Distributed Password Hash Computation on Commodity Heterogeneous Programmable Platforms

Branimir Pervan
branimir.pervan@fer.hr
University of Zagreb, Croatia

Josip Knezovic
josip.knezovic@fer.hr
University of Zagreb, Croatia

Katja Pericin
katja@reversinglabs.com
ReversingLabs Ltd.

Abstract

In this paper, we present the Cool Cracker Cluster CCC: a heterogeneous distributed system for parallel, energy–efficient, and high–speed bcrypt password hash computation. The cluster consists of up to 32 heterogeneous nodes with Zynq-7000–based SoCs featuring a dual–core, general–purpose ARM processor coupled with FPGA programmable logic. Each node uses our custom bcrypt accelerator which executes the most costly parts of the hash computation in programmable logic.

We integrated our bcrypt implementation into John the Ripper, an open source password cracking software. Message Passing interface (MPI) support in John the Ripper is used to form a distributed cluster. We tested the cluster, trying different configurations of boards (Zedboards and Pynq boards), salt randomness, and cost parameters finding out that password cracking scales linearly with the number of nodes. In terms of performance (number of computed hashes per second) and energy efficiency (performance per Watt), CCC outperforms current systems based on high–end GPU cards, namely Nvidia Tesla V100, by a factor of 2.72 and 5 respectively.

1 Introduction

Password–based authentication and password hashing are currently the most common methods of securely storing user credentials in computer–based information systems and services. However, the theft of personal data from websites causes substantial damage in compromising such systems. In addition, these leakages helped the perpetrators to gain valuable information about user passwords compromising other, previously not breached systems. In order to prevent the password leakage once the hashes have been obtained, the only barrier remaining for the attacker is the cost of cracking the password by computing the hashes from the list of password candidates. In general, hashing functions are fast to evaluate and prone to offline dictionary attacks. In order to address this issue, hashing functions used in password protection include costly looping and heavy usage of resources [9, 11, 12]. This technique parametrized inside the hashing functions as the so–called *cost parameter* enables algorithm resistance against the improvements in computation capabilities of future hardware such as Field-Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs) and Graphic-Processing Units (GPUs). Three most common password hashing functions supporting adjustable cost parameters are PBKDF2 [9], bcrypt [12] and scrypt [11].

In password–hashing applications, hashing algorithms additionally utilize randomly generated, user–specific salts to prevent the pre–computed rainbow tables attacks which use reverse lookup tables of precomputed hashes in order to avoid costly computation for common passwords. Random salts force the attacker to repeatedly guess (compute) password candidates for each password–salt pair, even in the case when two different users have the same password. The cost to recover the user password depends on the cost of a single password hash computation and the number of guesses. This implies that the cost of cracking the password can be increased by either increasing the strength of the chosen password or increasing the strength of the cryptographic hash function used.

To overcome high computational power demand, parts of password–cracking software, as well as cryptography processing in general, can be offloaded to GPUs or programmable hardware. For example, [5] shows a speedup of nearly 4 times of RSA decrypt primitives executed on GPU. Sprengers and Batina [13] demonstrated the implementation of MD5 hashing for GPU, speeding up the calculation for 25 to 30 times over the CPU implementation. They also report other authors achieving speedups when implementing hash functions for GPU, from 4 to 20 times over traditional CPU implementations, namely Bernstein et al. [2, 3] (ECC), Manavski et al. [8] (AES) and Harrison et al. [4] (RSA). To demonstrate a more recent example, a Hashcat community managed to achieve $\sim$54k cracks per second (c/s) using NVIDIA Tesla V100 GPU [6].

When considering large–scale attacks on lists of hashed passwords, one must take energy consumption into consideration, due to the fact that a single attack can last on a scale of days or even months and that both CPUs and GPUs consume significant amounts of energy. Implementing password-cracking software (or parts of) in programmable hardware can both mitigate energy consumption and provide higher performance; energy consumption since FPGA or ASIC chips have traditionally small energy footprint and high performance since hash calculations (or parts of) would be accelerated on customized hardware. An example of FPGA–accelerated hash calculation, namely bcrypt hash, can be found in [14].

In our research, we decided to focus on bcrypt algorithm, mostly because of its popularity in password hashing for web–based systems. By varying its cost parameter, one can easily answer to the increase in conventional computation capabilities. On top of that, it is the default password hash algorithm for OpenBSD and some Linux distributions such as OpenSUSE. In addition, it is reasonably harder to implement efficient bcrypt in GPUs than in FPGAs because of the high demand for cache–unfriendly RAM access.

The rest of the paper is organized as follows: In Section 2 we give the background intro in terms of general bcrypt hashing and our single-node heterogeneous implementation with custom accelerator implemented in programmable logic, comparing it with other high–end platforms. The high–level architecture of the cluster is given in Section 3. Section 4 describes our experiments and methodology used to measure the cluster's performance and energy efficiency. We also report the results for various test cases. Lastly, Section 5 sums up our conclusions and contributions, while listing other perspectives for future work.

2 Background

This section introduces the bcrypt hashing method and our previous research on energy–efficient, single–node implementation of bcrypt on the Zedboard [1]: a Zynq-7000 SoC platform featuring a two-core ARM–based processing system (PS) coupled with FPGA programmable logic (PL) [7].

2.1 Bcrypt

Bcrypt is a password hashing method based on Blowfish block cipher, which is structured as a 16–round Feistel network [12]. The key features are tunable cost parameter and pseudorandom access to memory which together constitute compute–intensive part of the algorithm aimed to resist the brute–force attacks. Blowfish encryption uses a 64–bit input and a P-box (in bcrypt, initially holding the password being hashed) to calculate addresses used to randomly access four 1 KB large S-boxes. Bcrypt hashing scheme has two phases, as listed in Algorithm 1. The first phase uses the *EksBlowfishSetup* procedure from Algorithm 2 to initialize the state (line 1). Next, the state obtained is used with Blowfish encryption in the electronic codebook (ECB) mode to encrypt the 192–bit string "OrpheanBeholderScryDoubt" 64 times (lines 3 and 4). The returned value is the resulting password hash [12].

Algorithm 1 bcrypt(cost, salt, key) [12]

```
1: $state \leftarrow EksBlowfishSetup(cost, salt, key)$
2: $ctext \leftarrow$ "$OrpheanBeholderScryDoubt$"
3: $repeat(64)$
4:     $ctext \leftarrow EncryptECB(state, ctext)$
5: $return\ Concatenate(cost, salt, ctext)$
```

Blowfish encryption is used by the *EksBlowfishSetup* algorithm listed in Algorithm 2. The encryption is used by *ExpandKey* function (lines 2, 4 and 5) to derive the state determined by the values stored in S-boxes and P-box. This algorithm has three inputs: *cost*, *salt* and *key*. The *cost* parameterizes how expensive the key setup process is (line 3), while *salt* is a 128-bit random value used to prevent the same passwords having the same hash value or efficient guessing the common passwords with precomputed hashes. The third parameter, the encryption *key*, is the user–chosen password [12]. Lines 4 and 5 of Algorithm 2 are swapped in actual implementations, including in OpenBSD's original implementation that predates the USENIX 1999 paper by two years [12].

Algorithm 2 EksBlowfishSetup(cost, salt, key) [12]

```
1: $state \leftarrow InitState()$
2: $state \leftarrow ExpandKey(state, salt, key)$
3: $repeat(2^{cost})$
4:     $state \leftarrow ExpandKey(state, 0, salt)$
5:     $state \leftarrow ExpandKey(state, 0, key)$
6: $return\ state$
```

2.2 Single–node hardware evaluation

This section provides our single–node implementation performance updated with the results on the Pynq board and compared with new platforms such as high–end GPUs and CPUs. Details of the implementation, communication, and partitioning of the work between the PS and PL are provided in our previous work [7]. In general, we designed and implemented the bcrypt accelerator cores we used in the PL and integrated the design into the popular open source password cracking program – John the Ripper (JtR) [10]. JtR, which runs on PS, performs control–oriented tasks and prepares the password candidates, while PL containing our bcrypt accelerators executes the most costly loop of the Algorithm 2 (lines 3 through 5). Password candidates are sent to our bcrypt accelerator cores which compute hashes in parallel and send them back to PS for comparison with the unknown hash under attack. This work–partitioning enabled us to exploit advanced password generation schemes already available in JtR together

Table 1: Post-Implementation Resource utilization per node

Resource	Utilization	Available	Utilization (%)
LUT	34,073	53,200	64.05
LUTRAM	416	17,400	2.39
FF	7,919	106,400	7.44
BRAM	140	140	100.00

with intelligent password selections and permutation, such as using Markov chains to predict more frequent letters used in natural language.

Table 1 shows the resource utilization for our Zedboard implementation. The limiting resource is the Block RAM (BRAM) which we fully utilize. We were able to instantiate the maximum of 56 bcrypt instances in PL each computing two bcrypt hashes resulting in 112 computations performed in parallel.

Fig. 1 shows the performance of our single–node Zynq-7000 SoC–based implementation and compares it with other popular platforms used for password guessing. We present two metrics: performance as the number of computed hashes or cracks per seconds (c/s) and energy efficiency as the number of computed hashes per Watt–second (c/Ws). Blue–shaded bars show the results for cost 5, while green–shaded bars show the results for cost 12. Selected cost settings are chosen because cost 5 is traditionally used in benchmarking, and cost 12 is more suitable for practical uses of password storage. For energy efficiency metric, we took the methodology in favor of compared platforms (desktop–class CPUs and GPUs) for which we take the specified Thermal design power (TDP) into calculation (without the power consumed by other parts of the system), while for our implementations we use the whole system power consumed during the program execution. We show the results using the logarithmic scale.

The first set of bars represent the result of our implementation for the Zedboard platform [1] (XC7Z020 chip), the second set gives the results on the same platform reported by Wiemer and Zimmermann [14]. Next, we report our implementation for the simpler Pynq board [15] (same XC7Z020 chip), and for more PL abundant ZC706 board [16](XC7Z045 chip). Finally, we report the results obtained for Intel's Xeon Phi 5110P (obtained from [7]), Nvidia's Tesla V100 16GB (obtained and extrapolated from cost 5 for cost 12 from [6]), Nvidia's GTX 1080 FE GPU (obtained from [6]), Intel Core i7-4510U 2.0 GHz CPU, and AMD Ryzen 7 1800X CPU. For all conducted tests on other platforms (CPUs and GPUs), we used the same *bleeding–jumbo* 1.8 version of JtR available at the Openwall's Git repository [10]. For our heterogeneous platforms, we replaced software–based bcrypt version with our accelerated version in PL. For CPU–based tests: Intel Core i7 and Amd Ryzen 7 we compiled JtR using OpenMP support and run the tests on Linux–based operating system using 8 and 16 GB memory respectively.

Compared to CPUs, our heterogeneous platforms achieve comparable performances with substantially better energy efficiency for both cost parameters. In fact, our Zynq-7045–based implementation has performance comparable to highly performant GPUs and Xeon Phi with an order of magnitude better energy efficiency. For cost 5, results obtained by Weiemer and Zimmerman for the same platform (Zedboard) are slightly better than ours. However, our cost 12 performance and energy efficiency slightly outperform their Zedboard results. Finally, our Zynq-7045 implementation (approximately 4 times more logic than Zynq-7020, and thus bcrypt cores) outperforms all compared CPU and GPU platforms in terms of energy efficiency (Xeon Phi and GTX 1080 FE for two orders of magnitude), and all but Nvidia's Tesla V100 in terms of performance. It also outperforms Zynq-7020–based implementations in terms of performance while retaining comparable energy efficiency. Note that energy efficiencies of Xeon Phi and GTX 1080 FE are evidently poor (below 1 c/Ws for cost 12), making them unusable for energy–aware real password attacks. On the other hand, our heterogeneous platforms with initial purchase costs comparable to high–end GPUs and Xeon Phi cards, and with orders of magnitude better energy efficiency could be effectively used for long–term password attacks, even for the cost settings of 10 and more that are used in contemporary password–based systems.

3 Cool Cracker Cluster CCC

In this section we describe our cluster–based implementation of bcrypt–accelerated **Cool Cracker Cluster CCC**. We exploit the fact that dictionary–based password cracking is an embarrassingly parallel problem: if a dictionary of password candidates is given, one simply has to divide the work of processing different segments of the dictionary and orchestrate it to available computational nodes.

Fig. 2 illustrates the whole hardware/software stack used in CCC . The computational unit in our cluster is a single node, in our case Zedboard [1] or the Pynq board [15]. Nevertheless, any board containing Zynq–based SoC could be used, which we exploit by extending our design to ZC706 board, although without adding it to the cluster. Zedboard and Pynq board offer equivalent computational capability in terms of cracks per second thus making our cluster consist of homogeneous nodes. Every node runs the JtR on the Linux–based OS with our bcrypt accelerator implemented in PL [7]. We utilize the Message Passing Interface (MPI) programming model integrated into the JtR for distributed password cracking. To form the cluster, we used standard Ethernet to interconnect computational nodes, making networking relatively easy and cheap in terms of infrastructure and energy consumption. In the cluster, one node acts as a master while the rest of the nodes are slaves. The master is responsible for:

- Starting, stopping, orchestrating and distributing work

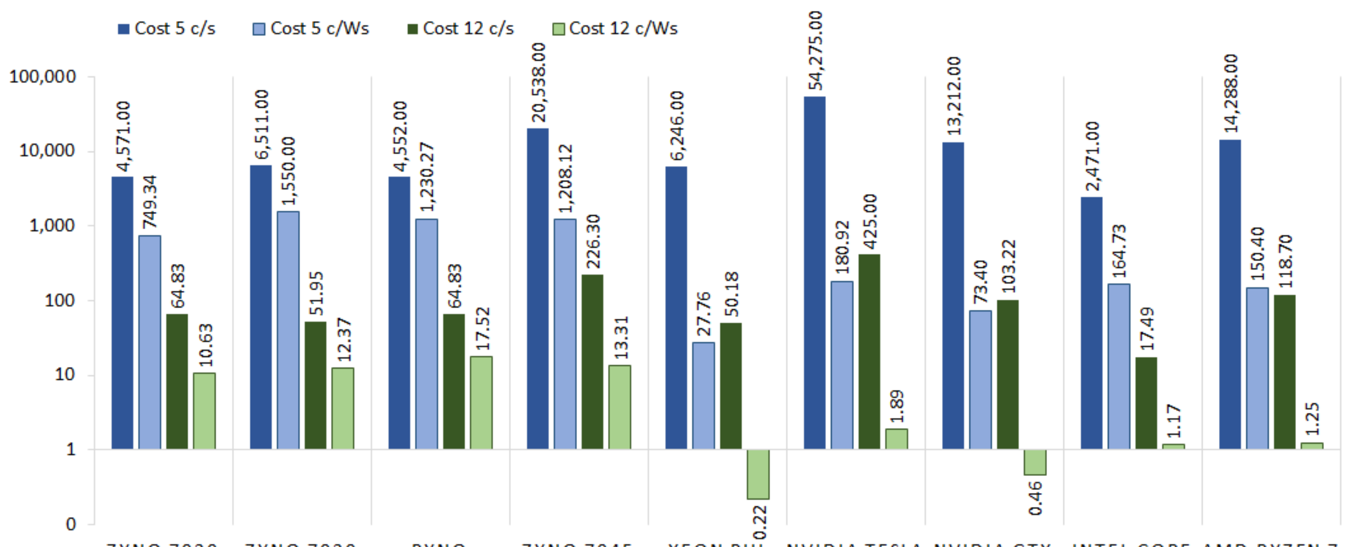

Figure 1: Performance and energy efficiency of various platforms

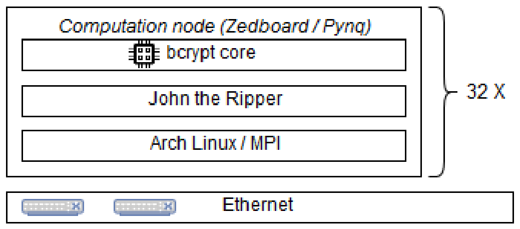

Figure 2: Block diagram of Cool Cracker Cluster

packages of a cracking job within slaves and himself;

- Syncing in-job events – since the cluster cracks one bcrypt hash at the time, synchronization has to be made if one computational node successfully cracks a hash or if a computational node finishes calculating hashes for its own portion of password candidates;
- Hosting the NFS network file system used to exchange data and events, dictionary with password candidates and list of hashes to be cracked.

We used one Zedboard as the master managing the job, making it a permanent resident of the cluster even in configurations with Pynq boards (i.e. configuration with 16 Pynqs, in fact, consists of 1 Zedboard and 15 Pynq boards). The most powerful cluster contained 32 heterogeneous computational nodes, 8 Zedboards (1 of which was the master) and 24 Pynq boards interconnected with two network switches. We were not able to reach experimentally the upper bound on the number of computational nodes, although we could argue that, at some point, the cluster scaling reaches the point of diminishing results due to initial setup time exceeding computing time together with the overhead of managing the large cluster.

4 Cluster Results

This section presents the results obtained by testing the CCC and describes the methodology we used to measure performance and energy efficiency.

4.1 Methodology

Table 2 presents the configurations of the CCC which we tested with the same parameters:

- A hashlist consisting of hashes of 5 preselected test passwords,
- A wordlist consisting of 2 million leaked passwords from the RockYou breach.

We modified the wordlist (dictionary) by randomly inserting our 5 test passwords in order to make the cluster nodes find the matching hash(es). To test the cluster thoroughly, passwords in the testing hashlist were hashed with two different cost settings (Algorithm 2, line 3) with both random and fixed salts, yielding four different testing hashlists. Although real applications of bcrypt algorithm commonly use higher costs, we retained cost 5 since the community uses it for benchmarking purposes. With cost 10 we tried to get an idea of how the cluster would behave on higher costs since using even higher costs would render the cluster unusable for testing if we wanted to retain consistency. For example,

Table 2: Configurations table

#	Consisted of	Number of concrete nodes		
		Zedboards	Pynqs	Switches
32	Mixed	8	24	2
24	Pynqs	1	23	2
24	Mixed	8	16	2
16	Pynqs	1	15	1
16	Mixed	8	8	1
8	Pynqs	1	7	1
8	Zedboards	8	0	1
8	Mixed	4	4	1
4	Pynqs	1	3	1
4	Zedboards	4	0	1
4	Mixed	2	2	1
2	Zedboards	2	0	1
2	Mixed	1	1	1
1	Pynq	0	1	0
1	Zedboard	1	0	0

Table 3: Idle power consumption [W] and integrated test performance [c/s]

#	Consisted of	Power [W]	Performance [c/s]
32	Mixed	125.7 ± 0.3	$146{,}333.6 \pm 96.8$
24	Pynqs	85.2 ± 0.3	$109{,}737.6 \pm 75.6$
24	Mixed	100.9 ± 0.3	$109{,}785.4 \pm 310.2$
16	Pynqs	58.2 ± 0.1	$73{,}113.8 \pm 223.7$
16	Mixed	75.6 ± 0.1	$73{,}175.4 \pm 217.2$
8	Zedboards	51.3 ± 0.1	$36{,}663.2 \pm 153.5$
8	Pynqs	33.5 ± 0.1	$36{,}545.6 \pm 39.2$
8	Mixed	40.0 ± 0.1	$36{,}570.9 \pm 112.0$
4	Zedboards	28.1 ± 0.1	$18{,}277.0 \pm 95.9$
4	Pynqs	20.3 ± 0.1	$18{,}289.6 \pm 90.8$
4	Mixed	23.1 ± 0.1	$18{,}237.9 \pm 57.2$
2	Zedboards	17.0 ± 0.1	$9{,}129.4 \pm 47.0$
2	Mixed	14.4 ± 0.0	$9{,}091.6 \pm 47.0$
1	Zedboard	5.1 ± 0.1	$4{,}564.7 \pm 5.0$
1	Pynq	2.7 ± 0.1	$4{,}552.6 \pm 4.7$

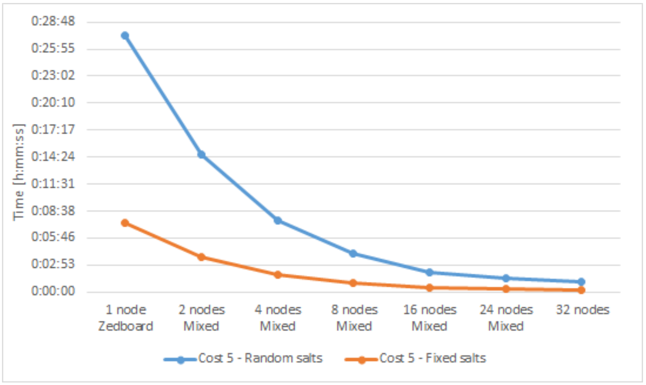

Figure 3: Total cracking time for cost 5 and mixed nodes

under the aforementioned assumptions, for cost 12 and one computational node, cracking would take more than 24 hours. Real applications generate random salts for every password which is to be hashed. We tested our cluster with fixed salts hashes as well to implicitly demonstrate how salt randomness radically improves the security of the hashed passwords and consequently the whole information system.

Every test set was run four times, on a clean startup of the cluster. The first measurement was discarded as it was only used to prevent "cold start" of the cluster, helping eliminate any known or unknown potential cache or similar effect. Presented results are statistically composed of the last three measurements in the form of average and standard deviation. Energy consumption of the cluster (compute nodes and network switches) was measured using high–quality wattmeter which sampled power network with a frequency of 1 Hz. After the test had started, we waited for approximately 30 seconds (latency needed to let the power consumption stabilize) and then took 10 measurement samples from the watt meter. Additionally, we measured the performance of the JtR's integrated testing (–*test*) option. The result shown in Table 3 is the average of 10 successive runs of JtR with –*test* flag. Idle power consumption was measured using the same methodology as when the cluster was under load. When idle, 10 samples were taken after the cluster stabilized.

4.2 Results

Performance and scaling results are presented using processed data tables and visualized through graphical representations. Due to the limited space and thus inability to completely describe test results, we chose to display Mixed configurations (Table 2) to balance between the difference in energy efficiency between Zedboards and Pynqs. This does not present a shortcoming since the performance of different permutations within the same number of computational nodes stays relatively constant and differs less than 0.5%. Furthermore, efficiency consistency is more important than the raw and absolute value of energy efficiency. On the other hand, to maximize energy efficiency, one should favor Pynq–based configurations, since we measured Pynq's power consumption to be near 46% lower than Zedboard's. This was most likely due to having fewer peripherals and simpler design, since it features the same SoC and power supply as Zedboard. Ground case data is shown in Table 3. Given figures present near-linear scaling minding the logarithmic distribution of values on the horizontal axis.

Fig. 3 shows the time required to finish the whole guessing job specified by subsection 4.1 for cost 5 with both fixed and random salts. Fixed salts times are significantly shorter for

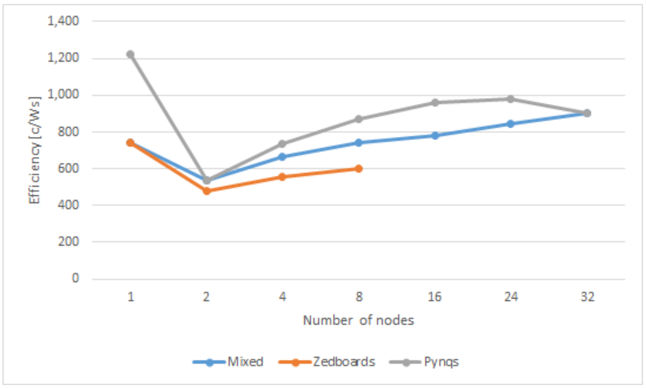

Figure 4: Performance–Power ratio for cost 5

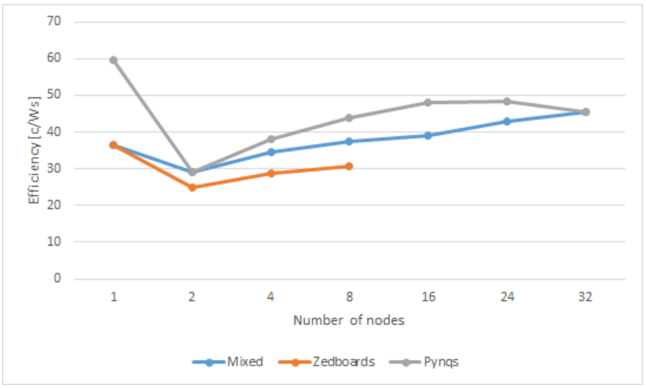

Figure 6: Performance–Power ratio for Cost 10

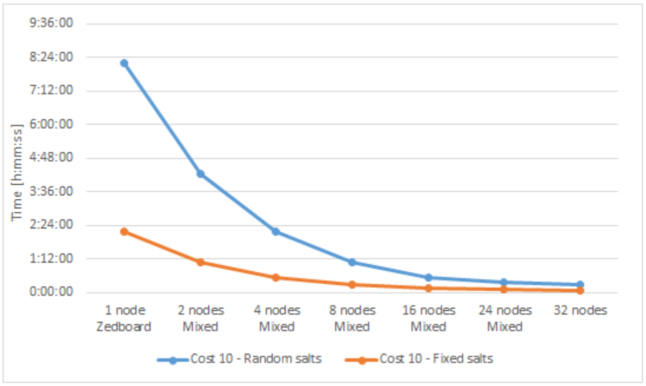

Figure 5: Total cracking time for cost 10 and mixed nodes

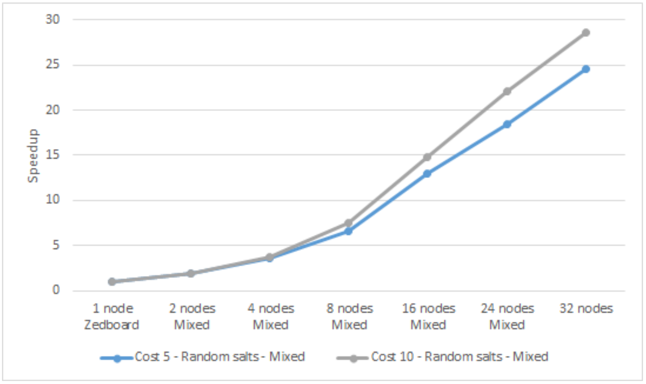

Figure 7: Cracking speedup scaled to 1 Zedboard

low node count due to JtR's generation of on–the–fly Rainbow tables. We obtained the linear scaling of the performance, i.e. linear decrease in job time for both cases. Fig. 4 gives the efficiency in c/Ws for cost 5 setting and for various configurations of cluster nodes. We observe a drop on 2 nodes when we move from Single node to MPI–based cluster which includes network switches. However, the gap is reduced as the number of nodes increases. As expected, Pynq–based configuration demonstrate the best energy efficiency. High drop in Pynq test case is caused by the fact that 2 node cluster, besides Pynq, featured one Zedboard (energy significantly more inefficient) and an additional network switch. However, we can as well observe the linear growth of efficiency after the initial drop. Mixed configuration generates false superlinear scaling since, on higher node count, more energy–efficient Pynq boards outnumber Zedboards.

Fig. 5 and 6 illustrate the same measurements for cost 10: total job time, and energy efficiency respectively. We again obtain a scalable linear decrease in cracking time, implying the increase in performance. Again, fixed salts times are significantly shorter for low node count due to JtR's generation of on–the–fly Rainbow tables. Energy efficiency, which is significantly lower than for cost 5 due to increased computational complexity, experiences the same drop when we move from a single node to multi–node cluster but on a much smaller scale than drop when running cost 5 test set. We observe less steep growth of efficiency due to smaller performance and less variance in performance in comparison to cost 5. It is worth mentioning that cost 10 can be considered to be used in production systems and that, with our cluster, a relatively trivial password can be efficiently revealed under 20 minutes.

Fig. 7 shows the speedup of our mixed CCC cluster configurations normalized to our initial single–node Zedboard for cost 5 and 10 with random salts. We observe linear scaling of the cluster cracking performance. We achieved slightly better scaling for cost 10 (speedup 28.55 for 32 nodes, 22.07 for 24 nodes) than for the cost 5 (speedup 24.65 for 32 nodes, 18.44 for 24 nodes).

Table 4 compares the cracking performance and energy efficiency of our 32–node CCC and NVIDIA Tesla V100 GPU as reported by Hashcat, another popular password guessing software [6]. The results shown are for cost 5 with random

Table 4: Nvidia Tesla V100–SXM2–16GB vs. cCc

Platform	Price	Performance	Efficiency
Tesla V100	$\sim$6100$	$\sim$54k c/s	$\sim$180 c/Ws
cCc (32 nodes)	$\sim$6400$	$\sim$147k c/s	$\sim$900 c/Ws

salts. Our cCc cluster outperforms Tesla GPU in both performance by a factor of 2.7 achieving $\sim$147k c/s compared to $\sim$54k c/s achieved with Tesla V100, and efficiency by a factor of 5 ($\sim$900 c/Ws compared to Tesla's $\sim$180 c/Ws. As for the initial acquisition costs, single Pynq board cost approximately $200 which makes $6400 for the 32–node system which in the roughly in the same price range of Tesla 16GB V100 card.

5 Conclusion and future work

In this paper, we present the heterogeneous distributed system for parallel and energy–efficient bcrypt password hash computation. The cluster consists of up to 32 heterogeneous nodes containing a dual–core ARM CPU PS and PL. We used the PL to implement our custom bcrypt accelerator. We further extend our design into the MPI–based distributed cluster of heterogeneous nodes, each capable to efficiently compute bcrypt hashes for provided password candidates. In terms of performance and energy efficiency, our cluster scaled well in both performance (c/s), and energy efficiency (c/Ws). Furthermore, this type of clustering has shown itself as a relatively simple yet powerful model, because heterogeneous boards used as computational nodes are: energy efficient, cheap, equipped with PL capable of performing accelerated tasks efficiently, and capable of running full–blown operating system which eases the programming of such platforms.

Our Future research will include investigating into heterogeneity on a higher level: how to incorporate multiple boards with different SoC chips into a functional cluster. By introducing additional, even more energy–efficient nodes, we aim to further increase the energy efficiency of the cluster. Currently, computational nodes in the cluster are tightly coupled, mostly due to using MPI in a communication and parallelization layer. Also, support for resilience, dynamic addition of nodes to the cluster, and dynamic work partitioning is in our future plans. We would also like to further optimize our hardware implementation of the bcrypt accelerator core and investigate into task scheduling techniques which would consider the optimal exploitation of the computational resources on nodes of different processing capabilities, i.e. mixing SoCs with different PL size.

References

[1] AVNET. Zedboard, March 2019.

[2] Daniel J. Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe, and Bo-Yin Yang. Ecc2k-130 on nvidia gpus. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010*, pages 328–346, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[3] Daniel J Bernstein, Hsueh-Chung Chen, Ming-Shing Chen, Chen-Mou Cheng, Chun-Hung Hsiao, Tanja Lange, Zong-Cing Lin, and Bo-Yin Yang. The billion-mulmod-per-second pc. In *Workshop record of SHARCS*, volume 9, pages 131–144, 2009.

[4] Owen Harrison and John Waldron. Practical symmetric key cryptography on modern graphics hardware. In *USENIX Security Symposium*, volume 2008, 2008.

[5] Owen Harrison and John Waldron. Efficient acceleration of asymmetric cryptography on graphics hardware. In Bart Preneel, editor, *Progress in Cryptology – AFRICACRYPT 2009*, pages 350–367, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[6] hashcat. hashcat - advamced password recovery, March 2019.

[7] Katja Malvoni, Solar Designer, and Josip Knezovic. Are your passwords safe: Energy-efficient bcrypt cracking with low-cost parallel hardware. In *8th {USENIX} Workshop on Offensive Technologies ({WOOT} 14)*, 2014.

[8] Svetlin A Manavski et al. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. *Signal Processing and Communications*, 2007, 2007.

[9] Ed K. Moriarty, B. Kaliski, and A.Rusch. Pkcs #5: Password-based cryptography specification version 2.1. RFC 8018, RFC Editor, January 2017.

[10] Openwall. John the Ripper password cracker, March 2019.

[11] Colin Percival and Simon Josefsson. The scrypt Password-Based Key Derivation Function. RFC 7914, RFC Editor, August 2016.

[12] Niels Provos and David Mazières. A Future-Adaptable Password Scheme. *Proceedings of the FREENIX Track:1999 USENIX Annual Technical Conference*, 1999.

[13] Martijn Sprengers and Lejla Batina. Speeding up gpu-based password cracking. SHARCS, 2012.

[14] Friedrich Wiemer and Ralf Zimmermann. High-speed implementation of bcrypt password search using special-purpose hardware. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–6, Dec 2014.

[15] Xilinx. Pynq: Python productivity for zynq, March 2019.

[16] Xilinx. Xilinx Zynq-7000 SoC ZC706 Evaluation Kit, March 2019.