Taylor & Francis
Taylor & Francis Group

Automatika

Journal for Control, Measurement, Electronics, Computing and Communications

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/taut20

Energy-efficient distributed password hash computation on heterogeneous embedded system

Branimir Pervan, Josip Knezović & Emanuel Guberović

To cite this article: Branimir Pervan, Josip Knezović & Emanuel Guberović (2022) Energy-efficient distributed password hash computation on heterogeneous embedded system, Automatika, 63:3, 399-417, DOI: 10.1080/00051144.2022.2042115

To link to this article: https://doi.org/10.1080/00051144.2022.2042115

Published online: 28 Feb 2022.

AUTOMATIKA
2022, VOL. 63, NO. 3, 399–417
https://doi.org/10.1080/00051144.2022.2042115

REGULAR PAPER

OPEN ACCESS Check for updates

Energy-efficient distributed password hash computation on heterogeneous embedded system

Branimir Pervan [a], Josip Knezović [a] and Emanuel Guberović [a,b]

[a]Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia; [b]Green Light Technologies Ltd., Zagreb, Croatia

ABSTRACT

This paper presents the improved version of the cool Cracker cluster (cCc), a heterogeneous distributed system for parallel and energy-efficient bcrypt password hash computation. The cluster consists of low-power heterogeneous embedded systems with programmable logic containing specialized hash computation accelerators, namely Xilinx Zynq-series SoC boards and ZTEX 1.15y board, with different performances measured in bcrypt hash computations per second [H/s]. Zynq based nodes use the improved version of our custom bcrypt accelerator. The cluster was formed around the open-source password cracking software package John the Ripper (JtR). To mitigate the different performances among the cluster nodes, we developed a password candidate distribution scheme based on the passwords' probability distribution. We performed an extensive evaluation of the cluster and the proposed distribution scheme. We also compared our cluster with various GPU implementations in terms of performance, energy-efficiency, and price-efficiency. We show that our solution outperforms other platforms such as high-end GPUs, by a factor of at least 3 in terms of energy-efficiency. In terms of the total operational costs, our cluster pays off after 4500 cracked passwords for a bcrypt hash with cost parameter 12, which makes it more appealing for real-world password-based system attacks.

ARTICLE HISTORY
Received 17 November 2021
Accepted 21 January 2022

KEYWORDS
Bcrypt; distributed computing; energy efficiency; heterogeneous hardware

1. Introduction

As more and more aspects of our lives become closely intertwined with the use of various Internet-based applications, many of which have only emerged in recent years, password-based authentication is still the most popular method of gaining access to them [1]. Although various alternatives, such as graphic based authentications, have been proposed [2], the simplicity of text-based passwords make their replacement a slow and even unlikely process.

The widespread use of password-based systems emphasized one of the major drawbacks they bring – the security vulnerability that ordinary user-generated passwords contain, as they are easily affected by common password guessing attacks. Although most people are aware of the importance of choosing strong passwords to protect their personal information, they often lack the necessary motivation to follow the suggested guidelines that could help them create stronger passwords [3]. The human aspect of this problem increases with the number of passwords people have to manage in their daily lives, as they use more and more password-based authorized applications. By using weaker passwords, they are often relieved of the burden of having to remember a larger number of passwords [4].

It is interesting to define password strength in terms of a very common attack method, the brute force attack, which attempts to find passwords for every possible combination in a given character set. Stronger passwords take a longer time to be cracked with brute force attacks, as more combinations need to be checked before the correct one is found. In general, password strength can be linked to the information content or entropy of the password, and both the length and the size of the character set size affect its value.

Entropy H is given as:

$$H = L * \log_2 N,$$

where L is password length and N is the character set size. It is noticeable that password length increase has a stronger effect on entropy then the increase in character set size. Some of the popular password strength metric services, such as "Password Meter" [5] and "How Secure is my Password" [6], use a bit more complex custom-crafted rule sets for their password evaluations.

In common server-client architectures, user data management, including authorization data, is stored at centralized locations. This creates an opportunity for potential attackers to steal this data for malicious usage. If text-based passwords were stored in their original form, with every successful attack they would be completely compromised. Encrypting password data with a special key helps to mitigate this problem to an extent,

CONTACT Branimir Pervan branimir.pervan@fer.hr Faculty of Electrical Engineering and Computing, University of Zagreb, Unska 3, Zagreb, Croatia

however, if a hacker gains access to the key itself, it is trivial to decrypt the encrypted data itself.

Hashing functions create irreversible password hashes, forcing potential attackers to hash many combinations and try to match them to the values that are stored. Attackers often use precomputed tables with cached outputs of hash functions called rainbow tables [7]. Adding random prefixes or postfixes, called salts, to passwords before applying hash functions invalidates common rainbow tables and makes brute-force attacks less effective, making hashed passwords further secured.

The three most common password hashing algorithms are PBKDF2 [8], bcrypt [9] and scrypt [10], with PBKDF2 and bcrypt (with reasonable settings of the cost parameter) being recommended by both OWASP [11] and IETF [12]. The process of updating or changing hash algorithms that are used is not completely trivial. One option, albeit very user-unfriendly, is to simply expire old user passwords and require users to enter new password values. Another option is to use the old hashed value as an input to the new hashing algorithm until the next time user successfully signs in when the new hash value is exchanged. Our research focuses on bcrypt algorithm, mostly because of its popularity in password hashing for web-based systems, and the legacy code base that will not be updated with the aforementioned algorithm update process and will use bcrypt for many years to come.

With the recent increase in computational power that attackers have at their disposal to conduct brute force attacks, the overall password strength needs to increase as well. In addition, bcrypt parameterizes the hash computation complexity with the cost parameter, which enables adjusting the space and compute complexity of the algorithm to match the increased performance of new hardware.

This paper provides the following contributions:

- Bcrypt accelerator in the embedded system with general-purpose CPU and FPGA logic, and the cluster of devices. We improved our previous accelerator hardware design and did appropriate software changes. The resulting optimization of the critical path in our design enabled us to improve the accelerator performance 1.5 times.
- Task-distribution algorithm for password candidates. We describe our task-distribution algorithm used to load balance among the nodes with different hash computation power in the cluster.
- We provide an in-detail analysis and experimental results on simulated real-world password cracking scenarios.

The rest of the paper is organized as follows: in Section 2, we give a brief description of the bcrypt hashing algorithm and the overview of the related work in the field of general password cracking and cracking on special-purpose hardware such as FPGA SoCs and GPUs. In Section 3, we describe single node implementation with, together with the architecture of our bcrypt accelerator implemented in programmable logic. We also describe the improvement of our accelerator on hardware and software levels, and briefly describe the evaluation of the single node implementation in terms of other notable examples in the field of password cracking on special-purpose hardware. In Section 4, we describe the architecture of our cluster as a distributed computing system and present our work distribution algorithm which minds for various performances in terms of bcrypt hash computations per second between different nodes in the cluster. In Section 5.2, we present testing methodology, experimental results in real-world scenarios, and the analysis of the cost-effectiveness of the cluster, minding the relatively high initial price. Finally, in Section 6, we conclude the paper with proposals for future research.

2. Background

In this section, we provide a brief overview of the bcrypt hashing algorithm and an overview of the related work in the field of hash function computation on specialized hardware.

2.1. bcrypt

Bcrypt [9] is a password hashing function designed by Provos and Maziéres, based on Blowfish block cipher [13], structured as a 16-round Feistel network. The key feature of the bcrypt hashing scheme is its adaptiveness, embodied through the tunable cost parameter, intended to increase as the computational strength progresses with the development of the hardware. Each increment of the cost parameter results in an exponential increase of the computational demand required to compute the hash. The cost parameter, coupled with pseudorandom access to memory, constitutes the compute- and memory-intensive part of the algorithm, making bcrypt resistant to brute-force attacks. Bcrypt also incorporates randomly generated salts by default to protect against rainbow (or reverse lookup) table attacks.

Algorithm 1 bcrypt [9].

```
1:  function BCRYPT(cost, salt, key)
2:      state ← EksBlowfishSetup(cost, salt, key)
3:      ctext ← "OrpheanBeholderScryDoubt"
4:      i = 0
5:      while i < 64 do
6:          ctext ← EncryptECB(state, ctext)
7:          i = i + 1
8:      end while
9:      return Concatenate(cost, salt, ctext)
10: end function
```

Bcrypt hashing scheme consists of two phases as shown in Algorithm 1. The first phase initializes the Blowfish state with the key setup Algorithm 2 (line 2). The obtained state is then used to encrypt the 192-bit string "OrpheanBeholderScryDoubt" 64 times with Blowfish encryption in the electronic codebook (ECB) mode (lines 5 and 6). Finally, a concatenated string consisting of cost, salt and hash value is returned presenting the final password hash (line 9) as illustrated below:

$$2b[cost][128 - bit\ salt][192 - bit\ hash\ value].$$

The first part of the hash is version, currently, *2b* as of February 2014, followed by the cost parameter used to generate the resulting hash. The rest of the hash is 16-byte (128 bits) salt and 24-byte (192 bits) hash value, both Radix-64 encoded to 22 and 31 characters, respectively.

Figure 1 shows the block diagram of the Blowfish encryption algorithm presented as the symmetric Feistel network with large S-boxes with randomized accesses dependant on the password to prevent fast access through caches in general-purpose CPU implementations. The algorithm consists of 16 rounds, each splitting input into left (L) and right (R) 32-bit halves and using P-box (denoted as K in the figure to avoid confusion since P is often used for denoting plaintext) and four S-boxes to perform the XORs, swaps and so-called F-function (which randomizes the looks-up into the S-boxes).

In Algorithm 1 (bcrypt), Blowfish encryption is used by the *EksBlowfishSetup* (line 2) and by *EncryptECB* (line 6). *EksBlowfishSetup*, a modified version of original Blowfish encryption algorithm called Expensive Key Schedule Blowfish, is in the core of bcrypt. It is shown in Algorithm 2 and uses the Blowfish encryption at lines 2, 3, 6 and 7. Inputs to the *EksBlowfishSetup* are: *cost*, *salt* and *key*. The *cost* parameterizes the expensiveness of the key setup process (line 5), *salt* is a random 128-bit value, and the third parameter (*key*) is the user-chosen password truncated to first 72 bytes.

InitState (line 2 of Algorithm 2) uses Blowfish encryption to populate P- and S-boxes with fractional parts of number π. Line 3 takes *salt* and *key* to permutate the values in P- and S-boxes, again using Blowfish. Lines 6 and 7 use Blowfish encryption in *ExpandKey* function to derive the state determined by the values stored in S-boxes and P-box. In actual implementations, including OpenBSD's original implementation, lines 6 and 7 are swapped.

It is important to note that many implementations of bcrypt truncate the input user password to the first 72 bytes and that the maximum value for the cost parameter is 31.

Algorithm 2 Expensive Blowfish key setup [9].

```
1:  function EksBlowfishSetup(cost, salt key)
2:      state ← InitState()
3:      state ← ExpandKey(state, salt, key)
4:      i = 0
5:      while i < 2^cost do
6:              state ← ExpandKey(state, 0, salt)
7:              state ← ExpandKey(state, 0, key)
8:          i = i + 1
9:      end while
10:     return state
11: end function
```

2.1.1. Cost extrapolation

As stated before, bcrypt hashing algorithm is adaptable by adjusting the expensiveness of the key setup process. The number of iterations of the loop (line 5) in the EksBlowfishSetup algorithm (2) exponentially depends on the cost, which enables us to approximate the performance of an algorithm implementation on a certain platform, by using the following equation:

$$X_{c_2} = X_{c_1} * 2^{(c_2 - c_1)}, \tag{1}$$

where X_{c1} is known (measured) performance of bcrypt implementation with cost parameter set to c_1, and c_2 is cost parameter we are approximating performance for on the same platform. For example, if performance for the cost 5 equals 1000 regardless of the metric, performance for cost 7 would equal 250 since the algorithm is approximately 4 times slower for cost 7 compared to cost 5.

2.2. Related work

Password cracking and strength evaluation is a compelling area that attracted an ample number of researchers in the previous decades intending to contribute to the problem of using weak user passwords in the distributed applications that use password authentications.

In [14], authors made an extensive empirical study on common password cracking techniques on real-world password datasets, varying in both application domain and user localization. Techniques used include a variety of state-of-the-art techniques for password guessing, including dictionary attacks, mangling using dictionaries and probabilistic context-free grammars, and Markov chain-based strategies. Conclusively, the study showed that the 'diminishing returns' principle applies, resulting in weak passwords being more common in the absence of an enforced password strength policy. The authors proposed proactive password checkers for users and security auditing tools that use an approximation of the number of guesses needed to crack the password.

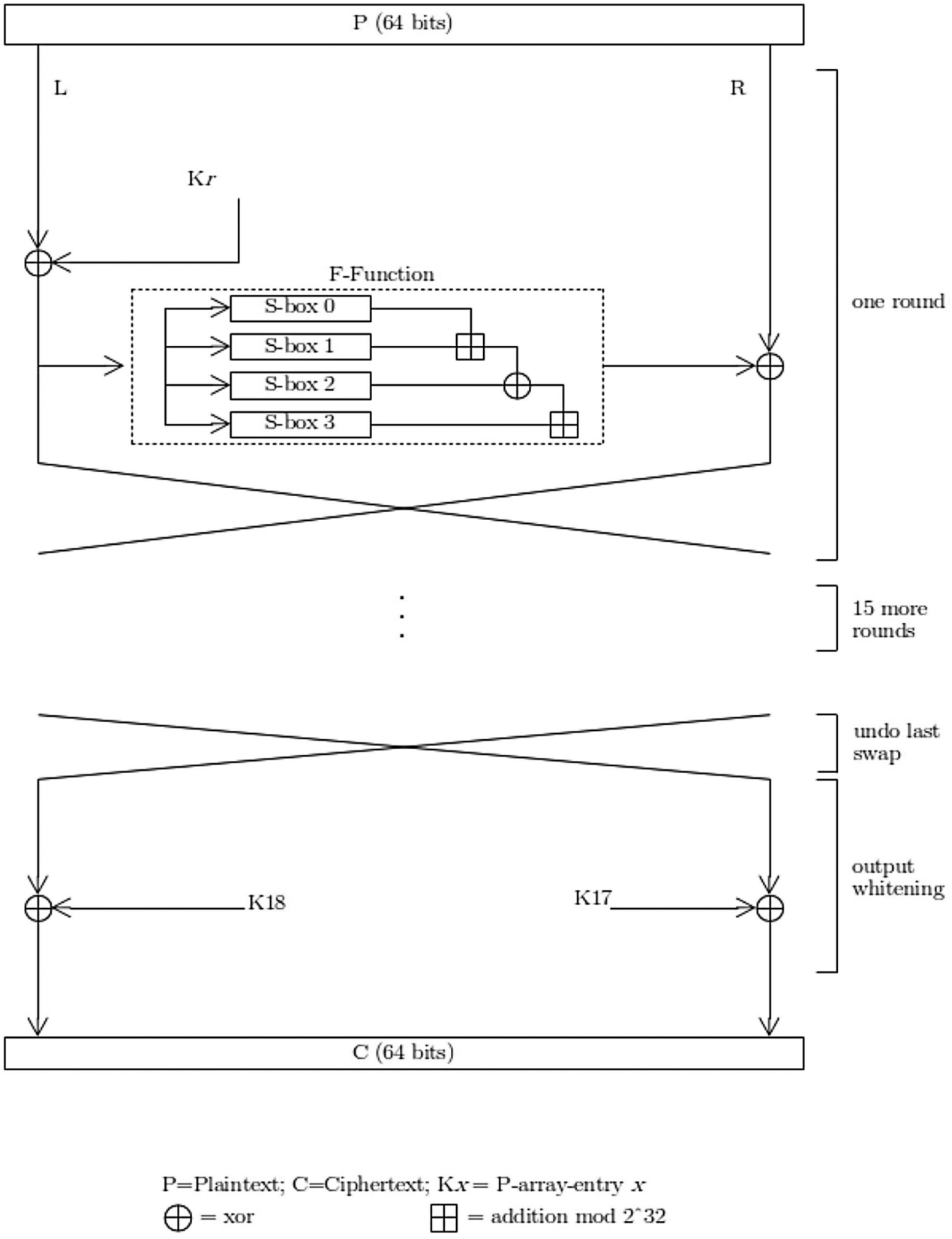

Figure 1. Blowfish diagram (Gid.vn, CC BY-SA 4.0, via Wikimedia Commons).

Besides assessing password acceptability with the approximation of its resilience to password cracking, another approach is to use statistical methods to evaluate password strengths. A fine-grained measurement of password strength was achieved [15] using adaptive metres based on Markov-models that estimate the strength of a password analogous to the probabilities of the n-grams that compose said password. Authors formally prove that this construction is secure even when the local data storage is compromised by an attacker. An interesting, password popularity approach, described in [16], uses a min-count data structure to achieve a

password strength acceptance based on the password frequency in combination with a minimum acceptable false-positive rate threshold.

Another popular attack vector in cybersecurity is passwords leak from application storage themselves, resulting in online services such as *HaveIBeenPwneed* [17] that notify users of potential security breaches. To mitigate the adverse effects of application storage password leaks, passwords are typically stored in hashed values. To understand the resistance of those password hashes against attacks using non-standard computing devices, in particular, FPGAs and GPUs, authors [18] investigated two popular password hashes, bcrypt and scrypt on custom implementation targeting these platforms.

While many of the common passwords cracking tools, including John the Ripper [19], have relied on CPU usage, GPUs started gaining focus in recent years with tools such as Hashcat [20]. However, research such as [21], where authors implemented a high-performance, a low-cost cluster consisting of 120 FPGAs called COPACOBANA (Cost-Optimized Parallel Code Breaker) machine, shows that depending on the actual algorithm, the parallel hardware architecture can outperform conventional computers by several orders of magnitude. In recent years secure substantial improvements were gained using hashing algorithms performance optimization techniques on FPGAs [22–24].

Bcrypt implementation on a cost-effective, low-power Xilinx Zynq-7020 FPGA consisting of 40 parallel bcrypt cores running at 100 MHz has shown to greatly outperform all other currently available implementations, improving password attacks on the same platform by at least 42%, computing 6511 password hashes per second for a cost parameter of 5 [25]. When considering accelerators usable in the domain of crypto analysis, there are a couple of notable examples in the field of collision search. In paper [26] authors present new hash collision search techniques that are inherently based on hardware reconfigurability featured by FPGA devices, while in [27] authors present a specific accelerator for SHA-1 collision search. One of the most recent papers available describes a hardware hash accelerator as a key component of blockchain applications [28].

Regarding other notable examples from the domain of cryptographic function, but in the context of the reconfigurable hardware, and with special attention to peculiarities of designing such hardware, authors in [29] propose a flexible framework for exploring, evaluating, and comparing SHA-2 designs. When it comes to SHA-2 hashing, authors in [30] conducted a comparative survey of different acceleration techniques regarding its acceleration.

Bcrypt implementations on homogeneous and heterogeneous multiprocessing platforms: Parallella board with 16- or 64-core Epiphany accelerator and ZedBoard showed better performance per Watt compared to CPU implementations [31]. These implementations were integrated into John the Ripper password cracker resulting in improved energy efficiency by a factor of 35+ compared to heavily optimized implementations on modern CPUs.

Finally, in our previous work presented in [32], we presented a distributed system for parallel and energy-efficient bcrypt password hash computation consisting of up to 32 identical heterogeneous nodes containing a dual-core ARM CPU processing system and FPGA programmable logic. The programmable logic was used to implement a custom bcrypt accelerator which was further extended into the MPI-based distributed cluster of heterogeneous nodes. We showed the capabilities and advantages of clustering these cheap, energy-efficient single-board nodes equipped with programmable logic capable of performing accelerated cryptographic operations. Furthermore, they can run a common Linux-based operating system which eases their programming and general usage.

3. Single node implementations

In this section, we present our previous and improved single-node implementation of the bcrypt-accelerated embedded system consisting of ARM CPUs and programmable logic in FPGA. We first evaluate their performance and compare them with similar heterogeneous, FPGA-based platforms, including [25]. Finally, we compare our new best-performing implementations with other interesting platforms such as high-end GPUs and CPUs.

3.1. Hardware implementation

Details of our previous base implementation including the work-partitioning and communication are provided in our previous work [31,32] with the block-scheme shown in Figure 2. We designed and implemented customized bcrypt accelerator cores in the programmable logic and integrated them (hardware and software) into the popular open-source password cracking program – John the Ripper [19]. Most of the program runs on the processing system (PS or host) with ARM Cortex CPU cores, and for the specific case of bcrypt hash computation PS (host) performs control-oriented tasks, prepares the password candidates, and then offloads the hash computation to the programmable logic part (PL) containing our customized bcrypt cores. Host copies data from DRAM via AXI GPIO interface to the Arbiter module in PL, which scatters the data to bcrypt cores (Bcrypt 0-N) storing them to BRAM (Block RAM) available in PL. BRAM is used as fast-access storage for synchronization flags, password candidates, salt, cost, P-, and S-boxes. The choice of storing to BRAM before instructing the cores

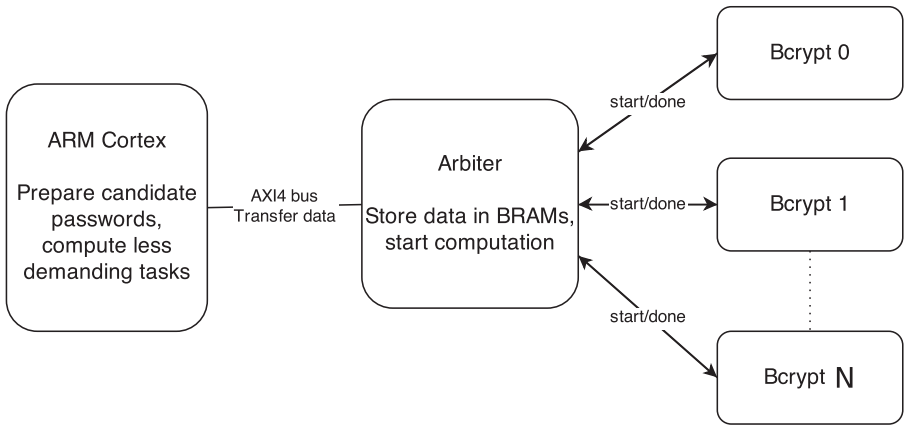

Figure 2. Block diagram of the bcrypt accelerated system.

Table 1. Post-implementation resource utilization per node.

Resource	Utilization	Available	Util. (%)
LUT	34,073	53,200	64.05
LUTRAM	416	17,400	2.39
FF	7919	106,400	7.44
BRAM	140	140	100
#bcrypt cores			28
Max. clock freq.			71 MHz

to start computing enabled bcrypt cores' fast access to data in one clock cycle. This also enables overlapping the communication (PS) and computation phases (PL) in hash computation. Computed hashes and read back from BRAM (in PL) to the host DRAM (in PS) for comparison with the current hash for which the password is guessed. The work distribution together with the password candidate generation implemented in the John the Ripper and executed on the host CPU (PS). This enabled us to exploit advanced password generation schemes included in JtR with intelligent password candidate selection and permutation, such as using Markov chains [19].

Table 1 shows the resource utilization for our ZedBoard implementation. The limiting resource is the amount of BRAM in programmable logic which we fully utilized. We were able to instantiate the maximum of 56 bcrypt instances in programmable logic each computing two bcrypt hashes resulting in 112 computations performed in parallel. At lower-cost settings such as 5, our design was limited by communication overhead, which impacted overall performance. However, the communication overhead was compensated at higher cost settings (above 8) because of the exponential growth in the complexity of the hash computation.

The bottleneck at small cost setting was caused by transferring 4KB large sets of S-boxes filled with initial values from the host (PS) to programmable logic which could not be completely overlapped with the hash computation in bcrypt accelerator cores. We subsequently improved the design by pre-storing initial values of S-boxes in BRAM, i.e. close to the bcrypt cores together with other data required: P-box, synchronization flags, password candidates, salt, and cost. We utilized whole available BRAM to store these data and initial values of S-boxes, thus removing the need to transfer them once the hash computation has started. This optimization removed the initial communication overhead and increased the hash-computation performance of our previous design for a small cost settings, which, albeit not relevant to practical uses, usually are used in benchmark reporting. For higher costs (8 and higher) the overhead was covered by the computation phase and the improved design did not make any difference in performance.

3.2. Single-node results

Figure 3 shows the performance of our previous (Old) and new, improved (New) single-node Zynq-based implementations. We compare them with a similar platform based on Zynq FPGA system with results reported by Wiemer and Zimmermann [25]. We present two metrics:

- Performance as the number of computed hashes per second [H/s], and
- Energy efficiency as the number of computed hashes per second per consumed Watt [H/s/W].

Blue-shaded bars in the Figure 3 show the mere performance metric [H/s], while green-shaded bars show the results for energy-efficiency performance metric [H/s/W]. We compare the results for two cost settings of the bcrypt: Cost 5 and Cost 12. Cost 5 is traditionally used in benchmarks, although not useful for practical password protection. Cost 12 is more suitable for practical usage and is recommended for bcrypt-based systems [11]. Power consumption was measured using wall socket Watt-metre under the full load of the tested platform, i.e. maximum number of bcrypt cores that could fit into the device. Our application, being compute intensive and compute bound, made power consumption constant and at its peak value throughout

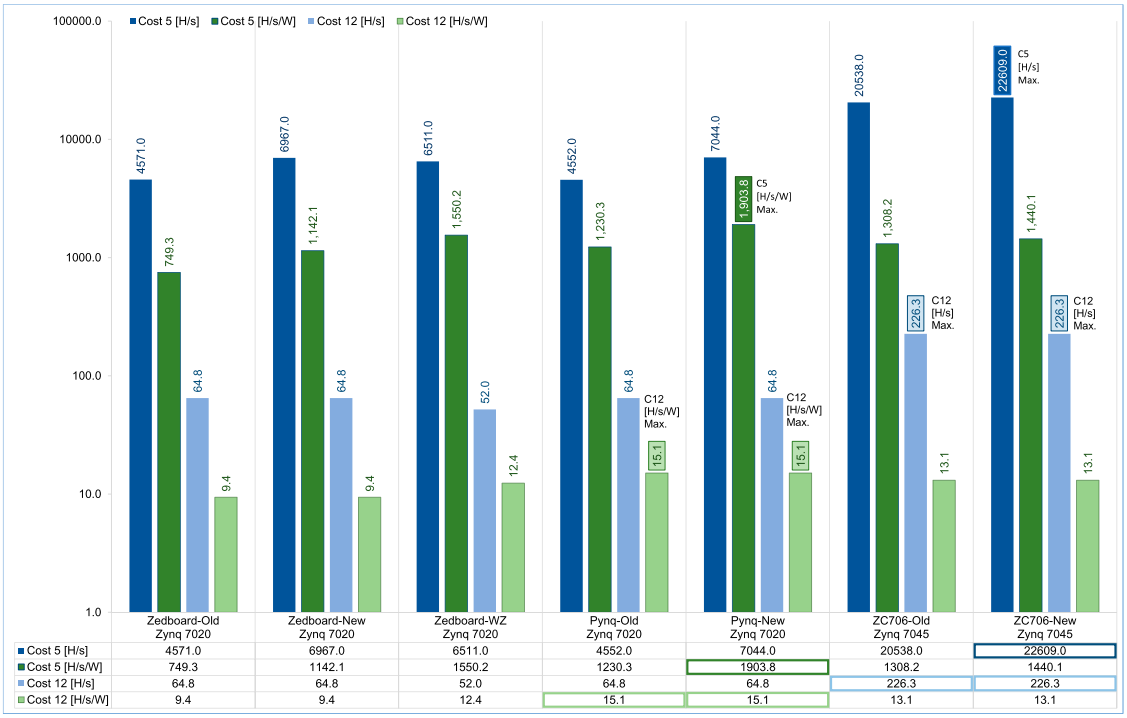

Figure 3. Performance of single-node heterogeneous FPGA-based platforms (Colour online).

the whole time of benchmark, which enabled us to calculate energy-efficiency easily by considering the power as consumed energy over the execution time and the achieved hash rate in c/s. Each benchmark with cost as parameter was run four times with presented results as averaged values excluding the first run. We show the results using the logarithmic scale.

The first cluster-set presents the results of our previous implementation for the ZedBoard platform [33] with Zynq-7020 chip denoted as Zedboard-Old. The second set gives the results of our improved implementation for the same board: Zedboard-New. Next, we show the results reported by Wiemer and Zimmermann [25] for the same board and chip, denoted as Zedboard-WZ. Following are the results of our old and new implementation on the Pynq board with the same Zynq-7020 chip [34] denoted as Pynq-Old and Pynq-New. The last cluster shows the result of our implementations on the ZC706 board [35] featuring Zynq-7045 chip which has the same architecture as Zynq-7020 chip (Zedboard and Pynq) but with 4 times more programmable logic resources and thus 4 times more bcrypt cores compared to Zedboard and Pynq implementations.

For cost 5 performance (dark blue bars), the best performance was achieved with our new implementation on ZC706-New (22,609.0 H/s) with 10% improvement over the old implementation ZC706-Old (20,538.0 H/s) . For Zynq-7020 chip-based boards (Zedboards) improvement of our new implementation, Zedboard-New and Pynq-New exceed 50% of the old implementations Zedboard-Old and Pynq-Old: 6967.0 H/s over 4571.0 H/s, and 7044.0 H/s over 4552.0 H/s, respectively. Our new Zedboard-New implementation with 6967.0 H/s outperforms Wiemer and Zimmermann's implementation Zedboard-WZ with 6511.0 H/s rate [25] for cost 5. For cost 12 performance (light blue bars), our ZC706 implementations, both old and new, performed equally best with the rate of 226.3 H/s. This is 3.5 times the performance of our Zedboard implementations for cost 12 (64.8 H/s) due to 4 times more resources in programmable logic and thus 4 times more bcrypt cores (112 over 28). Also, for cost 12 our new implementations on all platforms did not improve the results of our old variants, as the previously described bottleneck does not incur the performance for higher-cost settings. Both of our Zynq-7020 chip-based implementations (Zedboard and Pynq) outperform Wiemer and Zimmerman's implementation for cost 12: 64.8 H/s compared to 52.0 H/s.

Green-shaded bars in Figure 3 show the [H/s/W] metric (performance with energy efficiency). For cost 5 (dark green bars), the best results were obtained with Pynq-New: our new implementation on the Pynq board with Zynq-7020 chip, achieving 1903.8 H/s/W. For cost 12, i.e. light green bars, both our Pynq board implementations (Pynq-Old and Pynq-New) perform equally and again with the best results of 15.1 H/s/W. They are even more energy-efficient than our best-performing ZC706-New board which achieves efficiency of 13.1 H/s/W for cost 12. This is because Pynq board is less complex, low-cost board without an excessive number of interfaces and additional components that are not required for our bcrypt accelerator application.

Figure 4 compares the results of Zynq-based FPGA boards with high-end GPUs and CPU implementations.

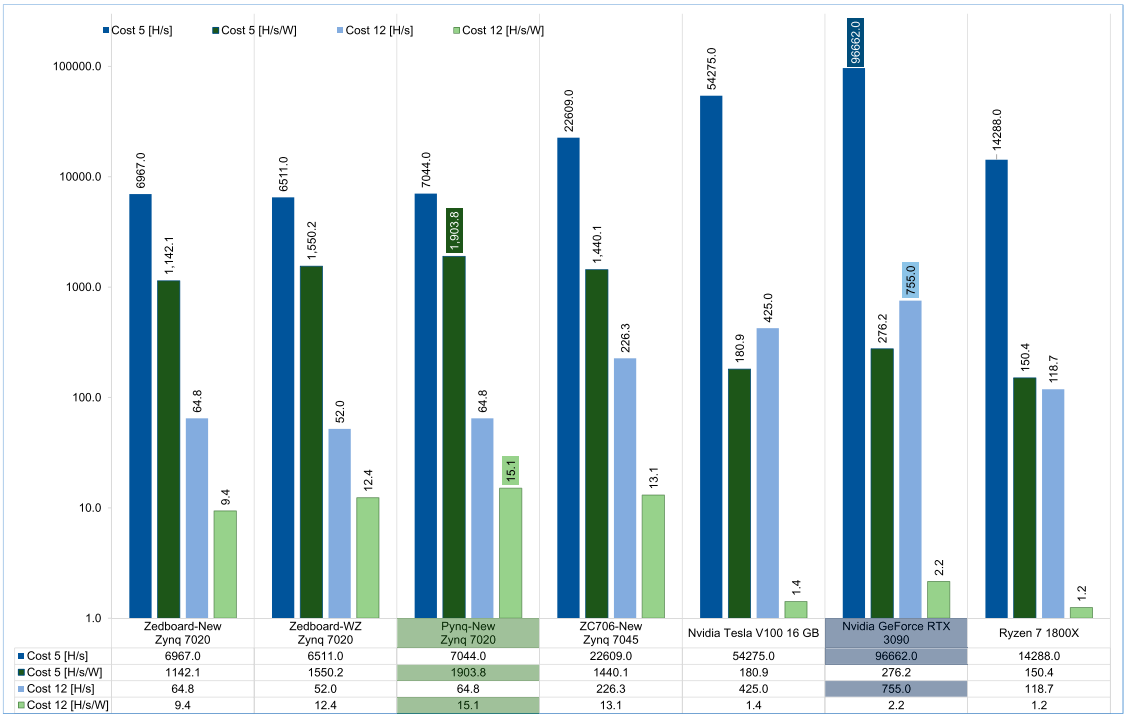

Figure 4. Comparison of the single-node heterogeneous platforms with notable GPUs (Colour online).

We report again performance as [H/s] rate and energy efficiency in [H/s/W].

For the energy-efficiency metric, for platforms that we were not able to experiment with and for which we obtained the results from other sources, we take the specified Thermal design power (TDP) as the estimate for the power consumption. In this case, we just consider the power consumption of the computing device (i.e. GPU) without the additional power consumption of the whole PC that hosts the device, while for our FPGA board implementations, we use the whole system power consumed during the program execution. This way, we compensate the usage of TDP figure in comparison with our implementations, although our previous experiments with similar platforms such as Intel's Xeon Phi and AMD HD 7970 GPU showed constant power consumption close to specified TDP during the benchmark executions [31]. We consider this a fair trade-off and a methodology that actually favours competing platforms over our implementations. Furthermore, we wanted to compare our solution to top-of-the-notch GPU hardware, which we could not physically acquire to measure their power consumption. These assumptions stay valid for the other comparisons between various platforms in this paper.

The first set of cluster repeats the results of our Zedboard-New implementation, followed by other relevant Zynq-based implementations. We then report the Nvidia's Tesla V100 16GB and Nvidia's GeForce RTX 3090 results, both obtained and extrapolated from reported cost 5 benchmarks [20]. The last set of bars show the AMD Ryzen 7 1800X CPU results we benchmarked. For all our own conducted benchmarks, we used the same *1.8.0-jumbo-1* version of John the Ripper available at the Openwall's Git repository [19]. For Zynq-based platforms, we removed the software-based bcrypt version with our accelerated version in programmable logic preserving all other settings and configurations of the program throughout all the benchmarked platforms. For AMD Ryzen 7, we compiled John the Ripper using OpenMP support and run the tests on a Linux-based operating system with 16 GB memory. Each bar is accompanied by the numeric result for the respective metric. We also highlight (shade) the best numeric results for both performance and energy-efficiency.

In terms of mere hash computing performance presented by the number of hashes per second [H/s] (blue bars), Nvidia's newest generation GPU GeForce RTX 3090 performed best achieving almost 96.7 kH/s and 755.0 H/s for cost 5 and 12, respectively. However, if we consider the energy efficiency [H/s/W] (green bars), our Pynq-New implementation achieved the best results of approximately 1.9 kH/s/W and 15.1 H/s/W, for costs 5 and 12, respectively. This is by order of magnitude better than GeForce RTX 3090 efficiency: 276.2 H/s/W for cost 5, and 2.2 H/s/W for cost 12. Also, our ZC706-New implementation with 4 times more programmable logic than Zedboard/Pynq, and thus 4 times more bcrypt-cores, scales in performance while maintaining the efficiency comparable to the best-performing Pynq-New platform.

In the rest of this paper, we show the scalability of our approach by forming a distributed cluster of such low-cost FPGA-based platforms which can obtain performance comparable to high-end GPUs and CPUs

with the advantage of much better energy efficiency, operational and purchasing costs than traditional GPU or CPU-based systems. We demonstrate that our platforms are better suited for real, long-term password attacks of the bcrypt-hashed password systems. This makes the whole system efficient and feasible for attacking contemporary systems with cost settings of 12 and more in terms of both the performance and purchasing/electricity costs.

4. Heterogeneous cluster architecture

In this section, we describe the implementation of our cluster with multiple heterogeneous nodes consisting of programmable logic for bcrypt acceleration. Dictionary-based password cracking is an embarrassingly parallel problem, since the dictionary can simply be easily split among the nodes. The worker nodes then work on different and disjoint parts of the dictionary.

4.1. Architecture

The base building unit of our cluster, called Cool Cracker Cluster cCc, is a computational node. The previous version, described in [32], was homogeneous at a cluster scale in terms of using heterogeneous nodes with the same Zynq-7020 SoC: Zedboards and Pynq boards. These nodes, although different in energy efficiency, having the same SoC chip (Zynq-7020) provided us with the same output performance. In this work, we mix up the nodes with various amounts of programmable logic and various processing capabilities in terms of number of hashes per second [H/s]. We started with the cluster consisting of 8 nodes:

- 2 ZedBoards with Xilinx Zynq-7020 SoC,
- 4 Pynq boards with Xilinx Zynq-7020 SoC,
- 1 ZC706 with Xilinx Zynq-7045 SoC,
- 1 ZTEX 1.15y Quad-Spartan 6 LX150 FPGA board originally used for crypto mining.

Each node, except for ZTEX node, is a standalone heterogeneous processing unit with general-purpose ARM CPU and programmable logic. The nodes run Arch Linux based on kernel version 4.6.0-xilinx. For intra-cluster communication for distributed password cracking, we used the Message Passing Interface (MPI) programming model built into the John the Ripper and embodied by the OpenMPI implementation, version 2.1.1. At the top of the stack, we used John the Ripper (JtR), version 1.8.0-jumbo-1, with an improved version of our bcrypt accelerator implemented in programmable logic, described in Section 3. The jumbo version of JtR was used because it already had an MPI implementation and support for the ZTEX board. The block diagram of main components in the single processing node of our cluster is shown in Figure 5.

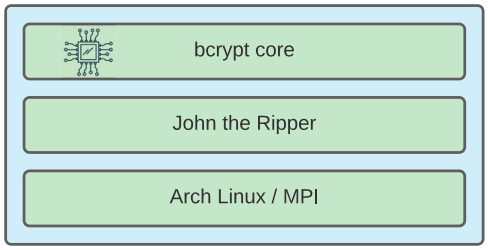

Figure 5. Single node.

The ZTEX 1.15y board does not have a general-purpose CPU but only FPGA: four Spartan-6 XC6SLX-150 chips with substantially more logic than other nodes, and thus capable of hosting more bcrypt computation cores [36]. To introduce it into the cluster, we used an additional Pynq hosting for running JtR (communication, password preparation), while hash computation is performed by ZTEX board connected via the USB 2.0 bus. On the Pynq hosting board, we also used John the Ripper 1.8.0-jumbo-1 which had already the support for bcrypt cracking developed independently of our work [37]. The same version of John the Ripper also included a bitstream with the hardware implementation of the accelerator for the ZTEX board. As a cluster interconnect, we used the standard 1 Gbps Ethernet switch to keep the networking cheap in terms of energy consumption and infrastructure. The block diagram of the whole cluster can be found in Figure 6.

Theoretically, there is no restriction on a device or its type that can be added to this cluster whatsoever with only requirement to support the specified version of John the Ripper (with MPI implementation) and that it can connect to a standard Ethernet network. Nodes can also be added without our accelerator implemented in programmable logic, such as standard desktop computers with or without GPU accelerator support (although this would severely impact the energy efficiency, as we will demonstrate in the rest of the paper).

While all nodes participated in the hash computation (except the Pynq hosted ZTEX board), one node also acted as the master, while the remaining nodes acted as workers. The master node is responsible for:

- job orchestration through MPI, i.e. starting, stopping, and distributing work packages between worker nodes; The work packages are basically the set of password candidates from the dictionary residing on the shared file which will be used for hash calculation. If the node exhausts its password candidates without computing the hash that is being under cracking, the node can repeat the process for the next hash in the list of hashes to be cracked, also residing in the shared file;
- synchronization of in-job events. This is necessary because, although distributing the password candidates from the dictionary, and guessing with the

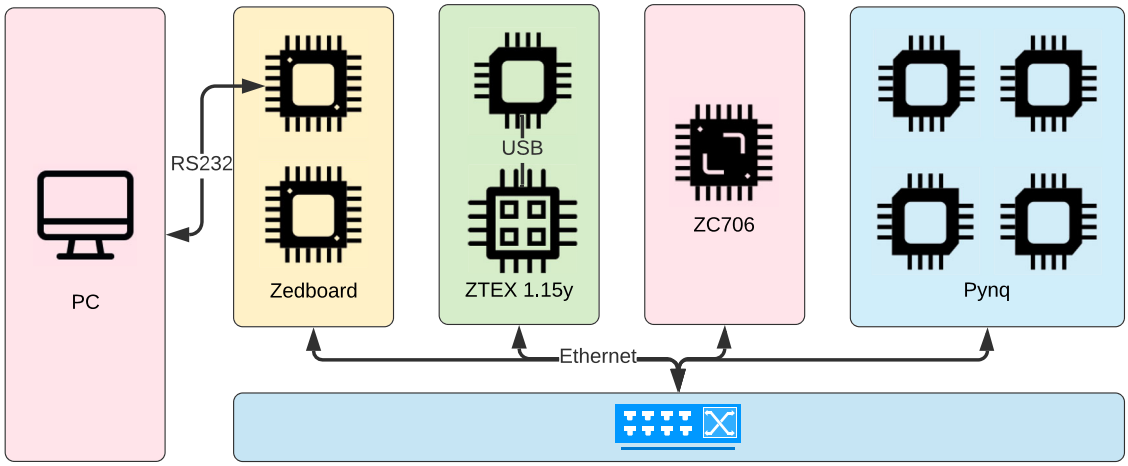

Figure 6. High level block diagram of the CCC.

different passwords, all nodes attack the same set of hashes to be cracked. To avoid unnecessary guessing, when one of the nodes successfully cracks a bcrypt hash (finds out the password that resulted in the hash), the other nodes are notified so that they do not need to repeated password guessing (hash computation) for that particular hash;

- hosting the NFS (network file system) used to exchange wordlists with hashes to be cracked and dictionaries of password candidates. In essence, the nodes share the dictionary (exhaustive list of password candidates), the hash list (the list of hashes to be guessed), and passwords list (list of cracked passwords).

4.2. Work distribution

The default implementation of MPI backed clustering provided by JtR assumes homogeneous clusters, i.e. each node in the cluster was assumed to have the same performance measured in hash computations per second [H/s]. Additionally, the password dictionaries are usually sorted with regard to password statistical probability (more probable passwords are first guessed). These facts made the existing password distribution for our cluster architecture with nodes with varying [H/s] rates inefficient as the algorithm simply fed the nodes every nth word in a word list with different initial offset, where n denotes the total number of nodes in the cluster. This simple work distribution algorithm is given in listing 3.

Algorithm 3 Simple work distribution.

```
1: procedure ProcessWord(ID, n, dictionary)
2:     offset = ID
3:     m = Count(dictionary)
4:     for i in Range(1, m) do
5:         CalculateHash(dictionary[offset + i * n])
6:     end for
7: end procedure
```

In the assumed case of homogeneous nodes, the distribution algorithm ensures that each node gets the fair share of more probable passwords and less probable passwords and thus ensuring the load balancing – each node will finish its work at roughly the same time.

In our case, the heterogeneity of our cluster implies that processing capabilities may vary between different nodes. This leads to sub-optimal performance with the existing password distribution algorithm, as some nodes (the more performance ones) potentially finish earlier and thus remain idle for some time while other nodes still work on their password candidates. We noticed this flaw by running the password cracking job on our cluster with word file containing more than 14 million passwords sorted by their statistical significance (more probable passwords first, followed by less probable passwords).

To mitigate this, we introduce the password candidate distribution scheme dependant on the probability of the candidate (the position in the dictionary) and the performance of the individual nodes in the cluster. Although the algorithm is relatively known, to our knowledge, there is no implementation of it in a similar context of the heterogeneous distributed cluster for password hash computations which considers passwords' probabilities and nodes' performance to optimize the overall cluster execution. The algorithm is initiated by the master node, which obtains performances of each node through their self-assessment. The self-assessment relies on the built-in test function of John the Ripper, which measures the time taken in computing bcrypt cost 5 hashes for a small set of passwords. Using this function, we are able to compute performance values for other costs by extrapolating from performance for cost 5. The performance measure used in algorithms is defined as number of hash computations per second [H/s]. The enumeration of the nodes in the cluster is done by MPI implementation used (OpenMPI), which enumerates the nodes sequentially, starting at 1, with a total of N nodes in the system.

For each node $k = 0 \ldots N$, where N is the number of nodes in the cluster, we define $s(k)$ as the performance of node k in [H/s]. Consequently, the cumulative performance of all of the nodes in the cluster s_{all} is:

$$s_{\text{all}} = \sum_{i=1}^{N} s(i). \tag{2}$$

We then define $s_{\text{previous}}(k)$ as the cumulative performance of all nodes listed before node k:

$$s_{\text{previous}}(k) = \sum_{i=1}^{k-1} s(i). \tag{3}$$

The algorithm for processing the dictionary is given in the Listing 4. The algorithm iterates over the dictionary until it exhausts it (lines 3 to 9). Each iteration except possibly the last one computes the hashes for the chunk of s_{all} passwords from the dictionary. Inside the iteration, each node determines the offset where its chunk of password candidates for this iteration resides within the complete dictionary (line 4 in Algorithm):

$$\mathit{offset}(k) = (\mathit{iter} * s_{\text{all}}) + s_{\text{previous}}(k). \tag{4}$$

After positioning itself in the new computation window starting at *offset* from the beginning of the dictionary, node k computes $s(k)$ consecutive password hashes (line 6). This process repeats until the end of the dictionary is reached (no more password candidates to guess). Therefore, the algorithm ensures that each node in the cluster gets a fair share of the most probable and least probable password candidates, which is critical for load balancing and performance of the whole job.

Algorithm 4 Work distribution.

```
1:  procedure ComputeHashes(dictionary, node k)
2:      iter = 0
3:      repeat
4:          offset(k) = (iter * s_all) + s_previous(k)
5:          for n in Range(0, s(k)) do
6:              CalculateHash(dictionary[offset(k) +
    n])
7:          end for
8:          iter = iter + 1
9:      until end of dictionary
10: end procedure
```

An example distribution of password candidates in a cluster with 3 nodes and performances of 3, 1 and 2 H/s is shown in Figure 7

$$N = 3, k = \{1, 2, 3\}, s(1) = 3, s(2) = 1, s(3) = 2. \tag{5}$$

The cluster aims to simulate real-world cracking scenarios, and this was the reason why the algorithm was implemented by striding the passwords instead of

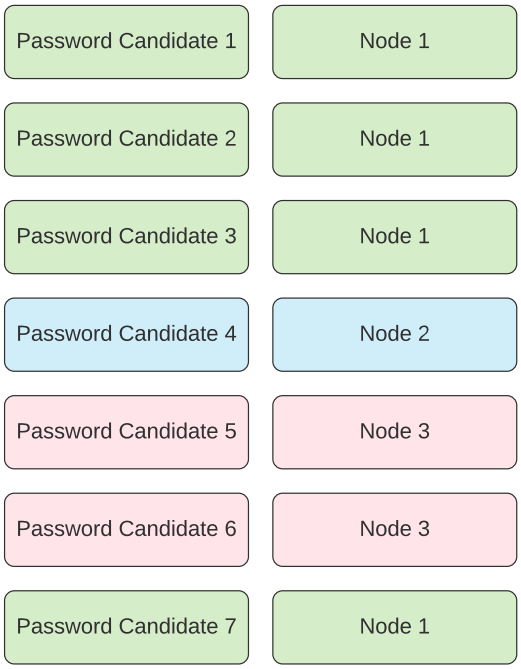

Figure 7. Distribution of password candidates.

dividing word lists into contiguous chunks of different sizes. As previously noted, dictionaries sort the passwords candidates by their probability in descending order (frequency of occurrence gathered with previous password leaks). By using the strided access pattern, the cluster nodes try the most likely passwords first and get their fair share of all passwords (more and less probable), reducing the time required to find passwords specified by input to the cluster, and thus better load balancing the work for the cluster with heterogeneous nodes.

Additionally, we considered using the standard producer-consumer paradigm, where nodes (consumers) would consume password candidates provided by a dictionary-managing node (producer). Worker nodes would send their requests for password candidates and the master node would respond with the candidate, all using MPI messages. While the above solution would scale well, it would not be able to keep up with the high speed of the specialized hardware implemented in programmable logic. Empirically, one MPI message generates a latency of about 11.5 ms, measured on a similar cluster in our lab. Our best performing node processes 93.25 H/s for cost 5, which means that it takes about 0.01 ms for 1 password candidate. Thus, the master node is *de facto* unable to generate enough password candidates to meet the requirements, since the latency caused by the network layer (MPI) is significantly larger than the time required by our best performing node to compute a single bcrypt hash. Moreover, it was easier and cheaper, in terms of engineering hours, to adapt the existing naive algorithm to our proposed algorithm than to change the paradigm

completely, since we did not find any clear added value from the paradigm shift.

5. Results

In this section, we present and discuss the results from the experiments with the cluster. We also compare our cluster in terms of performance, price-efficiency, and energy efficiency with current GPU solutions in terms of bcrypt hash computation reported by various sources. Given the relatively higher initial price of the cluster, we perform a price-efficiency analysis

5.1. Methodology

For experimental purposes, we used our cluster previously described in Section 4. The experiment consisted of utilizing the cluster in a classical password cracking task. We used a single password hashed using the bcrypt algorithm with cost settings of 5, 8, 10, and 12. We attempted a dictionary attack, where the dictionary consisted of passwords leaked in the famous RockYou attack that exposed over 32 million user accounts, many of which had passwords stored in plaintext. The dictionary contains 14 million password candidates, of which we used the top 7 million to make experiments finish in reasonable time while ensuring that the dictionary is large enough to test the performance, energy efficiency, and scalability.

In the dictionary, we intentionally omitted passwords whose hash representation we were trying to crack in the experiment. In this way, we wanted to ensure that each node try out all the candidates intended for that particular node. Otherwise, there would be a possibility that a node would reveal the original string of the hash to be cracked before all nodes reached the end of their packets, which would stop the cracking job for this particular hash to be guessed.

To get reliable results, we conducted experiments multiple times. More concretely, each specific test was executed four times with the results of the first run discarded. The results of the remaining three runs were averaged to obtain the power and performance metric we show in the results. Power consumption was measured using a plug-in watt-metre with all nodes and the network switch connected to the power grid through it, so we measured the consumption of all the components, i.e. the whole system. Since our application is extremely computed intensive, a peak value of the power consumption was constant throughout the whole cracking job, which enabled us to consider it as an average power equal to consumed energy over execution time. The average of the samples was used to determine the power consumption and calculate the energy-efficiency.

The mechanism used for data-sharing was the network file system (NFS) hosted on the master node, which contains a dictionary, a list of to-be-cracked hashes, and a text file containing the list of node addresses used by OpenMPI. Due to the constraints imposed by the OpenMPI, the absolute path to the John the Ripper executable had to be the same on every node in the cluster.

5.2. Experimental results

We report experimental results of individual nodes and the cluster in Table 2. Due to the limited space, we used abbreviations to denote data attributes. Performance (*Perf.*) is noted in bcrypt hash computations per second [H/s]. Energy-efficiency (denoted by *EE*) is given in [H/s/W], and is calculated by dividing performance and measured power consumption for the particular cost. Price-efficiency in [H/s/USD] is calculated by dividing performance and the most recent price of the component or the system as a whole in USD.

The results mostly conform with the theoretical model, as the performance between different cost parameter settings vary exponentially with the base 2. Some discrepancy is observed for cost 5, communication overheads both in-network and on-chip between the CPU and programmable logic being the most probable cause. The total performance of the cluster is near-sum of the performances of the individual nodes, as the cluster scaled well with the addition of the nodes. The workload was distributed in proportion to each node's speed and therefore maximum parallel efficiency was achieved.

The cumulative measured performance of the cluster for cost 12, which is approximately the sum of the performances of the individual nodes, is 1481.4 H/s. For a dictionary with 7 million password candidates, it is easy to calculate that the cracking job takes at most 1 h and 19 min. Considering a realistic cracking scenario with cracking the one bcrypt hash with cost parameter 12, our setup took 1 h and 18 min, which is in line with the theoretical maximum time required to either crack the password or search the complete dictionary. Considering larger costs that are used in modern production systems and are expected to be used in the future in production-grade contemporary systems, i.e. cost 16, and extrapolating the performance to that cost, the cracking job would take about 21 h. This means that our cluster can crack contemporary hashed passwords on a scale of tens of hours which is still within the acceptable cost/time limits for real attacks.

Figure 8 shows the speedups achieved when comparing the default implementation of John the Ripper with the version containing our implementation of the work distribution algorithm, both running on cCc. Each pair of bars in the figure represents the speedup achieved after adding yet another node to the cluster relative to the performance of 1 ZedBoard for cost 12, for both the

Table 2. Performances, energy-efficiency, price-efficiency.

		Cost 5			Cost 8			Cost 10			Cost 12		
	Price	Perf.	EE	PE	Perf.	EE	PE	Perf.	EE	PE	Perf.	EE	PE
ZedBoard	475	6967.0	1142.1	14.6	940.9	139.6	2.0	253.1	36.6	0.5	64.8	9.4	0.1
Pynq	260	7044.0	1903.8	27.1	939.9	223.7	3.6	253.1	59.5	1.0	64.8	15.1	0.2
ZC706	2800	22609.0	1440.1	8.0	3809.0	225.4	1.4	989.2	57.8	0.3	226.3	13.1	0.0
ZTEX	350	93250.0	2583.1	266.4	13338.0	369.5	38.1	3445.0	95.4	9.8	853.7	23.6	2.4
cCc	5400	157484.5	1892.8	29.1	23044.2	255.2	4.2	5916.4	65.0	1.0	1481.4	16.2	0.3

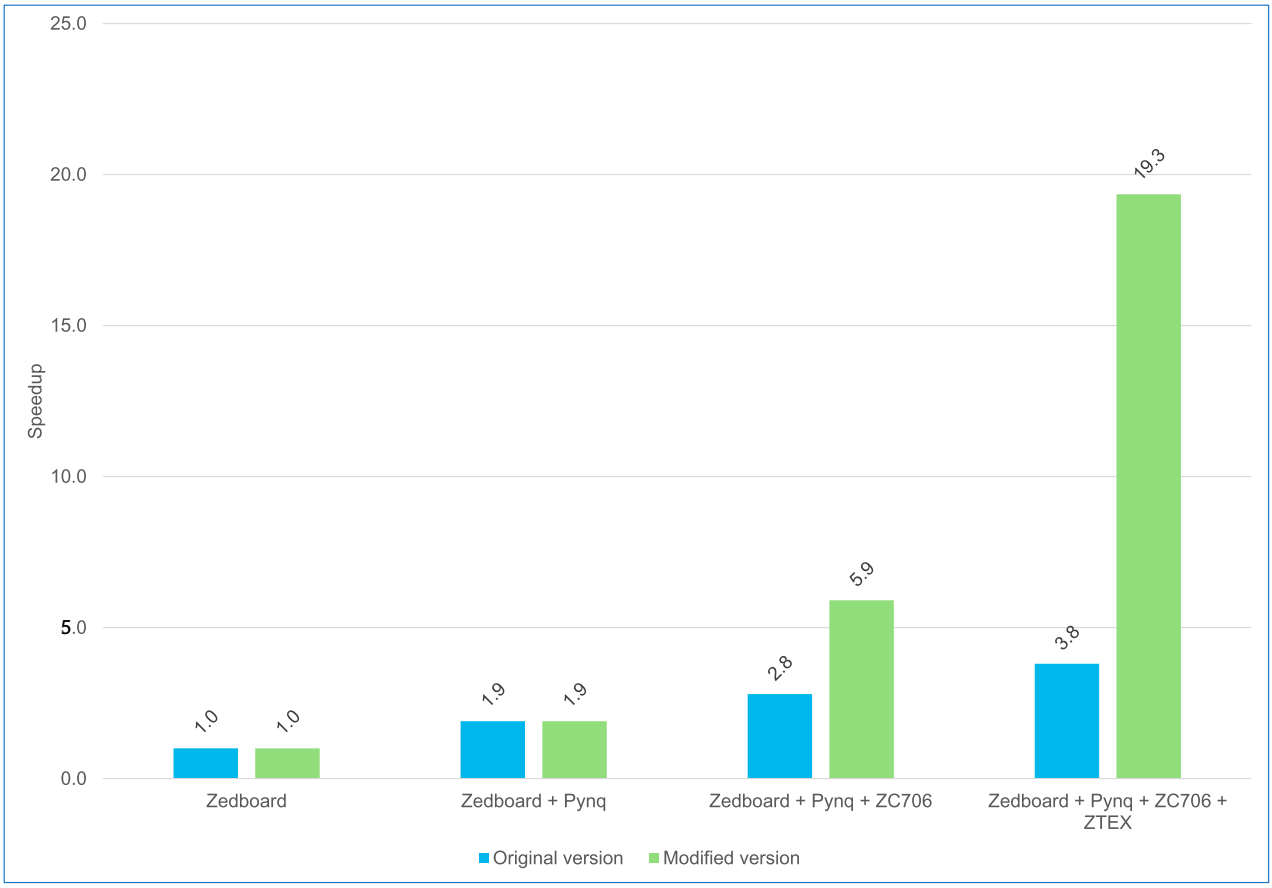

Figure 8. Relative speedup.

naive (default) work distribution and our novel password probability/node capability -dependent work distribution scheme. For the naive distribution, although the added node may be more capable than the previous node or remaining nodes in the cluster, adding it to the cluster always adds to the speedup in increments of at most one, since the work distribution does not take into account the heterogeneity of nodes' performances. In contrast, our work distribution, takes into account the performance of each individual node in the cluster together with the passwords probability-based sorting. This leads to significantly better speedups of in total 19.3 compared to 3.8 for the complete cluster (last pair of bars in Figure 8).

We also compared our cCc cluster against various GPUs. Their performance for cost 5, cost 12, prices, and TDPs used to calculate energy-efficiency and price-efficiency are listed in Table 3. Cost 12 performances are extrapolated from cost 5, and the TDPs in watts are taken from the manufacturers' official websites. For

Table 3. GPU performances, prices and TDPs.

			Performance	
GPU	TDP	Price	Cost 5	Cost 12
GTX 970	148	560	7039 [38]	55
GTX 980	165	630	8465 [38]	66.13
TITAN X	250	1500	12,313 [38]	96.19
GTX 1080 Ti	250	800	21,827 [38]	170.52
Tesla V100	300	6100	54,277 [39]	424.04
RTX 3090	350	3500	96,662 [40]	755.17
Tesla A100	250	12,300	138,375 [41]	1081.05
RX 6800 XT	300	2000	58,156 [42]	454.34

GPU price, which is in USD, we used data from various retailers, taking medial price. As before, performance is given in bcrypt hash computations per second [H/s], energy-efficiency in performance per watt [H/s/W], and price-efficiency in performance per USD or [H/s/USD].

Figure 9 shows the comparison in terms of energy efficiency. Although some of the GPUs seem to be obsolete, we still compared our cluster with them since

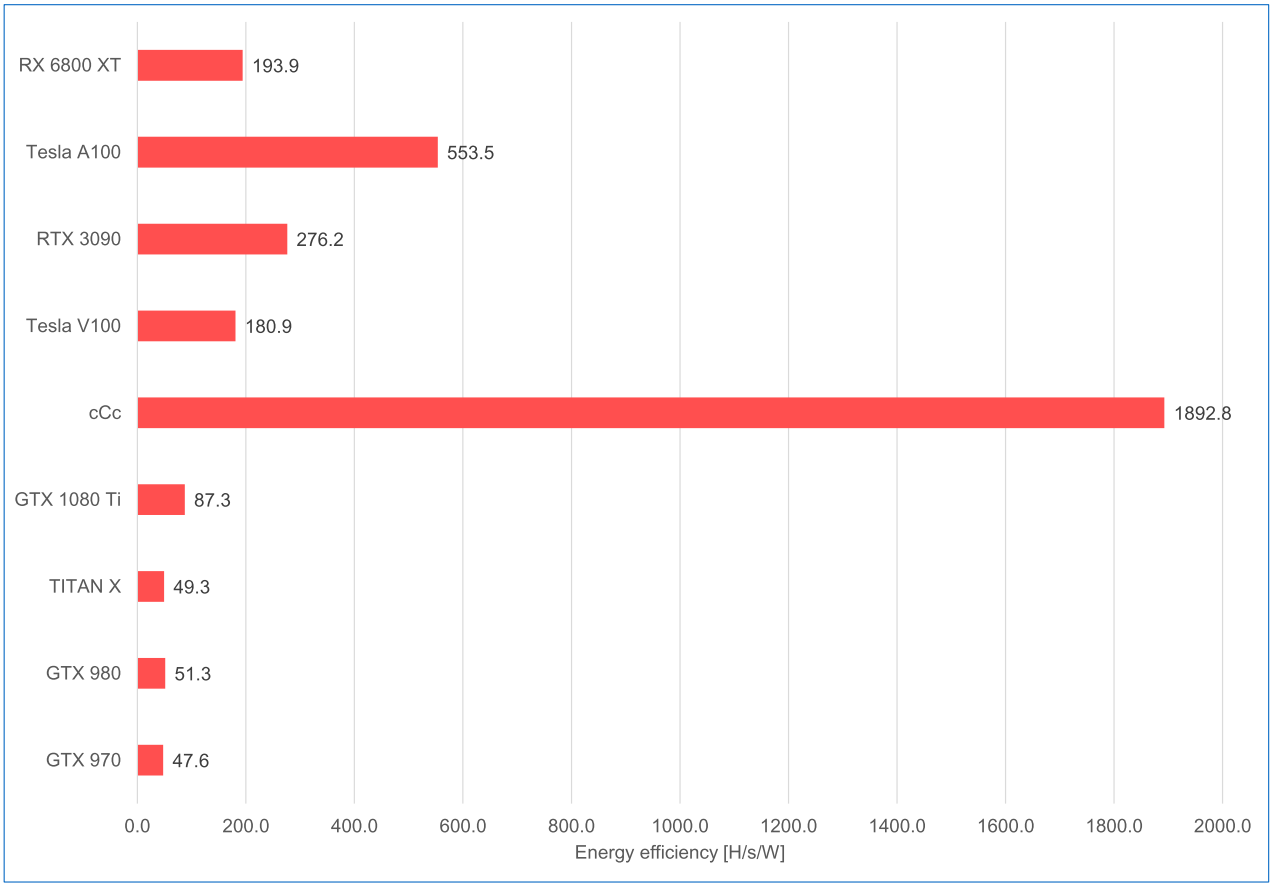

Figure 9. Energy-efficiency: GPUs vs. cCc.

these GPUs are comparable to the newer GPUs in terms of energy-efficiency and price-efficiency. Our cluster outperforms any GPU by at least a factor of 3. This is mainly because our custom implementation in programmable logic has once again proven that custom accelerators are the right choice for energy-efficient heavy computational tasks. It is worth noting that the energy efficiency of our cluster was calculated using actual (measured) power consumption, while the energy-efficiency of the GPUs was calculated using the manufacturers' TDPs. This actually favours the GPUs, as their actual power consumption is larger due to the need for other components to run cracking tasks on these GPUs.

Figure 10 gives the price-efficiency or [H/s/USD] for compared platforms. Our cCc cluster outperforms most GPUs, while being on par with RX 6800 XT, RTX 3090 and GTX 1080 Ti. It is worth noting that it is difficult to get market prices for the selected GPUs these days, including the ones we compare our cluster to, as cryptocurrency mining has caused their prices to skyrocket. The comparison is displayed for cost parameter value of 5, the numbers for GPUs coming from outside sources [38–42]. Some of them reported results for systems with multiple GPUs, but we did the comparison with single units. If we were to compare our cluster to systems with multiple GPUs, their prices would exceed the price of our cluster by order of magnitude, with no significant advantage in terms of energy-efficiency or price-efficiency.

To further increase the energy-efficiency of distributed systems like our cluster, Pynq boards should be preferred over ZedBoards as they consume less energy (up to 46%), and cost 2.5 times less (increased price-performance). Since it uses the same Zynq-7020 SoC and voltage rectifier, the significant energy savings likely result from the fact that it has fewer peripherals and a much simpler design. However, we have combined ZedBoards and Pynqs to demonstrate node-level heterogeneity and how different boards with the same SoCs deliver different performances and energy results.

5.3. Overall operational evaluation

To contextualize the capabilities of our cluster, we performed overall operational evaluation by combining performance, energy and cost which includes both: operational cost (i.e. electricity) and acquisition costs. In the previous Section 5.2, we showed that the cluster outperforms competing GPU solutions in both energy-efficiency [H/s/W] and price-efficiency [H/s/USD]. While the previous statement is true, to see the whole picture, we need to take into account the acquisition costs of the equipment. Although our cluster consists of reasonable priced nodes (at this time, even outdated

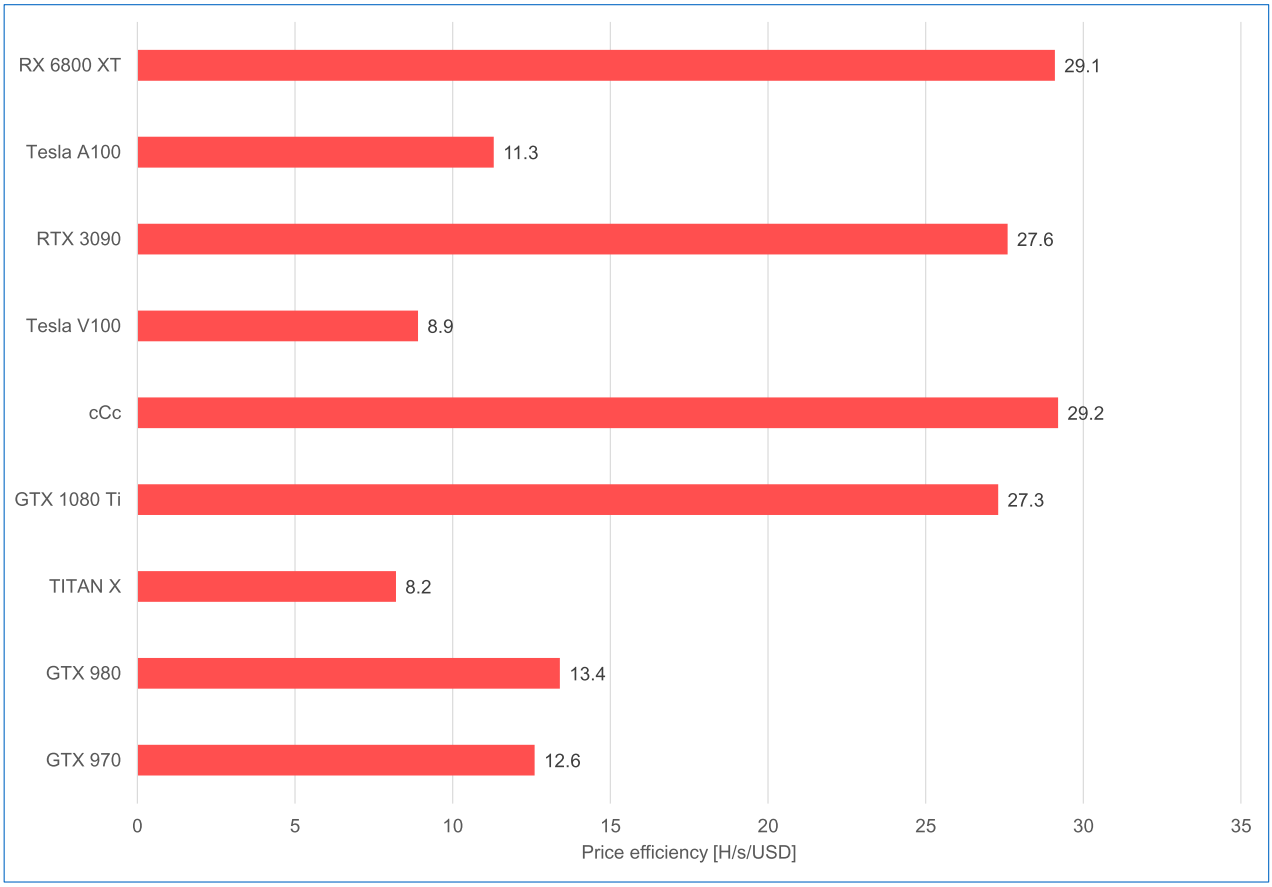

Figure 10. Price-efficiency: GPUs vs. cCc.

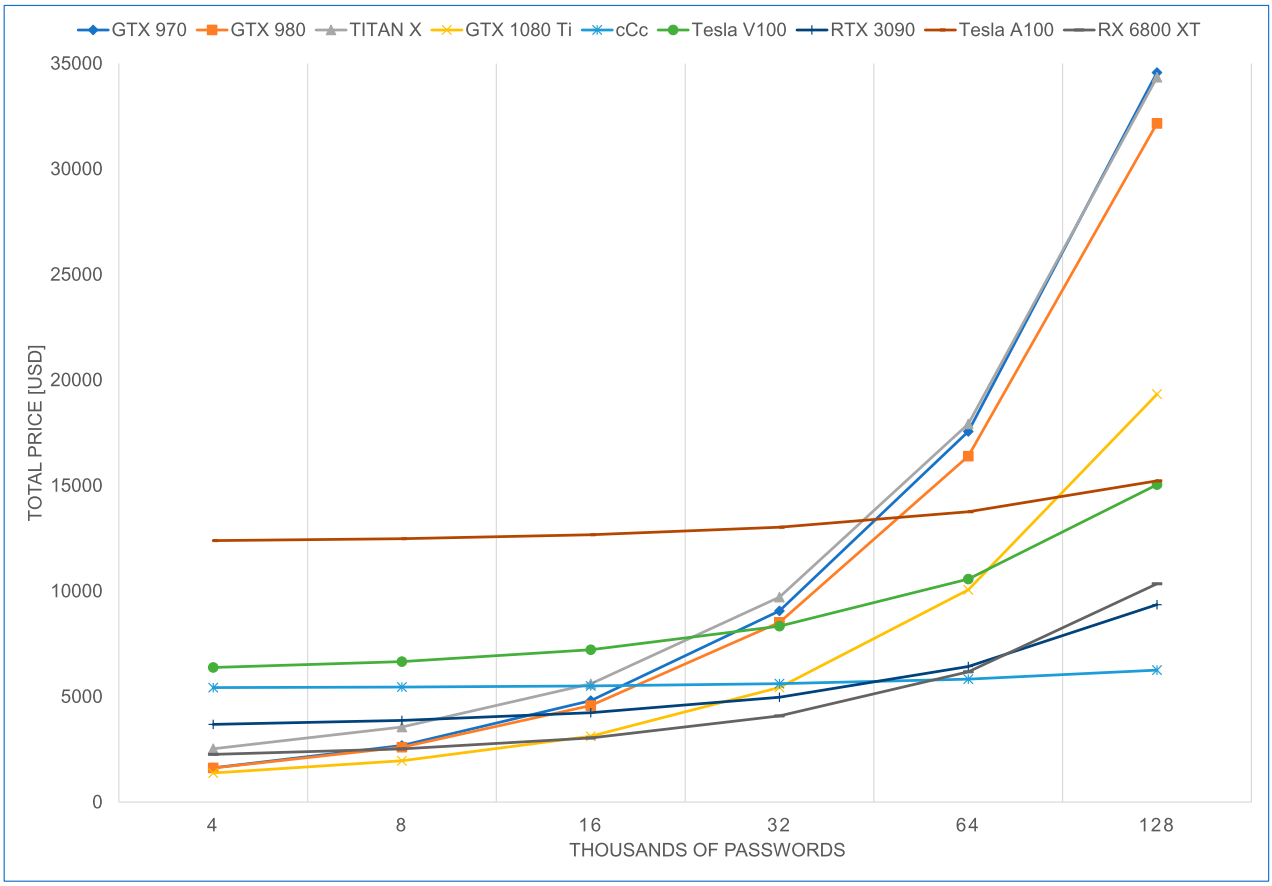

Figure 11. Total price vs. No. of passwords for Cost 5.

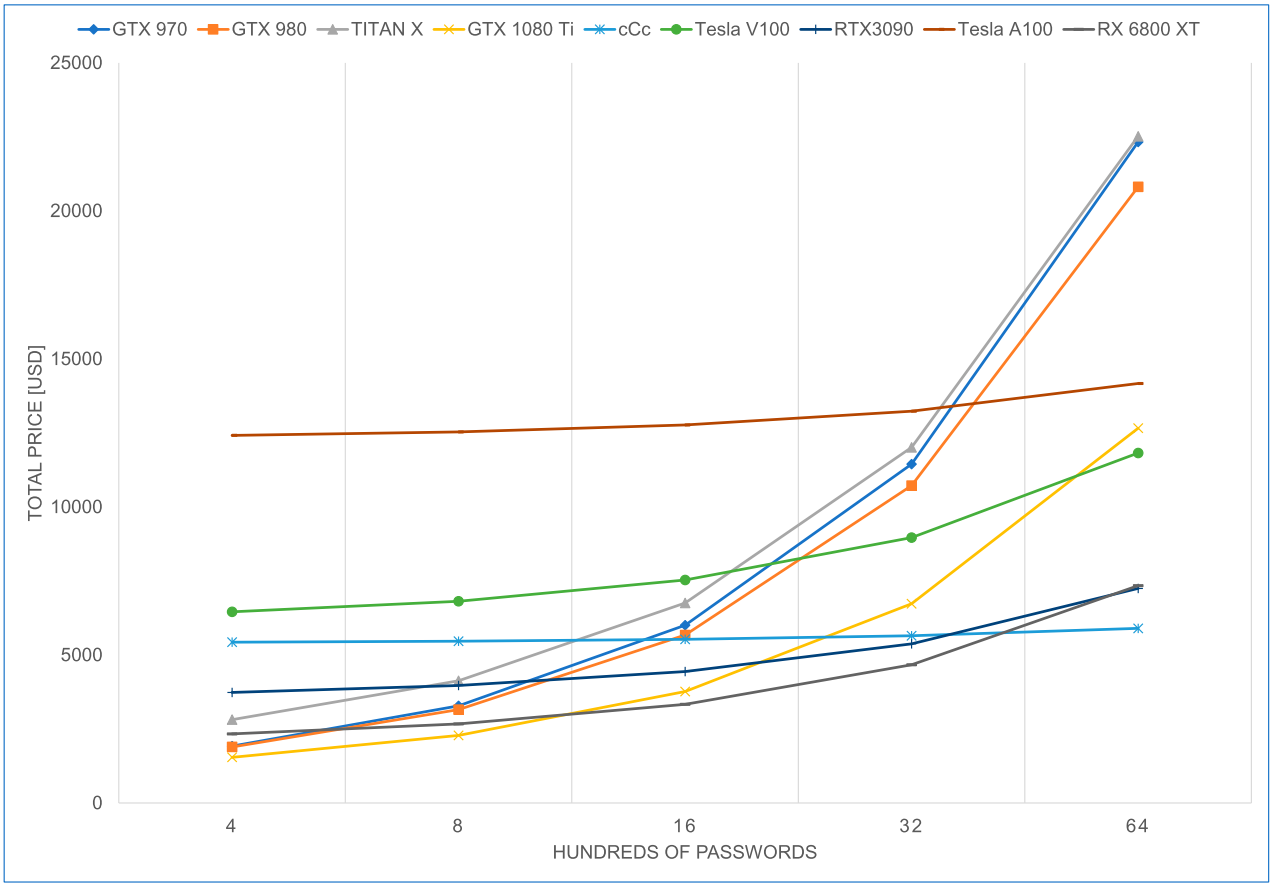

Figure 12. Total price vs. No. of passwords for Cost 12.

and replaced by new boards, so with the decreasing price tendency), it has a significantly higher initial acquisition cost than some of compared GPUs (with increasing price tendency due to high demand at the moment). As we will show in this section, the difference in initial cost is mitigated by the price of electricity used to run the cluster on real-world password cracking jobs, making our cluster more effective in terms of overall costs of operations.

The total operational cost of the aforementioned GPUs and our cluster is composed of the acquisition cost of the equipment and the electricity cost to run them. We calculated this total cost using the data presented in Table 3 in terms of the number of passwords cracked. To translate the number of passwords into time, we assumed that since we used the RockYou dictionary in our experiments, each password will eventually be cracked after trying half of the dictionary, which corresponds to 7 million password candidates. This assumption is not a drawback, as the real number of attempts per hash tends to be even larger, which ultimately can be more beneficial to our cluster as it is more energy-efficient. For example, the average number of characters in a single password is 8 [43], which means that for brute force attacks and no restrictions on the password (any combination of 95 characters, including uppercase, lowercase, special characters, digits) each password potentially requires $\approx 6.6 * 10^{15}$ attempts, or at least millions of attempts in the case of dictionary attacks. For the electricity cost, we used the average electricity price of in European Union, which is 0.2126 EUR/kWh provided by the Eurostat [44].

Figure 11 shows the analysis for cost 5. The X-axis represents the total number of cracked passwords in thousands, while the Y-axis represents the total price in USD. In terms of total operational costs, our cluster pays off relatively quickly, outperforming most GPUs after approximately 16,000 passwords. The final break-even point with the most energy-efficient GPU (RX 6800 XT) is 58,000 cracked passwords.

Figure 12 shows the analysis for cost 12. Our cluster outperforms most of the GPUs in the range of 1000–2000 passwords. The break-even point with the most energy-efficient GPU is 4500 passwords. It is important to note that bcrypt with cost 12 is still used in some current production systems. Eventually, the break-even point would come at even lower levels as the cost parameter increases, which best favours our energy-efficient cluster, since nowadays the safe cost parameter to consider for production systems is around 12, with a tendency to grow in the near future.

Both analyses of the total operational cost show that our cluster can be favoured over GPUs and high-end

CPUs for real-world usage as it pays off even for a relatively small number of passwords, since password leaks usually consist of millions of password hashes.

6. Conclusion

In this paper, we presented cluster-level heterogeneous system cCc (*cool Cracker cluster*) – a distributed system for parallel and energy-efficient bcrypt hash computation. In contrast to the previous version [32], we added a layer of heterogeneity, which includes nodes with different performances expressed in hash computations per second. On a single-node level, we also improved our bcrypt accelerator implemented in programmable logic for low-cost settings traditionally used in benchmarking. We used the spare capacity in BRAM to prestore initial values of S-boxes, thus removing the need to transfer them once the hash computation has started. This optimization resulted in a significant improvement in performance at cost 5 by a factor of 1.5. At a cluster level, we proposed a novel approach in the sense of implementation of an existing work distribution algorithm that takes into account the different performances of each node in the cluster and the probability of the password candidate (the position in the dictionary since passwords are sorted by their relevance), which to our knowledge has not yet been tried in a similar context. By using this algorithm, we managed to improve the overall cluster performance compared to the previous naive algorithm with equal work distribution which assumes no heterogeneity at the cluster-level. Experimental results on real-world cracking scenarios showed that our cluster satisfies the theoretical model and that it scales linearly with the addition of new nodes and with their performance capabilities.**red**JOSIPK: Ova zadnja recenica mi nije jasna, ja bih to izbacio.

In terms of energy-efficiency and price-efficiency, we achieve better results than current GPU solutions. Although our cluster has the third to the highest initial cost, it soon pays off and becomes affordable after 58,000 cracked passwords for cost 5 and after 4500 passwords for cost 12. It is worth noting that bcrypt with cost parameter 12 can be used in production-level systems and that leaks typically contain millions of password hashes, so there is more reason to favour our cCc over GPUs and CPUs.

As for our future research, we intend to improve the resilience and dynamic adaptation of the cluster. If a node fails during the cracking job, it could cause the whole job to restart, which is a significant threat to efficiency since production-level hashes cracking can take tens of hours or even days to complete. This could be mitigated by moving inter-node communication to a higher layer using the Actor model. We also intend to investigate distributed incremental cracking methods with accelerators implemented in programmable logic.

Acknowledgments

The authors would like to thank Mr. Eduard Strojan for conducting the described experiments in a laboratory while doing his Master thesis at the University of Zagreb. We also thank Solar Designer from Openwall for his support and valuable inputs he gave in discussions regarding John the Ripper with addition to FPGA and distributed password cracking, and Ms. Katja Peričin from ReversingLabs for the initial development and her work on bcrypt accelerator. We would also like to thank Mr. Luka Macan from the University of Zagreb for his assistance in laboratory work.

Data availability statement

The measurement data used to support the findings of this study are available from the corresponding author upon request.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Funding

The research was conducted as part of the employment of the authors, namely:
• Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia (Branimir Pervan, Josip Knezović, Emanuel Guberović).
• Green Light Technologies Ltd., Zagreb, Croatia (Emanuel Guberović).
This work has been supported in part by the project IZHRZ0_180625 Heterogeneous Computing Systems with Customizable Accelerators, funded under Croatian-Swiss Research Programme (CSRP) by the Croatian Science Foundation (HRZZ).

ORCID

Branimir Pervan http://orcid.org/0000-0003-3803-0910
Josip Knezović http://orcid.org/0000-0001-6975-4511
Emanuel Guberović http://orcid.org/0000-0001-9285-6858

References

[1] Herley C, Van Oorschot PC, Patrick AS. Passwords: if we're so smart, why are we still using them? In: Dingledine R, Golle P, editors. International vonference on Financial Cryptography and Data Security. Berlin: Springer; 2009. p. 230–237.
[2] Forget A. A world with many authentication schemes [PhD thesis]. Carleton University; 2013.
[3] Shay R, Komanduri S, Gage Kelley P, et al. Encountering stronger password requirements: user attitudes and behaviors. In: Cranor LF, General Chairr. Proceedings of the Sixth Symposium on Usable Privacy and Security. New York (NY): Association for Computing Machinery; 2010. p. 1–20.
[4] Klein DV. Foiling the cracker: a survey of, and improvements to, password security. Program Comput Softw. 1992;17(3):3.
[5] The Password Meter. The password meter; 2021 Mar. Available from: http://www.passwordmeter.com/

[6] Security.org. How secure is my password? 2021 Mar. Available from: https://www.security.org/how-secure-is-my-password/
[7] Oechslin P. Making a faster cryptanalytic time-memory trade-off. In: Boneh D, editor. Annual International Cryptology Conference. Berlin: Springer; 2003. p. 617–630.
[8] Kaliski B. Pkcs #5: password-based cryptography specification version 2.0. RFC 2898. Internet Engineering Task Force; 2000 Sept. Available from: https://www.ietf.org/rfc/rfc2898.txt
[9] Provos N, Mazières D. A future-adaptable password scheme. Proceedings of the FREENIX Track:1999 USENIX Annual Technical Conference; Monterey (CA). USENIX Association; 1999.
[10] Percival Tarsnap C, Josefsson S. The scrypt password-based key derivation function. RFC 7914. Internet Engineering Task Force; 2016 Aug. Available from: https://www.ietf.org/rfc/rfc7914.txt
[11] Open Web Application Security Project. Password storage cheat sheet. 2021 Mar. Available from: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
[12] Ed Moriarty K, Kaliski B, Rusch A. Pkcs #5: password-based cryptography specification version 2.1. RFC 8018, RFC Editor; 2017 Jan. Available from: https://www.rfc-editor.org/rfc/rfc8018.txt
[13] Schneier B. Description of a new variable-length key, 64-bit block cipher (blowfish). In: Anderson R, editor. International Workshop on fast Software Encryption. Berlin: Springer; 1993. p. 191–204.
[14] Dell'Amico M, Michiardi P, Roudier Y. Password strength: an empirical analysis. In: Mandyam G, Westphal C, General chairs. 2010 Proceedings IEEE INFOCOM; San Diego (CA). IEEE; 2010. p. 1–9.
[15] Castelluccia C, Dürmuth M, Perito D. Adaptive password-strength meters from Markov models. In: Hutton T, St.Amour L, Steering Group and Organizing Committee Co-Chairs. NDSS; San Diego (CA); 2012.
[16] Schechter S, Herley C, Mitzenmacher M. Popularity is everything: a new approach to protecting passwords from statistical-guessing attacks. In: Venema W, Program chair. Proceedings of the 5th USENIX Conference on Hot Topics in Security; Washington (DC). USENIX Association; 2010; p. 1–8.
[17] Hunt T. ';–have i been pwned? 2021 Mar. Available from: https://haveibeenpwned.com/
[18] Dürmuth M, Kranz T. On password guessing with GPUs and FPGAs. In: Mjølsnes S, editor. International Conference on Passwords. Cham: Springer; 2014. p. 19–38.
[19] Openwall. John the Ripper password cracker. 2021 Mar. Available from: https://www.openwall.com/john/
[20] hashcat. hashcat - advanced password recovery. 2021 Mar. Available from: https://hashcat.net/hashcat/
[21] Güneysu T, Kasper T, Novotnỳ M, et al. Cryptanalysis with COPACOBANA. IEEE Trans Comput. 2008;57(11):1498–1513.
[22] Al-Odat ZA, Ali M, Abbas A, et al. Secure hash algorithms and the corresponding FPGA optimization techniques. ACM Comput Surv (CSUR). 2020;53(5):1–36.
[23] Algredo-Badillo I, Feregrino-Uribe C, Cumplido R, et al. FPGA-based implementation alternatives for the inner loop of the secure hash algorithm sha-256. Microprocess Microsyst. 2013;37(6–7):750–757.
[24] Rodríguez-Henríquez F, Abbas Saqib N, Díaz Pérez A, et al. Cryptographic algorithms on reconfigurable hardware. Boston (MA): Springer Science & Business Media; 2007.
[25] Wiemer F, Zimmermann R. High-speed implementation of bcrypt password search using special-purpose hardware. In: Wirthlin M, Huebner M, Cumplido R, chairs. 2014 International Conference on Reconfigurable Computing and FPGAs (ReConFig14); Cancun, Mexico. IEEE; 2014. p. 1–6. doi:10.1109/ReConFig.2014.7032529
[26] Cilardo A, Mazzocca N. Exploiting vulnerabilities in cryptographic hash functions based on reconfigurable hardware. IEEE Trans Inf Forensics Secur. 2013;8(5):810–820.
[27] Cilardo A. The potential of reconfigurable hardware for HPC cryptanalysis of SHA-1. In: Al-Hashimi BM, General chair. 2011 Design, Automation & Test in Europe; Grenoble, France. IEEE; 2011. p. 1–6.
[28] Martino R, Cilardo A. Designing a SHA-256 processor for blockchain-based IoT applications. Internet Things. 2020;11:100254.
[29] Martino R, Cilardo A. A flexible framework for exploring, evaluating, and comparing SHA-2 designs. IEEE Access. 2019;7:72443–72456.
[30] Martino R, Cilardo A. SHA-2 acceleration meeting the needs of emerging applications: a comparative survey. IEEE Access. 2020;8:28415–28436.
[31] Malvoni K, Designer S, Knezovic J. Are your passwords safe: energy-efficient bcrypt cracking with low-cost parallel hardware. In: Bratus S, Lindner FX, workshop organizers. 8th USENIX Workshop on Offensive Technologies (WOOT 14); San Diego (CA); 2014.
[32] Pervan B, Knezovic J, Pericin K. Distributed password hash computation on commodity heterogeneous programmable platforms. In: Gantman A, Maurice C, workshop organizers. 13th USENIX Workshop on Offensive Technologies (WOOT 19); Santa Clara (CA); 2019.
[33] AVNET. Zedboard; 2019 Mar. Available from: https://www.zedboard.org
[34] Xilinx. Pynq: Python productivity for zynq; 2019 Mar. Available from: https://www.pynq.io
[35] Xilinx. Xilinx Zynq-7000 SoC ZC706 evaluation kit; 2019 Mar. Available from: https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html
[36] ZTEX. USB-FPGA module 1.15y; 2019 Mar. Available from: https://www.ztex.de/usb-fpga-1/usb-fpga-1.15y.e.html
[37] Solar Designer. bcrypt cracking on ztex 1.15y fpga boards (bcrypt-ztex); 2017 Jun. Available from: https://www.openwall.com/lists/john-users/2017/06/25/1
[38] Gosney JM. Maxwell/Pascal bcrypt benchmark; 2021 Mar. Available from: https://gist.github.com/epixoip/9d9b943fd580ff6bfa80e48a0e77520d
[39] Mathiopoulos I. Hashcat v4.0.0. benchmark on the Tesla V100; 2021 Mar. Available from: https://gist.github.com/iam1980/808f696a14b0c42b26621a01f91a8b18
[40] Croley S. Hashcat v6.1.1 benchmark on the Nvidia RTX 3090; 2021 Mar. Available from: https://gist.github.com/Chick3nman/e4fcee00cb6d82874dace72106d73fef
[41] Croley S. Hashcat v6.1.1 benchmark on the Nvidia Tesla A100 PCIE variant GPU; 2021 Mar. Available from: https://gist.github.com/Chick3nman/d65bcd5c137626c0fcb05078bba9ca89

[42] Gosney JM. RX 6800 XT Hashcat benchmarks; 2021 Mar. Available from: https://gist.github.com/epixoip/99085955a1145ff61ec83512a50421a7
[43] Statista. Average number of characters of leaked user passwords worldwide as of 2017; 2021 Mar. Available from: https://www.statista.com/statistics/744216/worldwide-distribution-of-password-length/
[44] Eurostat. Electricity price statistics; 2021 Feb. Available from: https://ec.europa.eu/eurostat/statistics-explained/index.php/Electricity_price_statistics