memory block kept results in no significant change in memory*time cost, anything lower than ½ rapidly becomes too expensive to compute. An attacker keeping every 4th node in memory suffers an additional computation factor of over 2000X for a 1M node graph. A reasonable attack against TwoCats would be to save ½ of the last ½ of memory. For 1,000,000 node graphs the gain is < 2% in the time*memory cost. If we increase spacing to 3 in the last ¼, then to 4 in the last 8th, and so on, the memory*time cost drops 3% versus the normal algorithm.

*Catena-3 Computation DAG:*




The Catena-3 DAG is computed using ¼ of the memory requirement compared to the size of it's computation DAG. However, an attacker is unlikely to succeed at improving his time*memory cost while using even one less memory location than ¼. Our algorithm pebbles Catena with 0 recomputation penalty for a ¼ pebble coverage just like TwoCats and Escrypt. With one fewer pebbles, the Catena pebbling penalty jumps to 1.8X for a 1024 node graphs, using fixed pebbles every 4. The penalty seems independent of graph size. When a Catena-3 sub-DAG is embedded in the first row, the penalty jumps to 3X, when using fixed pebbles

every 4 and pebbling nodes pointed to by short edges, which improves the situation on the first row.

Code for the pebbling application can be found at:

https://github.com/waywardgeek/TwoCats/tree/master/predict

## 5 Benchmark Results

The following benchmarks were run on my quad-core 3.4GHz i7-3660 Ivy Bridge development machine with 2 banks of 4GB CORSAIR Vengeance RAM, running Manjaro (Arch) Linux. Script version 1.1.6 was locally compiled with SSE2, and -O3 -march-native optimization. Catena was compiled from the available on 1/24/14 from github, using the waywardgeek branch I created. Escrypt was version 0.3.1, patched to upgrade it to 0.3.2, run in Scrypt-compatibility mode, and hand modified to only run a single print_scrypt call and then exit. Multiple threads did not seem to work properly in Scrypt, so it was run only single-threaded. However, the Escript implementation of Scrypt is currently the fastest implementation of Scrypt available. All of these benchmarks include memory allocation overhead. Executable were run with another CPU intensive task running and without, three times, and the best time was listed.

Peak cost is memory/CPU time. Average cost is computed based on my understanding of how each algorithm fills memory, and is an attempt to show the average memory usage / CPU time. In particular, Scrypt fills memory linearly in the first pass, and hashes it in the second leading to an average memory of about 0.75 of the peak, while Catena-3 fills memory linear in the first write pass, and holds steady in the next 5 read/write passes, leading to an average memory of 11/12 = 0.92. TwoCats does the worst of the three on average memory, as it fills continuously, at 0.5.

Bandwidth is total read/write passes * memory/CPU time. Catena can be done in 6 passes, which is what I assume in the table, though the implementation currently has 8, so Catena's bandwidth and runtime still have room for optimization.

I apologize to Alexander, the Escrypt author, and Christian, the Catena author, for using benchmarks from their pre-released code. Please feel free to ask me to correct/update/remove any data I have listed, or to request that new data be added. I feel it is important to list Catena data, since it shows the 3-4X peak time*cost reduction that I expect for a “resistant” PHS, and I have only listed data when using the same hash function as I had used in NoelKDF, which shows Catena in a good light from a benchmark perspective. Alexander mentioned that I should list the best Script data available, and his Escrypt implementation run in Scrypt compatibility mode is the best available. Also, as Alexander suggested, I have benchmarked an Escrypt Salsa20/2 version, which I built by deleting 3 of the 4 2-round calls in

SALSA20_8_BASE. The idea is to show how much faster Scrypt could be with a simple 3-line change.

| | Memory | CPU Time (s) | Compute Hardness | Bandwidth (GiB/s) | Peak Cost | Average Cost |
|---|---|---|---|---|---|---|
| Scrypt (1 thread) | 500 MiB | 0.83 | | 1.2 | 1.0X | 0.75X |
| Escrypt (1 thread) | 2 GiB | 2.71 | | 1.5 | 1.2X | 0.90X |
| Escrypt (2 threads) | 2 GiB | 1.40 | | 2.9 | | |
| Escrypt (4 threads) | 2 GiB | 0.81 | | 4.9 | | |
| Escrypt (8 threads) | 2 GiB | 0.61 | | 6.6 | | |
| Escrypt/Salsa20/2 (1 thread) | 2 GiB | .98 | | 4.1 | | |
| Escrypt/Salsa20/2 (8 threads) | 2 GiB | .50 | | 8.0 | | |
| Catena-3 | 1 GiB | 1.37 | | 4.4 | 1.2X | 1.1X |
| Catena-2 | 1 GiB | 1.06 | | 5.7 | 1.6X | 1.4X |
| TwoCats (1 thread) | 2 GiB | 0.52 | 52% | 7.7 | 6.4X | 3.2X |
| TwoCats (2 threads) | 2 GiB | 0.33 | 37% | 12.1 | | |
| Memmove (1 thread) | 2 GiB | 0.23 | | 17 | | |
| Memmove (2 threads) | 2 GiB | 0.18 | | 22 | | |

Compute hardness is calculated as the time spent doing serial multiplications compared to the total runtime. On the 3.4GHz quad-core i7 Ivy Bridge processor used in these benchmarks, multiplication takes 3 cycles, or 0.88ns. An additional cycle is required for the XOR operation, making 75% the maximum possible compute time hardness. Memory allocation overhead is on the order of 15-20%, making it difficult to achieve compute time hardness over 50% while hashing external RAM. A dedicated authentication server which allocates memory once for many authentications would achieve higher compute-time hardness.

For small memory in-cache hashing, TwoCats runs very fast, especially with SSE or AVX2 enabled. The Haswell machine was provided by Solar Designer, and is an Intel quad-core i7-4770. These numbers are for 1, 4, and 8 threads, with either 0 or 1 multiplication in the inner loop per 32 bytes hashed. Each hash requires 2 reads, doubling the bandwidth. All runs were made with the blocksize equal to 16KiB and subblocks size equal to 64, meaning there were many small random reads. In general, this slows down L1 cache limited loops by about 2X compared to having no small random reads. Each run was made hashing 1MiB repeated 2^16 times.

| | 1T 0M | 4T 0M | 8T 0M | 1T 1M | 4T 1M | 8T 1M |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| Haswell AVX2 | 79 GiB/s | 260 GiB/s | 350 GiB/s | 57 GiB/s | 199 GiB/s | 277 GiB/s |
| Ivy Bridge SSE2 | 57 GiB/s | 191 GiB/s | 191 GiB/s | 45 GiB/s | 142 GiB/s | 155 GiB/s |

AVX2 showed the most improvement over SSE2 when running in L1 cache, and with 8 threads. Even a single multiplication in the inner loop results in a multiplication time limited loop when running in L1 cache.

The L1 data bus in Ivy Bridge and Sandy Bridge is 16 bytes wide, with two lanes. On Haswell the data widths are doubled, and we can read 32 bytes at once. The theoretical top read bandwidth in Ivy bridge was 109 GiB/s, so it seems TwoCats is achieving about 50% with 1 thread. Similarly, it achieves about 36% of the possible L1 read bandwidth on Haswell. When sequential rather than random 64-byte reads are done, almost the full possible bandwidth is achieved on Ivy Bridge, at 92 GiB/s.

I compared resistance to cache timing attacks using my automated pebbler. Catena-3 seems to be the clear winner for a “resistant” PHS, as even reducing 1 pebble punishes an attacker significantly. However “hybrid” PHSs fill memory with all the computed results, so users do not benefit from the reduced memory consumption the way they do with Catena. For TwoCats, I tested Catena-2 and Catena-3, with and without an embedded Catena-3 graph in the first row (“Enhanced Catena”), and various other DAG styles, including the combination of Alexander's power-of-two sliding window and Christian's bit-reversal. As I enhanced the algorithm, the recomputation penalties decreased, but the relative order of results has not changed. To improve pebbling, I manually tuned my 3 parameters to minimize the recomputation penalty. Fixed pebbles at fixed spacing had the greatest impact. Enhanced Catena needs a heuristic to cover nodes pointed to by short edges, and the sliding-reverse DAGs needed a heuristic for fixing pebbles on nodes of high in degree.

| | Spacing | Max Degree | Min Edge Length | Recomputation Penalty |
|---|---|---|---|---|
| Catena-3 | 8 | 0 | 0 | 9X |
| Catena-2 | 5 | 0 | 0 | 2.6X |
| Enhanced Catena-3 | 8 | 0 | 25 | 89X |
| Enhanced Catena-2 | 5 | 0 | 25 | 6.5X |
| Sliding-Reverse | 16 | 3 | 0 | 973X |

All of these runs pebbled graphs with 128 pebbles. The Catan-3 and sliding-reverse graphs had 1024 nodes, while the Catena-2 graphs had 768 nodes.

## 6 Intellectual Property Statement

I, Bill Cox, place TwoCats, bothe the algorithm and all files and intellectual property associated with this project, into the public domain. I will file no patents on any idea used in this project. TwoCats includes source code from the official references for Blake2 and HKDF, and are is released under the MIT-like licenses. Also, twocats-test.c was copied from Catena's catena_test_vectors.c and is released under the MIT license.

TwoCats is and will remain available worldwide on a royalty free basis, and I am unaware of any patent or patent application that covers the use or implementation of the TwoCats algorithm.

## 7 No Hidden Weaknesses

I, Bill Cox, assert that TwoCats has no deliberately hidden weakness such as back doors, and I know of no weaknesses other than those discussed in this document in the Weaknesses section. No unusual constants are used in the code.

## 8 Bibliography


1. Christian Forler, Stefan Lucks, and Jakob Wenzel, Catena: A Memory-Consuming Password-Scrambling Framework, http://eprint.iacr.org/2013/525.pdf
2. Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. presented at BSDCan' 09, May 2009, 2009.
3. Secure Applications of Low-Entropy Keys, J. Kelsey, B. Schneier, C. Hall, and D. Wagner (1997)
4. On time versus space and related problems, 16th FOCS - Hopcroft, Paul, et al. - 1975
5. New developments in password hashing: ROM-port-hard functions, Alexander Peslyak, ZeroNights 2012, http://www.openwall.com/presentations/ZeroNights2012-New-In-Password-Hashing
6. Integer Hash Function, Thomas Wang, Jan 1997, updated Mar 2007 to Version 3.1, http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm, https://gist.github.com/aktau/6268616/raw/c3275bd0efc2f44ae3437282792ff0b2170624aa/inthash.md
7. Khovratovich, Dmitry, and Ivica Nikolić. "Rotational cryptanalysis of ARX." Fast Software Encryption. Springer Berlin Heidelberg, 2010.
8. Bryant, Randal E. "Graph-based algorithms for boolean function manipulation." Computers, IEEE Transactions on 100.8 (1986): 677-691.

9. Krawczyk, Hugo. "Cryptographic extraction and key derivation: The HKDF scheme." Advances in Cryptology–CRYPTO 2010. Springer Berlin Heidelberg, 2010. 631-648.
10. Yao, Frances F., and Yiqun Lisa Yin. "Design and analysis of password-based key derivation functions." Topics in Cryptology–CT-RSA 2005. Springer Berlin Heidelberg, 2005. 245-261.