How to use the negation map in the Pollard rho method

Peter Schwabe

中央研究院

Joint work with Daniel J. Bernstein and Tanja Lange

March 09, 2012

EiPSI Crypto Working Group, Utrecht

A few words about Taiwan and Academia Sinica

- Taiwan (台灣) is an island south of China
- About 36,200 km^2 large
- Territory of the Republic of China (not to be confused with the People's Republic of China)
- Capital is Taipei (台北)
- Marine tropical climate

A few words about Taiwan and Academia Sinica

- Taiwan (台灣) is an island south of China
- About 36,200 km^2 large
- Territory of the Republic of China (not to be confused with the People's Republic of China)
- Capital is Taipei (台北)
- Marine tropical climate
- 99 summits over 3000 meters (highest peak: 3952 m)
- Wildlife includes black bears, salmon, monkeys...

A few words about Taiwan and Academia Sinica

- Taiwan (台灣) is an island south of China
- About 36,200 km^2 large
- Territory of the Republic of China (not to be confused with the People's Republic of China)
- Capital is Taipei (台北)
- Marine tropical climate
- 99 summits over 3000 meters (highest peak: 3952 m)
- Wildlife includes black bears, salmon, monkeys...
- Academia Sinica is a research facility funded by ROC
- About 30 institutes
- More than 800 principal investigators, about 900 postdocs and more than 2200 students

A picture from Taiwan – Sun-Moon Lake (日月潭)

For more pictures check out http://cryptojedi.org/gallery/

The discrete-logarithm problem

- Let $G = \langle P \rangle$ be a finite cyclic group with generator P
- In the following: G is written additively

The discrete-logarithm problem

- Let $G = \langle P \rangle$ be a finite cyclic group with generator P
- In the following: G is written additively
- Given $Q \in G$, the discrete-logarithm problem (DLP) is to find $k \in \mathbb{Z}$, such that

$$k \cdot P = Q$$

The discrete-logarithm problem

- Let $G = \langle P \rangle$ be a finite cyclic group with generator P
- In the following: G is written additively
- Given $Q \in G$, the discrete-logarithm problem (DLP) is to find $k \in \mathbb{Z}$, such that
$$k \cdot P = Q$$
- For certain groups G this problem is the basis of many asymmetric cryptographic protocols
- Most importantly: $\mathbb{Z}/n\mathbb{Z}$ and elliptic-curve groups

Pollard's rho algorithm

- Does not use any additional structure (aside from the group structure)
- Best known algorithm to solve the DLP in generic groups of prime order

Pollard's rho algorithm

- Does not use any additional structure (aside from the group structure)
- Best known algorithm to solve the DLP in generic groups of prime order
- Uses a pseudorandom iteration function $f : G \to G$
- Start with $W_0 = n_0 P + m_0 Q$
- Iteratively apply f to obtain $W_{i+1} = f(W_i)$

Pollard's rho algorithm

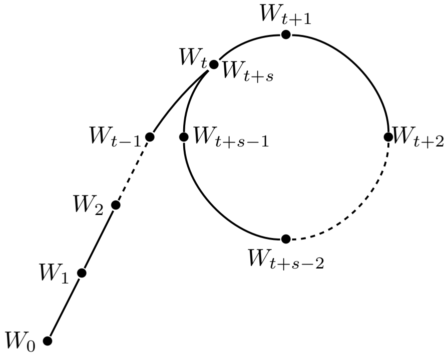
- Does not use any additional structure (aside from the group structure)
- Best known algorithm to solve the DLP in generic groups of prime order
- Uses a pseudorandom iteration function $f : G \to G$
- Start with $W_0 = n_0 P + m_0 Q$
- Iteratively apply f to obtain $W_{i+1} = f(W_i)$
- Update n_{i+1}, m_{i+1} from n_i, m_i (compute modulo $|G|$)
- f needs to preserve knowledge about the linear combination in P and Q

Pollard's rho algorithm

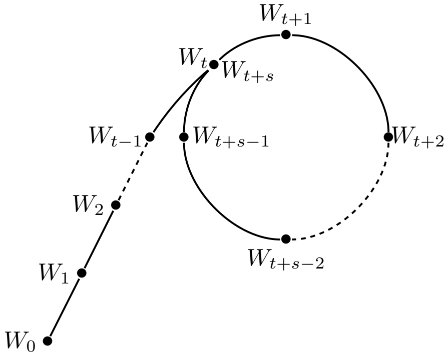
- Does not use any additional structure (aside from the group structure)
- Best known algorithm to solve the DLP in generic groups of prime order
- Uses a pseudorandom iteration function $f : G \to G$
- Start with $W_0 = n_0 P + m_0 Q$
- Iteratively apply f to obtain $W_{i+1} = f(W_i)$
- Update n_{i+1}, m_{i+1} from n_i, m_i (compute modulo $|G|$)
- f needs to preserve knowledge about the linear combination in P and Q
- If $W_i = W_j$ for $i \neq j$, then

$$n_i P + m_i Q = n_j P + m_j Q \Rightarrow$$
$$k = (n_j - n_i)/(m_i - m_j) \mod |G|$$

Pollard's rho algorithm II



Pollard's rho algorithm II

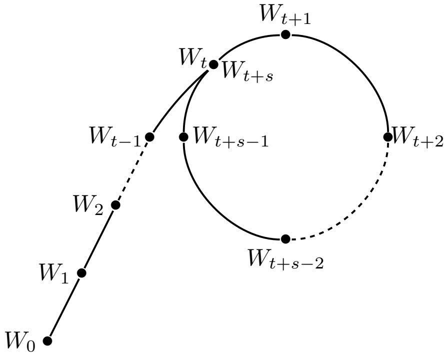


- Easy way to define f:

$$f(W) = n(W)P + m(W)Q,$$

with pseudorandom functions $n, m : G \to \mathbb{Z}/|G|\mathbb{Z}$

Pollard's rho algorithm II

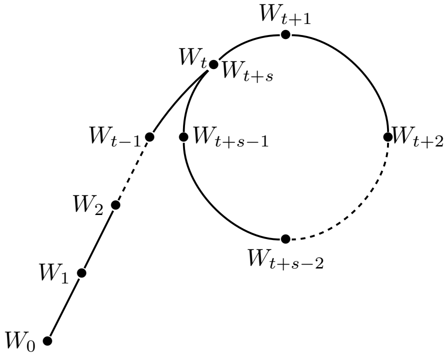


- Easy way to define f:

  $$f(W) = n(W)P + m(W)Q,$$

  with pseudorandom functions $n, m : G \to \mathbb{Z}/|G|\mathbb{Z}$
- Expected number of iterations until entering a cycle: $\sqrt{\frac{\pi|G|}{2}}$

Pollard's rho algorithm II



- Easy way to define f:

  $$f(W) = n(W)P + m(W)Q,$$

  with pseudorandom functions $n, m : G \to \mathbb{Z}/|G|\mathbb{Z}$
- Expected number of iterations until entering a cycle: $\sqrt{\frac{\pi|G|}{2}}$
- Detect cycles without storing all W_i: Floyd, Brent

Parallel Pollard

- Large instances of the DLP call for parallel computing
- Trivial parallelization of Pollard's rho algorithm on t computers gives speedup of $\sqrt{t}$

Parallel Pollard

- Large instances of the DLP call for parallel computing
- Trivial parallelization of Pollard's rho algorithm on t computers gives speedup of $\sqrt{t}$
- Much better: Use parallel approach by van Oorschot and Wiener:
- Client-Server approach, computation done on many clients
- Uses the notion of *distinguished points* (DPs), easy-to-determine property, such as "last d bits of the element's encoding are 0"

Parallel Pollard

- Large instances of the DLP call for parallel computing
- Trivial parallelization of Pollard's rho algorithm on t computers gives speedup of $\sqrt{t}$
- Much better: Use parallel approach by van Oorschot and Wiener:
- Client-Server approach, computation done on many clients
- Uses the notion of *distinguished points* (DPs), easy-to-determine property, such as "last d bits of the element's encoding are 0"
- Clients start from random points and iterate until they reach a DP
- Send starting point and DP to the server, restart from new random point

Parallel Pollard

- Large instances of the DLP call for parallel computing
- Trivial parallelization of Pollard's rho algorithm on t computers gives speedup of $\sqrt{t}$
- Much better: Use parallel approach by van Oorschot and Wiener:
- Client-Server approach, computation done on many clients
- Uses the notion of *distinguished points* (DPs), easy-to-determine property, such as "last d bits of the element's encoding are 0"
- Clients start from random points and iterate until they reach a DP
- Send starting point and DP to the server, restart from new random point
- Server searches in incoming points for collisions (same DP, different starting point)

Some notes on parallel Pollard

- Walks do not enter a cycle, shape is more like a λ

Some notes on parallel Pollard

- Walks do not enter a cycle, shape is more like a λ
- Choice of DP-property influences length of separate walks
- Fewer DPs: longer walks (on average), less storage, less communication
- More DPs: Less overhead after a collision

Some notes on parallel Pollard

- Walks do not enter a cycle, shape is more like a λ
- Choice of DP-property influences length of separate walks
- Fewer DPs: longer walks (on average), less storage, less communication
- More DPs: Less overhead after a collision
- Clients do not have to update n_i and m_i, simply do successful walks again to find coefficients

Additive walks

- Main cost of (parallalized) Pollard's rho algorithm: calls to the iteration function
- With $f(W) = n(W)P + m(W)Q$: two hash-function calls, one double-scalar multiplication

Additive walks

- Main cost of (parallalized) Pollard's rho algorithm: calls to the iteration function
- With $f(W) = n(W)P + m(W)Q$: two hash-function calls, one double-scalar multiplication
- Much more efficient: Additive walks
- Precompute r pseudorandom elements $R_0, \ldots, R_{r-1}$ with known linear combination in P and Q
- Use hash function $h : G \to \{0, r-1\}$
- Define $f(W) = W + R_{h(W)}$

Additive walks

- Main cost of (parallalized) Pollard's rho algorithm: calls to the iteration function
- With $f(W) = n(W)P + m(W)Q$: two hash-function calls, one double-scalar multiplication
- Much more efficient: Additive walks
- Precompute r pseudorandom elements $R_0, \dots, R_{r-1}$ with known linear combination in P and Q
- Use hash function $h : G \to \{0, r-1\}$
- Define $f(W) = W + R_{h(W)}$
- Now: only one hash-function call, one group addition

Additive walks

- Main cost of (parallalized) Pollard's rho algorithm: calls to the iteration function
- With $f(W) = n(W)P + m(W)Q$: two hash-function calls, one double-scalar multiplication
- Much more efficient: Additive walks
- Precompute r pseudorandom elements $R_0, \ldots, R_{r-1}$ with known linear combination in P and Q
- Use hash function $h : G \to \{0, r-1\}$
- Define $f(W) = W + R_{h(W)}$
- Now: only one hash-function call, one group addition
- Additive walks are noticably nonrandom, they require more iterations

Additive walks

- Main cost of (parallalized) Pollard's rho algorithm: calls to the iteration function
- With $f(W) = n(W)P + m(W)Q$: two hash-function calls, one double-scalar multiplication
- Much more efficient: Additive walks
- Precompute r pseudorandom elements $R_0, \ldots, R_{r-1}$ with known linear combination in P and Q
- Use hash function $h : G \to \{0, r-1\}$
- Define $f(W) = W + R_{h(W)}$
- Now: only one hash-function call, one group addition
- Additive walks are noticably nonrandom, they require more iterations
- Teske showed that large r provides close-to-random behaviour (e.g. $r = 20$)

Additive walks

- Main cost of (parallalized) Pollard's rho algorithm: calls to the iteration function
- With $f(W) = n(W)P + m(W)Q$: two hash-function calls, one double-scalar multiplication
- Much more efficient: Additive walks
- Precompute r pseudorandom elements $R_0, \ldots, R_{r-1}$ with known linear combination in P and Q
- Use hash function $h : G \to \{0, r-1\}$
- Define $f(W) = W + R_{h(W)}$
- Now: only one hash-function call, one group addition
- Additive walks are noticably nonrandom, they require more iterations
- Teske showed that large r provides close-to-random behaviour (e.g. $r = 20$)
- Summary: additive walks provide much better performance in practice

Application to elliptic-curve groups

- So far, everything worked in the generic-group model
- Now consider groups of points on elliptic curves
- Group elements are points (x, y)
- Efficient operation aside from group addition: negation
- For Weierstrass curves: $(x, y) \mapsto (x, -y)$

Application to elliptic-curve groups

- So far, everything worked in the generic-group model
- Now consider groups of points on elliptic curves
- Group elements are points (x, y)
- Efficient operation aside from group addition: negation
- For Weierstrass curves: $(x, y) \mapsto (x, -y)$
- Some curves have more efficiently computable endomorphisms, examples are Koblitz curves and BN curves

Application to elliptic-curve groups

- So far, everything worked in the generic-group model
- Now consider groups of points on elliptic curves
- Group elements are points (x, y)
- Efficient operation aside from group addition: negation
- For Weierstrass curves: $(x, y) \mapsto (x, -y)$
- Some curves have more efficiently computable endomorphisms, examples are Koblitz curves and BN curves
- Idea: Define iterations on equivalence classes modulo negation
- For example: always take the lexicographic minimum of $(x, -y)$ and (x, y)

Application to elliptic-curve groups

- So far, everything worked in the generic-group model
- Now consider groups of points on elliptic curves
- Group elements are points (x, y)
- Efficient operation aside from group addition: negation
- For Weierstrass curves: $(x, y) \mapsto (x, -y)$
- Some curves have more efficiently computable endomorphisms, examples are Koblitz curves and BN curves
- Idea: Define iterations on equivalence classes modulo negation
- For example: always take the lexicographic minimum of $(x, -y)$ and (x, y)
- This halves the size of the search space, expected number of iterations drops by a factor of $\sqrt{2}$

Putting it together

- Precompute $R_0, \ldots, R_{r-1}$
- Clients start at some random W_0
- Iteratively compute $W_{i+1} = |W_i + R_{h(W_i)}|$
- $|W|$ chooses a well-defined representative in $\{-W, W\}$

Putting it together

- Precompute $R_0, \ldots, R_{r-1}$
- Clients start at some random W_0
- Iteratively compute $W_{i+1} = |W_i + R_{h(W_i)}|$
- $|W|$ chooses a well-defined representative in $\{-W, W\}$
- Problem: *fruitless cycles*
  If $t = h(W_i) = h(W_{i+1})$

Putting it together

- Precompute $R_0, \dots, R_{r-1}$
- Clients start at some random W_0
- Iteratively compute $W_{i+1} = |W_i + R_{h(W_i)}|$
- $|W|$ chooses a well-defined representative in $\{-W, W\}$
- Problem: *fruitless cycles*
  If $t = h(W_i) = h(W_{i+1})$, and $|W_i + R_t| = -(W_i + R_t)$ we obtain the following sequence:

Putting it together

- Precompute $R_0, \ldots, R_{r-1}$
- Clients start at some random W_0
- Iteratively compute $W_{i+1} = |W_i + R_{h(W_i)}|$
- $|W|$ chooses a well-defined representative in $\{-W, W\}$
- Problem: *fruitless cycles*
  If $t = h(W_i) = h(W_{i+1})$, and $|W_i + R_t| = -(W_i + R_t)$ we obtain the following sequence:

$$W_{i+1} = f(W_i) = -(W_i + R_t)$$
$$W_{i+2} = f(W_{i+1}) = |-(W_i + R_t) + R_t| = |-W_i| = W_i$$

Putting it together

- Precompute $R_0, \ldots, R_{r-1}$
- Clients start at some random W_0
- Iteratively compute $W_{i+1} = |W_i + R_{h(W_i)}|$
- $|W|$ chooses a well-defined representative in $\{-W, W\}$
- Problem: *fruitless cycles*
  If $t = h(W_i) = h(W_{i+1})$, and $|W_i + R_t| = -(W_i + R_t)$ we obtain the following sequence:

$$W_{i+1} = f(W_i) = -(W_i + R_t)$$
$$W_{i+2} = f(W_{i+1}) = |-(W_i + R_t) + R_t| = |-W_i| = W_i$$

- Probability for such fruitless cycles: $1/2r$

Putting it together

- Precompute $R_0, \ldots, R_{r-1}$
- Clients start at some random W_0
- Iteratively compute $W_{i+1} = |W_i + R_{h(W_i)}|$
- $|W|$ chooses a well-defined representative in $\{-W, W\}$
- Problem: *fruitless cycles*
  If $t = h(W_i) = h(W_{i+1})$, and $|W_i + R_t| = -(W_i + R_t)$ we obtain the following sequence:

$$W_{i+1} = f(W_i) = -(W_i + R_t)$$
$$W_{i+2} = f(W_{i+1}) = |-(W_i + R_t) + R_t| = |-W_i| = W_i$$

- Probability for such fruitless cycles: $1/2r$
- Similar observations hold for longer fruitless cycles (length 4,6,. . .)
- Probability of a cycle of length $2c$ is $\approx 1/r^c$

Dealing with fruitless cycles

- Avoid frequent cycles by choosing large r
- Problem: Lookups become more expensive (cache issues)

Dealing with fruitless cycles

- Avoid frequent cycles by choosing large r
- Problem: Lookups become more expensive (cache issues)
- Avoid larger cycles by frequent distinguished points
- Early-abort walks after a certain number of iterations
- Problem: Large communication cost and storage

Dealing with fruitless cycles

- Avoid frequent cycles by choosing large r
- Problem: Lookups become more expensive (cache issues)
- Avoid larger cycles by frequent distinguished points
- Early-abort walks after a certain number of iterations
- Problem: Large communication cost and storage

Cycle detection

- For 2-cycles: Compare $h(W_i)$ and $h(W_{i+1})$
- Compare points

Dealing with fruitless cycles

- Avoid frequent cycles by choosing large r
- Problem: Lookups become more expensive (cache issues)
- Avoid larger cycles by frequent distinguished points
- Early-abort walks after a certain number of iterations
- Problem: Large communication cost and storage

Cycle detection

- For 2-cycles: Compare $h(W_i)$ and $h(W_{i+1})$
- Compare points

Escape strategies

- Retroactively adjust $h(W_i)$
- Determine unique point in cycle, add “special point” to escape
- Determine unique point in cycle, double this point
- Important: Escape point must be independent of the entrance point

How expensive are fruitless cycles

- In July 2009: Break of ECDLP on 112-bit curve over a prime field by Bos, Kaihara, Kleinjung, Lenstra, and Montgomery
- Computation carried out on a cluster of 214 Sony PlayStation 3 gaming consoles

How expensive are fruitless cycles

- In July 2009: Break of ECDLP on 112-bit curve over a prime field by Bos, Kaihara, Kleinjung, Lenstra, and Montgomery
- Computation carried out on a cluster of 214 Sony PlayStation 3 gaming consoles
- Iteration function did not use the negation map:

  > *"We did not use the common negation map since it requires branching and results in code that runs slower in a SIMD environment"*

How expensive are fruitless cycles

- In July 2009: Break of ECDLP on 112-bit curve over a prime field by Bos, Kaihara, Kleinjung, Lenstra, and Montgomery
- Computation carried out on a cluster of 214 Sony PlayStation 3 gaming consoles
- Iteration function did not use the negation map:

  > *"We did not use the common negation map since it requires branching and results in code that runs slower in a SIMD environment"*

- Paper at ANTS 2010 by Bos, Kleinjung, and Lenstra: Among many ways of dealing with fruitless cycles best speedup is 1.29, but

How expensive are fruitless cycles

- In July 2009: Break of ECDLP on 112-bit curve over a prime field by Bos, Kaihara, Kleinjung, Lenstra, and Montgomery
- Computation carried out on a cluster of 214 Sony PlayStation 3 gaming consoles
- Iteration function did not use the negation map:

  "We did not use the common negation map since it requires branching and results in code that runs slower in a SIMD environment"

- Paper at ANTS 2010 by Bos, Kleinjung, and Lenstra: Among many ways of dealing with fruitless cycles best speedup is 1.29, but

  "If the Pollard rho method is parallelized in SIMD fashion, it is a challenge to achieve any speedup at all. . . . Dealing with cycles entails administrative overhead and branching, which cause a non-negligible slowdown when running multiple walks in SIMD-parallel fashion. . . . [This] is a major obstacle to the negation map in SIMD environments."

What's the problem with SIMD?

- SIMD stands for *single instruction stream, multiple data streams*
- Same sequence of instructions carried out on different data
- Most commonly implemented through vector registers
- Branching means (in the worst case): Sequentially execute both branches

What's the problem with SIMD?

- SIMD stands for *single instruction stream, multiple data streams*
- Same sequence of instructions carried out on different data
- Most commonly implemented through vector registers
- Branching means (in the worst case): Sequentially execute both branches
- Computing power of the Cell processor in the PlayStation 3 is in the Synergistic Processor Elements (SPEs)
- Instruction set of the SPEs is purely SIMD

What's the problem with SIMD?

- SIMD stands for *single instruction stream, multiple data streams*
- Same sequence of instructions carried out on different data
- Most commonly implemented through vector registers
- Branching means (in the worst case): Sequentially execute both branches
- Computing power of the Cell processor in the PlayStation 3 is in the Synergistic Processor Elements (SPEs)
- Instruction set of the SPEs is purely SIMD
- SIMD becomes more and more important on all modern microprocessors

What's the problem with SIMD?

- SIMD stands for *single instruction stream, multiple data streams*
- Same sequence of instructions carried out on different data
- Most commonly implemented through vector registers
- Branching means (in the worst case): Sequentially execute both branches
- Computing power of the Cell processor in the PlayStation 3 is in the Synergistic Processor Elements (SPEs)
- Instruction set of the SPEs is purely SIMD
- SIMD becomes more and more important on all modern microprocessors
- Question: Can we really not get the factor-$\sqrt{2}$ speedup with SIMD?

Our approach

- Solve ECDLP on elliptic curve over $\mathbb{F}_p$
- Begin with simplest type of negating additive walk
- Starting points W_0 are known multiples of Q
- Precomputed table contains r known multiples of P

Our approach

- Solve ECDLP on elliptic curve over $\mathbb{F}_p$
- Begin with simplest type of negating additive walk
- Starting points W_0 are known multiples of Q
- Precomputed table contains r known multiples of P
- Use (relatively) large r (in our implementation: 2048)

Our approach

- Solve ECDLP on elliptic curve over $\mathbb{F}_p$
- Begin with simplest type of negating additive walk
- Starting points W_0 are known multiples of Q
- Precomputed table contains r known multiples of P
- Use (relatively) large r (in our implementation: 2048)
- $|(x, y)|$ is (x, y) if $y \in \{0, 2, 4, \ldots, p-1\}$, $(x, -y)$ otherwise

Our approach

- Solve ECDLP on elliptic curve over $\mathbb{F}_p$
- Begin with simplest type of negating additive walk
- Starting points W_0 are known multiples of Q
- Precomputed table contains r known multiples of P
- Use (relatively) large r (in our implementation: 2048)
- $|(x, y)|$ is (x, y) if $y \in \{0, 2, 4, \ldots, p-1\}$, $(x, -y)$ otherwise
- *Occasionally* check for 2-cycles:
  - If $W_{i-1} = W_{i-3}$, set $W_i = |2 \cdot \min\{W_{i-1}, W_{i-2}\}|$
  - Otherwise set $W_i = W_{i-1}$

Our approach

- Solve ECDLP on elliptic curve over $\mathbb{F}_p$
- Begin with simplest type of negating additive walk
- Starting points W_0 are known multiples of Q
- Precomputed table contains r known multiples of P
- Use (relatively) large r (in our implementation: 2048)
- $|(x, y)|$ is (x, y) if $y \in \{0, 2, 4, \ldots, p-1\}$, $(x, -y)$ otherwise
- *Occasionally* check for 2-cycles:
  - If $W_{i-1} = W_{i-3}$, set $W_i = |2 \cdot \min\{W_{i-1}, W_{i-2}\}|$
  - Otherwise set $W_i = W_{i-1}$
- With even lower frequency check for 4-cycles, 6-cycles etc.
- Implementation actually checks for 12-cycles (with very low frequency)

Eliminating branches

- Compute $|(x,y)|$ as $(x, y+\epsilon(p-2y))$, with $\epsilon = y \mod 2$

Eliminating branches

- Compute $|(x, y)|$ as $(x, y + \epsilon(p - 2y))$, with $\epsilon = y \mod 2$
- Amortize $\min$ computations across relevant iterations, update $\min$ while computing iterations

Eliminating branches

- Compute $|(x, y)|$ as $(x, y + \epsilon(p - 2y))$, with $\epsilon = y \mod 2$
- Amortize $\min$ computations across relevant iterations, update $\min$ while computing iterations
- Always compute doublings, even if they are not used
- Select W_i from W_{i-1} and $2W_{\min}$ without branch
- Selection bit is output of branchfree comparison between W_{i-1} and W_{i-1-c} when detecting c-cycles

Eliminating branches

- Compute $|(x, y)|$ as $(x, y + \epsilon(p - 2y))$, with $\epsilon = y \mod 2$
- Amortize $\min$ computations across relevant iterations, update $\min$ while computing iterations
- Always compute doublings, even if they are not used
- Select W_i from W_{i-1} and $2W_{\min}$ without branch
- Selection bit is output of branchfree comparison between W_{i-1} and W_{i-1-c} when detecting c-cycles
- All selections, subtractions, additions and comparisons are linear-time
- Asymptotically negligible compared to finite-field multiplications in EC arithmetic

Optimization and analysis

- Checking for fruitless cycles every w iterations
- Probability for fruitless cycle: $w/2r$
- Average wasted iterations if fruitless cycle occured: $w/2$

Optimization and analysis

- Checking for fruitless cycles every w iterations
- Probability for fruitless cycle: $w/2r$
- Average wasted iterations if fruitless cycle occured: $w/2$
- Checking without finding a fruitless cycle wastes one iteration

Optimization and analysis

- Checking for fruitless cycles every w iterations
- Probability for fruitless cycle: $w/2r$
- Average wasted iterations if fruitless cycle occured: $w/2$
- Checking without finding a fruitless cycle wastes one iteration
- Overall loss: $1 + w^2/4r$ per w iterations

Optimization and analysis

- Checking for fruitless cycles every w iterations
- Probability for fruitless cycle: $w/2r$
- Average wasted iterations if fruitless cycle occured: $w/2$
- Checking without finding a fruitless cycle wastes one iteration
- Overall loss: $1 + w^2/4r$ per w iterations
- Minimize $1/w + w/4r$: Take $w \approx 2\sqrt{r}$

Optimization and analysis

- Checking for fruitless cycles every w iterations
- Probability for fruitless cycle: $w/2r$
- Average wasted iterations if fruitless cycle occured: $w/2$
- Checking without finding a fruitless cycle wastes one iteration
- Overall loss: $1 + w^2/4r$ per w iterations
- Minimize $1/w + w/4r$: Take $w \approx 2\sqrt{r}$
- Slowdown from fruitless cycles by a factor of $1 + \Theta(1/\sqrt{r})$

Optimization and analysis

- Checking for fruitless cycles every w iterations
- Probability for fruitless cycle: $w/2r$
- Average wasted iterations if fruitless cycle occured: $w/2$
- Checking without finding a fruitless cycle wastes one iteration
- Overall loss: $1 + w^2/4r$ per w iterations
- Minimize $1/w + w/4r$: Take $w \approx 2\sqrt{r}$
- Slowdown from fruitless cycles by a factor of $1 + \Theta(1/\sqrt{r})$
- Negligible if $r \to \infty$ as $p \to \infty$

Solving the 112-bit ECDLP faster

- Software by Bos et al. takes expected 65.16 PS3 years to solve DLP
- Our software takes expected 35.6 PS3 years for the same DLP

Solving the 112-bit ECDLP faster

- Software by Bos et al. takes expected 65.16 PS3 years to solve DLP
- Our software takes expected 35.6 PS3 years for the same DLP
- (very-close-to) factor-$\sqrt{2}$ speedup through negation map

Solving the 112-bit ECDLP faster

- Software by Bos et al. takes expected 65.16 PS3 years to solve DLP
- Our software takes expected 35.6 PS3 years for the same DLP
- (very-close-to) factor-$\sqrt{2}$ speedup through negation map
- Faster iterations
  - Faster arithmetic in $\mathbb{Z}/(2^{128}-3)\mathbb{Z}$ (prime field has order $(2^{128}-3)/76439$)
  - Non-standard radix $2^{12.8}$ to represent elements of $(2^{128}-3)/76439$
  - Careful design of iteration function, arithmetic, and handling of fruitless cycles

Solving the 112-bit ECDLP faster

- Software by Bos et al. takes expected 65.16 PS3 years to solve DLP
- Our software takes expected 35.6 PS3 years for the same DLP
- (very-close-to) factor-$\sqrt{2}$ speedup through negation map
- Faster iterations
  - Faster arithmetic in $\mathbb{Z}/(2^{128}-3)\mathbb{Z}$ (prime field has order $(2^{128}-3)/76439$)
  - Non-standard radix $2^{12.8}$ to represent elements of $(2^{128}-3)/76439$
  - Careful design of iteration function, arithmetic, and handling of fruitless cycles
- Negligible overhead (in practice!) from fruitless cycles

Solving smaller DLPs

- We have a faster implementation to solve the DLP
- But we don't have a cluster of > 200 PlayStations
- How can we demonstrate that the implementation indeed works?

Solving smaller DLPs

- We have a faster implementation to solve the DLP
- But we don't have a cluster of > 200 PlayStations
- How can we demonstrate that the implementation indeed works?
- Implementation solves ECDLPs on elliptic curves
  $E : y^2 = x^3 - 3x + b$
- Repeatedly solve DLP on curves with smaller subgroups (choose different b), specifically:

Solving smaller DLPs

- We have a faster implementation to solve the DLP
- But we don't have a cluster of > 200 PlayStations
- How can we demonstrate that the implementation indeed works?
- Implementation solves ECDLPs on elliptic curves $E : y^2 = x^3 - 3x + b$
- Repeatedly solve DLP on curves with smaller subgroups (choose different b), specifically:
  - 32237 experiments in a subgroup of order $\approx 2^{50}$
  - 257241 experiments in a subgroup of order $\approx 2^{55}$
  - 33791 experiments in a subgroup of order $\approx 2^{60}$
- Rate of DPs per hour matches expectations
- Median number of DPs required to solve DLP matches expectations

Solving smaller DLPs

- We have a faster implementation to solve the DLP
- But we don't have a cluster of > 200 PlayStations
- How can we demonstrate that the implementation indeed works?
- Implementation solves ECDLPs on elliptic curves $E : y^2 = x^3 - 3x + b$
- Repeatedly solve DLP on curves with smaller subgroups (choose different b), specifically:
  - 32237 experiments in a subgroup of order $\approx 2^{50}$
  - 257241 experiments in a subgroup of order $\approx 2^{55}$
  - 33791 experiments in a subgroup of order $\approx 2^{60}$
- Rate of DPs per hour matches expectations
- Median number of DPs required to solve DLP matches expectations
- Confident performance extrapolation to 112-bit DLP

Left-out details

- Paper has way more details on the implementation

Left-out details

- Paper has way more details on the implementation
- Hand-optimized assembly implementation (not online yet)

Left-out details

- Paper has way more details on the implementation
- Hand-optimized assembly implementation (not online yet)
- Various tricks in the design of the iteration function

Left-out details

- Paper has way more details on the implementation
- Hand-optimized assembly implementation (not online yet)
- Various tricks in the design of the iteration function
- Entertaining history on "How not to use negation in Pollard's rho method"

Left-out details

- Paper has way more details on the implementation
- Hand-optimized assembly implementation (not online yet)
- Various tricks in the design of the iteration function
- Entertaining history on "How not to use negation in Pollard's rho method"
- Paper is online, e.g. at http://cryptojedi.org/papers/#negation