High-Speed Implementation of bcrypt Password Search using Special-Purpose Hardware

Friedrich Wiemer, Ralf Zimmermann
Horst Görtz Institute for IT-Security (HGI), Ruhr-University Bochum, Germany
Email: {friedrich.wiemer, ralf.zimmermann}@rub.de

***Abstract*—Using passwords for user authentication is still the most common method for many internet services and attacks on the password databases pose a severe threat. To reduce this risk, servers store password hashes, which were generated using special password-hashing functions, to slow down guessing attacks. The most frequently used functions of this type are PBKDF2, bcrypt and scrypt.**

In this paper, we present a novel, flexible, high-speed implementation of a bcrypt password search system on a low-power Xilinx Zynq 7020 FPGA. The design consists of 40 parallel bcrypt cores running at 100 MHz. Our implementation outperforms all currently available implementations and improves password attacks on the same platform by at least 42%, computing 6,511 passwords per second for a cost parameter of 5.

I. Introduction

In the modern world, we constantly use online services in our daily life. As a consequence, we provide information to the corresponding service providers, e. g., financial services, email providers or social networks. To prevent abuse like identity theft, we encounter access-control mechanisms at every step we make. While it is one of the older mechanisms, password authentication is still one of the most frequently used authentication methods on the internet even with the emerging advanced login-procedures, e. g., single sign-on or two-factor authentications.

To authenticate users for online services, these passwords are stored on corresponding servers. As a consequence, an attack on these databases, followed by a leak of the information, pose a very high threat to the users and may form a single point of failure, if the passwords are stored in plain text. Recent examples of such leaks are the eBay[1] or Adobe[2] password leaks, where several million passwords were stolen. To prevent these attacks or at least raise the barrier of abuse, passwords must be protected on the server. Instead of storing the password as plaint text, a cryptographic hash of the password is kept. In this case, a successful attacker has to recover the passwords from the hash value, which should in theory be infeasible due to the properties of the hash function. To prevent time-memory trade-off techniques like rainbow tables, the password is combined with a randomly chosen salt and the tuple

$$(s, h) = (salt, \text{hash}(salt, password))$$

is stored. Improvements to exhaustive password searches with the aim to determine weak passwords exists. As passwords are often generated from a specific character set, e. g., using digits, upper- and lower-case characters, and may be length-restricted, e. g., allowing six to eight characters, the search space can be reduced considerably. This enables password recovery by brute-force or dictionary attacks. Recently, more advanced methods, e. g., probabilistic context-free grammars [1] or Markov models [2], [3] were analyzed to improve the password guesses and the success rate and thus reduce the number of necessary guesses.

Apart from the generation of suitable password candidates, the implementation has a high impact on the success. On general-purpose CPUs, generic tools like *John the Ripper* (JtR)[3] or target-specific tools like *TrueCrack*[4], addressing a specific algorithm, in this case TrueCrypt volumes, use algorithmic optimizations to gain a speedup when testing multiple passwords. To further improve efficiency, not only the CPU may be used: modern GPUs feature a large amount of parallel processing cores at high clock-frequencies in combination with large memory. As a prominent example, *HashCat*[5] utilizes this platform for high-performance hash computations.

The major problem remains that hash functions are very fast to evaluate and thus enable fast attacks. *Password-hashing functions* address this issue. These functions map a password to key material for further usage and explicitly slow down the computation time by making heavy use of the available resources: the computation should be fast enough to validate an honest user, but render password guessing infeasible. One key idea to prevent future improvements in architectures from breaking the efficiency of these function are flexible cost parameters. These adjust the function in terms of time and/or memory complexity.

The current standardized password-based key-derivation function is PBKDF2 which is part of the Public-Key Cryptography Standards (PKCS) [4]. Non-standardized alternatives are bcrypt [5] and scrypt [6]. While the three functions are considered secure, each has its own advantages and disadvantages. This lead to the currently running password hashing competition (PHC)[6], which aims at providing well-analyzed alternatives. Another purpose is discussing new ideas and different security models with respect to the impact of special-purpose hardware like modern GPUs, Application-Specific Integrated Circuits (ASICs) or Field-Programmable Gate Arrays (FPGAs).

Usually, the overall cost of large-scale attacks on cryp-

[1] cf. http://www.ebayinc.com/in_the_news/story/ebay-inc-ask-ebay-users-change-passwords
[2] cf. https://adobe.cynic.al/
[3] cf. http://www.openwall.com/john
[4] cf. http://code.google.com/p/truecrack
[5] cf. http://hashcat.net/oclhashcat
[6] cf. https://password-hashing.net/

978-1-4799-5944-0/14/$31.00

tographic functions – and thus the feasibility of the attack – is dominated by the power costs. For this reason, specialized hardware achieves excellent results due to its low power consumption, especially when compared to general-purpose architectures. This makes special-purpose hardware very attractive for cryptanalysis in general [7], [8], [9], [10], as well as in the context of password-hashing functions [11].

For the remainder of this paper, we focus on bcrypt as the target function in the scope of efficient password-guessing attacks and start with an overview on the currently available implementations. In Section II-A, we describe the bcrypt algorithm and the influence its tunable cost parameter.

With the goal of benchmarking energy-efficient password cracking, [12] provided several implementations of bcrypt on low-power devices, including an FPGA implementation in December 2013. The authors used the zedboard (cf. II-B), which combines an ARM processor and an FPGA, and split the workload on both platforms. The FPGA computes the time-consuming cost-loop of the algorithm while the ARM manages the setup and post-processing. They reported up to 780 passwords per second (pps) for a cost parameter of 5 and identified the highly unbalanced resource usage as a drawback of the design. In August 2014, [13] presented a new design, improving the performance to 4571 pps for the same device and parameter, using the ARM only for JtR to generate candidates and to transfer initialization values to the FPGA. When they further optimized performance, the zedboard became unstable (heat and voltage problems). Due to these issues they also report a higher theoretical performance of 8122 pps (derived from cost 12) and 7044 pps (simulated using the larger Zynq 7045 FPGA).

Contribution: In this paper, we provide a practical and efficient implementation of bcrypt on a low-power FPGA-platform. Compared to the previous implementations on the same device, we achieve a performance gain of 8.35 and 1.42, respectively. In addition, we implemented a simple on-chip password generation to utilize free area in the fabric, which splits a pre-defined password space and generates all possible brute-force candidates. This creates a self-contained, fully functional system (which may still use other sources for password candidate checking), which we compare to other currently available attack-platforms.

Outline: The rest of the paper is structured as follows: We first introduce the necessary background information in Section II, before we describe our implementation details in the subsequent section and discuss our results, followed by an evaluation of the costs of different attack scenarios, in Section IV. Finally, our conclusion and perspectives for future work form the last section.

II. Background

In this section, we introduce the bcrypt algorithm and outline the computationally expensive steps to motivate the design decisions we made. The second part gives a short overview of our two target FPGA families and their features.

A. The bcrypt password hash

Provos and Mazières published the bcrypt hash function [5] in 1999, which, at its core, is a cost-parameterized, modified version of the Blowfish encryption algorithm. The key concepts are a tunable cost parameter and the pseudo-random access of a 4 KByte memory. bcrypt is used as the default password hash in OpenBSD since version 2.1 [5]. Additionally, it is the default password hash in current versions of Ruby on Rails and PHP.

bcrypt uses the parameters *cost*, *salt*, and *key* as input. The number of executed loop iterations is exponential in the *cost* parameter, cf. Algorithm II.1 (`EksBlowfishSetup`). The computation is divided into two phases: First, Algorithm II.1 initializes the internal state, which has the highest impact on the total runtime. Afterwards, Algorithm II.2 (`bcrypt`) encrypts a fixed value repeatedly using this state.

In its structure, bcrypt makes heavy use of the Blowfish encryption function. This is a standard 16-round Feistel network, which uses SBoxes and subkeys determinded by the current $state$. Its blocksize is 64-bit and during every round, an f-function is evaluated: it uses the 32-bit input as four 8-bit addresses for the SBoxes and computes $(S_0(a) + S_1(b)) \oplus S_2(c) + S_3(d)$ `EksBlowfishSetup` is a modified version of the Blowfish key schedule. It computes a state, which consists of 18 32-bit subkeys and four SBoxes – each 256 $\times$ 32 bits in size – which are later used in the encryption process. The state is initially filled with the digits of π before an `ExpandKey` step is performed: After adding the input key to the subkeys, this step successively uses the current state to encrypt blocks of its *salt* parameter and updates it with the resulting ciphertext. In this process, `ExpandKey` computes 521 Blowfish encryptions. If the salt is fixed to zero, the function resembles the standard Blowfish key schedule. An important detail is that the input key is only used during the very first part of the `ExpandKey` steps. `bcrypt` finally uses `EncryptECB`, which is effectively a Blowfish encryption.

B. Special-Purpose Hardware

While general-purpose hardware, i. e., CPUs, offers a wide variety of instructions for all kinds of programs and algorithms, usually only a smaller subset is important for a specific task. More importantly, the generic structure and design of the architecture might impose restrictions and become cumbersome, i. e., when registers are too small or memory access latency becomes a bottleneck. Reconfigurable hardware like FPGAs and special-purpose hardware like ASICs are far more specialized – they are dedicated to a single task.

An FPGA consists of a large area of programmable logic resources (the fabric), e. g., lookup tables (LUTs), shift registers, multiplexers and storage elements, and a fixed amount of dedicated hardware modules, e. g., memory cores (BRAM), digital signal processing units or even processor hardcores, and can be specialized for a given task. In this work, we target two different Xilinx FPGA families. The main platform is zedboard, more precisely its Zynq-7000 XC7Z020 FPGA. It is located in the low-power low-cost segment. The second target is the Virtex-7 XC7VX485T FPGA which is a high-performance device.

The Zynq-7000 consists mainly of a dual-core ARM Cortex A9 CPU, while the fabric area and resources are comparable to an Xilinx Artix-7 FPGA. The zedboard allows easy access to the logic inside the fabric and memory modules via direct

Algorithm II.1: EksBlowfishSetup

Input: cost, salt, key
Output: state

$state \leftarrow$ InitState();
$state \leftarrow$ ExpandKey($state, salt, key$);
Repeat (2^{cost}) **begin**
  $state \leftarrow$ ExpandKey($state, 0, salt$);
  $state \leftarrow$ ExpandKey($state, 0, key$);
end
return $state$;

Algorithm II.2: bcrypt

Input: cost, salt, key
Output: hash

$state \leftarrow$ EksBlowfishSetup($cost, salt, key$);
$ctext \leftarrow$ "OrpheanBeholderScryDoubt";
Repeat (64) **begin**
  $ctext \leftarrow$ EncryptECB($state, ctext$);
end
return *Concatenate*($cost, salt, key$);

memory access and provides several interfaces, e. g., AXI4, AXI4-Stream, AXI4-Lite or Xillybus. It is a good choice for hardware/software co-design and in the context of this work provides a self-contained system including complex means for password generation and fast hardware designs. The Virtex-7 on the other hand, offers a five times larger fabric area and seven times more memory cores at the cost of more power consumption and a higher device price.

III. Implementing bcrypt on FPGAs

In this section, we describe our FPGA implementation of a multi-core bcrypt cracker, capable of both on-chip password generation and offline dictionary attacks. We start with the general design decisions, the results of an early version of our design and discuss the choices we made to improve the overall design.

An efficient implementation should result in a balanced usage of the available dedicated hardware and fabric resources and maximize the number of parallel instances on the device. In the case of bcrypt, using one dual-port BRAM resource to store two SBoxes saves LUT resources, results in high clock frequencies and relaxes the routing without creating wait-states. To increase the utilization of the memory, we focused on shared memory access without adding clock cycles to the main computation. In the final design, one bcrypt core occupies three BRAM blocks with two additional global memory resources for the initialization values. This leads to an upper bound of 46 cores per zedboard (ignoring any extra BRAM usage of the interface).

Considering a brute-force attack to benchmark the capabilities of the FPGA, the interface can be minimalistic. We use a bussystem with minimal bandwidth capacity, resulting in a small on-chip area footprint. For this scenario, we chose the following setup: During start-up, the host transfers a 128-bit target salt and a 192-bit target hash to the FPGA. These values are kept in two registers to allow access during whole computation time. After filling the registers, all bcrypt cores start to work in parallel. The password candidates are generated on-chip. When the attack completes, a successful candidate is transferred back to the host.

Our earlier design was built out of fully independent bcrypt cores. Each core contained its own password register as well as the memory for the initialization values. This effectively removed all cross-dependencies and resulted in very short routing delays and thus very high clock frequencies. Due to Blowfish's simple Feistel structure, only a small amount of combinatoric logic was needed: Since the main work is done

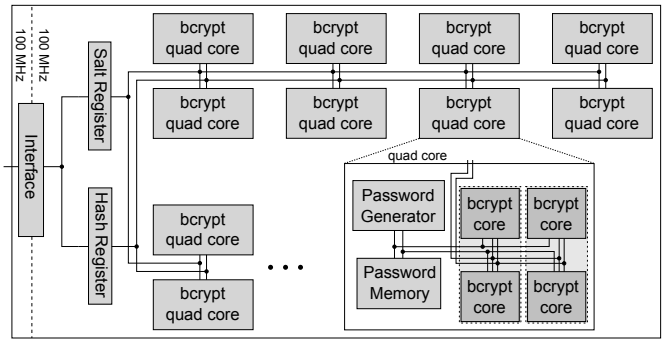

Figure 1. Schematic Top-Level view of FPGA implementation. The design uses multiple clock-domains: a (slow) interface clock and a fast bcrypt clock. Each quad-core accesses the salt- and hash registers and consists of a dedicated password memory, four bcrypt cores and a password generator.

via BRAM lookups. Nevertheless, storing the password in fabric consumes far too much area and resulted in an unbalanced implementation. However, the timing results indicated that more than 100 MHz should be possible.

In order to reduce the area footprint, we tried to share resources and analyzed the algorithm for registers that are not constantly accessed by all cores. We first removed the initialization memory and used the free register resources to implement a pipeline and buffer the signals such that the critical path was unaffected by the change. Due to the required memory access and the dual port properties, we also combined four bcrypt cores with one password generator and password memory. These *quad-cores* can schedule password accesses with negligible overhead.

These changes reduced the area consumption by roughly 20% at the cost of one additional BRAM resource per quad-core. Figure 1 shows the resulting design using multiple parallel and independent quad-cores. Every bcrypt core starts its operation with the initialization of the 256 SBox entries. Within this timeslot, the password generator produces four new passwords and writes them into the password memory. By using the dual-port structure of the memory, two bcrypt cores access their passwords in parallel. While these first two cores uses the BRAM, the second pair of cores is stalled. This leads to a delay of 19 clock cycles between both pairs.

The bcrypt core spends most of the time within Blowfish encryptions, as these are used during the `ExpandKey` (521 times) and `EncryptECB` (3 times) steps. Thus, optimizing the Blowfish core heavily improves the overall performance. A naïve implementation needs two clock cycles per Blowfish round: one to calculate the input of the f-function – and thus

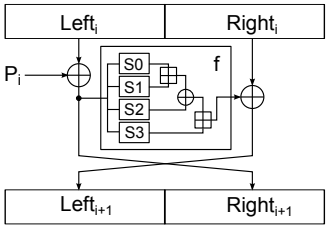

Figure 2. The normal Feistel-structure of one standard Blowfish round. Note that the final XOR operation may be moved along the datapath. By delaying it to the next round, we can resolve data dependencies and compute one Blowfish round in one clock cycle more efficiently.

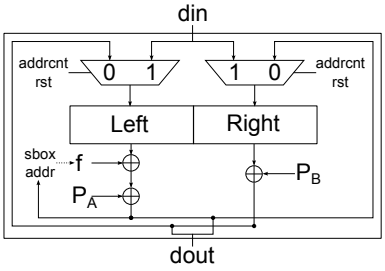

Figure 3. Blockdiagram of Blowfish core. The computation of the delayed f-function is integrated into the left half and the result of the modified data-path forms the memory address for the next f-function.

the addresses to the SBox entries – and one to compute the XOR operation on the f-function output and the subkey.

Figure 2 shows the standard Blowfish Feistel round. We moved the XORs along the datapath, changing the round boundaries. This delay allows us to prefetch the subkeys from the memory and resolve data-access dependencies to reduce the cycle count to one per round.

The resulting Blowfish core is depicted in Figure 3. All of the three XOR operations – the f-function's output and the subkeys P_A and P_B – are computed in every round, removing all multiplexers from the design. As this would change the Blowfish algorithm, we use the reset of the BRAM output registers to suppress any invalid XOR operations during the computation. This design leads to a very minimalistic control logic and a very small Blowfish design in terms of area. Concerning the critical path, the maximum delay comes from the path from the SBox through the evaluation of the f-function.

We have roughly a fourth of the available slices left when we reach the limit of available memory blocks. These resources can be utilized for the password generation. In its simplest form, this is very efficient on-chip, as it only requires a small amount of logical resources. For each password byte, one counter and register store the current states. The initialization value differs for every core and determines the search space. The logic always generates two subsequent passwords and enumerates over all possible combinations for a given character set and maximum password length. When the state has been updated correctly, it is mapped into ASCII representation and written into the password memory. The generation process finishes during the 256 initialization clock cycles, leaving enough time to buffer the signals and ensure a low amount of levels-of-logic.

Please note that with this design, even a slow and simple interface capable of sending 320 bits and a start flag can use the system for brute-force attacks. A more complex interface – capable of fast data-transfer or even direct memory access of the BRAM cores – easily enables dictionary attacks, as new passwords are transferred directly into the password memory during the long bcrypt computation. The on-chip password generation may be removed or modified to work in a hybrid mode.

IV. Results

In this section we will present the results of our implementation. We used Xilinx ISE 14.7 and – if needed Xilinx Vivado 2014.1 – during the design flow and verified the design both in simulation and on the zedboard after Place and Route.

Table I provides the post place-and-route results of the full design on the zedboard. We implemented the design using ten parallel bcrypt quad-cores and a Xillybus interface. The design achieves a clock frequency of 100 MHz. The optimizations from Section III reduced the LUT consumption to roughly 600 LUTs, the amount of BRAMs to 3.25 per single core. We therefore can fit ten quad-cores – and thus 40 single cores – on a zedboard, including the on-chip password generation.

The bcrypt cores need constant cycles c for hash generation, in detail:

$$\begin{aligned} c_{\text{Reset}} &= 1 & c_{\text{Init}} &= 256 \\ c_{\text{Delay}} &= 19 & c_{\text{Pipeline}} &= n,\ (n = 2) \\ c_{\text{bf}} &= 18 & c_{\text{updateP}} &= 9 \cdot (c_{\text{bf}}) \\ c_{\text{key xor}} &= 19 & c_{\text{updateSBox}} &= 512 \cdot (c_{\text{bf}}) \end{aligned}$$

$$\begin{aligned} c_{\text{ExpandKey}} &= c_{\text{key xor}} + c_{\text{upP}} + c_{\text{upSBox}} = 9,397 \\ c_{\text{EncryptECB}} &= 3 \cdot 64 \cdot (c_{\text{bf}} - 1) = 3,264 \end{aligned}$$

Following these values, one bcrypt hashing needs

$$\begin{aligned} c_{\text{bcrypt}} &= c_{\text{Reset}} + c_{\text{Pipeline}} + c_{\text{Init}} + c_{\text{Delay}} + \\ &\quad (1 + 2^{\text{cost}+1} \cdot c_{\text{ExpandKey}}) + c_{\text{EncryptECB}} \\ &= 12,939 + 2^{\text{cost}+1} \cdot 9,397 \end{aligned}$$

cycles to finish. This leads to a total of 614,347 cycles per password (cost 5) and 76,993,163 (cost 12), respectively.

In order to compare the design with other architectures, especially with the previous results on the zedboard, we measured the power consumption of the board during a running attack and used (ocl)Hashcat to benchmark a Xeon E3-1240 CPU (4 cores@3.1 GHz) and a GTX 750 Ti (Maxwell architecture) as representatives for the classes of CPUs and GPUs. Furthermore, we synthesized our quad-core architecture on the Virtex 7 XC7VX485T FPGA, which is available on the VC707 development board, and estimated the number of available cores with respect to the area a new interface may occupy. We assume a worst-case upper bound of 20W as the power consumption for the full evaluation board. For the CPU

Table I. RESOURCE UTILIZATION OF DESIGN AND SUBMODULES.

	LUT	FF	Slice	BRAM
Overall	64.8%	13.06%	93.29%	95.71%
quad-core	2,777	720	801	13
single core	617	132	197	3
Blowfish core	354	64	71	0
Password Generator	216	205	81	0

Table II. COMPARISON OF MULTIPLE IMPLEMENTATIONS AND PLATFORMS, CONSIDERING FULL SYSTEM POWER CONSUMPTION.

	cost parameter 5		cost parameter 12			
	$\frac{\text{Hashes}}{\text{Second}}$	$\frac{\text{Hashes}}{\text{Watt Second}}$	$\frac{\text{Hashes}}{\text{Second}}$	$\frac{\text{Hashes}}{\text{Watt Second}}$	Power	Price
zedboard	6,511	1,550	51.95	12.37	4.2 W	$319
Virtex-7	51,437	2,572	410.4	20.52	20 W	$3,495
Xeon E3-1240	6,210	20.7	50	0.17	300 W	$262
GTX 750 Ti	1,920	6.4	15	0.05	300 W	$120
[13] Epiphany 16	1,207	132.64	9.64	1.06	9.1 W	$149
[13] zedboard	4,571	682.24	64.83	9.68	6.7 W	$319

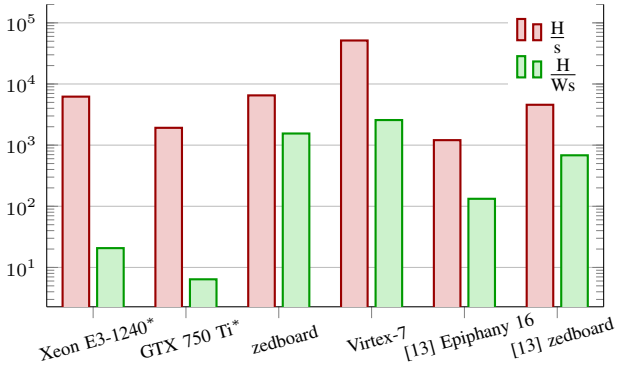

Figure 4. Comparison of different implementations for cost parameter 5. Left bars (red) show the hashes-per-seconds rate, right bars (green) the hashes-per-watt-seconds rate. Results with * were measured with (ocl)Hashcat. The axial scale is logarithmic.

and the GPU attack, we also consider the complete system. While there are smaller power supplies available, we consider a 300W power supply, which is the recommended minimum for the GPU to run stable.

Table II compares the different implementation platforms for cost parameter of 5 and 12. For better comparison, Figure 4 shows the performance and efficiency graphically only for the first case. Our zedboard implementation outperforms the previous implementation from [13] by a factor of 1.42, computing 6511 pps at a measured power consumption of only 4.2W compared to 6.7W of the previous implementation. Thus, this implementation yields also a better power efficiency of 1550 pps per watt, which is more than twice as efficient as the previous implementation. The CPU attack on a Xeon computes 5% less pps, at a significantly higher power consumption. Even considering only the power consumption of the CPU itself of 80W, the efficiency of the zedboard is still about 20 times higher. The estimated Virtex-7 design shows that the high-performance board is a decent alternative to the zedboard: it outperforms all other platforms with 51437 pps and has a very high power-efficiency rating. The drawback is the high price of $3495 for the development board.

To analyze the full costs of an attack, including the necessary power consumption (at the price of 10.08 cents per kWh[7]), we consider two different scenarios. The first uses the fairly low cost parameter of 5 for a simple brute-force attack on passwords of length 8 with 62 different characters and requires the runtime to be at most 1 month. We chose the considerably low cost parameter for comparison with the related work, as it is typically used for bcrypt benchmarks. However, this value is insecure for practical applications, where a common choice seems to be 12, which is also used in the related work. Thus, we use this more reasonable parameter in the second setting. Here, the adversary uses more sophisticated attacks and aims for a reduction of the number of necessary password guesses and for a reduced runtime of one day per cracked password: We consider an adversary with access to meaningful, target-specific, custom dictionaries – for example generated through social engineering – and derivation rules. In [11], the authors trained on a random subset of 90% from the leaked RockYou passwords to attack the remaining 10% and estimated that $4 \cdot 10^9$ guesses are needed for about 67% chance of success, which we use as a basis for the computational power.

Figure 5 shows the costs of running brute-force attacks in the first scenario. To achieve the requested amount of password tests in one month, we need 13564 single CPUs, 43872 GPUs, 10361 CPUs + GPUs, 12999 zedboards or 1645 Virtex-7 boards. The figure shows the total costs considering acquisition costs (fixed cost) and the power consumption. It reveals the infeasibility of CPUs for attacking password hashes, and even more clearly the efficiency of special-purpose devices. Even high-performance FPGAs like the Virtex-7 are more profitable after only a few password cracks, than a combination of CPU and GPU.

Figure 6 shows the costs of attacking multiple passwords in the second scenario. Here, we need 30 CPUs, 102 GPUs, 23 CPUs + GPUs, 38 zedboards or 4 Virtex-7 boards. With the higher cost parameter our current zedboard implementation does not yield similar good results and thus [13] implementation is currently better suited for this attack when mounted on a zedboard. With the higher cost parameter, their implementation can conceal an interface bottleneck coming from the initialization of the bcrypt cores. As our implementation does not suffer from this bottleneck, we can run several cores on a bigger FPGA without negative consequences. Please note that the Virtex-7, after amortizing its acquisition costs, outperforms every other platform (reaching the break-even point with [13] zedboard after attacking about 1500 passwords).

V. CONCLUSION AND FUTURE WORK

In this paper, we presented a highly optimized bcrypt implementation on FPGAs. We use a quad-core structure to achieve an optimal resource utilization and gain a speed-up of 42% and – due to lower power consumption – increased

[7]Taken from the "Independent Statistics & Analysis U.S. Energy Information Administration", average retail price of electricity United States all sectors. http://www.eia.gov/electricity/data/browser

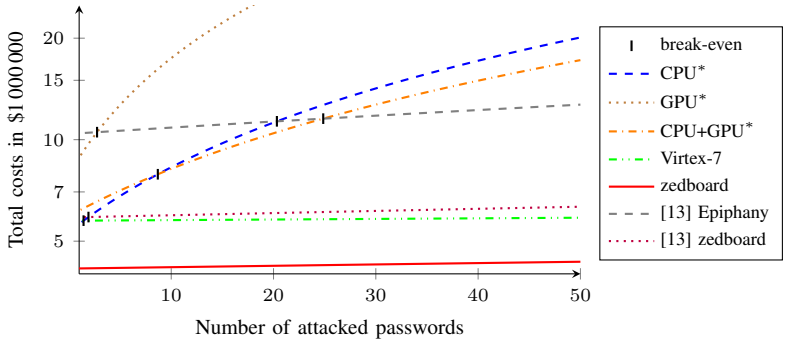

Figure 5. Total costs in *millions USD* for attacking n passwords of length 8 from a set of 62 characters, with logarithmic scale. Each attack finishes within *one month*. Both the acquisition costs for enough devices and the total power costs where considered.

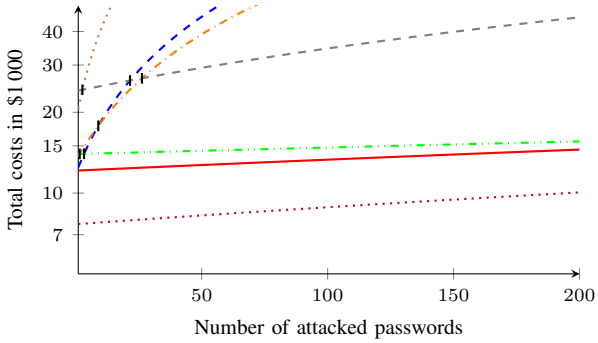

Figure 6. Total costs in *thousands USD* for attacking n passwords of length 8 from a set of 62 characters using a cost parameter of 12 (which is commonly recommended), with logarithmic scale. Each attack finishes within *one day*, with a *dictionary attack* where 65% are covered ($4 \cdot 10^9$ Tests).

power-efficiency by 127% compared to the previous results on the same device. In the design we presented, the critical path is still within the Blowfish core, resulting in a moderate clock-frequency of 100 MHz. An idea to improve this is to pipeline the encryption within a quad-core, interleaving the computations of the core. This may shorten the critical path further, allowing higher clock frequencies and more parallel bcrypt cores due to shared resources.

We showed that it is possible to utilize the remaining fabric area to implement a small on-chip password generation, which is adaptable and may be combined with a dictionary attack, e. g., for prefix and suffix modifications. These possibilities should be evaluated and further analyzed, as the password generation has a high impact on the success rate. Even more importantly, using only off-chip password generation, i. e., by using a CPU to generate passwords and transfer them to the FPGA, introduces two potential bottlenecks: the software implementation itself and the data bus. With the combination of off-chip creation and on-chip modification, it should be possible to reduce the risk of these bottlenecks even in large and highly parallelized clusters: We can use the password generator construction for simple mangling rules and relax the interface or dedicate several cores to brute-force attacks, while others work on a dictionary. This leads to more possible trade-offs in terms of interface speed vs. area consumption.

In our attack scenarios, we considered modern representatives of CPUs as well as GPUs and benchmarked the (ocl)Hashcat bcrypt implementation on these platforms. We compared the total costs of low-power and high-performance devices in two scenarios: simple brute-force with a fixed runtime of 1 month (cost 5) and an advanced attack with a timeframe of 1 day (cost 12). In both cases, the high power consumption of CPUs and GPUs renders large-scale attacks infeasible, as our FPGA implementation not only outperforms these devices but also requires significantly less power.

Interestingly, in combination with improved methods to derive suitable password candidates, the overall costs when using a fast and power-efficient FPGA implementation are not as high as expected for reasonable parameters. As a result, we should evaluate and adjust the parameters used in practice to withstand the advances in technology and intelligent password generation.

References

[1] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek, "Password Cracking Using Probabilistic Context-Free Grammars," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2009, pp. 391–405.

[2] A. Narayanan and V. Shmatikov, "Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff," in *Proc. 12th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2005, pp. 364–372.

[3] C. Castelluccia, A. Chaabane, M. Dürmuth, and D. Perito, "Omen: An improved password cracker leveraging personal information," Available as arXiv:1304.6584, 2013.

[4] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0," RFC 2898, Sept. 2000, http://tools.ietf.org/html/rfc2898.

[5] N. Provos and D. Mazières, "A Future-Adaptable Password Scheme," in *USENIX Annual Technical Conference, FREENIX Track*, 1999, pp. 81–91.

[6] C. Percival, "Stronger Key Derivation via Sequential Memory-Hard Functions," Presentation at BSDCan'09. Available online at http://www.tarsnap.com/scrypt/scrypt.pdf, 2009.

[7] T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp, "Cryptanalysis with COPACOBANA," *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1498–1513, November 2008.

[8] T. Güneysu, C. Paar, G. Pfeiffer, and M. Schimmler, "Enhancing COPACOBANA for advanced applications in cryptography and cryptanalysis," in *Proceedings of the Conference on Field Programmable Logic and Applications (FPL 2008)*, 2008, pp. 675–678.

[9] T. Gendrullis, M. Novotný, and A. Rupp, "A Real-World Attack Breaking A5/1 within Hours," *IACR Cryptology ePrint Archive*, vol. 2008, p. 147, 2008.

[10] R. Zimmermann, T. Güneysu, and C. Paar, "High-Performance Integer Factoring with Reconfigurable Devices," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, Aug 2010, pp. 83–88.

[11] M. Dürmuth, T. Güneysu, M. Kasper, C. Paar, T. Yalçin, and R. Zimmermann, "Evaluation of Standardized Password-Based Key Derivation against Parallel Processing Platforms," in *Computer Security – ESORICS 2012*, 2012, pp. 716–733.

[12] K. Malvoni, "Energy-efficient bcrypt cracking," Bergen, Norway, Dec. 2013, presentation given at PasswordCon Bergen, 2013. Slides online at: http://www.openwall.com/presentations/Passwords13-Energy-Efficient-Cracking/Passwords13-Energy-Efficient-Cracking.pdf.

[13] K. Malvoni, Solar Designer, and J. Knezovic, "Are Your Passwords Safe: Energy-Efficient Bcrypt Cracking with Low-Cost Parallel Hardware," in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, Aug. 2014. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/malvani