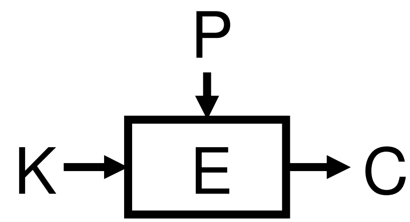
Cracking Unix passwords using FPGA platforms

Nele Mentens, Lejla Batina,
Bart Preneel, Ingrid Verbauwhede
COSIC, Katholieke Universiteit Leuven
SHARCS
25.02.2005

Outline

- Time-memory trade-off
- Unix password hashing
- Time-memory trade-off for Unix password hashing
- Implementation options and results
- Future work
- Conclusion

Time-memory trade-off



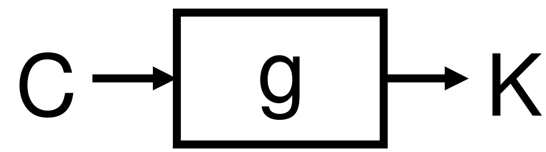
- Encryption $C = E_K(P)$
- Fixed and known plaintext

  $\Rightarrow E_K(P)$ is a one-way function
- Attack scenario: find K for given C
- Straightforward methods:
  - exhaustive key search
  - precomputation table with all (K,C)-pairs
- Time-memory trade-off (Hellman, 1980):
  - less time than exhaustive key search
  - less memory than precomputation

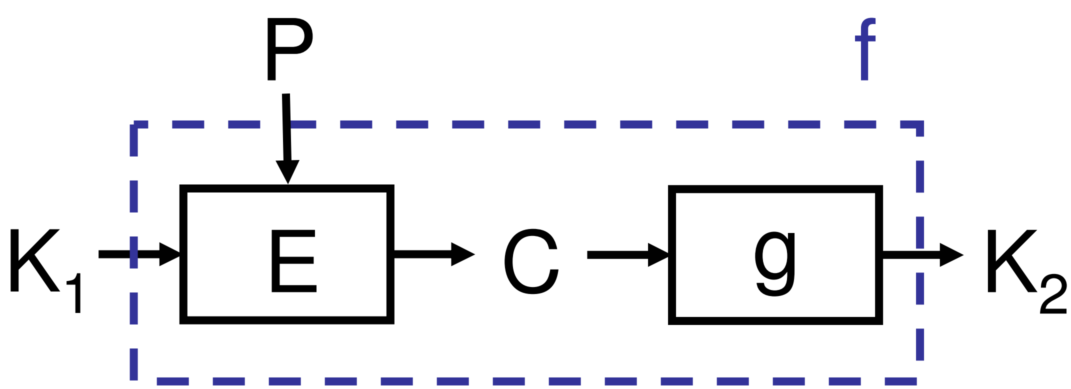
Time-memory trade-off

Two functions are defined:

- $g: \{0,1\}^n \rightarrow \{0,1\}^k$ called reduction function maps a ciphertext to a key.

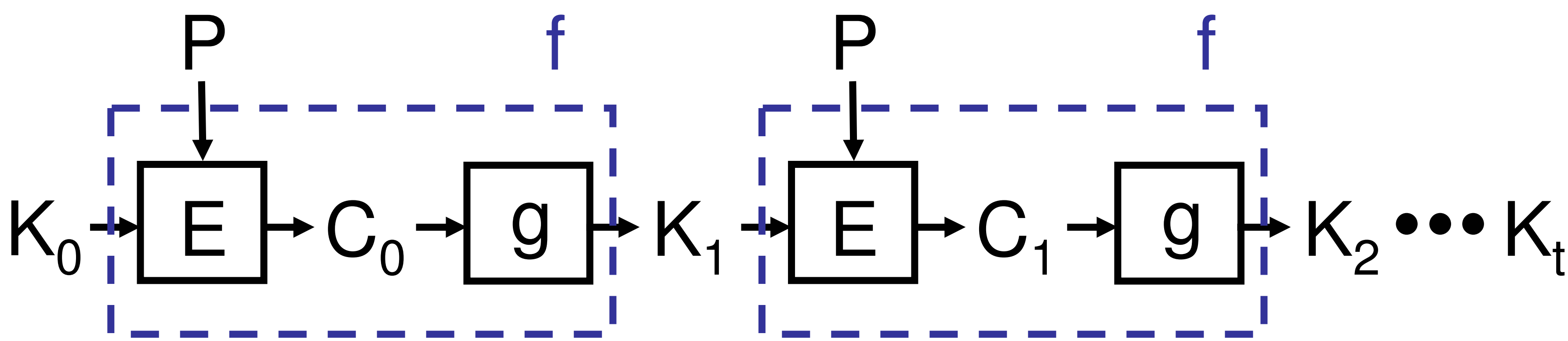


- $f: \{0,1\}^k \rightarrow \{0,1\}^k$ or $f(K) = g(E_K(P))$ maps a key to a key.

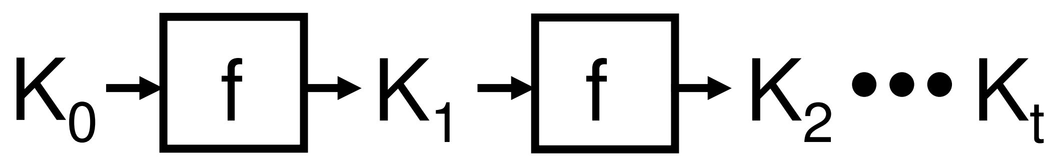


Time-memory trade-off

Now a chain of length t can be constructed:



or



Time-memory trade-off

Original idea from Hellman:

- m chains of length t
- Only the start point (SP) and the end point (EP) of a chain are stored in a table.

Time-memory trade-off

Preparation of the attack (off-line part):

- Start from a key and apply a repeated sequence of encryptions and reduction functions.
- The length of this sequence (chain) is t.
- Start from another key and do the same.
- Repeat this until m chains have been computed.
- Create a table with m start point-end point pairs.

Time-memory trade-off

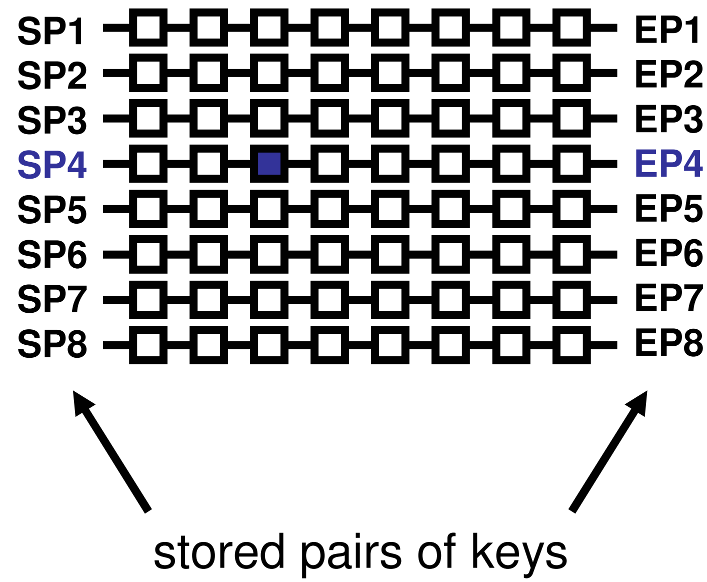
Attack (on-line part):

- Start from the given ciphertext C_a and do the chain computations (repeated sequence of encryptions and reduction functions) until there is a match with the end point of a chain.
- Start from the start point of this chain and compute all intermediate ciphertexts until C_a is found. The key just before C_a is the right one.

Time-memory trade-off

Attack example:

- Start from C_a until EP4 is found.
- Start from SP4 until C_a is found.
- K_2 of chain 4 is the key we need.



$$C_a \to \boxed{g} \to K_a \to \boxed{f} \to K_b \to \boxed{f} \to K_c \to \boxed{f} \to K_d \to \boxed{f} \to K_e \to \boxed{f} \to K_f = \text{EP4}$$

$$\text{SP4} \to \boxed{E} \to C_1 \to \boxed{g} \to K_1 \to \boxed{E} \to C_2 \to \boxed{g} \to K_2 \to \boxed{E} \to C_3 = C_a$$

Time-memory trade-off

improvements:

- distinguished points (Rivest, Borst *et al.*, Stern): only store end points with a special property e.g. last 20 bits are 0

  $\Rightarrow$ reduced number of memory accesses but variable length chains

- rainbow tables (Oechslin): use a different reduction function in every iteration

  $\Rightarrow$ decreased probability of merging chains

Time-memory trade-off

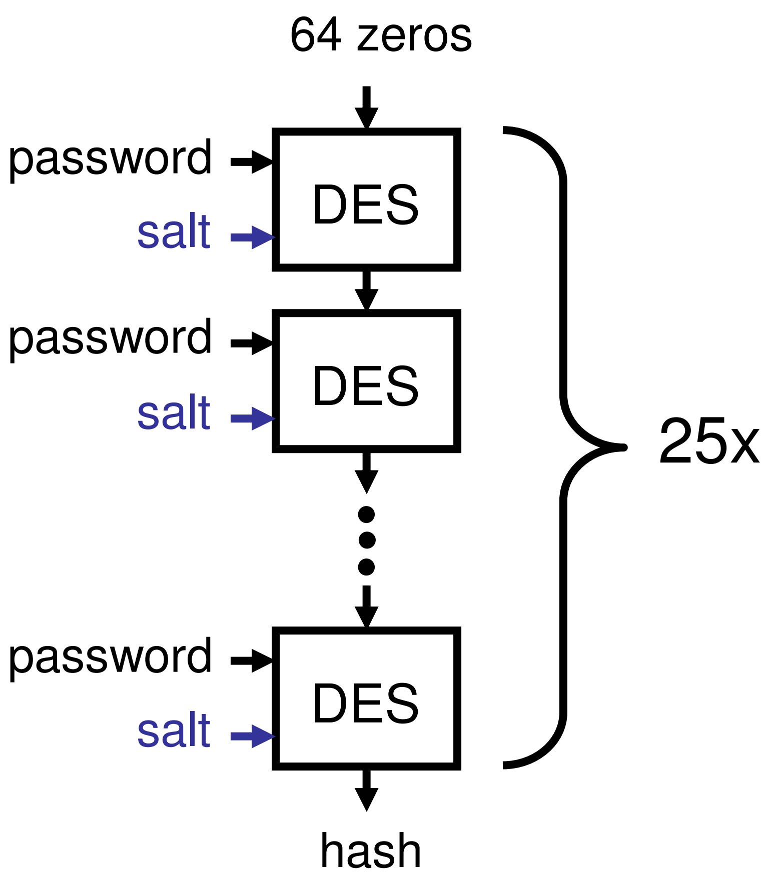
Cost for success probability 86% for 1 rainbow table with $m = 2^{2k/3}$ and $t = 2^{k/3}$

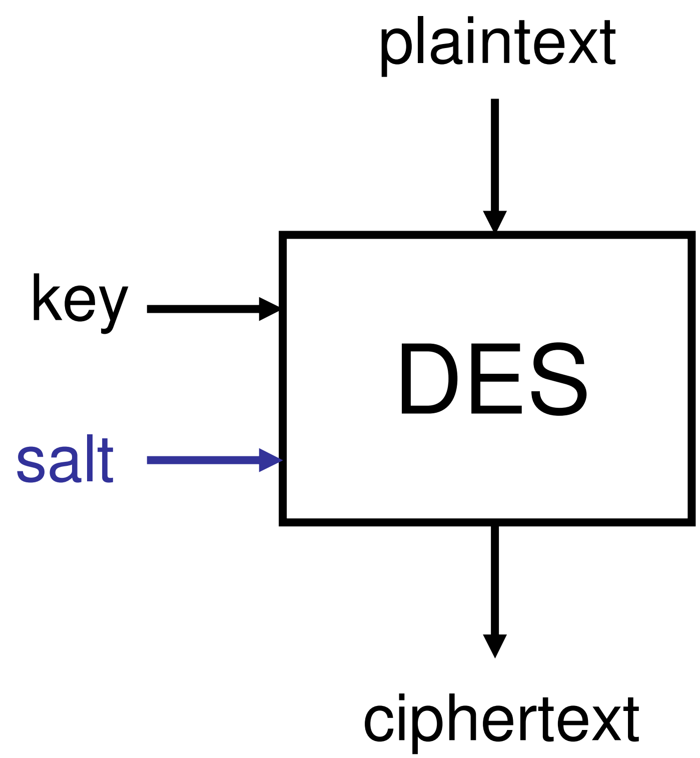
- Precomputation: time 2^k and memory $2^{2k/3}$
- Recovery of one key: time $2^{2k/3}$

Improved analysis based on full cost: see Michael Wiener’s talk

Unix password hashing

The Unix password system uses 25 modified DES blocks.



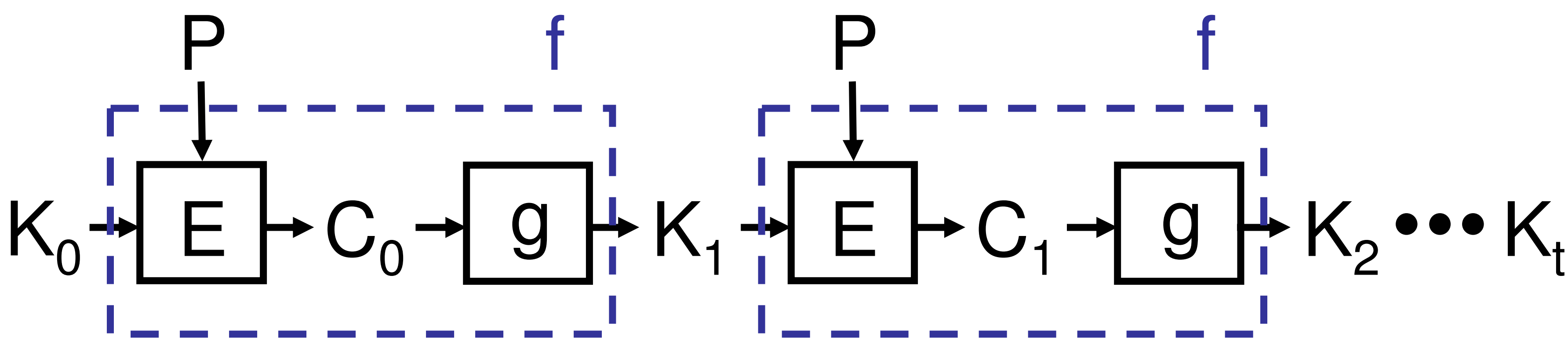


Unix password hashing

/etc/passwd:

- write-protected file
- contains username, salt and hash
- data are stored as ASCII characters

Time-memory trade-off for Unix password hashing



K_i = password – assume k=48

P = 64 zeros

E = 25 modified DES blocks

C_i = hash

g = reduction function

Implementation options and results

Options for the reduction functions:

S-boxes, xor functions, bit swaps

- All options have low hardware complexity.
- For rainbow tables we need one general reduction function from which different reduction functions can be derived.
- Our reduction function is an xor with a counter, which has a different value for each reduction function.

Implementation options and results

Generation of the tables (off-line part):

- Implementation platform:
  BEE2 designed at UC Berkeley
- Variant of time-memory trade-off:
  rainbow tables
- Generation of start points will be done in the FPGA using a counter. The counter in the reduction function can be re-used for this purpose.

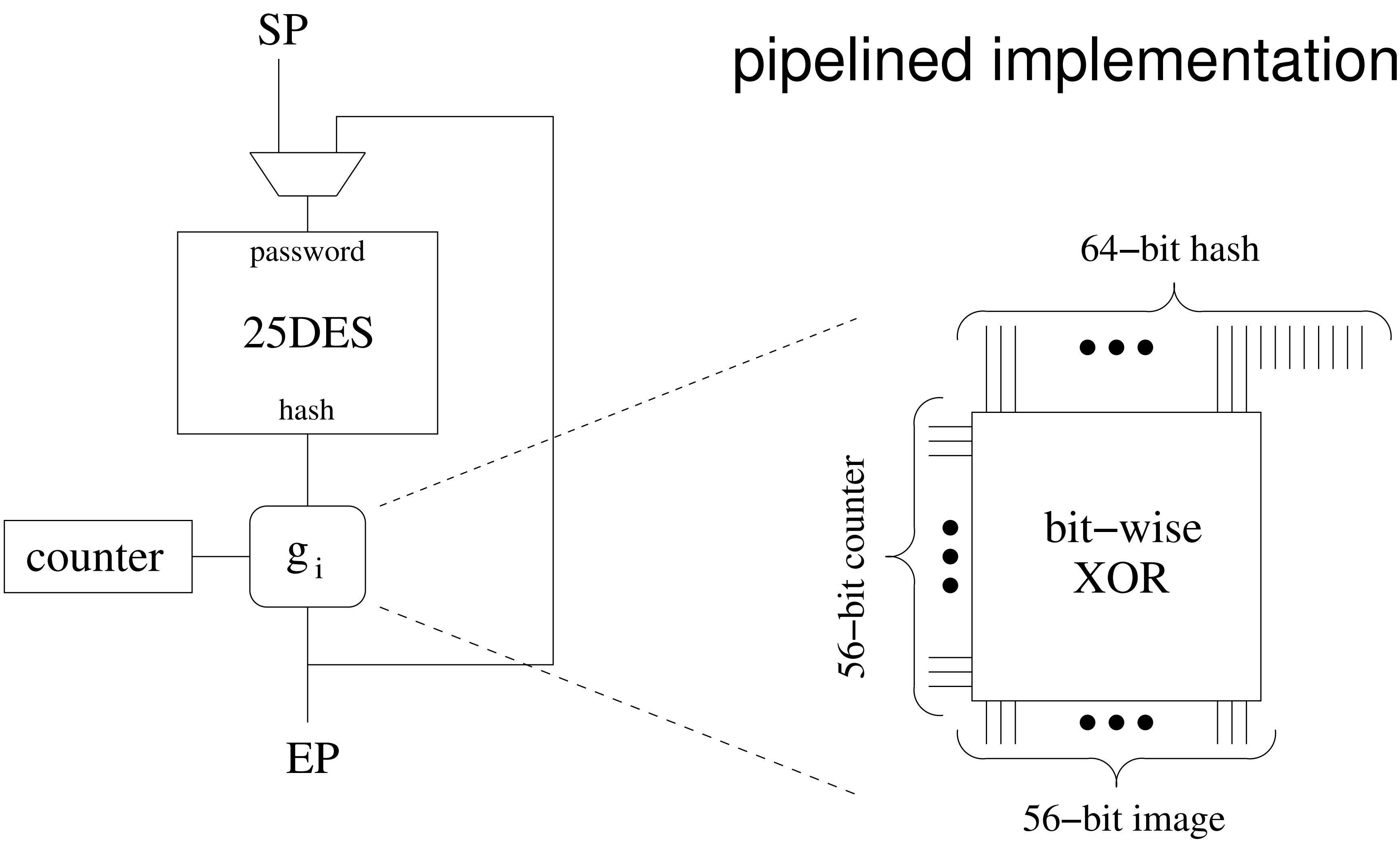
Implementation options and results

The BEE2 platform:

- One BEE2 module consists of five Virtex-II-Pro-70 FPGAs.
- Each BEE2 module has 20 GB DDR-RAM and a 10 Gb/s ethernet connector.
- The platform is modular. Currently it consists of 2 modules, but 10 more are being produced.
- The platform can handle frequencies up to 200-300 MHz.
- The cost per module is ± $7500

Implementation options and results

pipelined implementation



Implementation options and results

Some numbers on the precomputation part:

- Computation for one salt takes 8 days on 1 BEE2 module.
- Precomputation for all salts in one year requires 92 modules.
- Memory complexity per salt is

$$2^{\frac{2}{3}48} * 14 \text{ Bytes} = 56 \text{ GBytes}$$

Implementation options and results

Some numbers on the on-line part:

- Recovering a password using one Virtex-4 takes

$$\frac{2^{16}\left(2^{16}-1\right)}{2}1.5\mu s < 1\,hour$$

- Using 25 pipelining steps it will only take a few minutes.

Implementation options and results

Comparison with other implementations:

	platform	algorithm	speed(enc/s)
Biham, 1997	64-bit Alpha computer	56-bit DES	2M
UCL, 2002	Virtex1000	40-bit DES	66M
Oechslin, 2003	P4, 1.5 GHz, 500 MB RAM	56-bit DES	0.7M
this work	BEE2	25 x 56-bit modified DES	100M

Future work

- Perform the attack for one salt
- Optimize the choice of parameters
- Examine how many tables would be optimal
- Try this on PlayStation 3

Conclusions

- FPGA implementation of the Unix password system
- FPGA creates the table inputs for the off-line part of the time-memory trade-off
- Decisions need to be made on other aspects of the attack