RUHR-UNIVERSITÄT BOCHUM

Speed and Area-Optimized Password Search of bcrypt on FPGAs

1. July 2014

Chair for Embedded Security

Ruhr University Bochum

Friedrich Wiemer and Ralf Zimmermann

Outline

Motivation

Why do we care?

- password cracking has an inherent parallel structure
- FPGAs enable to exploit true parallelism

- bcrypt claims to resist hardware optimizations
- implementation presented last year[1] (GSoC) can be optimized

[1] http://www.openwall.com/presentations/Passwords13-Energy-Efficient-Cracking/

What is bcrypt?

Introduced in 1999 by Provos and Mazières.[2]

bcrypt

- cost-parameterized
- uses EksBlowfishSetup (key schedule) and encryption

EksBlowfish

- block cipher
- based on blowfish
- expensive key schedule
- encryption as in blowfish

Implemented in OpenBSD 2.1, Ruby on Rails, and PHP as standard password hash.

[2] http://www.usenix.org/events/usenix99/full_papers/provos/provos.pdf

What is bcrypt?

Introduced in 1999 by Provos and Mazières.[2]

bcrypt

- cost-parameterized
- uses EksBlowfishSetup (key schedule) and encryption

EksBlowfish

- block cipher
- based on blowfish
- expensive key schedule
- encryption as in blowfish

Implemented in OpenBSD 2.1, Ruby on Rails, and PHP as standard password hash.

[2] `http://www.usenix.org/events/usenix99/full_papers/provos/provos.pdf`

EksBlowfish

Pseudo code

Algorithm 1: EksBlowfishSetup

```
Input: cost, salt, key
Output: state
$state \leftarrow \text{InitState}()$;
$state \leftarrow \text{ExpandKey}(state, salt, key)$;
Repeat ($2^{cost}$) begin
    $state \leftarrow \text{ExpandKey}(state, 0, salt)$;
    $state \leftarrow \text{ExpandKey}(state, 0, key)$;
end
return $state$;
```

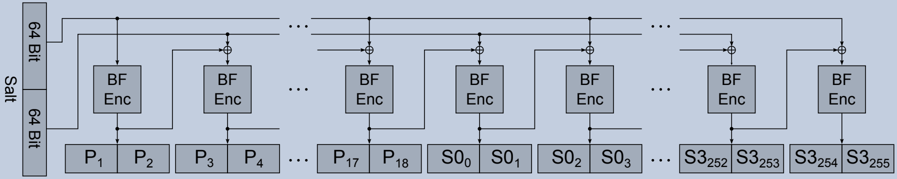
ExpandKey

RUB

ExpandKey(state, salt, key)

- modified version of blowfish key schedule
- XORes key onto subkeys
- state update – *does not use key*
- 521 `bf_enc` calls

State Update



bcrypt

Pseudocode

Algorithm 2: bcrypt

```
Input: cost, salt, key
Output: hash
$state \leftarrow \text{EksBlowfishSetup}(cost, salt, key)$;
$ctext \leftarrow$ “OrpheanBeholderScryDoubt”;
Repeat (64) begin
    $ctext \leftarrow \text{EncryptECB}(state, ctext)$;
end
return $Concatenate(cost, salt, ctext)$;
```

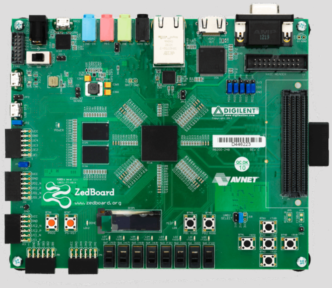
Design

Target Platforms
zedboard

Low cost, low power FPGA

Specs

- Zynq-7000 XC7Z020 FPGA (comparable to Artix7)
- ARM Cortex A9 CPU
- HDMI, VGA, Ethernet, Audio, USB, JTAG, SD Card, Buttons. . .
- 140 36 kbit BRAMs



Target Platforms
RIVYERA

High Performance FPGA cluster

Specs

- 64 Spartan-6 LX150 (available with up to 128)
- i7 CPU
- 16 GB RAM
- 268 36 kbit BRAMs per FPGA

Up to 10 RIVYERAs can be fit into a standard rack.

Optimization Goal

Low Area Footprint

Design

API

- keep API minimalistic ⇒ low bandwidth interface
- transfer target *salt* and *hash*, start cracker
- on success, read *password*

Password Generation

- on-chip password generation
- no bandwidth
- split password range at synthesis-time
- alternatively replace generator with interface for dictionary attacks

Design

bcrypt Core

- memory for SBoxes, subkeys, key and initial values
- estimated almost no logic needed for algorithm
  consists mainly of BRAM access

Problematic

- memory for password storage
  registers require too much logic
- one core consumes more LUTs than expected
  need to optimize FSM, share resources

Design

Quad Core

- bundle four bcrypt cores together
- store four passwords in one BRAM
- during initialization, password generator can access BRAM
- core0, core1 and core2, core3 access the BRAM alternately

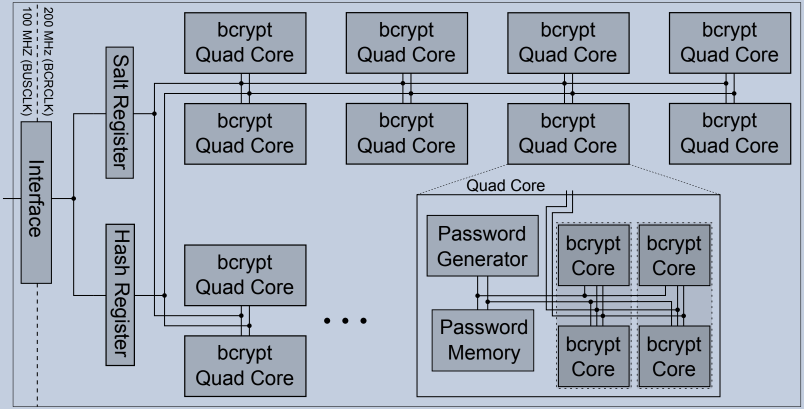
Advantages

- cores need access only during begin of `ExpandKey`
  thus, they only diverge by 36 clock cycles
- saves more than 4000 LUTs per quad core
  20 % per single core

Resulting Design

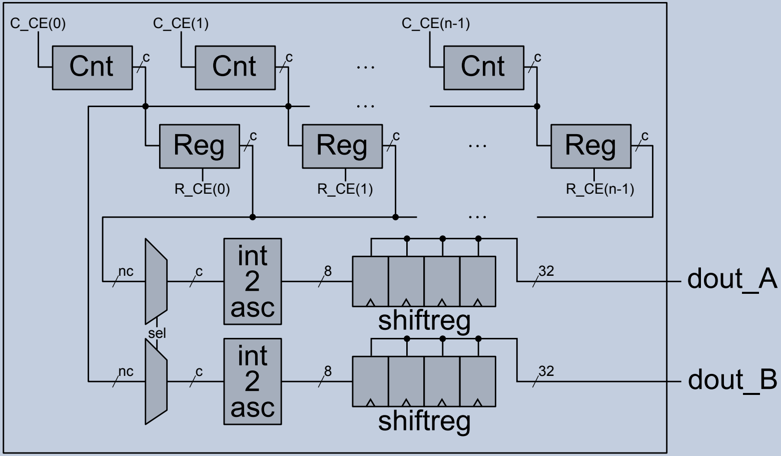
RUB

Quad Core



Resulting Design

Password Generation



Resulting Resources

zedboard

- estimations for one zedboard:

  28 cores as upper bound, BRAMs as limiting resource
- first design attempt (password in registers):

  12 cores fit, LUT utilization way to high
- Quad Core Design:

  24 cores fit, while using “big” interface

RIVYERA

- Quad Core Design:

  48 cores per Spartan-6 LX150 $\Rightarrow$ 3072 cores on whole RIVYERA

Demo

Results

Compared to OpenWall

- OpenWall (GSoC) 780 H/s (cost 5) – zedboard 3,200 H/s

[3] http://www.openwall.com/crypt/

Results

Compared to OpenWall

- OpenWall (GSoC) 780 H/s (cost 5) – zedboard 3,200 H/s

cost Parameter for Target (CPU) runtime

	1 ms	10 ms	100 ms	1000 ms
bcrypt cost	3.69	7.03	10.3	13.6
i5-2400	1015 H/s	101.88 H/s	11.16 H/s	1.15 H/s

resembles performance of server-side implementation[3]

[3] `http://www.openwall.com/crypt/`

Results

Compared to CPUs and GPUs

RUB

Hash-rates

	Hashs per Second			
Target CPU runtime	1 ms	10 ms	100 ms	1000 ms
zedboard	**9,230**	**916.25**	**98.77**	**9.93**
Spartan-6 LX150	9,230	916.25	98.77	9.93
Virtex-6 LX240T	55,380	5,497.5	592.62	59.58
RIVYERA	590,720	58,640.0	6,321.28	635.52
i7-3820 QM	4,800	477.71	51.0	4.6
Xeon E3-1240	14,738	1497.0	172.7	16.8
GTX 480	2,868	319.4	33.7	2.7
GTX 750 Ti	4,794	478.8	52.7	4.6

Results

Besides pure Hash-rate

zedboard

- costs \$300–\$400 — cheaper Zynq-7000 also boards available
- consumes up to 5 W
- no overhead, everything is includes on board

GTX 480

- \$500 in 2010
- 430 W

GTX 750 Ti

- \$150
- 60 W

Optimizations

To be done

- 200 MHz on zedboard, but only 100 MHz on RIVYERA
- low FF/LUT ratio
- unused FFs can be used to buffer signals

  $\Rightarrow$ shorter critical path

  $\Rightarrow$ higher clock speed
- optimize the interface $\Rightarrow$ 28 cores per zedboard

Questions?

Thank you for your attention!