TigerKDF:
A Compute Time and Sequential Memory Hard Key Derivation Function

Bill Cox

waywardgeek@gmail.com

For Consideration in the Password Hashing Competition

Chapel Hill, NC, February 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. TigerPHS is a new "hybrid" sequential-memory-hard KDF which is compute time hardened through sequential multiply operations, limiting a custom ASIC's speed advantage to about 2X. TigerPHS greatly increases the time*memory cost per guess of custom ASIC attacks, while greatly reducing benefits from time-memory trade-offs. TigerPHS's "hybrid" design also provides good defense against cache timing attacks, without sacrificing time*memory performance.*

1 Introduction

TigerPHS is a sequential-memory and compute-time hard key derivation function (KDF) that maximizes an attackers time*memory cost for guessing passwords. TigerPHS:

- 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
- Use parallelism for improved protection
- 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

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

Like Scrypt, TigerPHS 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 appends 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 that maximizes performance.

All else being equal, TigerPHS has about a 5X lower time*peak memory cost against cache-timing attacks than resistant KDFs, and about a 7X lower time*average memory cost. At the same time, TigerPHS will have about a 3.5X higher time*peak memory cost and a 2.3X higher time*average memory cost against the more common brute force attacks. An unpredictable KDF may have nearly zero time*memory cost in this case. This is discussed 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, TigerPHS employs multiplication based compute-time hardening, forcing ASIC attackers to run no more than about 2X faster than a recent PC, regardless of his 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. TigerPHS does in the second loop. GPUs are discussed more in Section 3.

TigerPHS currently runs using a 32-byte state and Blake2s in between block hashes. It runs well on 32-bit architectures. A 64-byte version of TigerPHS is a natural extension, possibly using Blake2b.

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

https://github.com/waywardgeek/TigerPHS

1.1 Credits

While I deserve credit for this document, my code, and the inspiration for naming a key derivation function 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, the 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 TigerPHS to improve

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

their own entry. I will feel honored rather than offended by entries copying TigerPHS's multiplication chain hardening, or even significant portions of TigerPHS's algorithm directly.

TigerPHS 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
- Blake2s hashing between SIMD “lanes” between block hashes. 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. TigerPHS 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 TigerPHS, 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 Designer's sliding power-of-two window to create the memory access pattern used in the “resistant” first loop.

I 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 TigerPHS, and Blake2 significantly improved the performance of TigerPHS. 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.

2 Algorithm Specification

2.1 Simplified "Unpredictable" TigerPHS

A simplified version of "unpredictable" TigerPHS is:

Simplified unpredictable TigerPHS:

```
Inputs: hashlen, password, salt, memlen
Output: hash[hashlen]

state[0 .. 7] = conv2Uint32(H(32, password, salt))
mem[0 .. hashlen-1] = H(hashlen, state)
prevAddr = 0
toAddr = hashlen
for i = 1 .. memlen/hashlen-1:
        fromAddr = state[0] % i
        for j = 0 .. hashlen/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]
output H(hashlen, conv2Uint8(mem[memlen-hashlen .. memlen-1]))
```

The function H is a cryptographically strong hash function such as Blake2s, which is used in the reference implementation.

In short, this algorithm hashes the previous hash block written to memory with a pseudo-random prior hash block to generate the next hash block to write to memory. It does this in 8 parallel "lanes" corresponding to the 8 32-bit state registers, and the lanes do not depend on each other. At the end, the last 8 values are hashed together with H to form the derived key.

While simple, this algorithm can be improved by (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. Hashing larger blocks than hashlen to reduce cache miss penalties (Scrypt)
3. Hash between lanes in between block hashes to mix data and introduce cryptographically strong hashes now and then (Solar Designer)
4. Introduce SSE/AVX2 parallelism to hash 8 chains in parallel (Solar Designer)
5. Introduce small unpredictable reads to thwart GPU attacks (Solar Designer)
6. Introduce repetitions to increase compute time and L1 cache bandwidth for memory-limited systems (Solar Designer & Bill Cox)
7. Introduce a "resistant" loop before the "unpredictable" loop to harden against cache-timing attacks (Catena)
8. Introduce inter-thread memory hashing to thwart TMTOs (Solar Designer)
9. Introduce thread level parallelism to take advantage of multi-core CPUs (Scrypt)
10. Introduce client-independent update (Catena)
11. 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 KDF 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. Many potential enhancements have been left out of TigerPHS to reduce complexity.

2.2 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 1GiB twice in 1.4 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 multipliers running in parallel.

In contrast, TigerPHS 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 $0.88ns*2^{31}*8/32 = 0.47$ seconds. TigerPHS performs this calculation in 0.83 seconds, only 76% longer than an optimally multiplication bound loop.

Definition: Multiplication-Hard Hash Function

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. TigerPHS has a multiplication to simple operation ratio of 1-to-1. It does one multiply and one XOR sequentially in every loop.

The TigerPHS hash function below implements upgrades 1-6 mentioned above:

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

```
hashBlocks(uint32 state[8], uint32 mem[], uint32 blocklen, uint32 subBlocklen,
            uint64 fromAddr, uint64 prevAddr, uint64 toAddr,
            uint8 multiplies, uint32 repetitions):
    numSubBlocks = blocklen/subBlocklen
    for i = 0 .. 7:
        oddState[i] = state[i] | 1
     v = 1
    for r = 0 .. repetitions-1:
        for i = 0 .. numSubBlocks-1:
            randVal = mem[fromAddr]
            p = prevAddr + subBlocklen*(randVal & (numSubBlocks – 1))

            # Compute the multiplication chain, preferably in CPU registers
            for j = 0 .. subBlocklen/8 - 1:
                for k = 0 .. multiplies:
                    v = v * oddState[k]
                    v ^= randVal
                    randVal += v >> 32

            # Hash 32 bytes of memory, preferably in the SIMD unit
            for k = 0 .. 7:
                state[k] = (state[k] + mem[p++]) ^ mem[fromAddr++]
                state[k] = (state[k] >> 24) | (state[k] << 8)
                mem[toAddr++] = state[k]

    hashWithSalt(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 8 32x32->64 multiplies per inner loop, along with every 32 bytes of hashed memory. A block is 4094 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 difficulty. After the new hashed block is written to memory, hashWithSalt calls Blake2s to mix bits between the state registers.

2.3 The “Resistant” loop

To provide stronger resistance againt offline brute-force guessing atttacks, TigerPHS runs an “unpredictable” second loop called “hashWithPassword”. This function implements features 7 and 8 above. The function is:

```
hashWithoutPassword(uint32 state[8], uint32 mem[], uint32 p, uint64 blocklen,
        uint32 blocksPerThread, uint32 multiplies, uint32 repetitions,
        uint32 parallelism, uint32 completedBlocks):

    uint64 start = blocklen*blocksPerThread*p
    uint32 firstBlock = completedBlocks
    if completedBlocks == 0:
        # Initialize the first block of memory
        for i = 0 .. blocklen/8 - 1:
            hashWithSalt(mem + start + 8*i, state, i)
            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(state, mem, blocklen, blocklen, fromAddr, prevAddr, toAddr,
            multiplies, repetitions)
```

This function is called SLICE number of times, where SLICE is 16. 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.

2.4 The "Unpredictable" Loop

The unpredictable loop follows execution of the resistant loop. This loop provides TigerPHS 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 1/4th memory or less. Also, the unpredictability of this loop keeps him from using parallel cores to recompute values before they are needed. A "resistant" KDF 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(uint32 state[8], uint32 mem[], uint32 p, uint64 blocklen,
                uint32 blocksPerThread, uint32 multiplies, uint32 repetitions,
                uint32 parallelism, uint32 completedBlocks):

        uint64_t 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(state, mem, blocklen, TIGERPHS_SUBBLOCKLEN, fromAddr,
                        prevAddr, toAddr, multiplies, repetitions)
```

This function differs from the simplified version of TigerKFD 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:

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

2.5 Thread Management

The function hashMemory initializes thread states, and launches the threads once per slice, first on hashWithoutPassword, and then hashWithPassword. It implements feature 9 above: thread level parallelism. The decision to support multithreading was difficult, but in the end, the huge performance and security improvement warrants the additional complexity. In comparison, pepper can take advantage of multiple CPU cores, but not without giving an attacker a simple TMTO. The function is:

```
hashMemory(uint8 hash[], uint8 hashSize, uint32 mem[], uint8 memCost, uint8 timeCost,
        uint8 multiplies, uint8 parallelism, uint32 resistantSlices):

    # Determine parameters that meet the memory goal
    repetitions = 1 << timeCost
    (parallelism, blocklen, blocksPerThread) =
        TigerPHS_ComputeSizes(memCost, timeCost, parallelism)

    # Convert hash to 8 32-bit ints.
    uint32_t hash256[8]
    hash256 = TigerPHS_hkdfExtract(hash, hashSize)

    # Initialize thread states
    uint32 states[parallelism]
    for p = 0 .. parallelism-1:
        states[p] = copy(hash256)
        hashWithSalt(states[p], p)

    for slice = 0 .. SLICES - 1:
        for p = 0 .. parallelism-1:
            if slice < resistantSlices:
                hashWithoutPassword(states[p], mem, p, blocklen,
                    blocksPerThread, multiplies, repetitions, parallelism,
                    slice*blocksPerThread/SLICES)
            else:
                hashWithPassword(states[p], mem, p, blocklen,
                    blocksPerThread, multiplies, repetitions, parallelism,
                    slice*blocksPerThread/SLICES)

    # Apply a crypto-strength hash
```

```
addIntoHash(hash256, mem, parallelism, blocklen, blocksPerThread)
hash = TigerPHS_hkdfExpand(hashSize, hash256);
```

The function TigerPHS_ComputeSizes does the following:

- Compute blocksPerThread as (1024 << memCost)/(parallelism*blocklen)
- If blocksPerThread is < 256, set it to 256
- If needed, lower blocklen to nearest power of 2 to keep from having too much memory
- If blocklen is < 64, set it to 64
- If needed to reduce memory further, decrease parallelism

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

The function addIntoHash simply adds the last 8 words written to memory by each thread into state.

2.6 Garlic (aka memCost)

The final upgrade is adding Catena style “garlic” and protecting against memory leaks, which are features 10 and 11 above. Catena's “garlic” which I implemented in TigerPHS, enables a very cool 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 TigerPHS 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 KDFs: attackers have a much higher chance of gaining access to leaked memory, enabling him to abort incorrect password guesses early. Because memory-hard KDFs 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, TigerPHS overwrites 1/ 32^{nd} 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 $1/64^{th}$ of memory is filled. At this point, TigerPHS 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 TigerPHS main hash function is:

```
bool TigerPHS(uint8 hash[], uint8 hashSize, uint8 startMemCost, uint8 stopMemCost,
              uint8_t timeCost, uint8 multiplies, uint8 parallelism,
              bool updateMemCostMode):

    uint32 mem[1024 << stopMemCost]

    # 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 || (!updateMemCostMode && i + 6 < startMemCost)):
            resistantSlices = SLICES/2
            if i < startMemCost:
                resistantSlices =  SLICES
            hashMemory(hash, hashSize, mem, i, timeCost, multiplies,
                parallelism, resistantSlices)
            if i != stopMemCost:
                # Not doing the last hash is for server relief support
                TigerPHS_hkdf(hash, hashSize)

    # 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. Feel free to copy it for your PHC entry.

There are some details in tigerphs-common.c that are boiler-plate for a KDF, such as hashing the password, salt and data with HKDF. Refer to the reference source code for these details.

3 Security Analysis

There are reasons to believe the design of TigerPHS is secure. The analysis of TigerPHS 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 KDF, 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 KDF. 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 advanced 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

3.1 Key Reversal and Derived Key Randomization

TigerPHS uses HKDF-SHA256[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 HKDF_Extract, including sizes of all variable sized data, making the initial key derivation in TigerPHS “strongly secure”, according to Yao and Yin's definition[10].

```
hash256 = HKDF_Extract(salt, hashSize || passwordSize ||  password |
        dataSize || datastartMemCost || timeCost|| multiplies || parallelism)
```

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 TigerPHS because the password length is hashed upfront

By default, Blake2s is used as the hashing algorithm (PRF) for PBKDF2, but this is configurable in tigerphs.h. This insures there is no currently known way of reversing the key which is better than brute-force guessing, and also that changes in the password and associated data cause only randomized effects in the derived key. In addition, PBKDF2 is called again at the end of the algorithm with only the final hash as input. This eliminates concerns about PBKDF2 such as chosen input parameter attacks. In between, memory

hashing and multiplication-intensive hashing strive to lose little entropy, but the results are isolated from the starting application of the cryptographic hash, and are added into the hash. This loop is repeated per level of applied "garlic", meaning the total entropy loss should be no higher than what is lost for 2 + 3*garlic rounds of the cryptographic hash function: 2 for the initial PBKDF2, and 3 for each application of garlic for the isolation hash and PBKDF2.

As this is negligible entropy loss between the initial and final calls to PBKDF2, TigerPHS provides comparable base security with respect to reversibility and derived key randomization as PBKDF2-BLAKE2S.

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. If one exists, the LiteCoin community would be very interested.

TigerPHS 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 in an Scrypt TMTO attack, 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 TMTO 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 without a high recomputation cost. The TMTO hardness of TigerPHS 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 make the 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. If it were possible to compute the last round of Salsa20 hashing without first taking time to compute all the prior rounds, attackers would be very interested in hearing about it.

3.4 Hardened Compute Time Cost

TigerPHS introduces the concept of multiplication based compute-time hardening of a KDF. By sequentially calling a hash function dominated by the runtime of serial multiplications, an attacker is forced to spend a comparable time computing the TigerPHS 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:

```
uint64 v = 1
for i = 0 .. 7:
        oddState[i] = state[i] | 1
for i = 0 ..  numSubBlocks-1:
        randVal = mem[f++]
        for j = 0 .. subBlocklen/8 – 1:
                for k = 0 .. 7:
                        v = (uint32)v * (uint64)oddState[k]
                        v ^= randVal
                        randVal += v >> 32
hashWithSalt(state, v)
```

The oddState variables are allocated in regular CPU registers, while the stat 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, but moving data from between the SIMD unit and the scalar integer unit is very slow. The value of mem[f] is updated pseudorandomly by the memory hashing code, and only interacts with the compute-time hardening code through randValue. At the end of the loop, the resulting value of v is used as salt when 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 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. TigerPHS includes similar small random reads in the second loop (but not the first), which by default is 64 bytes compared to Bcrypt's 16. TigerPHS should be at least as hard against GPU attacks when run with 32KiB as Bcrypt with 4KiB.

Against ASICs, TigerPHS 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. TigerPHS'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.

3.6 Memory Leak Attacks

The single largest weakness in memory-hard KDFs 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.

TigerPHS 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.

3.7 Weaknesses

TigerPHS inherits any weaknesses in HKDF-SHA256, just as Scrypt inherited weakness from PBKDF2. However, HKDF was designed by the author of HMAC to be a secure next-generation KDF which is free from the known flaws in previous algorithms such as PBKDF2. Unlike PBKDF2, HKDF-SHA256 has no known input collisions, and there are no chosen-input attacks. I can find only one security weakness in HKDF: the input key material is not padded in any way other than by SHA256. I recommend that users of TigerPHS try to use fixed-length passwords to throughout their application. This insures no branching or memory addressing specific to the password length ever happens.

It is possible for an attacker to make better use of TigerPHS 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 TigerPHS 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 TigerPHS 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 KDF increases as the square of the hashing speed, this was an easy decision to make. However, TigerPHS 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.

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 TigerPHS'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 KDF). 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 TigerPHS 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 significantly better pebbling algorithms can be found, reducing the effectiveness of defending against cache-timing attacks. Other algorithms come with proofs for lower bounds on recomputation, but the

proven lower bounds are 2 orders of magnitude lower than the efficiency of the best currently known pebbling attacks[3]. TigerPHS uses the algorithm that proved most resistant computationally to the pebbling code I designed. I believe it will remain more resistant over time than the alternatives considered. This is the best choice, in my opinion, for a first-loop algorithm in a hybrid KDF, based on my experience with pebbling attacks, and I know of no better pebbling algorithm.

Against simple brute-force attacks, TigerPHS 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 is a lower bound, so the security risk here is low.

While simplicity has been a major goal of TigerPHS from the start, improved security has been a higher concern. Several decisions were made in favor of security over simplicity, including the two-loop "hybrid" architecture.

The mathematical analysis of TigerPHS's hashing functions could be improved. The multiplier/XOR chains have no formal proof that there exists no simple mathematical shortcut to reduce the computation time. I have a little number theory background, but I am certainly no expert. However, I do have extensive experience representing Boolean equations and manipulating them to reduce their size and delay in ASICs. Multiplications have no efficient Boolean level canonical representation that I am aware of (BDDs and related forms are all exponential), and a logic equation representation of the multiplication chain that it will explode exponentially if I push the XOR operations through the multipliers using any common circuit design techniques such as Shannon Decomposition.

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

4 Efficiency Analysis

TigerPHS 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.

[3] Catena-3 has a proven lower bound of $T \geq G^3/128S^2$ and G is only 1/4th the size of the DAG

4.1 The Cost of Cache Timing Attack Resistance

The “resistant” KDFs 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” KDFs which have computation DAGs with max fanout degree 2. When pebbling a resistant KDF'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 KDFs having a 3-4X lower memory cost than unpredictable or hybrid KDFs.

When CPU limited, run-times should be approximately the same for resistant, hybrid, and unpredictable KDFs when using the same hashing algorithm, though some KDFs 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 KDFs. However, resistant KDFs do not suffer from a memory bandwidth degradation in comparison to hybrid and unpredictable KDFs. 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 TigerPHS, 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 KDFs versus unpredictable KDFs 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 SIMD Optimization

Having a parallelism parameter enables TigerPHS to be SIMD optimized. In particular, if parallelism is a multiple of 2 or 4, the inner hashing loop should be executable as 2 or 4 32-bit loops in parallel. Memory for first loop can be interleaved, enabling efficient 128-bit parallel operations. The second loop has each parallel task reading from a pseudo-unpredictable memory location (fromAddr), and will require more memory bandwidth as a result, but interleaving memory still enables reading from prevAddr and writing to toAddr 128-bits at a time for 4 parallel tasks.

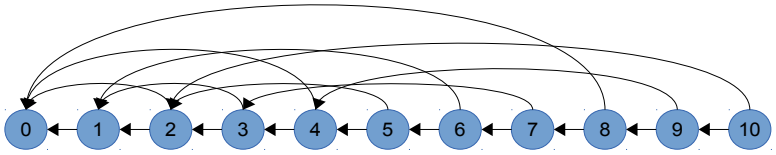
4.3 Computation DAGs

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.

TigerPHS's "resistant" Computation DAG:

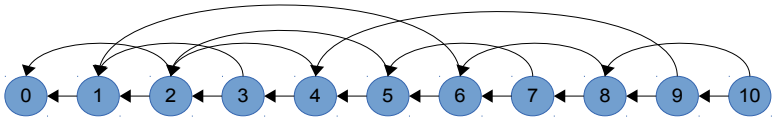


One visual give-away that this is a sliding-power-of-two bit-reversal DAG is that every power of 2 node points to node 0. To compute the destination of an edge, take the binary representation of the source node, remove the leading 1, and reverse the bits. For example, node 6 is 110, which becomes 10 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.

TigerPHS's "unpredictable" Computation DAG:

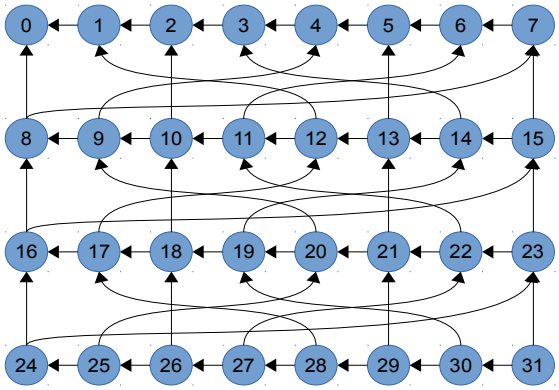


TigerPHS "unpredictable" computation DAGs have the same linear chain, but instead of waiting until all memory is written before doing password dependent pseudo-unpredictable reads, it reads and hashes while writing. This creates a unpredictable-ish looking graph where edges on average point back ¼ from 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 TigerPHS 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 TigerPHS 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 TigerPHS 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/TigerPHS/tree/master/predict

5 Benchmar 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. TigerPHS does the worst of the three on average memory, as it fills linearly 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, and 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 a 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
TigerPHS (1 thread)	2 GiB	0.52	52% (-M4)	7.7	6.4X	3.2X
TigerPHS (1	2 GiB	0.45	40% (-M3)	8.9		

thread)						
TigerPHS (2 threads)	2 GiB	0.33	36% (-M4)	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, TigerPHS 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 and subblocks size equal to 16KiB, meaning there were no small random reads. In general, this slows down L1 cache limited loops by about 2X. Each run was made hashing 64KiB repeated 2^20 times (about 1 million).

	1T 0M	4T 0M	8T 0M	1T 1M	4T 1M	8T 1M
Haswell AVX2	109 GiB/s	404 GiB/s	701 GiB/s	82 GiB/s	307 GiB/s	547 GiB/s
Ivy Bridge SSE2	92 GiB/s	222 GiB/s	580 GiB/s	80 GiB/s	198 GiB/s	274 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.

I compared resistance to cache timing attacks using my automated pebbler. Catena-3 seems to be the clear winner for a “resistant” KDF, as even reducing 1 pebble punishes an attacker significantly. However “hybrid” KDFs fill memory with all the computed results, so users do not benefit from the reduced memory consumption the way they do with Catena. For TigerPHS, 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	7	0	0	10X
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	1176X

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.

Not too much should be read into these numbers as they are simply upper bounds. However, sliding-reverse was chosen for TigerPHS's first loop based in part on these results. Also, the sliding-reverse DAG does a better job in the second loop at forcing attackers to keep results in memory.

6 Intellectual Property Statement

I, Bill Cox, place TigerPHS, 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. TigerPHS includes source code from the official references for Blake2 and HKDF, and are is released under the MIT-like licenses. Also, tigerphs-test.c was copied from Catena's catena_test_vectors.c and is released under the MIT license.

TigerPHS 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 TigerPHS algorithm.

7 No Hidden Weaknesses

I, Bill Cox, assert that TigerPHS 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.