Fast Exhaustive Search for Polynomial Systems in $\mathbb{F}_2$

GPUs for Brute-Force Enumeration

presented by: Bo-Yin Yang

Institute of Information Science
and TWISC, Academia Sinica
Taipei, Taiwan
by@crypto.tw

CHES'10 @UCSB on Aug18

Outline

Our Results

The Problem

Solve for n variables in $\mathbb{F}_2$ from $m \geq n$ "generic" equations of low degree.

The Results: Generalized Gray Code Enumeration

- Data for 48 vars and 48 to 64 equations below, with estimate in USD to solve a Patarin IP challenge (64-64, quartic) in 1 month.
- Beat all Gröbner Bases solvers for practical sizes, generic case.

Time (minutes)			Testing platform				#cores	est. cost
$d=2$	$d=3$	$d=4$	GHz	Arch.	Name	USD	(#used)	(USD)
1217	2686	3191	2.2	K10	Phenom 9550	120	4(1)	54,000
1157	1992	2685	2.3	K10+	Opteron 2376	184	4(1)	113,316
142	240	336	2.3	K10+	Opteron 2376×2	368	8(8)	
780	1364	1819	2.4	C2	Xeon X3220	210	4(1)	60,720
671	1176	1560	2.83	C2+	Core2 Q9550	225	4(1)	55,575
179	294	390	2.83	C2+	Core2 Q9550	225	4(4)	
761	1279	1856	2.26	Ci7	Xeon E5520	385	4(1)	78,720
95	154	225	2.26	Ci7	Xeon E5520×2	770	8(8)	
41	73	271	1.3	G200	GTX 280	n/a	240	n/a
21	36	126	1.25	G200	GTX 295	500	480	15,500

Exhaustive Search in degree $d \geq 2$ – Summary

1. Complexity of each iteration : $\binom{n}{d} \rightarrow O(d \lg^{\beta(d)} n)$
2. Internal state: $1 \rightarrow \binom{n}{d-1}$

Headline (for all degree $d \geq 2$)

Exhaustive search requires d XOR per candidate vector.

3. Easy to implement efficiently

Exhaustive Search in degree $d \geq 2$ – Summary

1. Complexity of each iteration : $\binom{n}{d} \to O(d \lg^{\beta(d)} n)$
2. Internal state: $1 \to \binom{n}{d-1}$

Headline (for all degree $d \geq 2$)

Exhaustive search requires d XOR per candidate vector.

3. ~~Easy to implement efficiently~~... (more on that later)

Exhaustive Search in degree $d \geq 2$ – Summary

1. Complexity of each iteration : $\binom{n}{d} \rightarrow O(d \lg^{\beta(d)} n)$
2. Internal state: $1 \rightarrow \binom{n}{d-1}$

Headline (for all degree $d \geq 2$)

Exhaustive search requires d XOR per candidate vector.

3. ~~Easy to implement efficiently~~… (more on that later)

4. …on GPUs too !

Naive Search for n $\mathbb{F}_2$-vars in m eqs of degree-d?

A deg $= d$ equation over $\mathbb{F}_2$ has $\binom{n}{k}$ terms at degree k for $k \leq d$.

- $\approx m\binom{n}{d}$ bit operations (OPs) per input, naively.
- $\approx 2\binom{n}{d}$ OPs per input if we compute serially and **early-abort**
- If logical operations are w-wide, we can bit-slice to speed up $\sim w\times$

"Folklore" Gray Code Search

Define $b_i(x) :=$ index of the i-th non-zero bit in binary representation of x (-1 if no such), then standard Gray code: $G(k) = G(k-1)\ \texttt{XOR}\ 2^{b_1(k)}$.
With e_i representing the unit vector in the i-th direction, define:

$$d_i f(\mathbf{x}) := f(\mathbf{x} + e_i) - f(\mathbf{x}),$$

then identifying a codeword $\mathbf{x}$ with a vector in $(\mathbb{F}_2)^n$, we have

$$f(G(k)) = f(G(k-1)) + d_{b_1(k)} f(G(k-1)).$$

When f quadratic: differentials $d_i f$'s affine, can evaluate in time $O(n)$.
Per-input time complexity down to $O(n)$, no change in memory use.

Recursive Gray Code Search

- Every 2 iterations x_0 (lowest bit of Gray Code) flips.

Recursive Gray Code Search

- It always goes: flip x_0, flip some other x_i, flip x_0.
- Hence, each time x_0 is flipped, the partial derivative to x_0 has last been seen (evaluated) at an input 1 bit away.

Recursive Gray Code Search

- It always goes: flip x_0, flip some other x_i, flip x_0.
- Hence, each time x_0 is flipped, the partial derivative to x_0 has last been seen (evaluated) at an input 1 bit away.
- x_i (not low bit) always changes when the i bits below $= 10 \cdots 0$, and if we strike out the last i bits of a Gray Code sequence, we get Gray Code with each entry repeated 2^i times, and x_i as new low bit.
- Hence any differential w.r.t. x_i also last evaluated 1 bit away.
- Mathematical Induction pushes this to any higher degree.

Recursive Gray Code Search

- It always goes: flip x_0, flip some other x_i, flip x_0.
- Hence, each time x_0 is flipped, the partial derivative to x_0 has last been seen (evaluated) at an input 1 bit away.
- x_i (not low bit) always changes when the i bits below $= 10\cdots0$, and if we strike out the last i bits of a Gray Code sequence, we get Gray Code with each entry repeated 2^i times, and x_i as new low bit.
- Hence any differential w.r.t. x_i also last evaluated 1 bit away.
- Mathematical Induction pushes this to any higher degree.

Quadratic Example (with $d_{ij}f(\mathbf{x}) := d_i f(\mathbf{x}+e_j) - d_i f(\mathbf{x})$)

$$\begin{aligned}
f(1) - f(0) &= d_0 f(0) \\
f(11) - f(1) &= d_1 f(1) = d_1 f(0) + d_{01} \\
f(10) - f(11) &= d_0 f(11) = d_0 f(10) = d_0 f(0) + d_{01} \\
f(110) - f(10) &= d_2 f(10) = d_2 f(0) + d_{12} \\
f(111) - f(110) &= d_0 f(110) = d_0 f(11) + d_{02} \\
\text{2 XORs/input} &= \ldots (n/2)\times \text{ speedup, } n\times \text{ RAM use}
\end{aligned}$$

Part of 5-bit Gray Code Example Table

Note that the initial appearance of each δ requires a setup.

index	code	b_1	b_2	b_3	b_4	actions(quadratic)	actions(quartic)
00000	00000	-1	-1	-1	-1		
00001	00001	0	-1	-1	-1	$\delta \mathrel{+}= \delta_0$	$\delta \mathrel{+}= \delta_0$
00010	00011	1	-1	-1	-1	$\delta \mathrel{+}= \delta_1$	$\delta \mathrel{+}= \delta_1$
00011	00010	0	1	-1	-1	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,1})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \delta_{0,1})$
00100	00110	2	-1	-1	-1	$\delta \mathrel{+}= \delta_2$	$\delta \mathrel{+}= \delta_2$
00101	00111	0	2	-1	-1	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,2})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \delta_{0,2})$
00110	00101	1	2	-1	-1	$\delta \mathrel{+}= (\delta_1 \mathrel{+}= \mathbf{C}_{1,2})$	$\delta \mathrel{+}= (\delta_1 \mathrel{+}= \delta_{1,2})$
00111	00100	0	1	2	-1	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,1})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= (\delta_{0,1} \mathrel{+}= \delta_{0,1,2}))$
01000	01100	3	-1	-1	-1	$\delta \mathrel{+}= \delta_3$	$\delta \mathrel{+}= \delta_3$
01001	01101	0	3	-1	-1	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,3})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \delta_{0,3})$
01010	01111	1	3	-1	-1	$\delta \mathrel{+}= (\delta_1 \mathrel{+}= \mathbf{C}_{1,3})$	$\delta \mathrel{+}= (\delta_1 \mathrel{+}= \delta_{1,3})$
01011	01110	0	1	3	-1	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,1})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= (\delta_{0,1} \mathrel{+}= \delta_{0,1,3}))$
01100	01010	2	3	-1	-1	$\delta \mathrel{+}= (\delta_2 \mathrel{+}= \mathbf{C}_{2,3})$	$\delta \mathrel{+}= (\delta_2 \mathrel{+}= \delta_{2,3})$
01101	01011	0	2	3	-1	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,2})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= (\delta_{0,2} \mathrel{+}= \delta_{0,2,3}))$
01110	01001	1	2	3	-1	$\delta \mathrel{+}= (\delta_1 \mathrel{+}= \mathbf{C}_{1,2})$	$\delta \mathrel{+}= (\delta_1 \mathrel{+}= (\delta_{1,2} \mathrel{+}= \delta_{1,2,3}))$
01111	01000	0	1	2	3	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,1})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= (\delta_{0,1} \mathrel{+}= (\delta_{0,1,2} \mathrel{+}= \mathbf{C}_{0,1,2,3})))$
10000	11000	4	-1	-1	-1	$\delta \mathrel{+}= \delta_4$	$\delta \mathrel{+}= \delta_4$
10001	11001	0	4	-1	-1	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,4})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \delta_{0,4})$
10010	11011	1	4	-1	-1	$\delta \mathrel{+}= (\delta_1 \mathrel{+}= \mathbf{C}_{1,4})$	$\delta \mathrel{+}= (\delta_1 \mathrel{+}= \delta_{1,4})$
10011	11010	0	1	4	-1	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,1})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= (\delta_{0,1} \mathrel{+}= \delta_{0,1,4}))$
10100	11110	2	4	-1	-1	$\delta \mathrel{+}= (\delta_2 \mathrel{+}= \mathbf{C}_{2,4})$	$\delta \mathrel{+}= (\delta_2 \mathrel{+}= \delta_{2,4})$
10101	11111	0	2	4	-1	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,2})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= (\delta_{0,2} \mathrel{+}= \delta_{0,2,4}))$
10110	11101	1	2	4	-1	$\delta \mathrel{+}= (\delta_1 \mathrel{+}= \mathbf{C}_{1,2})$	$\delta \mathrel{+}= (\delta_1 \mathrel{+}= (\delta_{1,2} \mathrel{+}= \delta_{1,2,4}))$
10111	11100	0	1	2	4	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,1})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= (\delta_{0,1} \mathrel{+}= (\delta_{0,1,2} \mathrel{+}= \mathbf{C}_{0,1,2,4})))$
11000	10100	3	4	-1	-1	$\delta \mathrel{+}= (\delta_3 \mathrel{+}= \mathbf{C}_{3,4})$	$\delta \mathrel{+}= (\delta_3 \mathrel{+}= \delta_{3,4})$
11001	10101	0	3	4	-1	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,3})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= (\delta_{0,3} \mathrel{+}= \delta_{0,3,4}))$
11010	10111	1	3	4	-1	$\delta \mathrel{+}= (\delta_1 \mathrel{+}= \mathbf{C}_{1,3})$	$\delta \mathrel{+}= (\delta_1 \mathrel{+}= (\delta_{1,3} \mathrel{+}= \delta_{1,3,4}))$
11011	10110	0	1	3	4	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= \mathbf{C}_{0,1})$	$\delta \mathrel{+}= (\delta_0 \mathrel{+}= (\delta_{0,1} \mathrel{+}= (\delta_{0,1,3} \mathrel{+}= \mathbf{C}_{0,1,3,4})))$

Part of 5-bit Gray Code Example Table

Note that the initial appearance of each δ requires a setup.

index	code	b_1	b_2	b_3	b_4	actions(quadratic)	actions(quartic)
00000	00000	-1	-1	-1	-1		
00001	00001	0	-1	-1	-1	δ += δ_0	δ += δ_0
00010	00011	1	-1	-1	-1	δ += δ_1	δ += δ_1
00011	00010	0	1	-1	-1	δ += (δ_0 += $\mathbf{C}_{0,1}$)	δ += (δ_0 += $\delta_{0,1}$)
00100	00110	2	-1	-1	-1	δ += δ_2	δ += δ_2
00101	00111	0	2	-1	-1	δ += (δ_0 += $\mathbf{C}_{0,2}$)	δ += (δ_0 += $\delta_{0,2}$)
00110	00101	1	2	-1	-1	δ += (δ_1 += $\mathbf{C}_{1,2}$)	δ += (δ_1 += $\delta_{1,2}$)
00111	00100	0	1	2	-1	δ += (δ_0 += $\mathbf{C}_{0,1}$)	δ += (δ_0 += ($\delta_{0,1}$ += $\delta_{0,1,2}$))
01000	01100	3	-1	-1	-1	δ += δ_3	δ += δ_3
01001	01101	0	3	-1	-1	δ += (δ_0 += $\mathbf{C}_{0,3}$)	δ += (δ_0 += $\delta_{0,3}$)
01010	01111	1	3	-1	-1	δ += (δ_1 += $\mathbf{C}_{1,3}$)	δ += (δ_1 += $\delta_{1,3}$)
01011	01110	0	1	3	-1	δ += (δ_0 += $\mathbf{C}_{0,1}$)	δ += (δ_0 += ($\delta_{0,1}$ += $\delta_{0,1,3}$))
01100	01010	2	3	-1	-1	δ += (δ_2 += $\mathbf{C}_{2,3}$)	δ += (δ_2 += $\delta_{2,3}$)
01101	01011	0	2	3	-1	δ += (δ_0 += $\mathbf{C}_{0,2}$)	δ += (δ_0 += ($\delta_{0,2}$ += $\delta_{0,2,3}$))
01110	01001	1	2	3	-1	δ += (δ_1 += $\mathbf{C}_{1,2}$)	δ += (δ_1 += ($\delta_{1,2}$ += $\delta_{1,2,3}$))
01111	01000	0	1	2	3	δ += (δ_0 += $\mathbf{C}_{0,1}$)	δ += (δ_0 += ($\delta_{0,1}$ += ($\delta_{0,1,2}$ += $\mathbf{C}_{0,1,2,3}$)))

Initialization Costs

When $i_1 > i_2 > \cdots > i_k$, the differential $d_{i_1,\, i_2, \ldots,\, i_k}$ appears initially as

$$\frac{\partial^k}{\partial x_{i_1} \partial x_{i_2} \cdots \partial x_{i_k}} f(e_{i_1 - 1} + e_{i_2 - 1} + \cdots + e_{i_k - 1})$$

sum of $\sum_{j=0}^{d-k} \binom{k}{j}$ coefficients total $\approx \binom{n}{d-1} d$ XORs.

Partial (s-of-n) Evaluation

Substituting s variables out of n to get 2^s subsystems in $n - s$ variables.

Partial (s-of-n) Evaluation

Substituting s variables out of n to get 2^s subsystems in $n - s$ variables.

Why Do We Need Partial Evaluation

- Required for Parallel Processing
- Convenient for memory management
- Optimization (for Enumeration and Check)

Partial (s-of-n) Evaluation

Substituting s variables out of n to get 2^s subsystems in $n - s$ variables.

Why Do We Need Partial Evaluation

- Required for Parallel Processing
- Convenient for memory management
- Optimization (for Enumeration and Check)

Things to Note

- Coeffs of deg-k terms are deg $= d - k$ poly in the s vars.
- Highest order coefficients are constant, can be shared.
- Store coefficients of subsystems out to buffer DRAM, may require two-stages if memory is limited (e.g., GPUs).
- Basically same code as for enumeration

Early Abort and Architectural Concerns

- Machine test-for-0 on w-bit-wide word, does $\underline{w \text{ ANDs} + \text{branch}}$.
- Evaluating many eqs. wastes less than not using machine instructions.
- So "enumerate" on $f^{(0)} \cdots f^{(w-1)}$ then "check" 2^{n-w} passers.

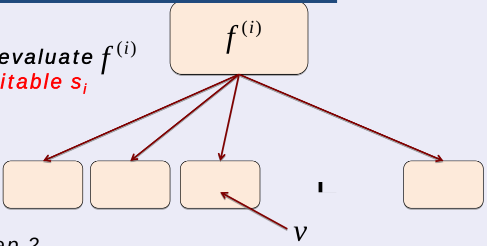
Early Abort and Architectural Concerns

- Machine test-for-0 on w-bit-wide word, does $\underline{w \text{ ANDs} + \text{branch}}$.
- Evaluating many eqs. wastes less than not using machine instructions.
- So "enumerate" on $f^{(0)} \cdots f^{(w-1)}$ then "check" 2^{n-w} passers.

Complete Granularity: Enumeration and Partial Eval as One?

$V^{(i)} := \{\mathbf{x} \in \{0,1\}^n : f^{(0)}(x) = f^{(1)}(x) = \cdots = f^{(i-1)}(x) = 0\}$

To compute $V^{(i)}$ from $V^{(i-1)}$ for each i



Step 1.
partial evaluate $f^{(i)}$ with suitable s_i

Step 2.
evaluate $f^{(i)}(v)$ by substituting $v \in V^{(i-1)}$ into the corresponding subsystem

If components of $f(\mathbf{x}) = 0$ are filters, what is the best #vars to partially evaluate from the equation $f^{(i)}$?

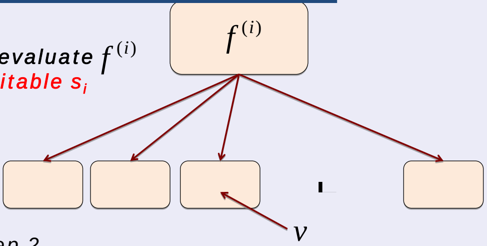
Early Abort and Architectural Concerns

- Machine test-for-0 on w-bit-wide word, does $\underline{w \text{ ANDs} + \text{branch}}$.
- Evaluating many eqs. wastes less than not using machine instructions.
- So "enumerate" on $f^{(0)} \dots f^{(w-1)}$ then "check" 2^{n-w} passers.

Complete Granularity: Enumeration and Partial Eval as One?

$V^{(i)} := \{\mathbf{x} \in \{0,1\}^n : f^{(0)}(x) = f^{(1)}(x) = \cdots = f^{(i-1)}(x) = 0\}$

To compute $V^{(i)}$ from $V^{(i-1)}$ for each i



Step 1. partial evaluate $f^{(i)}$ with suitable s_i

Step 2. evaluate $f^{(i)}(v)$ by substituting $v \in V^{(i-1)}$ into the corresponding subsystem

If components of $f(\mathbf{x}) = 0$ are filters, what is the best #vars to partially evaluate from the equation $f^{(i)}$? Needs $\left(2 \cdot \sum_{j=0}^{d} \binom{n-s}{j}\right)$ times 2^{n-s} in $V^{(i)}$. Plus must partial evaluate 2^s times $\sum_{j=0}^{d}(n-j)\binom{n-s}{j}$ $(n, n, n-1, \dots, 0, \dots)$ is generally good if we search for some good s_i.

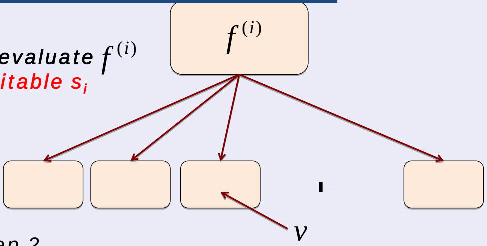
Early Abort and Architectural Concerns

- Machine test-for-0 on w-bit-wide word, does $\underline{w \text{ ANDs} + \text{branch}}$.
- Evaluating many eqs. wastes less than not using machine instructions.
- So "enumerate" on $f^{(0)} \cdots f^{(w-1)}$ then "check" 2^{n-w} passers.

Complete Granularity: Enumeration and Partial Eval as One?

$$V^{(i)} := \{\mathbf{x} \in \{0,1\}^n : f^{(0)}(x) = f^{(1)}(x) = \cdots = f^{(i-1)}(x) = 0\}$$

To compute $V^{(i)}$ from $V^{(i-1)}$ for each i



Step 1.
partial evaluate $f^{(i)}$ with suitable s_i

Step 2.
evaluate $f^{(i)}(v)$ by substituting $v \in V^{(i-1)}$ into the corresponding subsystem

If components of $f(\mathbf{x}) = 0$ are filters, what is the best #vars to partially evaluate from the equation $f^{(i)}$? Needs $\left(2 \cdot \sum_{j=0}^{d} \binom{n-s}{j}\right)$ times 2^{n-s} in $V^{(i)}$. Plus must partial evaluate 2^s times $\sum_{j=0}^{d} (n-j)\binom{n-s}{j}$ $(n, n, n-1, \ldots, 0, \ldots)$ is generally good if we search for some good s_i.

"optimal" is $(n, n, n-\ell, n-\ell-1, \ldots, h+1, h, 0, \ldots, 0)$ for small (h, ℓ).

Check vs Enumerate

Changing check width

If per-input cost to enumerate is c, cost to check (verify) v, width w, and we want to change the width to $w' < w$, then we must check

$$c + 2^{-w} v \overset{?}{>} c' + 2^{-w'} v'$$

if $w' < w$, it simplifies to checking $c - c' \overset{?}{>} 2^{-w'} v'$.

Check vs Enumerate

Changing check width

If per-input cost to enumerate is c, cost to check (verify) v, width w, and we want to change the width to $w' < w$, then we must check

$$c + 2^{-w} v \overset{?}{>} c' + 2^{-w'} v'$$

if $w' < w$, it simplifies to checking $c - c' \overset{?}{>} 2^{-w'} v'$.

CPU with SSE2 instructions: Optimal Width=16

With SSE2, all widths are powers of two. Assuming that $w' = w/2$ and $c' \approx c/2$, then it is better to use w'-wide if $v' < 2^{w'} c'$.

Check vs Enumerate

Changing check width

If per-input cost to enumerate is c, cost to check (verify) v, width w, and we want to change the width to $w' < w$, then we must check

$$c + 2^{-w} v \overset{?}{>} c' + 2^{-w'} v'$$

if $w' < w$, it simplifies to checking $c - c' \overset{?}{>} 2^{-w'} v'$.

CPU with SSE2 instructions: Optimal Width=16

With SSE2, all widths are powers of two. Assuming that $w' = w/2$ and $c' \approx c/2$, then it is better to use w'-wide if $v' < 2^{w'} c'$.
In all our tests when $w' = 16$, v' is about 400, 1500, and 5000 cycles for degrees 2, 3, 4 respectively, and c' on Intel CPUs is around 0.4, 0.7, 0.9 cycles respectively (with AMD CPUs slower by up to a factor of 2).

Check vs Enumerate

Changing check width

If per-input cost to enumerate is c, cost to check (verify) v, width w, and we want to change the width to $w' < w$, then we must check

$$c + 2^{-w} v \overset{?}{>} c' + 2^{-w'} v'$$

if $w' < w$, it simplifies to checking $c - c' \overset{?}{>} 2^{-w'} v'$.

CPU with SSE2 instructions: Optimal Width=16

With SSE2, all widths are powers of two. Assuming that $w' = w/2$ and $c' \approx c/2$, then it is better to use w'-wide if $v' < 2^{w'} c'$.
In all our tests when $w' = 16$, v' is about 400, 1500, and 5000 cycles for degrees 2, 3, 4 respectively, and c' on Intel CPUs is around 0.4, 0.7, 0.9 cycles respectively (with AMD CPUs slower by up to a factor of 2).
Since $2^{10} < v'/c' < 2^{14}$ in all cases, this points to our checking $w' = 16$ equations at a time being better than either 32 or 8.

Our Programs

Three Stages

- Partial Evaluation: at least partly done on the CPU.
- Enumerate: this accounts for more than 90% of runtime.
- Checking: always done on the CPU.

Code Generation

We have perl scripts that generate the programs given some parameters, which are very regular and straight-line code, for both GPU and CPU. These generated programs also capture our ideas about memory space allocation and register spilling, etc.

Vector Code on Modern x86-64 CPUs

The SSE2 Instruction Set

- SSE2 instruction let you act in parallel on 8-, 16-, 32-, or 64-bit chunks simultaneously. Here we do 8 chunks at a time.
- Turns out that more than SSE2 instruction set does not help much.
- Intel Core's can dispatch 3 operations on XMM registers per cycle, AMD CPUs only 2.
- Since we are working with more than 1 entries in an XMM register, it takes extra work to extract, particularly the `PMOVMSKB` instruction.
- Caches are basically large enough and good enough to mask the latencies except on the old version of the K10.

Vector Code on Modern x86-64 CPUs

The SSE2 Instruction Set

- SSE2 instruction let you act in parallel on 8-, 16-, 32-, or 64-bit chunks simultaneously. Here we do 8 chunks at a time.
- Turns out that more than SSE2 instruction set does not help much.
- Intel Core's can dispatch 3 operations on XMM registers per cycle, AMD CPUs only 2.
- Since we are working with more than 1 entries in an XMM register, it takes extra work to extract, particularly the `PMOVMSKB` instruction.
- Caches are basically large enough and good enough to mask the latencies except on the old version of the K10.

Notes

- Branches are relatively cheap, so we use them instead of conditionals.
- s is smaller than for GPUs since we do not need as many threads. But for both CPUs and GPUs, s increase roughly linear as n.

Performance Testing on PCs

Timing (mins) and Cycle counts at $n = 48$, $m = 64$

deg = 2		deg = 3		deg = 4		CPU	
kernel	cycle	kernel	cycle	kernel	cycle	GHz	arch
1217	0.57	2686	1.26	3191	1.50	2.2	K10
1157	0.57	1992	0.98	2685	1.32	2.3	K10+
780	0.40	1364	0.70	1819	0.93	2.4	C2
671	0.41	1176	0.71	1560	0.94	2.83	C2+
761	0.37	1279	0.62	1856	0.89	2.26	Ci7

Notes

- Cycle count is almost strictly a function of arch
- scaling with any modern core is almost perfect

Performance Testing on PCs

Timing (mins) and Cycle counts at $n = 48$, $m = 64$

deg = 2		deg = 3		deg = 4		CPU	
kernel	cycle	kernel	cycle	kernel	cycle	GHz	arch
1217	0.57	2686	1.26	3191	1.50	2.2	K10
1157	0.57	1992	0.98	2685	1.32	2.3	K10+
142	0.56	240	0.94	336	1.32	$\times 8$ Cores	
780	0.40	1364	0.70	1819	0.93	2.4	C2
671	0.41	1176	0.71	1560	0.94	2.83	C2+
761	0.37	1279	0.62	1856	0.89	2.26	Ci7

Notes

- Cycle count is almost strictly a function of arch
- scaling with any modern core is almost perfect

Sample Code

C Intrinsics Code

```
...
diff0 ^= deg2_block[ 1 ];
res ^= diff0;
Mask = _mm_cmpeq_epi16(res, zero);
mask = _mm_movemask_epi8(Mask);
if(mask) check(mask, idx, x^155);
...
```

Sample Code

C Intrinsics Code

```
...
diff0 ^= deg2_block[ 1 ];
res ^= diff0;
Mask = _mm_cmpeq_epi16(res, zero);
mask = _mm_movemask_epi8(Mask);
if(mask) check(mask, idx, x^155);
...
```

Assembly Code

```
.L746:
      movq       976(%rsp), %rax  //
      pxor       (%rax), %xmm2    // d_y ^= C_yz
      pxor       %xmm2, %xmm1     // res ^= d_y
      pxor       %xmm0, %xmm0     //
      pcmpeqw    %xmm1, %xmm0     // cmp words for eq
      pmovmskb   %xmm0, %eax      // movemask
      testw      %ax, %ax         // set flag for branch
      jne        .L1266           // if needed, check and
.L747:                            // comes back here
```

Performance Analysis

Instruction counts

- For quadratics 2 XORs, then 1 COMPARE, then a `PMOVMSKB`, then a test-and-branch. Hence, 6 instructions including 3 XMM loads. Ideal value is 3 cycles/loop. Actual value is slightly higher.
- For cubics/quartics, 7/8 XMM instructions respectively.

Performance Analysis

Instruction counts

- For quadratics 2 XORs, then 1 COMPARE, then a `PMOVMSKB`, then a test-and-branch. Hence, 6 instructions including 3 XMM loads. Ideal value is 3 cycles/loop. Actual value is slightly higher.
- For cubics/quartics, 7/8 XMM instructions respectively.

Comparison to PS3

Version used for cryptanalysis is sold at a subsidy (US$300)

- $6\times$ 3.2GHz synergetic processing elements (SPEs) usable,
- each SPE does 128-bit wide logical OP/cycle in main pipeline,
- with a secondary pipeline to handle bookkeeping.

Hence max. $6\times$ 3.2GHz 128-bit OPs.

Performance Analysis

Instruction counts

- For quadratics 2 XORs, then 1 COMPARE, then a `PMOVMSKB`, then a test-and-branch. Hence, 6 instructions including 3 XMM loads. Ideal value is 3 cycles/loop. Actual value is slightly higher.
- For cubics/quartics, 7/8 XMM instructions respectively.

Comparison to PS3

Version used for cryptanalysis is sold at a subsidy (US$300) max. 6× 3.2GHz 128-bit OPs.

The competition

- AMD K10+ can do 4 core $\times$ 2 = 8 XMM OPs/cycle.
- Intel Quad Core's can do 4 core $\times 3 = 12$ XMM OPs/cycle.
- The K10+ has utilization ratio $\sim 7/8$, So a 3.2GHz Phenom IIx4 (or 2.66GHz Ci7) always beats a Cell.

nVidia G200 GPU on GeForce GTX 280 / Tesla C1060

470 mm^2, TDP: 204 watts (TSMC 55nm), $>$1.4 bil transistor, w. $\approx$ 1GB DRAM

1062.72 GFLOPS (single-precision), 159 GB/s mem bandwidth vs. 106.56 GFLOPS, 25.6 GB/s of Intel Core i7 at 3.33 GHz

- 30 "Multiprocessors" (MPs, Real Units of GPU Computing)
  - 8 ALUs, each can do 1 FMADD/cycle @ 1.296GHz
  - 2 Special Function Units, each 2 actions/cycle (incl. FMUL)
  - 16k registers (each 32bits), 16kB shared memory
  - 8kB constant cache into 64kB constant area
  - Latency: 22-26 cycles from SRAM 20 cycles, $>$ 400 cycles from DRAM.
  - Opportunitistic but in-order dispatcher, 1 64-bit instruction decode per 4 cycles, $\geq$ 128 lightweight hardware threads (4 "warps") needed for performance. We don't even mention the Magic Textile Units.
- How to Program with CUDA
  - Program in C-like language `*.cu` file
  - Compile with `nvcc` to pseudomachine code (`*.ptx`).
  - Load with nVidia driver, load/launch from the main program.
  - Buggy compiler and can only jump to first 16k instructions (!).

GPU Code and Performance Analysis

2.6 SP cycles for quadratics

```
    ...
    diff0 ^= deg2_block[ 3 ];   // d_y^=d_yz
    res ^= diff0;               // res^=d_y
    if( res == 0 ) y = z;       // cmov
    if( res == 0 ) z = code233; // cmov
    diff1 ^= deg2_block[ 4 ];
    res ^= diff1;
    if( res == 0 ) y = z;
    if( res == 0 ) z = code234;
    diff0 ^= deg2_block[ 0 ];
    res ^= diff0;
    if( res == 0 ) y = z;
    if( res == 0 ) z = code235;
     ...
```

The "`if (X=0) Y=Z;`" bit is actually the magic words to emit a predicated move, a 32-bit "half instruction".

GPU Code and Performance Analysis

2.6 SP cycles for quadratics

The inner loop has 5-6 instructions but < 3 cycles, clearly the SFU can dispatch some instructions, very well.

```
...
xor.b32 $r19, $r19, c0[0x000c]       // d_y^=d_yz
xor.b32 $p1|$r20, $r17, $r20
mov.b32 $r3, $r1
mov.b32 $r1, s[$ofs1+0x0038]
xor.b32 $r4, $r4, c0[0x0010]
xor.b32 $p0|$r20, $r19, $r20         // res^=d_y
@$p1.eq mov.b32 $r3, $r1
@$p1.eq mov.b32 $r1, s[$ofs1+0x003c]
xor.b32 $r19, $r19, c0[0x0000]
xor.b32 $p1|$r20, $r4, $r20
@$p0.eq mov.b32 $r3, $r1             // cmov
@$p0.eq mov.b32 $r1, s[$ofs1+0x0040] // cmov
...
```

Switching to predicated branch instead of predicated move makes each input take some 3.5 cycles on average!

CPU vs GPU

Need to rerun a thread

Due to artifacts of GPU programming, threads return “no possible solution”, “one possible solution”, and “too many possible solutions”. The last is rerun completely on CPU.

CPU vs GPU
Need to rerun a thread

Due to artifacts of GPU programming, threads return "no possible solution", "one possible solution", and "too many possible solutions". The last is rerun completely on CPU.

GPU Thread-Checking Probability

If we have n variables, pre-evaluate s, and check w equations via Gray Code, then the probability of a subsystem with 2^{n-s} vectors including at least two candidates $\approx \binom{2^{n-s}}{2}(1-2^{-w})^{2^{n-s}}(2^{-w})^2 \approx 1/2^{2(s+w-n)+1}$, provided that $n < s + w$.

CPU vs GPU
Need to rerun a thread

Due to artifacts of GPU programming, threads return “no possible solution”, “one possible solution”, and “too many possible solutions”. The last is rerun completely on CPU.

GPU Thread-Checking Probability

If we have n variables, pre-evaluate s, and check w equations via Gray Code, then the probability of a subsystem with 2^{n-s} vectors including at least two candidates $\approx \binom{2^{n-s}}{2}(1-2^{-w})^{2^{n-s}}(2^{-w})^2 \approx 1/2^{2(s+w-n)+1}$, provided that $n < s+w$.

Example

For $n = 48$, $s = 22$, $w = 32$, the thread-recheck probability is about 1 in 2^{13}, and we must re-check about 2^9 threads (using Gray Code).

CPU vs GPU

Need to rerun a thread

Due to artifacts of GPU programming, threads return “no possible solution”, “one possible solution”, and “too many possible solutions”. The last is rerun completely on CPU.

GPU Thread-Checking Probability

If we have n variables, pre-evaluate s, and check w equations via Gray Code, then the probability of a subsystem with 2^{n-s} vectors including at least two candidates $\approx \binom{2^{n-s}}{2}(1-2^{-w})^{2^{n-s}}(2^{-w})^2 \approx 1/2^{2(s+w-n)+1}$, provided that $n < s + w$.

Example

For $n = 48$, $s = 22$, $w = 32$, the thread-recheck probability is about 1 in 2^{13}, and we must re-check about 2^9 threads (using Gray Code).

It gains little with conditionals (a packed SUBTRACT update a counter then 3 bookkeeping instructions, still 3 loads) over branching on CPU and a thread-check is expensive, hence the latter.

Performance Comparison at 32 variables, 64 equations

Cycle Counts (SP vs core)

deg	C2	C2+	K10	K10+	Ci7	GTX280	GTX295
2	0.40	0.40	0.58	0.57	0.41	2.87	2.93
3	0.65	0.66	1.21	0.91	0.66	4.66	4.90
4	0.95	0.96	1.41	1.32	1.00	15.01	14.76

Performance Comparison at 32 variables, 64 equations

Cycle Counts (SP vs core)

deg	C2	C2+	K10	K10+	Ci7	GTX280	GTX295
2	0.40	0.40	0.58	0.57	0.41	2.87	2.93
3	0.65	0.66	1.21	0.91	0.66	4.66	4.90
4	0.95	0.96	1.41	1.32	1.00	15.01	14.76

2.66GHz Core i7 vs 1.242GHz GT200 GPU

- For quadratics, 1 Core i7 $\approx$ $15.2SP \lesssim 2MP$
- For cubics, 1 Core i7 $\approx$ $16.6SP \gtrsim 2MP$
- For quartics 1 Core i7 $\approx$ $38.4SP \approx 4.8MP$

Performance Comparison at 32 variables, 64 equations

Cycle Counts (SP vs core)

deg	C2	C2+	K10	K10+	Ci7	GTX280	GTX295
2	0.40	0.40	0.58	0.57	0.41	2.87	2.93
3	0.65	0.66	1.21	0.91	0.66	4.66	4.90
4	0.95	0.96	1.41	1.32	1.00	15.01	14.76

2.66GHz Core i7 vs 1.242GHz GT200 GPU

- For quadratics, 1 Core i7 $\approx$ $15.2SP \lesssim 2MP$
- For cubics, 1 Core i7 $\approx$ $16.6SP \gtrsim 2MP$
- For quartics 1 Core i7 $\approx$ $38.4SP \approx 4.8MP$

Why is quartics so different?

GPU Memory Pressure Partly Explained

Each thread on a G200b has less than 150 “slots” of 32-bit fast SRAM space (registers and shared memory) to store the differentials. So they must inevitably overflow (spill) to slow DRAM (“local memory”).

GPU Memory Pressure Partly Explained

Each thread on a G200b has less than 150 "slots" of 32-bit fast SRAM space (registers and shared memory) to store the differentials. So they must inevitably overflow (spill) to slow DRAM ("local memory").

Frequency of Differential Access

The differential $\delta_{i_1>i_2>\cdots>i_k}$ is accessed once every 2^{i_1+1} inputs. We allocate the most often used differentials to fast SRAM memory and read the remaining from slow DRAM.

Each MP with 8 ALUs (SPs) can potentially dispatch 8 instruction per cycle or more, but the DRAM controllers can only load once every cycle to each MP. Hence, we can have at most one DRAM load out of every 8 instructions ($< 12.5\%$) if we wish to get full use from ALUs.

GPU Memory Pressure Partly Explained

Each thread on a G200b has less than 150 “slots” of 32-bit fast SRAM space (registers and shared memory) to store the differentials. So they must inevitably overflow (spill) to slow DRAM (“local memory”).

Frequency of Differential Access

The differential $\delta_{i_1>i_2>\cdots>i_k}$ is accessed once every 2^{i_1+1} inputs. We allocate the most often used differentials to fast SRAM memory and read the remaining from slow DRAM.

Each MP with 8 ALUs (SPs) can potentially dispatch 8 instruction per cycle or more, but the DRAM controllers can only load once every cycle to each MP. Hence, we can have at most one DRAM load out of every 8 instructions ($< 12.5\%$) if we wish to get full use from ALUs.

For quartics (with parameters) we chose through empirical testing, some 23% of instructions involve a DRAM operand, so throughput of an MP is at halved at least, and in fact, this is observed.

Summary

Take-Away Point

GPUs are good for cryptanalysis.

Summary

Take-Away Point

GPUs are good for cryptanalysis and easier than FPGAs.

Devil is in the Details

Easy to get "working", but hard to get "just right".

Summary

Take-Away Point

GPUs are good for cryptanalysis and easier than FPGAs.

Devil is in the Details

Easy to get "working", but hard to get "just right".

Take-Away Point

For not-too-overdetermined $\mathbb{F}_2$ systems in the practical range, Brute Force works better than $\mathbf{F_4}$ and other Gröbner basis solvers. This affects, for example, security guarantees of `QUAD`-type stream ciphers.

Summary

Take-Away Point

GPUs are good for cryptanalysis and easier than FPGAs.

Devil is in the Details

Easy to get "working", but hard to get "just right".

Take-Away Point

For not-too-overdetermined $\mathbb{F}_2$ systems in the practical range, Brute Force works better than $\mathbf{F_4}$ and other Gröbner basis solvers. This affects, for example, security guarantees of `QUAD`-type stream ciphers.

Future

A full version will be uploaded to the ePrint Archive prior to journal submission, and we may have packages for people to download and use.

Credits

- This talk describes joint work with
    - Charles Bouillaguet, ENS, France
    - Adi Shamir, Weizmann Institute (visiting ENS)
- I am also joined by my conspirator Doug Cheng and kids from our lab
    - Tony (Tung) Chou, EE, Nat'l Taiwan U
    - Hsieh-Chung Isidore Chen, IIS, Academia Sinica
    - Ruben Niederhagen, IIS & TU Eindhoven

  Also many thanks to Ming-Shing Chen for general support.

Thanks for Listening!

- Questions or comments?