john-devkit: specialized compiler for hash cracking

Aleksey Cherepanov
lyosha@openwall.com

May 26, 2015

Positive Hack Days

General

john-devkit

- is an experiment
  - not yet embraced by John the Ripper developer community
- is a code generator
- on input: algo written in special language and a list of optimizations to apply
- on output: C file for John the Ripper

John the Ripper (JtR)

- the famous hash cracker
- primary purpose is to detect weak Unix passwords
- supports 200+ hash formats (types)
- supports several kinds of compute devices:
  - CPU, Xeon Phi
    - scalar
    - SIMD: SSE2+/AVX/XOP, AVX2, MIC/AVX-512, AltiVec, NEON
  - GPU
    - OpenCL, CUDA
  - FPGA, Epiphany
    - currently for bcrypt only

Problems of JtR development

- scalability of programmers is low due to 200+ formats: sometimes it is hard to apply even 1 optimization to all formats:
  - important formats get the optimization first
  - each additional format to optimize eats more time
- support for each device needs a separate implementation
- readability degrades when various cases are handled by preprocessor

Aims of john-devkit

- to separate crypto algorithms, optimizations, and output code for various devices
- to include optimizations specific for hash cracking and John the Ripper
- to provide better syntax
- to retain or improve performance
- to provide precise control over optimizations
- to support various devices: CPU, GPU, FPGA
- to give great output for great input (not for any input)
- to be simple

Early results

- john-devkit is not mature
- 7 formats were implemented separating crypto primitives, optimizations, and device specific code
- good speeds (over default implementation in JtR):
  - raw-sha256 +22%
  - raw-sha224 +20%
  - raw-sha512 +6%
  - raw-sha384 +5%
- bad speeds (but expose interesting features of john-devkit):
  - raw-sha1 -1%
  - raw-md4 -11%
  - raw-md5 -15%
- optimizations implemented: interleave, vectorization, unroll of loops, early reject, additional batching (loop around algo)
- all formats got all optimizations without effort

Optimizations

Cracking process

- we are in attacker's position
- we have a lot of candidates to try
  - high parallelism
- high level algo:
  - load hashes (once)
  - generate some candidates
  - compute hashes (or only parts)
  - reject most of wrong candidates
  - check probable passwords precisely (rare case)
  - generate next batch of candidates and repeat
- formats are integrated into this process using OOP-like calls over function pointers

Optimizations

- some optimizations do not affect internals of crypto algorithms in any way and may be added to any algorithm
  - additional loop around algo to process more candidates in 1 call
  - OpenMP support
- other optimizations affect crypto algorithms
  - vectorization (SIMD)
  - precomputation
    - e.g. first few steps in MD*/SHA* for partially changed input
  - reversal of operations
    - e.g. last few steps in MD*/SHA*, DES final permutation
  - loop unrolling
  - interleaving
  - bitslicing
  - and others

Bitslice

- splits numbers into bits and computes everything through bitwise operations
- optimization focuses on minimization of Boolean formula (or circuit)
- Roman Rusakov generated current formulas for S-boxes of DES used in John the Ripper with custom generator
  - it took 3 months
- Billy Bob Brumley demonstrated application of simulated annealing algorithm to scheduling of DES S-box instructions
- so code generation is not new for John the Ripper (not even speaking about C preprocessor)

Other solutions

OpenCL

- is the first thing I hear when I say about output for both CPU and GPU
- has quite heavy syntax (based on C)
- knows nothing about John the Ripper
- does not have automatic bitslicing

Dynamic formats in John the Ripper

- were implemented by Jim Fougeron
- separate crypto primitives from formats
  - so md5($p) and md5(md5($p)) have one code base
  - work at runtime
- john-devkit aims to be able to do similar thing but at compile time and with ability to optimize better
  - so md5(md5($p)) would get more optimizations (at price of separate code)

C Macros

- allow to do things, but are not smart
- an example of loop unroll in Keccak defining all useful variants:

```
[...]
#elif (Unrolling == 3)
#define rounds \
    prepareTheta \
    for(i=0; i<24; i+=3) { \
        thetaRhoPiChiIotaPrepareTheta(i  , A, E) \
        thetaRhoPiChiIotaPrepareTheta(i+1, E, A) \
        thetaRhoPiChiIotaPrepareTheta(i+2, A, E) \
        copyStateVariables(A, E) \
    } \
    copyToState(state, A)
#elif (Unrolling == 2)
#define rounds \
    prepareTheta \
    for(i=0; i<24; i+=2) { \
        thetaRhoPiChiIotaPrepareTheta(i  , A, E) \
        thetaRhoPiChiIotaPrepareTheta(i+1, E, A) \
    } \
    copyToState(state, A)
```

X-Macro

- is a tricky way to use macros, most likely with a separate file to be included multiple times:
  - the file has code with variable parts
  - these parts are defined before #include
- so #include provides a "template engine"
- example from NetBSD's libcrypt:

```
[...]
#define HASH_Init SHA1Init
#define HASH_Update SHA1Update
#define HASH_Final SHA1Final
#include "hmac.c"
```

john-devkit technical details

From Python to C in john-devkit

- code in intermediate language (IL) is generated from algorithm description
- the code is modified by several functions chosen by user
- C code is generated from the modified the code using a template

Intermediate Language (IL)

- while algorithms are written in Python with modified environment, john-devkit uses flat representation of code using its own instruction language called intermediate language
- some instructions of this language express constructions specific to hash cracking
  - for instance, state variables of hash functions are defined by special instruction
- intermediate language is very simple
- intermediate language is intended to be rich to express common constructions natively to simplify optimization

Example of specific instruction

- separate instruction is used to define state variable, so john-devkit uses a filter to replace initial state with values for SHA-224 having code for SHA-256:

```
def override_state(code, state):
    consts = {}
    for l in code:
        if l[0] == 'new_const':
            consts[l[1]] = l
        if l[0] == 'new_state_var':
            consts[l[2]][2] = str(state.pop(0))
    return code
```

Positive Hack Days

Openwall

Optimizations specific to password cracking

- use knowledge not available to regular compiler:
- code can be moved between some functions of format
- the functions have different probability to be called
  - so main computation is always called
  - check of probable candidates is very rare
    - it almost implies a successful guess (for strong hashes),
  - also hashes are loaded only once while there are millions of candidates being hashed every second

Specific optimization: early reject

- hashes are long
- some output values may be computed a bit quicker than others
- a 32-bit or 64-bit one value is usually enough to reject almost all wrong candidates
- so john-devkit drops instructions for computation of other output values in main working function and places full implementation into function for precise check of possible password
- main implementation is vectorized while full implementation is scalar because it checks only 1 candidate

Specific optimization: steps reversal

- some operations can be reversed
  - if r = i + C, we know r, and C is a constant, then i = r - C
  - John the Ripper learns "r" when it loads hashes
- john-devkit can sometimes reverse operations, replacing "forward" computation during cracking (applied per candidate password) with reverse computation at startup (applied per hash)

Full Python

- is available to define algorithms
- the environment has some objects with overloaded instructions to produce code in IL in a global variable instead of running it right away
- but any Python code can be used
  - it is evaluated fully before further steps of code generation
  - but to produce good output some additional markup may be needed

Full Python, example

- a part of MD4 definition adapted right from RFC 1320:

```
def make_round(func, code):
    res = ''
    func = re.sub('([abcdks])', r'{\1}', func)
    parts = re.compile(r'\[(.)(.)(.)(.)\s+(\d+)\s+(\d+)\]'
            ).findall(code)
    for a, b, c, d, k, s in parts:
        res += func.format(**vars()) + "\n"
    return res

exec make_round('a = rol((a + F(b, c, d) + X[k]), s)',
'''     [ABCD  0  3]  [DABC  1  7]  [CDAB  2 11]  [BCDA  3 19]
        [ABCD  4  3]  [DABC  5  7]  [CDAB  6 11]  [BCDA  7 19]
        [ABCD  8  3]  [DABC  9  7]  [CDAB 10 11]  [BCDA 11 19]
        [ABCD 12  3]  [DABC 13  7]  [CDAB 14 11]  [BCDA 15 19]
''')
```

Conclusions

- john-devkit demonstrates practical application of code generation approach
- john-devkit is a real way to automate programmer's work at such scale

Thank you!

- Thank you!
- code: https://github.com/AlekseyCherepanov/john-devkit
- more technical detail will be on john-dev mailing list
- my email: lyosha@openwall.com