Attacking Memory-Hard scrypt with Near-Data-Processing

Jiwon Choe
Brown University
jiwon_choe@brown.edu

R. Iris Bahar
Brown University
iris_bahar@brown.edu

Tali Moreshet
Boston University
talim@bu.edu

Maurice Herlihy
Brown University
mph@cs.brown.edu

ABSTRACT

In a traditional DRAM-based main memory architecture, a memory access operation requires much more time and energy than a simple logic operation. This fact is exploited to build time-consuming and power-hungry memory-hard cryptographic functions that serve the purpose of hindering brute-force security attacks.

The security of such memory-hard functions depends entirely on the non-trivial costs of memory access. However, various compute-capable memory technologies have recently emerged as promising ways to reduce the memory access bottleneck, yet no one has looked into how they may impact the security of memory-hard cryptographic functions. In this preliminary work, we investigate the impact of near-data-processing (NDP) on scrypt, a widely used memory-hard password-based key-derivation function, and discuss the opportunities to further undermine scrypt using compute-capable memory.

CCS CONCEPTS
- Hardware → Memory and dense storage; • Security and privacy → Hash functions and message authentication codes.

ACM Reference Format:

1 INTRODUCTION

In a traditional DRAM-based main memory architecture, a memory access operation requires much more time and energy than a simple logic operation. This fact is exploited to build time-consuming and power-hungry memory-hard cryptographic functions, which serve the purpose of hindering brute-force security attacks. The computation cost of the memory-hard function is negligible for an honest user, who would compute it only once, but the cumulative computation cost is significant and therefore prohibitive for a brute-force attacker, who would need to compute the function a large number of times.

To this end, Colin Percival [34] defined the memory-hard algorithm: an algorithm that requires amount of memory approximately proportional to the number of operations to be performed. If sufficiently large amount of memory is required, not only would the compute time and power be bounded by memory access, but the algorithms would also be resistant to brute-force attacks using customized hardware. Memory is expensive and takes up large chip area, and therefore requiring large amounts of memory for a single function computation limits the amount of customized hardware that can be built to execute large-scale parallel attacks.

The security of memory-hard functions depends entirely on the non-trivial costs of memory access. However, various compute-capable memory technologies have recently emerged as promising ways to address the problems of slow and energy-intensive memory access [18, 44]. Compute-capable memory supplements memory devices with compute units, so that simple data-intensive computations can be done near memory (near-data-processing) or even in memory (processing-in-memory). There has been extensive research in improving application performance and reducing energy consumption using compute-capable memory [1–4, 11–13, 15–17, 21–23, 25–33, 37–39, 45–47], but to the best of our knowledge, no one has looked into how compute-capable memory may impact the security of memory-hard cryptographic functions.

In this preliminary work, we investigate the impact of near-data-processing (NDP) on scrypt [34, 35], a widely used maximally memory-hard password-based key derivation function. We show that the scrypt algorithm can be accelerated with a simple NDP architecture and provide realistic evaluations with a cycle-accurate, full-system NDP architecture framework. We also suggest how scrypt can be further accelerated with various compute-capable memory technologies.

2 SCRYPT OVERVIEW

Scrypt is a sequential memory-hard [34] password-based key derivation function, meaning that the fastest sequential algorithm to solve the function is memory-hard, and it is impossible for a parallel algorithm to asymptotically achieve a significantly lower cost. The algorithm was first proposed in 2009 [34] and was published as RFC 7914 [35] in 2016.

Algorithm 1 shows the scrypt algorithm as described in [34, 35]. Lines 1–4 give the high-level flow of scrypt. Inputs P and S are password and salt phrases, respectively, and dkLen is the desired key length. The password and salt are first passed through
PBKDF$_2^{SHA256}$\footnote{PBKDF2 iteratively applies a designated pseudorandom function on the password and salt a specified number of times to generate a cryptographic key. In scrypt, SHA256 is used as the pseudorandom function and is iterated only once. SHA256 is easy to compute and is not memory-hard.} to generate a 128$r$p-byte string. The generated string is divided into $p$ equal-length blocks, and the SMix function is called on each of them. The results from the SMix function are concatenated back together to be used as the salt in a final PBKDF$_2^{SHA256}$ call, which takes the original password and new salt to generate a final $dkLen$-byte output key.

$p$, $N$, and $r$ are scrypt-specific parameters. $p$ determines the number of times SMix is called in scrypt (lines 3–4). It is referred to as the parallelization parameter, for the $p$ SMix calls are independent of one another and can be computed in parallel. $N$ is a cost parameter passed to the SMix function; it controls the CPU and memory usage of scrypt by requiring the SMix function to compute, store, and pseudorandomly access $N$ different BLockMix hashes (lines 5–12), $r$ is the block size parameter that determines the size of a block that the BLockMix function operates on (lines 13–14).

The SMix function is central to the scrypt algorithm and makes up the memory-hard component of scrypt. The scrypt RFC [35] recommends the block size parameter to be $r = 8$. With this parameter, the initial input block to SMix is only 1kB in size and can easily fit in cache. However, the SMix function expands this 1kB block into an array of $N$ blocks, and the blocks are iteratively accessed in a pseudorandom order, based on the contents of the previously-accessed block. Assuming a sufficiently large $N$, the SMix function is bound by memory access and makes up the non-trivial cost of running scrypt.

### Algorithm 1 scrypt algorithm

1. function SCRYPT$(P, S, P, N, r, dkLen)$
2. $\langle B_0|B_1|...|B_{p-1} \rangle \leftarrow$ PBKDF$_2^{SHA256}(P, S, 1, 128rp)$
3. for $i = 0$ to $p - 1$
   4. $B_i \leftarrow$ SMix$_r(B_i, N)$
   5. return PBKDF$_2^{SHA256}(P, B_0|B_1|...|B_{p-1}, 1, dkLen)$
6. function SMix$_r(B, N)$
7. $X \leftarrow B$
8. for $i = 0$ to $N - 1$
   9. $V_i \leftarrow X$
   10. $X \leftarrow$ BlockMix$_{Salsa20/8,r}(X)$
   11. for $i = 0$ to $N - 1$
      12. $j \leftarrow INTEGERIFY(X)$ mod $N$
      13. $X \leftarrow$ BlockMix$_{Salsa20/8,r}(X \oplus V_j)$
      14. return $X$

For a preliminary investigation into scrypt’s vulnerability to compute-capable memory, we implement and evaluate the scrypt algorithm on a generic near-data-processing architecture.

### 3 SCRYPT ACCELERATED WITH NEAR-DATA-PROCESSING

As a preliminary investigation into scrypt’s vulnerability to compute-capable memory, we implement and evaluate the scrypt algorithm on a generic near-data-processing architecture.

#### 3.1 Generic NDP Architecture

Figure 1 describes the generic NDP architecture that our work is based on. NDP architectures are implemented via 3D die-stacked memory, in which a logic die is stacked together with multiple DRAM dies. The memory is divided into vertical sections, referred to as NDV vaults, and each NDP vault has a tightly coupled compute unit, referred to as the NDP core, placed in the stacked logic die. The NDP core’s low-latency memory access is enabled by its physical proximity to the NDP vault and the high-performance through-silicon via (TSV) interconnect. NDP cores are generally assumed to have minimal functionality with exclusive access to data in its coupled NDP vault. Data-intensive parts of computation can be offloaded to the NDP cores to exploit the low-latency memory access.

We assume that the NDP core is a simple, lightweight processor without cache. Instead, each NDP core is equipped with a small scratchpad memory to which data in the NDP vault can be read in via DMA. The scratchpad memory also stores the NDP core’s program memory, and a reserved portion of the scratchpad memory is memory-mapped into host address space for the NDP core’s communication with host processors.

#### 3.2 NDP-Aware scrypt Implementation

As described in Section 2, SMix makes up the memory-hard component of scrypt, and therefore we offload it to the NDP core. PBKDF$_2^{SHA256}$ computations are not memory-hard and are run on the host processor.

An SMix call runs entirely on a single NDP core-vault pair. The host processor communicates the 128$r$-byte input block $B$ and parameters $r$ and $N$ for SMix through the memory-mapped portion of the NDP core’s scratchpad memory. The output of SMix is also communicated back to the host via the memory-mapped region. The 128$r$-N-byte array $V$ generated in SMix (lines 7–9 of Algorithm 1)
Figure 2: The host-NDP interaction and data placement for the SMix function in the NDP-aware scrypt implementation.

Table 1: Evaluation framework details.

<table>
<thead>
<tr>
<th>Host Configuration</th>
<th>NDP Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>processor 8 in-order processors (ARMv7 Cortex-A15)</td>
<td>NDP core 1 in-order processor/vault (ARMv7 Cortex-A15)</td>
</tr>
<tr>
<td>L1 cache: 32kB icache, 64kB dcache, private, 2-way set-associative, 0.8 ns dcache access latency, 256B/block</td>
<td>scratchpad memory: 40kB/NDP core, stores program memory 8kB reserved for memory-map</td>
</tr>
<tr>
<td>L2 cache: 2MB, shared, 8-way set associative, 1.8ns access latency, 256B/block</td>
<td>DMA capability between scratchpad and NDP vault</td>
</tr>
<tr>
<td>memory: 2GB</td>
<td>NDP vault: 128MB/vault</td>
</tr>
</tbody>
</table>

does not have cache, every one of these xor instructions would incur DRAM operations to access the small portion of the block being xor-ed. Reading a word-length portion of interest from \( V_j \) in memory goes through the following process: the DRAM row containing the portion is activated, the corresponding columns are selected, and then the bits are transferred to the NDP core. Each of these steps with non-negligible delays would all be repeated for every word in \( V_j \), even though the DRAM row contains several contiguous words of \( V_j \). Therefore, the entire block \( V_j \) must be read into scratchpad memory using DMA in order to eliminate redundant DRAM row activations.

Table 2: Scrypt execution times on host and NDP with varying values of \( N (r = 8, p = 1) \).

<table>
<thead>
<tr>
<th>execution time (seconds)</th>
<th>host</th>
<th>NDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N = 16384 )</td>
<td>2.223813</td>
<td>1.507814</td>
</tr>
<tr>
<td>( N = 32768 )</td>
<td>4.455462</td>
<td>3.014112</td>
</tr>
<tr>
<td>( N = 65536 )</td>
<td>8.910643</td>
<td>6.026549</td>
</tr>
</tbody>
</table>

Table 3: Scrypt execution times on host and NDP with varying values of \( r (N = 16384, p = 1) \).

<table>
<thead>
<tr>
<th>execution time (seconds)</th>
<th>host</th>
<th>NDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r = 8 )</td>
<td>2.223813</td>
<td>1.507814</td>
</tr>
<tr>
<td>( r = 16 )</td>
<td>4.434392</td>
<td>3.002565</td>
</tr>
<tr>
<td>( r = 32 )</td>
<td>8.848431</td>
<td>5.986616</td>
</tr>
</tbody>
</table>

From the evaluation, we see that offloading the SMix function to the NDP core yields a 1.5x speedup in scrypt execution time, regardless of the \( N \) and \( r \) values. Note that this speedup would not be affected much by varying \( p \) either, for an increased \( p \) would only require more NDP core/vault pairs to run in parallel.

5 OPEN PROBLEMS & DISCUSSION

Parts of the scrypt algorithm have the potential to be further accelerated with compute-capable memory. For example, the Salsa20 stream cipher [7] used in BlockMix (line 17 of Algorithm 1) is simply bitwise add-rotate-xor operations repeated over several rounds on a 64-byte block, and the BlockMix function output is just a reordering of the 64-byte output blocks from Salsa20/8. These functions have the potential to be accelerated with specialized near-memory accelerators or even with processing-in-memory (PIM). In fact, computing bitwise operations in memory has been frequently explored in PIM research [1, 16, 21, 30, 39], but extending the prior PIM work to accelerate scrypt computation is still an open problem.

Scrypt is only one of many memory-hard cryptographic hash functions. Argon2 [9], Catena [20], Lyra2 [5], and yescrypt [36] are...
are all memory-hard password hashing algorithms that received recognition in the Password Hashing Competition. In particular, Argon2 was the winner of this competition, and its implementations using compute-capable memory would be interesting to look into.

More recently, memory-hard algorithms are being explored not only as password hashing algorithms, but also as proof-of-work (PoW) puzzles for blockchain mining. Ethash [19] (used in Ethereum [42]), Equihash [10] (used in Zcash [24]), and Cuckoo Cycle [41] (used in Cortex [14]) are some examples of memory-hard algorithms being used as Blockchain PoW puzzles. Building accelerators for these memory-hard PoW puzzles can undermine the tamper-proof quality of blockchains, making this an interesting area of future work. Wu et al. [43] have proposed a memory architecture-aware accelerator design for Ethash, but further work remains in applying compute-capable memory to accelerate memory-hard puzzles.

6 CONCLUSION

Our results show that even the simplest NDP hardware can yield a stable 1.5x speedup in evaluating the script function. Although the 1.5x speedup may not be a great threat to the security of script, we pose an important research question: how much can script be accelerated with compute-capable memory, and at what point would script be considered insecure?

ACKNOWLEDGEMENTS

We thank Erfan Azarkhish for his efforts in implementing the original MCSoCsim simulator [6] and providing it as open-source software. This work was supported by National Science Foundation grants 1561807 and 1908806.

REFERENCES


https://password-hashing.net/
Attacking Memory-Hard scrypt with Near-Data-Processing


