Attacking Memory-Hard scrypt with Near-Data-Processing

Jiwon Choe
Brown University
jiwon_choe@brown.edu

Tali Moreshet
Boston University
talim@bu.edu

R. Iris Bahar
Brown University
iris_bahar@brown.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:
Jiwon Choe, Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. 2019. Attacking Memory-Hard `scrypt` with Near-Data-Processing. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3357526.3357570

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 parellel 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

MEMSYS '19, September 30-October 3, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-7206-0/19/09…$15.00
https://doi.org/10.1145/3357526.3357570

Algorithm 1 `scrypt` algorithm

```
1:  function Scrypt(P, S, p, N, r, dkLen)
2:      (B_0||B_1||...||B_{p-1}) ← PBKDF2_SHA256(P, S, 1, 128rp)
3:      for i = 0 to p − 1 do
4:          B_i ← SMix_r(B_i, N)
        return PBKDF2_SHA256(P, B_0||B_1||...||B_{p-1}, 1, dkLen)
5:  function SMix_r(B, N)
6:      X ← B
7:      for i = 0 to N − 1 do
8:          V_i ← X
9:          X ← BlockMix_{Salsa20/8, r}(X)
10:     for i = 0 to N − 1 do
11:         j ← Integerify(X) mod N
12:         X ← BlockMix_{Salsa20/8, r}(X ⊕ V_j)
        return X
13: function BlockMix_{Salsa20/8, r}(B_0||B_1||...||B_{2r-1})
14: ▷ each B_i must be 64-bytes (enforced by Salsa20/8 definition)
15:     X ← B_{2r-1}
16:     for i = 0 to 2r − 1 do
17:         X ← Salsa20/8(X ⊕ B_i)
18:         Y_i ← X
        return Y_0||Y_2||...||Y_{2r-2}||Y_1||Y_3||...||Y_{2r-1}
```

`PBKDF2`$_{SHA256}$[1] to generate a $128rp$-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 `PBKDF2`$_{SHA256}$ 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`.

[1] `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.

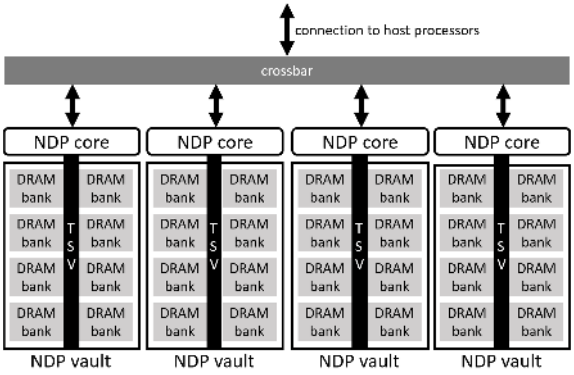

Figure 1: Generic near-data-processing architecture. Our investigations of `scrypt` with NDP are based on this generic architecture.

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 *NDP 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. `PBKDF2`$_{SHA256}$ 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 $128r$-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 $128rN$-byte array V generated in `SMix` (lines 7–9 of Algorithm 1)

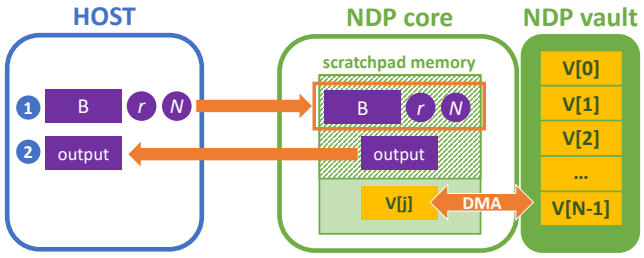

Figure 2: The host-NDP interaction and data placement for the SMix function in the NDP-aware scrypt implementation.

Table 1: Evaluation framework details.

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

is stored in the NDP vault. However, the pseudorandomly chosen $128r$-byte block V_j (lines 11–12) is always read into the scratchpad memory prior to the bitwise-xor operation in line 12. Figure 2 describes the host-NDP interaction and the data placement for the SMix function in the NDP-aware scrypt implementation.

Reading the random blocks into scratchpad memory is necessary in order to reduce redundant DRAM activity that causes delays and power consumption that cannot be reduced by NDP, as was identified in [15]. Because the NDP core is simple and does not have any sophisticated functionality, the bitwise-xor is expected to be executed as a sequence of simple xor instructions that operate on word-length data. Since the NDP core also 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.

4 EVALUATION

Our evaluations are made on Brown-SMCSim[2], a gem5 [8]-based cycle-accurate, full-system NDP architecture simulator with real hardware constraints. Table 1 summarizes the details of the evaluation framework.

We referred to code in the scrypt git repository [40] to implement the scrypt algorithm on Brown-SMCSim. Our host-based and NDP-based scrypt implementations and the Brown-SMCSim framework are available as open-source at https://github.com/jiwon-choe/Brown-SMCSim/tree/scrypt.

We compare the total execution time of scrypt with the SMix function executed on the host processor and on the NDP core. We varied the scrypt parameters for these measurements – Table 2 shows the execution times with varying values of N; table 3 shows the execution times with varying values of r. For all experiements, p was set to 1, and the desired key length was set to 64 bytes. We used the password and salt "pleaseletmein" and "SodiumChloride" that were used to generate some of the test vectors provided in the RFC [35].

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

execution time (seconds)		
	host	**NDP**
$N = 16384$	2.223813	1.507814
$N = 32768$	4.455462	3.014112
$N = 65536$	8.910643	6.026549

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

execution time (seconds)		
	host	**NDP**
$r = 8$	2.223813	1.507814
$r = 16$	4.434392	3.002565
$r = 32$	8.848431	5.986616

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 Salsa20/8 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]

[2]Originally SMCSim [6], extensively modified to conform to the NDP architecture design described in 3.1. Brown-SMCSim had been used for evaluations in [15].

are all memory-hard password hashing algorithms that received recognition in the Password Hashing Competition[3]. 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 `scrypt` function. Although the 1.5x speedup may not be a great threat to the security of `scrypt`, we pose an important research question: how much can `scrypt` be accelerated with compute-capable memory, and at what point would `scrypt` be considered insecure?

ACKNOWLEDGEMENTS

We thank Erfan Azarkhish for his efforts in implementing the original SMCSim simulator [6] and providing it as open-source software.

This work was supported by National Science Foundation grants 1561807 and 1908806.

REFERENCES

[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 481–492.

[2] Paula Aguilera, Dong Ping Zhang, Nam Sung Kim, and Nuwan Jayasena. 2016. Fine-Grained Task Migration for Graph Algorithms using Processing in Memory. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 489–498.

[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 105–117.

[4] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 336–348.

[5] Ewerton R Andrade, Marcos A Simplicio, Paulo SLM Barreto, and Paulo CF dos Santos. 2016. Lyra2: Efficient password hashing with high security against time-memory trade-offs. *IEEE Trans. Comput.* 65, 10 (2016), 3096–3108.

[6] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2016. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In *International Conference on Architecture of Computing Systems*. Springer, 19–31.

[7] Daniel J Bernstein. 2008. The Salsa20 family of stream ciphers. In *New stream cipher designs*. Springer, 84–97.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[9] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. 2016. Argon2: new generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 292–302.

[10] Alex Biryukov and Dmitry Khovratovich. 2017. Equihash: Asymmetric proof-of-work based on the generalized birthday problem. *Ledger* 2 (2017), 1–30.

[11] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 316–331. https://doi.org/10.1145/3173162.3173177

[12] Amirali Boroumand, Saugata Ghose, Brandon Lucia, Kevin Hsieh, Krishna Malladi, Hongzhong Zheng, and Onur Mutlu. 2017. LazyPIM: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters* (2017).

[13] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. 2019. CoNDA: Efficient Cache Coherence Support for Near-data Accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 629–642. https://doi.org/10.1145/3307650.3322266

[14] Ziqi Chen, Weiyang Wang, Xiao Yan, and Jia Tian. [n. d.]. Cortex-AI on Blockchain. ([n. d.]).

[15] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R Iris Bahar. 2019. Concurrent Data Structures with Near-Data-Processing: an Architecture-Aware Implementation. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19)*. ACM, New York, NY, USA. https://doi.org/10.1145/3323165.3323191

[16] Zamshed Chowdhury, Jonathan D. Harms, S. Karen Khatamifard, Masoud Zabihi, Yang Lv, Andrew P. Lyle, Sachin S. Sapatnekar, Ulya R. Karpuzcu, and Jian-Ping Wang. 2018. Efficient In-Memory Processing Using Spintronics. *IEEE Comput. Archit. Lett.* 17, 1 (Jan. 2018), 42–46. https://doi.org/10.1109/LCA.2017.2751042

[17] Palash Das, Shivam Lakhotia, Prabodh Shetty, and Hemangee K Kapoor. 2018. Towards Near Data Processing of Convolutional Neural Networks. In *VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), 2018 31st International Conference on*. IEEE, 380–385.

[18] Reetuparna Das. 2017. Blurring the Lines between Memory and Computation. *IEEE Micro* 37, 6 (2017), 13–15.

[19] Ethereum Wiki. 2017. Ethash. (2017). https://github.com/ethereum/wiki/wiki/Ethash

[20] Christian Forler, Stefan Lucks, and Jakob Wenzel. [n. d.]. *Catena: a memory-consuming password-scrambling framework*. Technical Report. Citeseer.

[21] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-Memory Data Parallel Processor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3173162.3173171

[22] M. Gao, G. Ayers, and C. Kozyrakis. 2015. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 113–124. https://doi.org/10.1109/PACT.2015.22

[23] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. 2016. Accelerating Linked-list Traversal Through Near-Data Processing. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/2967938.2967958

[24] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. 2016. Zcash protocol specification. *Tech. rep. 2016–1.10. Zerocoin Electric Coin Company, Tech. Rep.* (2016).

[25] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *Computer Design (ICCD), 2016 IEEE 34th International Conference on*. IEEE, 25–32.

[26] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. FloatPIM: In-memory Acceleration of Deep Neural Network Training with High Precision. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 802–815. https://doi.org/10.1145/3307650.3322237

[27] Mohsen Imani, Saransh Gupta, and Tajana Rosing. 2018. GenPIM: Generalized Processing In-Memory to Accelerate Data Intensive Applications. (2018).

[28] Dong-Ik Jeon, Kyeong-Bin Park, and Ki-Seok Chung. 2018. HMC-MAC: Processing-in Memory Architecture for Multiply-Accumulate Operations with Hybrid Memory Cube. *IEEE Comput. Archit. Lett.* 17, 1 (Jan. 2018), 5–8. https://doi.org/10.1109/LCA.2017.2700298

[29] Jeremie S Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. 2018. GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies. *BMC genomics* 19, 2 (2018), 89.

[30] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 173.

[31] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-Memory Computing. In *Proceedings of the 29th ACM*

[3] https://password-hashing.net/

Symposium on Parallelism in Algorithms and Architectures (SPAA '17). ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/3087556.3087582

[32] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 457–468.

[33] Lifeng Nai and Hyesoon Kim. 2015. Instruction Offloading with HMC 2.0 Standard: A Case Study for Graph Traversals. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS '15)*. ACM, New York, NY, USA, 258–261. https://doi.org/10.1145/2818950.2818982

[34] Colin Percival. 2009. Stronger key derivation via sequential memory-hard functions. (2009).

[35] C. Percival and S. Josefsson. 2016. *The scrypt Password-Based Key Derivation Function*. RFC 7914. RFC Editor.

[36] Alexander Peslyak. 2014. Yescrypt-a password hashing competition submission. *Password Hashing Competition. v0 edn* 14 (2014).

[37] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+ logic devices on MapReduce workloads. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 190–200.

[38] Paulo C Santos, Geraldo F Oliveira, João P Lima, Marco AZ Alves, Luigi Carro, and Antonio CS Beck. 2018. Processing in 3D memories to speed up operations on complex data structures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 897–900.

[39] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 273–287.

[40] Tarsnap. 2019. Tarsnap/scrypt. (2019). https://github.com/Tarsnap/scrypt

[41] John Tromp. 2015. Cuckoo cycle: a memory bound graph-theoretic proof-of-work. In *International Conference on Financial Cryptography and Data Security*. Springer, 49–62.

[42] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[43] Kun Wu, Guohao Dai, Xing Hu, Shuangchen Li, Xinfeng Xie, Yu Wang, and Yuan Xie. 2019. Memory-Bound Proof-of-Work Acceleration for Blockchain Applications. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*. ACM, New York, NY, USA, Article 177, 6 pages. https://doi.org/10.1145/3316781.3317862

[44] Yuan Xie. 2018. Intelligent Memory Architecture with New Memory Technologies. *Computer Architecture Today Blog* (2018). https://www.sigarch.org/intelligent-memory-architecture-with-new-memory-technologies/ https://www.sigarch.org/intelligent-memory-architecture-with-new-memory-technologies/.

[45] Salessawi Ferede Yitbarek, Tao Yang, Reetuparna Das, and Todd Austin. 2016. Exploring specialized near-memory processing for data intensive operations. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 1449–1452.

[46] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: throughput-oriented programmable processing in memory. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 85–98.

[47] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 544–557.