Bill Cox

waywardgeek@gmail.com

For Consideration in the Password Hashing Competition

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

Chapel Hill, NC, January 2014

Abstract. *Scrypt demonstrated a new way to thwart off-line brute-force password guessing attacks, as the first successful sequential memory hard key derivation function (KDF). 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 on a massively parallel scale. NoelKDF is a new "hybrid" sequential-memory-hard KDF which is compute time hardened through sequential multiply operations in its inner loop, limiting a custom ASIC core speed advantage to about 2X. NoelKDF greatly increases the time*memory cost per core of custom ASIC attacks, while greatly reducing any benefit from time-memory trade-offs. NoelKDF's "hybrid" design also provides good defence against against cache-timing attacks, without sacrificing time*memory performance.*

1 Introduction

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

- Insures high memory cost for an attacker by filling memory rapidly
- Insures high runtime for an attacker through serial multiply operations that are difficult to speed up on an ASIC
- Mostly eliminates time-memory trade-off benefits to attackers
- Provides effective defence against cache-timing attacks
- Can use parallelism present on the machine for improved protection
- Is suitable for desktop PCs, web servers, embedded applications, and high end graphics cards
- Supports client independent hash updates
- Performs well hashing many GB of DRAM, or just a few MB of cache
- Is named after a cat

In this paper, memory-hard KDFs which do no password derived memory addressing are called "pure", while those that do so early in hashing are called "dirty", and combinations of the two are called "hybrid". "Pure" memory-hard KDFs are 100% immune to cache-timing attacks, while "dirty" KDFs can be attacked on a massively parallel scale once an attacker knows what memory addresses are accessed when hashing a correct password. "Hybrid" KDFs can have good resistance against cache-timing attacks while at the same time can be as effective against brute-force guessing attacks as dirty KDFs. Scrypt[2] is a "hybrid" KDF, and suffers far less from cache-

timing attacks than if it were “dirty”, but since it was not designed specifically to thwart cache-timing attacks or TMTO attacks, its resistance can be improved.

Like Scrypt, NoelKDF uses a 2-loop architecture, where in the first loop is a “pure” KDF. No password derived memory addresses are accessed in the first loop. The second loop is a “dirty” KDF, which significantly improves defence against offline brute-force guessing attacks as well as limiting TMTO options for an attacker.

Against cache-timing-attacks, NoelKDF has a time*memory cost about 16X lower than when no addressing information is known since the second loop can be aborted early, and the first loop can be computed efficiently with about 1/8th of the total memory. However, NoelKDF's “dirty” second loop provides about a 3-4X improved time*memory cost than “pure” KDFs, As a result, NoelKDF should have only about a 5-to-1 disadvantage against cache-timing attacks relative to “pure” KDFs. NoelKDF trades off some cache-timing attack resistance in order to retain full resistance against brute-force guessing attacks.

While I deserve credit for this poor document, my fairly awesome code, and the inspiration for naming a key derivation function after my cat, most of the good ideas found here come from the authors of Escript and Catena[1]. The cache timing defence strategy is only possible due to the excellent work and generosity of ideas from Christian Forler, while many of the other good ideas came from Alexander Peslyak (aka Solar Designer). I consider them both to be the unofficial primary authors of this work until such time as they prefer to be listed in the top spots officially. Since they have generously offered their ideas to the other PHC entrants, I accept the help they offered in the spirit intended, and hope that in some small ways I have helped the eventual winning entry to be better. In particular, any PHC author should feel free to borrow any ideas in NoelKDF to improve their own entry.

2 Algorithm Specification

2.1 Simplified “Dirty” NoelKDF

NoelKDF was originally a “dirty” KDF, until Christian Forler demonstrated an effective strategy against cache-timing attacks, which sent the other PHC authors scrambling. A simplified version of “dirty” NoelKDF is:

Simplified Dirty NoelKDF:

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

mem[0 .. hashlen-1] = H(hashlen, password, salt)
value = 0
prevAddr = 0
destAddr = hashlen
for i = 1 .. memlen/hashlen:
   randAddr = value % i
   for j = 0 .. hashlen-1:
      value = value*(mem[prevAddr++] | 3) + mem[randAddr++]
      mem[destAddr++] = value
output H(hashlen, mem[memlen-hashlen .. memlen-1], salt)
```

The function H is a cryptographically strong hash function such as PB-KDF-SHA256, which is used in the reference implementation.

2.2 Multiplication-Hard KDFs

Speeding up multiplication on custom ASICs versus a modern CPUs is difficult, and unlikely to result in a significant speed-up without exotic technologies such as liquid nitrogen cooling. In comparison, the Salsa20/8 hash function used in Scrypt[2] is likely to run at about 1ns per Salsa20/8 round per core, generating 64GB/second of data on a well optimized advanced technology custom ASIC. In comparison, Scrypt on my development machine hashes 0.5GB twice in 1.8 seconds, which is about 100X slower. This is because it has only 16 levels of ADD/XOR logic per 32-bit register, of which there are 16, making Salsa20/8 it a bit less complex than 16 Booth-encoded 32x32 multipliers running in parallel. On the development machine, 32x32 bit multiplies have a 0.9ns latency.

NoelKDF is hardened against such attacks through the use of a multiplication-hard hash function, which cannot be significantly sped up on custom AISCs compared to advanced CPUs. Each iteration depends on the previous, and must be computed with one sequential multiply followed by one sequential addition. A lower bound on the runtime is:

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

On my development machine, a 32-bit multiply operation has latency multTime = 0.9ns. Hashing 2GB of data this way would require a minimum of $0.9ns*2^{31}/4 = 0.48$ seconds. The reference version of NoelKDF performs this calculation in 0.69 seconds, or only 44% 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 se-

quential simple operations, where simple operations are the usual single-cycle ALU operations: add, sub, XOR, ANS, 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 multiplication, while all of the simple operations are computed in 1. This means in a multiplication-hard hash function, it may be possible to spend at least 50% of the compute time on multiplications, assuming other tasks can execute in parallel. NoelKDF has a multiplication to simple operation ratio of 1-to-1. It does one multiply and one addition sequentially in every loop. The other operations, including increments, OR, and memory read and write can be done in parallel.

NoelKDF is sequential-memory-hard in the sense described in Catena[1]. Additional CPUs cannot significantly speed up it's computations, since the random address computed in the second loop cannot be known until the prior loop iteration has completed. If an attacker decides to cheat and use less memory, he will suffer a penalty when he has to recompute the missing data, just as in Scrypt[2]. Even stronger, NoelKDF memory blocks depend on not just the previous, as in Script, but also on a password dependent pseudo-random previous block. Attackers will have to recompute any missing input blocks that are accessed. With NoelKDF's 2-fanouts per node, this can grow exponentially through the computation DAG if an attacker has too little coverage of computed results in memory.

2.3 Algorithm Enhancements

By far, the most extensive enhancement over the original NoelKDF algorithm is the addition of a “pure” first loop. Several other enhancements were needed as well.

2.3.1 Speed Enhancements

For small hashlen, cache miss delays will dominate the runtime. Therefore NoelKDF has a separate blocklen parameter to enable high speed with short hash lengths.

The reference implementation, when run with 2 threads, achieves 12GB/s memory bandwidth on a machine I estimate to have a 25GB/s maximum. A SIMD optimized version with temporal cache hints is likely to achieve 22GB/s, as running 2 copies of 2-thread noelkdf at the same time increases memory bandwidth to 22GB/s.

While the original NoelKDF algorithm was optimized for external DRAM hashing, the current version is euqally suitable for running in small amounts of memory entirely from CPU cache. This is enabled by choosing a few MB rather than a GB or more, setting blockSize to a small value such as 64, which is well matched to cache line sizes, and setting “repetitions” to high enough of a value to make the runtime long enough.

2.3.2 Improvements Over Pepper

Pepper is a small number of secret bits of the salt, which are sent as 0's to the user. When authenticating, he must try all the possible salt guesses until he guesses the correct salt. This has the impact of increasing the compute effort for both an attacker and defender. If pepper is chosen well, the user can try all the salt guesses in parallel, feeling little increase in latency, while an attacker feels the full impact.

This scheme has some problems. First, it gives an attacker a free TMTO, and eliminating these trade-offs is a major goal of NoelKDF. Second, if we use pepper in a server-relief system, it is likely that the user will attempt to authenticate with the server multiple times, increasing network traffic. Finally, it seems to be naturally applied externally to the password KDF, and wont be able to take full advantage of SIMD instructions when used that way, especially when the inner loop hash function is very simple as is the case with NoelKDF.

To overcome these limitations, NoelKDF executes with a user-specified level of parallelism. A pthread version of NoelKDF has been developed to demonstrate the use of this parameter, while the reference version simply computes the parallel tasks in series.

2.3.3 Garlic and Data

NoelKDF “borrows” the concept of “garlic” from Catena[1], along with other goodies such as “client independent update”, and of course a cache-timing resistance strategy. I also copied catena_test_vectors.c to generate vectors for NoelKDF. Please read the Catena paper for a more in-depth description of these features. Client independent updates are where a sysadmin can increase the difficulty of computing a password hash without having to know the user's password.

A data input has also been added, following Catena's naming convention, which allows a software developer to provide application specific data such as secondary authentication keys to the hash. For example, when a server has a password specific key that is decrypted using the server's master key during password authentication, that password specific key can be added as data to the hash. This renders brute force attacks impractical to attackers who do not have the master key. Similarly, if a user provides a key file, a hash of that file can be used as data, again making brute-force attacks impractical. A similar concept appears in Escrypt (local parameters) and Blakerypt (session keys).

2.3.4 Additional Enhancements

In the second “dirty” loop, the prior memory block to hash is no longer chosen uniformly randomly. Instead it follows a cube-law, reducing the average edge length from ½ to ¼, significantly improving TMTO resistance. Having short edges causes the DAG depth to be considerably higher, with typical sub-DAGs of attackers using only 1/8th of memory being around 1/8th the size of the entire DAG, even with decent

strategies for pebble placement, such as evenly spacing them, covering high degree nodes, and covering nodes with short incoming edges. With a uniform distribution for selecting the prior memory block to hash, only 1 in 100 edges are shorter than 1% of the node number, while with the cubed distribution, 21.5% are this short. The average edge length from node *i* is computed as:

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

One intended use for NoelKDF is specifically in applications like TrueCrypt where only parameters that are indistinguishable from true random data can be used. Salt is such a parameter, but garlic, memory-size, and such are not. In these cases, the application can choose default settings for most values, and select an appropriate garlic level. After hashing the user's password, the resulting derived key is hashed one more time with a strong cryptographic hash and stored along with the salt in the encrypted volume. When verifying a password later, the application compares the resulting hash after each level of garlic is applied, and stops if it matches. This can run as a Password Halting Puzzle[3].

2.4 The NoelKDF Algorithm

Given an a user's password, salt, optional application specific data, and a PBKDF2 based cryptographic hash function H(hashlen, password, salt), the algorithm for NoelKDF is:

NoelKDF Algorithm:

```
function NoelKDF_HashPassword(hashsize, password, salt, memsize, garlic, data,
        blocksize, parallelism, repetitions):
    if data != null:
        derivedSalt = H(hashsize, data, salt)
        hash = H(hashsize, password, derivedSalt)
    else:
        hash = H(hashsize, password, salt)
    wordHash = toUint32Array(hash)
    memlen = memsize/4
    blocklen = blocksize/4
    numblocks = memlen/(2*parallelism*blocklen)
    mem = array of 2*numblocks*blocklen*parallelism*2^garlic 32-bit integers
    for i = startGarlic .. stopGarlic:
        value = 0
        for p = 0 .. parallelism-1:
            hashWithoutPassword(p, wordHash, mem, blocklen, numblocks, repetitions)
            value += mem[2*p*numblocks*blocklen+blocklen-1]
        for p = 0 .. parallelism-1:
            hashWithPassword(p, mem, blocklen, numblocks, parallelism, repetitions, value)
        xorIntoHash(wordHash, mem, blocklen, numblocks, parallelism)
        numblocks *= 2
    return toUint8Array(wordHash)
```

```
function hashWithoutPassword(p, wordHash, mem, blocklen, numblocks, repetitions):
    start = 2*p*numblocks*blocklen
    mem[start .. start+blocklen-1] = toUint32Array(H(blocklen*4,
            toUint8Array(wordHash), toUint8Array([p])))
    value = 1
    mask = 1
    toAddr = start + blocklen
    for i = 1 .. numblocks-1:
        if mask << 1 <= i:
            mask = mask << 1
        reversePos = bitReverse(i, mask)
        if reversePos + mask < i:
            reversePos += mask
        fromAddr = start + blocklen*reversePos
        value = hashBlocks(value, mem, blocklen, fromAddr, toAddr, repetitions)
        toAddr += blocklen

function hashWithPassword(p, mem, blocklen, numblocks, parallelism, repetitions, value):
    startBlock = 2*p*numblocks + numblocks
    start = startBlock*blocklen
    toAddr = start
    for i = 0 .. numblocks-1:
        v = (uint64)value
        v2 = v*v >> 32
        v3 = v*v2 >> 32
        distance = (i + numblocks – 1)*v3 >> 32
        if distance < i:
            fromAddr = start + (i – 1– distance)*blocklen
        else:
            q = (p + i) % parallelism
            b = numblocks – 1 – (distance - i)
            fromAddr = (2*numblocks*q + b)*blocklen
        value = hashBlocks(value, mem, blocklen, fromAddr, toAddr, repetitions)
        toAddr += blocklen

function hashBlocks(value, mem, blocklen, fromAddr, prevAddr, toAddr, repetitions):
    prevAddr = toAddr - blocklen
    for r = 0 .. repititions-1:
        for j = 0 .. blocklen-1:
            value = value*(mem[prevAddr + j] | 3) + mem[fromAddr + j]
            mem[toAddr + j] = value
    return value

function xorIntoHash(wordHash, mem, blocklen, numblocks, parallelism):
    for p = 0 .. parallelism-1:
        pos = 2*(p+1)*numblocks*blocklen - length(wordHash)
        for i = 0 .. length(wordHash)-1:
            wordHash[i] ^= mem[pos+i]
```

```
function bitReverse(value, mask):
    result = 0
    while mask != 1:
        result = (result << 1) | (value & 1)
        value >>= 1
        mask >>= 1
    return result
```

The first “pure” loop uses a power-of-two sliding window with a bit-reversal function to select the target within this window. Alexander Peslyak suggested the power-of-two sliding window to provide a more uniform distribution of incoming edges, while Christian Forler invented using a bit-reversal distribution of edges to thwart cache-timing attacks.

The second “dirty” loop is the NoelKDF cubed-distribution. When running on multiple threads, each thread computes it's results from all of the threads results computed in the first loop. This forces an attacker to keep all of the first loop results in memory while the second loops run. While an attacker can run the second loop sequentially, he will still be forced to have most of memory loaded at once (or suffer a substantial recomputation penalty), greatly increasing his time*memory cost.

2.5 The NoelKDF Hash Function

NoelKDF introduces a new non-cryptographic hash function that is meant to be compute time hardened and fast rather than producing data that is undetectably non-random. For a memory-hard KDF, there is simply no good reason to have the hash function in the inner loop produce cryptographically random data. For the inner loop hash function, there are two primary concerns:

1. Entropy loss needs to be insignificant
2. An attacker must have to compute mem[0 .. i-1] before he can compute mem[i]

Even SHA-256 leaks entropy, just too little to measure, since multiple inputs hash to the same output. NoelKDF uses the following inner loop hash function:

value = value(mem[prevAddr] | 3) + mem[randAddr]*

This is motivated from a desire for the hash function to be similar to a permutation. With permutations, no entropy is lost. If mem[prevAddr] and mem[randAddr] were replaced with a PRNG stream seeded only from the password and salt, then this would be a true permutation, and would be reversible. Being reversible is undesirable, as it gives attackers more TMTO options.

For this hash function to work well and not lose significant entropy, mem[prevAddr] needs to be highly scrambled relative to value. With a default page size of 4096, value is hashed 4096 times since the last time mem[prevAddr] was hashed into value. Page sizes of 256 passed the dieharder tests, indicating that even 256 iterations of the hash is far enough away. A side benefit of choosing Alexander's sliding-power-

of-two window in the first loop is that the 1st block, which is generated using PBKDF-SHA256, is rehashed into the stream at every node in the DAG that is a power of two, potentially reintroducing lost entropy in case there were any.

A final reason for the OR is to eliminate the possibility that an attacker can compute mem[i] without first computing mem[0 . i-1]. The OR is the only non-linear operation in the hash function.

Passing the dieharder tests is not a goal in itself, but simply additional evidence that NoelKDF does not leak entropy quickly. Dieharder tests are run with a blocklen of 256 rather than 4096. This is because the 32-bit value use to mix between values in a block is an awfully thin pipe for mixing 4KB quickly. The dieharder tests seem to detect correlations between sequential memory blocks if they are too long. However, this does not mean the hash function is losing entropy. It simply means that the dieharder tests can detect the low amount of mixing between large adjacent pages. The effect is similar to what would happen if we wrote out every value as it is computed in Salsa20/8. Writing out data every 8 rounds easily passes the dieharder tests, but if we wrote out all 64 computed values for every round of Salsa20/2, there is no way this stream would pass tests for randomness. However, it would generate considerably more data much more quickly while not losing entropy, and attackers would still be forced to compute it all. This would be a very effective strategy for a fast memory-hard KDF hash function, except that Salsa20/8, as well as most cryptographically strong hashes, use CPU operations like ADD, SHIFT, and XOR which can be sped up in custom ASICs by a large factor.

3 The Cost of Pure Cache Timing Resistance

The “pure” 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 free ½ memory attack against all “pure” KDFs requiring no recomputations. When pebbling a pure 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 pure KDFs having a 3-4X lower memory cost than dirty or hybrid KDFs.

When CPU limited, run-times should be approximately the same for pure, hybrid, and dirty 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 dirty KDFs.

I have written a pebbling application that attempts to pebble NoelKDF, 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 pure KDFs vs dirty KDFs is typically about a 3-4X penalty in time*memory cost, though in some cases it may be in the 2-3X range.

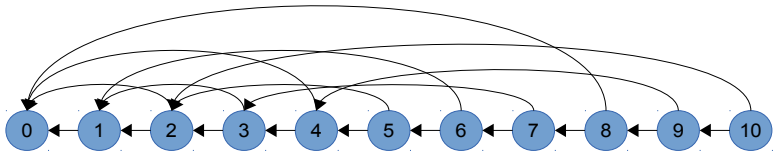
3.1 Computation DAGs

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. It's directed acyclic computation graph is a linear chain:

Scrypt Computation DAG:

Script is vulnerability 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.

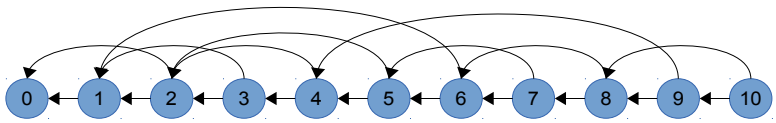
NoelKDF's "Pure" Computation DAG:



One visual give-away that this is a sliding-power-of-two bit-reversal DAG is that every power of 2 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.

NoelKDF's "Dirty" Computation DAG:

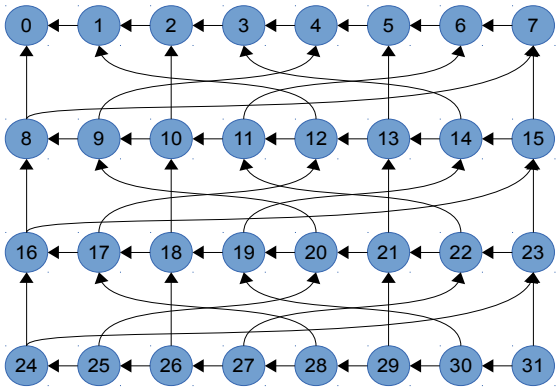


NoelKDF "dirty" computation DAGs have the same linear chain, but instead of waiting until all memory is written before doing password dependent pseudo-random reads, it reads and hashes while writing. This creates a random-ish looking graph where edges on average point back ¼ from their position, and 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 NoelKDF graph recomputation penalty grows to a substantial portion of the entire graph for each missing node requiring recomputatoin 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 NoelKDF 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 NoelKDF 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 pebbing 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/noelkdf/tree/master/predict

4 Results

The following benchmarks were run on our quad-core i7 CPU development machine running Manjaro (Arch) Linux, using the default install package for Scrypt, and the version of Catena available on 1/24/14 from github, using the waywardgeek branch I created. All were run in single-threaded mode.

	Memory	CPU Time (seconds)	Time*Memory
Scrypt	500MB	1.8	1.0X
Catena-3	1GB	1.37	2.6X
Catena-2	1GB	1.06	3.4X
NoelKDF	1GB	0.4	9X

The Catena-3 code used the same exact hash function as NoelKDF, as I literally copied and pasted the hashing code from noelkdf-ref.c into catena-multhash.c.

Next, I compared resistance to cache timing attacks with my automated pebbler. Catena-3 seems to be the clear winner for a “pure” KDF, as even reducing 1 pebble punishes an attacker. However “hybrid” KDFs fill memory with all the computed results, so users do not benefit from the reduced memory consumption they way they do with Catena. For NoelKDF, 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
Enanced Catena-3	8	0	25	89X
Enhanced Catena-2	5	0	25	6.5X
Sliding-Reverse	16	3	0	1176X

Not to much should be read into these numbers as they are simply upper bounds. However, sliding-reverse was chosen for hybrid-NoelKDF 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.

5 Conclusion

A natural choice for memory-hard KDFs seems to exist between the extremes of "pure" and "dirty" KDFs. "Hybrid" KDFs, while maybe 5X-ish less resistant to cache-timing attacks are 3-4X-ish more resistant against the more common brute-force guessing attacks, all else being equal.

Further, substantial resistance to ASIC attacks can be had through multiplication-time hardened hashing functions.

NoelKDF is just such a hybrid KDF.

6 Intellectual Property Statement

I, Bill Cox, place noelkdf-ref.c, the algorithm it contains, and all other files and intellectual property associated with this project into the public domain. I will file no patents on any idea used in this project. NoelKDF includes sha.c and sha.h which I copied from the scrypt source code, and which is released under the BSD license, and noelkdf-test.c was copied from Catena's catena_test_vectors.c and is released under the MIT license.

NoelKDF 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 NoelKDF algorithm.

7 No Hidden Weaknesses

I, Bill Cox, assert that NoelKDF has no deliberately hidden weakness such as back doors, and I know of no weaknesses other than those discussed in this document. 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. Xavier Boyen. Halting Password Puzzles – Hard-to-break Encryption from Human-memorable Keys. In 16th USENIX Security Symposium—SECUR-

ITY 2007, pages 119–134. Berkeley: The USENIX Association, 2007. Available at http://www.cs.stanford.edu/~xb/security07/.

4. Secure Applications of Low-Entropy Keys, J. Kelsey, B. Schneier, C. Hall, and D. Wagner (1997)