ECC mod `8^91+5`

Daniel R. L. Brown[*]

January 31, 2018

Abstract

The field size $8^{91}+5$ for elliptic curve cryptography offers simplicity, security, and efficiency.

1 Introduction

This report describes parameters for elliptic curve cryptography (ECC). Specifically, it focuses on the field size, the prime $8^{91}+5$. This field size has several advantages, and a few disadvantages. The main advantage is weak and theoretical protection against exhaustive weakening.

The appendices of the report include an example curve defined over this field size.

1.1 Background

Theoretically, a cryptographic algorithm's parameters are possibly the output of an exhaustive search for secretly weak values of the parameters, rendering the parameters *exhaustively weakened* by their author. Bernstein and several of his co-authors [BCC+14] list some ways this might occur in the elliptic curve cryptography (ECC).

In elliptic curve cryptography, several standards—NIST, Brainpool, and CFRG—partially counter the theoretical threat of exhaustively weakened parameters, including the parameter of field size.

Ideally, a countermeasure to this theoretical threat does not sacrifice efficiency (or other security properties). None of the currently standardized field sizes for ECC are ideal.

[*]Certicom/BlackBerry. `dbrown@certicom.com`

1.1.1 A general countermeasure

A general countermeasure to the concern of parameters exhaustively weakened is to choose *compact* values for the parameters. In other words, highly-compressible, or low-complexity, or simple-as-possible, parameters partially resist exhaustive weakening.

The heuristic reasoning of this countermeasure is that exhaustive weakening means traversing a large space of candidate parameters. By Shannon's theory of information a large space of values requires a large set of information to be encoded. So, exhaustively weakened parameters must be non-compact in that they must be encoded with an amount of information sufficient to encode the entire search space of the hypothetical attack.

Historical precedents for compactness Choosing compact parameters is quite an old concept, with plenty of historical precedence.

In cryptography, compact parameters have often been called called *nothing-up-my-sleeve* (NUMS) parameters.

Often, NUMS parameters are also intended to be random, and as such are selected using some kind of simple pseudorandom function applied to a compact value. With regard to field size, choosing a pseudorandom field has a large cost in software performance. Since this decreases the usability for a given security level, the performance lost can also be viewed as a security loss.

More recently, in ECC, the term *rigid* has been used (introduced by Bernstein and Lange [BL13]). This new term is used in contexts devoid of pseudorandomness, which, among other things, helps distinguish it from the usual pseudorandom NUMS approach.

In more general context, such as science and philosophy, the strategy of compact parameters is an instance of the principle often called *Occam's razor*. This principle is to do nothing more complicated than necessary. In other words, choose the simplest suitable option.

Choosing compact parameters is an effort to minimize the amount of information used to encode parameters, while still remaining secure against known attacks.

Parameter compactness provides no protection against parameter weaknesses that are positively correlated with the compactness of the parameters. For an example in ECC, smaller field sizes are generally weaker and generally more compact (at least when represented in plain decimal). As we shall see, there are reasonable systems of representing (compressing) primes in which compactness potentially helps in finding more secure and more efficient primes.

1.1.2 The ECC field size parameter

A fundamental parameter in elliptic curve cryptography is the field size. As we will be seen in later in the report restricting compact values of this parameter, leads one to some secure and efficient choices of the parameters. Nonetheless, it also is reasonable to ask about the converse: is there a need to seek to compact parameters in their own right? In other words, is the field size, as a parameter, susceptible to exhaustive weakening?

Some weak evidence of the need for possibility of exhaustively weakened ECC field sizes are some the NIST prime field sizes. For example, Bernstein seems to suggest [Ber14] that some NIST field sizes, such as NIST P-256, are more difficult to implement securely than other similar prime field sizes (Curve25519): "What is a few copies? . . . Variable time. . . . Trouble . . . Even worse." It has also been argued that the NIST field sizes are slower than similar field sizes (such as Curve25519) without offering a substantial increase in security: thereby indirectly hindering the availability of security to efficiency-constrained users.

Regarding the affected NIST prime fields, it is much less clear that these minor security defects could have been a result of an exhaustive search. For evidence of exhaustive weakening, one should exhibit a large space of candidate parameters, and a search criterion. It is not quite clear how to do this for NIST prime fields, because they can be characterized as a belonging to a small class of generalized Mersenne primes. In the unlikely case that the weaknesses of these NIST prime were intentional, rather than accidental, the attackers were more likely to have honed into the weakness rather exhaustively search for it. Against such direct attacks, compactness of parameters are mostly ineffective: only astute insight in identifying the weakness is effective.

Disregarding the NIST curve examples, perhaps stronger evidence for susceptibility of the field size in ECC to exhaustive weakening can be contrived as follows. Again, the ideal form of evidence is a large set of candidate primes, of which a small portion are vulnerable to a security flaw, and further that the vulnerable primes are best found by exhaustive search. Hypothetically, consider a security flaw in which just a few field values are likely to induce an hardware word overflow in a typical implementation, similarly to the Izu–Takagi attack [IT03]. These faulty field values, being so few and unsuspected, would not be detected during normal operation or by random testing. The adversary, though, supply these faulty value induced faults in the target implementation, and then potentially learn some secret bits.

1.2 A fast and compact field size for ECC

The prime field size $8^{91}+5$ can be represented using decimal integers and basic number-theoretic operations (including an exponential operation), as:

$$\texttt{8\^{}91+5} \tag{1}$$

meaning it has a *decimal-exponential complexity* (DEC) of 6 symbols (or less, if it has a shorter expression).

An earlier version of this notion was discussed by Bernstein, Hamburg, Krasnova and Lange [BHKL13] when they introduced the curve Curve1174 and pointed out that “it is even more concisely expressible than the existing Curve25519 curve”, seeming to suggest that concision is a good thing.

This prime is very close to a power of two: differing just by five. Primes close to a power of two, are generally known to yield efficient modular arithmetic, mainly because computer hardware deals most efficiently with powers of two.

The prime $8^{91}+5$ is larger than 2^{256}. Field size of about 2^{256} is currently considered the approximate minimum threshold of adequate security against Pollard rho attacks, at least under simplified arguments. A field size meeting this minimum can be used to establish a 128-bit symmetric, without undermining the security of 128-bit symmetric encryption scheme. A field size meeting this minimum is believed to withstand all currently-implemented attack algorithms under all future foreseeable computing power[1].

2 Simplified and sophisticated comparisons

Keep it simple. Simplify if possible. As simple as possible. No unnecessary complications.

All these sayings suggest a strategy for cryptographic algorithm selection. Choose the simplest parameter among the suitable parameters. Any simpler parameter must be rejected for some simple reason.

2.1 Extract simplified rules in general

Unfortunately, cryptology is a sophisticated subject. Claims about the superiority (in efficiency or security) of one cryptographic algorithms can require sophisticated arguments. Sophisticated arguments can be.

[1]Excluding quantum computing power, if you regard that as foreseeable.

- difficult for non-experts to verify,
- dependent on variable inputs (such as computer hardware characteristics),
- susceptible to sabotage by subtle subterfuge.

Fortunately, in some cases, even sophisticated arguments in cryptography have parts that are simple. If these simpler principles play a significant role, then they should be isolated and extracted for clarity.

Simplified arguments are often only a rule of thumb, providing only an approximate answer. They can be applied as a first step, to be refined with more sophisticated arguments. Sometimes, simplified arguments give good answers, but cause one to miss some other good answers.

More sophisticated arguments might overturn simplified arguments. A simplified argument is strong if such overturns are the exception. In that case the simplified argument can be called a *heuristic*. Sometimes the underlying justification for a heuristic is largely coincidence, accident, or serendipity.

Of course, it can be argued that any simplifications identified, including the ones below, are actually a sophistication in their own right. There may be many sophisticated way to simplify a sophisticated argument.

This report searches for the simplest primes of a minimum size, and preliminarily rejects some of them a heuristic efficiency analysis. The search itself is slightly sophisticated, but does not truly undermine simplicity of the resulting primes. Their simplicity is established in two ways:

- by an intrinsic, self-evident compactness, requiring only a few basic symbols to represent,
- by the absence of proposal with similar properties.

The searching discussed in the report is an approach to verify the latter absence.

2.2 Simplified rules for ECC

Elliptic curve cryptography is especially sophisticated. We try to limit the sophistication by identifying some simplified rules.

2.2.1 Simplified number-theoretical rules

A basic understanding ECC seems to require some knowledge of finite fields, which in turn, requires knowledge of prime numbers. In other words, this involves a small amount of understanding of number theory, but for simplicity, we try to keep this to minimum.

The number theory involved in ECC also involves modular arithmetic, and, usually, computing powers of numbers (exponentiation). So, we include modular arithmetic and powers in a minimal set of sophistications. Most numerate people, and even number theorists, have primarily learned to use decimal notation for arithmetic with small integers.

One can easily argue that binary notation, or another base, is more natural, but we resort to these bases only as needed, viewing it as an extra sophistication, being something taught outside of primary school.

So, for this reason, we first seek out prime field sizes that can be expressed using the fewest symbols in a basic number-theoretical language of decimal-exponential expressions.

For example, the prime field size $8^{91}+5$ can be expressed in 6 symbols as `8^91+5`. Of course, many other primes can be expressed 6 or fewer symbols as decimal exponential expressions. More sophisticated arguments may be needed to decide between these many choices.

2.2.2 Simplified efficiency considerations

Efficient implementation of ECC software requires somewhat sophisticated understanding of computer hardware. This may vary with the type of hardware, but it is good to extract a simplified argument. If possible, we should try to simplify these considerations, and distill some simple common principles to efficient implementation, to be applied as a rule of thumb.

A simplified argument in efficient implementation is that computer hardware uses binary integer arithmetic, and therefore arithmetic involving powers of two is generally arithmetic than powers of odd integers. A slightly more sophisticated variant of this argument is that much computer hardware offers efficient bit-shift operations, which provide multiplication by powers of two, and bit-wise and operations, which provide modular reduction by a power of two.

This simplified argument can sometimes be applied at a glance to eliminate prefer the prime $8^{91}+5$ over the prime $7^{98}+2$. A more sophisticated analysis might reverse this preliminary conclusion.

2.2.3 Simplified minimum and maximum field sizes

A too small field size will be insecure; a too large field will be too inefficient. Therefore, we should seek simplified rule (and argument) for a minimum and maximum field size.

Under the most basic understanding of ECC, a minimum field size is a security condition, needed to avoid making Pollard rho attack feasible. Accordingly, a simplified strategy places a priority on the rule for a minimum field size, letting the maximum field size have a secondary importance.

Minimum field size We first consider some simplified rules for minimum field sizes, to be followed by some simplified justifications of these rules.

- The prime field size is at least 2^{256}.
- The prime field size is at least approximately 2^{256}.
- The prime field size is at least approximately 2^{224}.
- The prime field size is at least 2^{80}.

The second rule is a little vague, but one can hope that the other simplified rules are sufficient to resolve the resulting ambiguity. The second rule allows for curves like Curve25519.

The simplified justifications for these simplified minimum field size rules. These justifications are given simplest first, and roughly the support the corresponding simplified rules above.

- ECC is often used to establish a 128-bit symmetric key, which takes 2^{128} steps to attack by exhaustive search. Pollard rho attacks against ECC is approximated most simply as taking square root of the field size steps. Using an ECC field size smaller 2^{256} would then take fewer than 2^{128} steps to find the symmetric key, decreasing its effective security.
- The size of the ECC steps is usually larger than the size of the steps used to attack the symmetric key. So, ECC can actually tolerate field sizes smaller than 2^{256} without undermining the effective security of the symmetric key. (Furthermore, ECC has a more secure than symmetric-key trade-off between an adversary's success rate and amount of computation.)
- The actual number of steps Pollard rho depends is larger than just a simple square root, by a constant factor.
- Matching the symmetric key sizes, to avoid decreasing security, is irrelevant, if the total amount of adversary computation to attack the ECC remains infeasible. Some sort of computational estimate of how

large a field an ECC adversary could feasibly attack may be used to establish a minimum field rather than matching the attack cost against the symmetric cipher (which is also infeasible, one hopes).

Maximum field size A maximum field size is an efficiency condition. In this report, we take a simplified rule of preferring a more efficient field given a choice between two sufficient secure field sizes, but not imposing any maximum field size. This simplified maximum field size rule has the obvious effect that we will never reject a curve under maximum field size rule.

More sophisticated rules might impose a maximum field size, based on some estimate of minimum efficiency, or perhaps some other reason such as minimum data usage efficiency, or even just as an arbitrary way to converge towards interoperability and ultimately deployment.

For example, the SEC1 standard required primes p at the 128-bit security level to belong to the interval $[2^{255}, 2^{256}]$, perhaps largely under its stated goal of promoting interoperability. In hindsight, this interval might be too narrow, and too arbitrary.

2.3 Formalizing complexity

A task here is to formally define the set of decimal (or binary, octal or hexadecimal) exponential expressions. This task can be broken into two sub-tasks:

- defining the syntax of expressions (what strings qualify as a decimal exponential expression), and
- defining the semantics of the expressions (how to evaluate the strings as integers).

Given the syntax of expressions, one can count the number of expressions of a given length. Given the semantics, one can further refine this information to study the the distribution of the integers of a given length, and using some heuristic to estimate how many ought to be prime, and so on, and then, finally, enumerating the primes of a given complexity.

2.3.1 Syntax

We treat the decimal case. For other bases, the analysis is similar.

A decimal exponential expression is a string consisting of the alphabet of 17 characters,

```
1 2 3 4 5 6 7 8 9 0 ( ) + - * ^ /
```
(2)

and obeying certain syntactical rules defined below. The rules for a valid expression are recursively defined.

The characters fall into three classes. The first 10 are digits, with 0 having a special syntactical role. The next two characters are parentheses, which must be nested and matched as usual and as implied by the following rules. The last five characters are operators.

A numeric expression is consists of digits only, with the first digit nonzero. Numeric expressions are included among valid expression.

A closed expression is either a numeric expression or a valid expression that is enclosed in parenthesized.

A string is valid expression if and only if it is the concatenation of an odd number of sub-strings where the odd-order sub-strings (first, third, fifth, ...) are closed expressions and the even-order sub-strings (second, fourth, ...) are just operators, a single-character each.

One can use generating series to count numeric, closed, and valid expressions of a given length:

$$N(x) = 9x + 90x^2 + 900x^3 + \dots \tag{3}$$

$$C(x) = N(x) + x^2V(x) \tag{4}$$

$$V(x) = C(x)(1 - 5xC(x))^{-1} \tag{5}$$

Using this system of equations to calculate lower-degree terms seem to yield:

$$V(x) = 9x + 90x^2 + 1314x^3 + 17190x^4 + 231849x^5 + 3100140x^6 + \dots \tag{6}$$

with possibly 41572305 (over forty million) valid DEC-7 expressions.

Taking the base-2 logarithm of the coefficients of the series V provides a calibrate idea of how much Shannon entropy is encoded into a decimal expression. This may also allow calibration between decimal exponential complexity and the complexity to other bases.

Typical mathematical notation allows one to optionally omit the operator for multiplication when clear from context. Such omission slightly complicates the syntax, and increase the number expressions of a given length.

An even richer class of expressions (and numbers) can be obtained by allowing the right unary operator `!` to mean a factorial. One might also consider other standard math functions.

To aid in our analysis it is help to have a notion of the shape of a decimal exponential expression. The shape is obtained by replacing each digit by `#` and each operator by `?`. Proofs about the decimal exponential can then be divide into cases by shape.

Table 1 lists all 13 possible shapes of DEC-6 numbers. In the second column of 8 shapes uses parentheses trivially, which will evaluate to DEC-4 of DEC-2 numbers. The top five elements of the first column use one or less operations, which will likely give a composite or a too small number.

`######`	`(####)`
`####?#`	`(##)?#`
`###?##`	`(#)?##`
`##?###`	`##?(#)`
`#?####`	`#?(##)`
`##?#?#`	`(##?#)`
`#?##?#`	`(#?##)`
`#?#?##`	`((##))`

Table 1: The sixteen shapes of DEC-6 numbers

Using the generating series approach above suggests that the number of shapes of DEC-1 to DEC-7 number are 1, 1, 3, 4, 10, 16, 37.

2.3.2 Semantics

The evaluation of some valid expressions (as defined above) is quite unambiguous, while for others it is a little unclear. To fully formalize the system, one must resolve this lack of clarity. Ideally, one can either dictate a unique way to evaluate each expression into an integer (or a rational number, or a complex number), or else concede the expressions have multiple interpretations. Here are some issues:

- How to decide an order of evaluation between two or more top-level operations:
  - Which operators have precedence over others?
  - How to similar adjacent operators associate?
- How to handle division?
  - Require exact division (with errors returned otherwise)?
    - Allow fractions to appear temporarily?
    - Allow irrational and imaginary numbers to appear temporarily?
  - Allow rounded division?

    * Towards zero?
    * Towards negative infinity?

For example, Table 2 lists some rules for evaluating an alternating sequence of operands and operators. Consecutive ^ operators are to be evaluated first, from right to left. Replace the resulting sub-expressions by the values (now quite large, generally), resulting in a partially evaluated expression with any ^ operators. Next look for group of consecutive operators consisting of the operators * and / only. In each sub-expression evaluate the operators from left to right.

Operators	Associativity
^	right
* /	left
+ -	left

Table 2: Order of precedence and associativity

Many of the semantic questions above only apply to longer expressions, so it is quite possible to reason about DEC-6 primes without answering all of the questions exactly.

For simplicity, we will assume that the / operator requires exact division, and inexact division is invalidates the expression. Similarly, a negative result in the - also invalidates the expression (which helps to avoid raises an integers to an negative power).

For example, regardless of these semantic detail, a heuristic analysis, suggests that only the operator ^ helps to make expressions much larger than numeric expression of similar length. Consequently, although they are very many valid expressions, many will be too small to be use for elliptic curve cryptography. Only those with at least one occurrence of the operator ^ will be interesting. For primes, one also needs a + or -, since products of perfect powers are not generally prime. (Ignoring rounded division, here.)

Note that straight line programs (SLP) are another way to measure integer complexity, and do not use exponentiation. Instead, a SLP can record intermediate variables and multiply them without re-copying them. In this way, an SLP can be also shorter than a numeric expression.

At least for short expressions, such as six or seven symbols, these considerations are very limiting on the expressions. For longer expressions, which are most relevant in the binary setting, useful expression can be more complicated.

Once one develops an idea of how many expressions yield integers in the target range, one can next try to estimate how many of these are prime. Then one can evaluate the expressions and test them for primality.

The number of primes in the target range gives a somewhat normalized measure of uniqueness, that is somewhat less arbitrary than the number of symbols. (The number of symbols varies between the base.)

Lemma 1. *All DEC-1 to DEC-6 primes of size at least* 2^{80} *have a decimal exponential expression of shape* `#^##?#`*.*

Proof. First we can eliminate parentheses from the shape. Parentheses can be removed if they surround no operators, meaning a numeric expression (digits only). Otherwise parentheses surround at least one operator, which has at least two operands. So, a parenthetic sub-expression requires at least five symbols (the two parentheses, the operator and the two operands). For DEC-6 or less expressions, this leaves only one more symbol, which violates the syntax, since one symbol is not enough to hold an operator and operator. (If we allow elided multiplication operators, then we see the value must be composite.)

Expressions without parenthesis have a shape consisting of `#` and `?` symbols with no two of the latter adjacent. One can easily check that the number of such shapes of a given length is a Fibonacci number.

The largest DEC-6 or less expression with no operators is `999999`, which is less than $2^{20} < 2^{80}$.

An expression with only one operator cannot be prime if the operator is `^` or `*` unless one of the operands is one and the other a prime with an expression of shorter length. If the single operator is `+` or `-` or `/`, then it is not too hard to verify that the largest DEC-6 expression if `9999+9`, which is less than $2^{14} < 2^{80}$.

The only remaining expression have two operators, because three operators require four operands and hence seven or more symbols.

By the earlier reasoning, we do not expect both operators to be in the set of `*` and `^`, because that would result in a composite number, or equal prime expression of shorter length.

If neither of the operators is `^` then the largest possible expression seems to be `99*9*9`, which is too small again.

So one of the operators is `^` and the other one of `+` or `-` or `/`.

If the second operator is `^` the expression will be too small, or the first operator will be `+` and the expression can be re-arranged by commutativity.

If the remaining expression have form `##^#?#`, the largest possible expression seems to `99^9+9` which is less than $128^9 + 128^9 = 2^{64} < 2^{80}$.

The leaves only expression of the form `#^##?#`. □

Continuing the proof slightly: if we still insist on exact division, then the second operator must be `+` or `-`, otherwise the result would be composite. If we allow division with rounding, then the primes that result (if any) have no known efficiency advantages, so should be preliminarily rejected.

2.3.3 DEC-5 field sizes

The largest DEC-5 (or less) prime that I could find was $6^9 - 7$, which is too small (as expected by the previous lemma).

Some suitably large composite finite field sizes are DEC-5 or less numbers. For example, 2^{283}. But these all seem to be low-characteristic finite fields, which is both a security risk, due to improved index calculus attacks, and an efficiency defect, due hardware support of integer multiplication.

2.4 Some DEC 6 primes

Because no suitable DEC5 or less primes were found, our simplified rules indicate to next look for suitable DEC6 primes.

The following six DEC-6 primes are at least approximately 2^{256}, which is our second simplified rule for minimum field size.

$$6^{98} - 7 \quad (\approx 2^{253.3}) \tag{7}$$
$$8^{91} + 5 \quad (\approx 2^{273.0}) \tag{8}$$
$$7^{98} - 2 \quad (\approx 2^{275.1}) \tag{9}$$
$$9^{87} + 4 \quad (\approx 2^{275.8}) \tag{10}$$
$$8^{95} - 9 \quad (\approx 2^{285.0}) \tag{11}$$
$$9^{99} + 4 \quad (\approx 2^{313.8}) \tag{12}$$

These are all probable primes. As expected, these all have the shape `#^##?#`. I did not find other sufficiently large primes, though I did not do a full exhaustive search, or a proof.

Note that (non-thorough) searches for DEC-6 numbers allowing rounded division did not yield primes in this range.

2.4.1 Rejecting four of the six field sizes

The two of the six primes with base $b = 8$ are probably most efficient, under our simplified efficiency rule of being close to a power of two. The other four primes are quite far from a power of two.

Perhaps sophisticated arguments can salvage the other primes, but preliminarily they should be rejected, at least until such sophisticated arguments arise.

A theoretical argument to focus on base 4 and 8 More generally, when considering primes of the form $b^m + d$, for small b, d, m, one does not expect the prime to very close to a power of two unless, b is also a power of two, due to the famous abc conjecture in number theory. The conjecture generalized Fermat's last theorem, and predicts, among many other things, that perfect powers cannot be too close.

Assuming this conjecture, it would be extremely surprising to find $b^m + d = 2^n + t$ for small numbers b, m, d, n, t. Of course, resorting to this conjecture is highly sophisticated, so it cannot form part of the simplicity argument. Rather, it is merely an aid to those searching for simple primes close to a power of two.

2.4.2 Ways to verify the search results

The user may want to independently verify the claim that $8^{91}+5$ and $8^{95}-9$ are the only DEC-6 primes larger than 2^{256} of the efficient form $8^m + d$. In particular, the user wants to be sure that no primes have been falsely excluded, and further test these two numbers are actually prime.

Fortunately, tests for compositeness, with proofs, and probabilistic primality tests are quite easy. For example, the J programming language snippet

```
(#~1 p:({:+8x^{.)"1)>,{(85+i.15);i:9
```
(13)

will report only these two probable primes among $8^m + d$ with $85 \leq m \leq 99$ and $-9 \leq d \leq 9$. A user might also try to use NTL or Sage, or some other software that has primality testing.

Formulaic composites For the excluded composites, a user can also try to identify small or structured prime factors, just to be extra sure. Clearly two divides $8^m + d$ for even d. One can see immediately that $8^m - 1$ is divisible by 7 and $8^m + 1 = (2^m + 1)(4^m - 2^m + 1)$ is divisible by $2^m + 1$. Therefore, when seeking primes, one can restrict to $d = \pm 3, \pm 5, \pm 7, \pm 9$, leaving eight possible values for d. Taking $85 \leq m \leq 99$, since $8^{85} = 2^{255}$, leaves fifteen possible values for m. The leaves $8 \times 15 = 120$ candidates $8^m + d$.

Heuristic expectations The famous prime number theorem, suggests a heuristic that the expected number of primes out of these 120 candidates is about $120/\log(2^{256}) \approx 0.7$. The fact that two probable primes were found is slightly more than this heuristic prediction, but is within the range of plausibility.

Small integer factors Using the first small ten small primes (up to 29), eliminates 69 of these numbers. This would be a feasible, but tedious, hand-verifiable calculation. If one wants to find small factors without using a big integer software, one can eliminate quite a few more numbers. For example, 105 candidates have a factor less than 10^5 and 109 of the candidates have a factor less than 10^7.

Composite witnesses It seems that for all the candidates $n = 8^m + d$, other than the two probable primes, $8^{91} + 5$ and $8^{95} - 9$, it holds that $2^{n-1} \neq 1 \bmod n$, proving that each such n is composite. This calculation requires 120 modular exponentations modulo 300-bit or less integers.

2.4.3 Tiebreakers between the best DEC-6 primes

Our the simplified rules have now reduced the decision down to just field sizes for ECC. Unfortunately, our simplified rules do not provide a way to choose between. Consequently, we must extend our rules, becoming a little more sophisticated, but still trying to be as simple as possible.

Complexity in other bases In binary and octal, instead of decimal, the complexity of $8^{91} - 9$ seems to be one greater than that of $8^{91} + 5$, mainly due to 9 requiring more symbols than 5 (in binary and octal).

This argument is sophisticated in that it invokes other bases, and like decimal exponential complexity is still only heuristic. It seems unfair to resolve a heuristic with another heuristic. Rather, heuristic seem better as a simplified way to resolve gaps in the simplified arguments.

Field speed Both field's elements can be represented in five 64-bit integers. This gives a simplified argument suggests that the fields should have similar speeds in software.

If one follows the conventional practices of using a curve of size close to the field size, with a low co-factor, then using field $8^{91} + 5$ should be slightly faster, because the scalar multipliers will be smaller integers.

If we accept that principle that a minimum field size of 2^{256}, is adequate for security, then the better speed of $8^{91} + 5$ makes it preferable.

The prime $8^{95} - 9$ is just below a power of two, meaning that Fermat inversion is not as optimal as for $8^{91} + 5$, which is just above a power of a prime. Perhaps $8^{95} - 9$ does have a Fermat inversion algorithm nearly as efficient as $8^{91} + 5$, but I expect it to be somewhat more complicated.

Pollard rho The prime $8^{95} - 9$ is larger than $8^{91} + 5$, so potentially provides a greater security against Pollard rho.

However, if one wants a more secure Pollard rho as a security precaution, then one might also wish to re-consider one's other security precautions. For example, one might put a greater priority on having a larger field size, so might extend one's search to DEC-7 primes.

Implement-ability In a five-limb implementation, $8^{95} - 9$ requires larger limb values, so it might need more work to avoid overflow of the 64-bit words. In other words, it might suffer from greater overflow pressure. Future implementations should resolve this issue more clearly.

2.4.4 Smaller DEC-6 primes

Upon decreasing the minimum field size, the number of available DEC-6 primes increases. Seven remaining DEC-6 primes with base 4 or 8 which are at least 2^{160} are:

$$4^{80} + 7, \qquad 4^{83} - 5, \qquad 4^{87} - 3, \qquad 4^{87} + 7 \tag{14}$$

$$8^{68} + 7, \qquad 8^{71} + 3, \qquad 8^{81} - 9. \tag{15}$$

Only the last of these seven primes, $8^{81} - 9$, also exceeds approximately 2^{224}, thereby complying with our third simplified rule.

The prime $8^{81} - 9$ could likely be implement in four limbs of 64-bit integers, so it is not unreasonable to hope it can be faster than $8^{91} + 5$, which requires five limbs. On the other hand, its limb sizes may be around 2^{61}, which is closer to overflowing. The extra reduction steps needed to prevent overflow might outweigh the gain from using fewer limbs.

2.5 Select DEC-7 primes

If DEC-6 primes are unsuitable, perhaps because they are: too small to resist Pollard rho to sufficiently degree of precaution, or else too inefficient (compared to alternatives of similar size), or else too difficult to implement correctly, then, under our simplified argument for choosing field sizes, we should next look for a DEC-7 prime.

As one might expect, DEC-7 primes are considerably more plentiful than DEC-6 primes. Using DEC-7 primes gives one far more options. It seems that decimal exponential complexity exhausts its utility as a simplified heuristic for ECC field sizes once we get to DEC-7 primes.

2.5.1 Alternative bases for exponential complexity

Instead of giving up, one can try to extend the simplified heuristic of exponential complexity by leaving decimal turning to other bases. The natural choices are base 2 (binary), 16 (hexadecimal), or 8 (octal). These bases are those used most often in computer software development. Larger bases such as, 32, 60, 64, 256 and 2^{32} might even be worth looking at.

Some interesting DEC-7 primes have been proposed for ECC: including $2^{521}-1$, $2^{255}-19 = 8^{85}-19$ and $2^{336}-3$. Most of these have some efficiency advantages (relative to their size), and perhaps also security in the sense of implementation-fault resistance.

One can then ask how the alternative-base exponential complexity heuristics fare compared to these more sophisticated recommendations. Perhaps future work can carry out such a comparison.

A few interesting primes are highlighted in Table 3. The expressions in the table are not proven to be minimal. In particular, the binary expression are longest, and seem to have most potential for shortening.

The rather large prime $2^{3^6}-9$ was added to Table 3 because it has quite a low binary exponential complexity (BEC) of 14 symbols, mainly due to the low binary exponential complexity of the exponent 729. I am not aware of low complexity of this exponent contributing to either security or efficiency. So, this prime $2^{729}-9$, may serve to illustrate that the binary exponential complexity is not a good heuristic for ECC.

2.5.2 The "competitive" pseudo-Mersenne primes

Bernstein and his co-authors [BCC+14] wrote:

> For pseudo-Mersenne primes larger than 2^{224} the only possibly competitive ones are: $2^{226}-5$; $2^{228}+3$; $2^{233}-3$; $2^{235}-15$; $2^{243}-9$; $2^{251}-9$; $2^{255}-19$; $2^{263}+9$; $2^{266}-3$; $2^{273}+5$; $2^{285}-9$; $2^{291}-19$; $2^{292}+13$; $2^{295}+9$; $2^{301}+27$; $2^{308}+27$; $2^{310}+15$; $2^{317}+9$; $2^{319}+9$; $2^{320}+27$; $2^{321}-9$; $2^{327}+9$; $2^{328}+15$; $2^{336}-3$; $2^{341}+5$;

Bit-security	Decimal	Binary	Hexadecimal	Octal
243	8^81-9	10^11^101-1001	8^51-9	2^3^5-11
	6	14	6	8
255	8^85-19	10^11111111-10011	2^FF-13	2^377-23
	7	17	7	8
273	8^91+5	1000^1011011+101	8^5B+5	2^421+5
	6	16	6	7
285	8^95-9	1000^1011111+1001	8^5F-9	2^435-11
	6	17	6	8
336	2^336-3	1000^1110000-11	8^70-3	2^520-3
	7	15	6	7
521	2^521-1	10^1000001001-1	2^209-1	2^1011-1
	7	15	7	8
729	2^729-9	10^11^110-1001	8^F3-9	2^3^6-11
	7	14	6	8

Table 3: Some primes with some of their shorter exponential expressions

> $2^{342}+15$; $2^{359}+23$; $2^{369}-25$; $2^{379}-19$; $2^{390}+3$; $2^{395}+29$; $2^{401}-31$; $2^{409}+29$; $2^{414}-17$; $2^{438}+25$; $2^{444}-17$; $2^{452}-3$; $2^{456}+21$; $2^{465}+29$; $2^{468}-17$; $2^{488}-17$; $2^{489}-21$; $2^{492}+21$; $2^{495}-31$; $2^{508}+15$; $2^{521}-1$.

The justification for their list presumably includes some argument more sophisticated than just decimal-exponential complexity. For example, all the primes above are within distance 31 to a power of two, while one expects to many primes of similar decimal-exponential complexity not meeting this condition.

Peering at their list through the lens of decimal exponential complexity, one sees that each prime is specified in decimal exponential notation with an expression of complexity of 7 or 8. However, in each case, the base is fixed to 2, whereas some of the complexity-8 expressions can be re-expressed as complexity 7 by changing the base from 2 to 8, but only if the exponent of two is below 300. Furthermore, some primes in their list can be represent with decimal exponential complexity of 6, which are the three primes discussed earlier in this report: $8^{81}-9$, $8^{91}+5$ and $8^{95}-9$, which are written as $2^{243}-9$, $2^{273}+5$ and $2^{285}-9$ in their list.

The DEC-7 primes in their list appear to be: $2^{226}-5$; $2^{228}+3$; $2^{233}-3$; $2^{251}-9$; $8^{85}-19$; $2^{263}+9$; $2^{266}-3$; $8^{97}-19$; $2^{295}+9$; $2^{317}+9$; $2^{319}+9$; $2^{321}-9$; $2^{327}+9$; $2^{336}-3$; $2^{341}+5$; $2^{390}+3$; $2^{452}-3$; $2^{521}-1$. This list includes the $2^{255} - 19$ and $2^{291} - 19$.

The DEC-8 primes in their list seem to be: $2^{235} - 15$; $2^{292} + 13$; $2^{301} + 27$; $2^{308} + 27$; $2^{310} + 15$; $2^{320} + 27$; $2^{328} + 15$; $2^{342} + 15$; $2^{359} + 23$; $2^{369} - 25$; $2^{379} - 19$; $2^{395} + 29$; $2^{401} - 31$; $2^{409} + 29$; $2^{414} - 17$; $2^{438} + 25$; $2^{444} - 17$; $2^{456} + 21$; $2^{465} + 29$; $2^{468} - 17$; $2^{488} - 17$; $2^{489} - 21$; $2^{492} + 21$; $2^{495} - 31$; $2^{508} + 15$.

If future implementation work shows that the DEC-6 or DEC-7 primes in this list are more efficient (or more secure) than the similar-sized DEC-8 primes in this list, then one might conclude the decimal exponential complexity is better vindicated as a heuristic than intuition suggest it deserves.

Regardless, if a DEC-8 prime is superior to a DEC-7 prime, it is reasonable for a novice to seek a strong argument for the superiority, on the grounds of avoiding unnecessary complexity.

A Comparison to standard curves

To be completed.

A.1 Comparison to Curve25519

Bernstein introduced Curve25519 in 2005. After about a decade, it has now been adopted by CFRG and deployed in various ways. Its field size is usually expressed as though it were DEC-8 prime: `2^255-19`, which is of course, fine. But from the perspective of this paper, we seek a shorter expression to convey its lack of complexity. The exponent of two is divisible by three, so we may re-write the prime as $8^{85} - 19$, suggesting that it is actually a DEC-7 prime, which minimal expression `8^85-19`.

Furthermore, the field $2^{255} - 19$ is about 2^{22} times smaller the field of size $8^{81} + 5 = 2^{273} + 5$. This means that Pollard rho is expected to be about 2^{11} times faster in Curve25519 than in an elliptic curve defined over the field of size $8^{91} + 5$.

Of course, Curve25519 has some advantages over the field of size $8^{91} + 5$. In particular, elements of the field of size $2^{255} - 19$:

- can fit into 32 bytes, which might be a convenient number of bytes for network communications;
- can fit into four 64-bit words, which might help to reduce the number of 64-bit multiplications (in hardware with very fast addition);

- can fit into five 64-bit double-floating point numbers (each holding 53 bits), which might help in hardware with faster floating point arithmetic,
- can fit with lots of room to spare into five 64-bit integer, with the extra spare allowing for fewer reductions operations (such as after additions operations).

As usual in ECC, these efficiency advantages of Curve25519, largely due to its smaller size, can be weighed against the security advantage of the larger field size.

A.2 Comparison to NIST P-256

NIST curve P-256 uses the prime:

$$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \approx 2^{256.0} \tag{16}$$

which has a decimal exponential complexity of at most 24 symbols, as in `2^256-2^224+2^192+2^96-1`. Perhaps it has a shorter decimal exponential expression: it seems like an interesting to find the shortest expression.

Arguably, P-256 fares poorly under the decimal exponential complexity to the point of being unfair. Perhaps, it is only a coincidence that the field size P-256 has been claimed to be a little too complex to implement [Ber14] and also too complex in terms of decimal exponential expressions.

Obviously, the P-256 prime has some special structure that is not captured well by the decimal exponential complexity. For example, in the J programming language it can be expressed in 17 symbols, as `-/2x^32*8 7 6 0 3`, which includes space symbols. Nevertheless, since this is still ten more than the mere J symbols `5+8x^91` needed to express $8^{91}+5$, so it is not clear how to get the complexity under other measures even close to that of $8^{91}+5$, or even $2^{255}-19$.

A.3 Comparison to secp256k1

Recall that SEC1 imposes the requirement that $2^{255} < p < 2^{256}$. The SEC2 recommended curve secp256k1 uses the prime

$$p = 2^{256} - 2^{32} - 977 \tag{17}$$

Note that the curve secp256k1 has j-invariant 0, and therefore has complex multiplication by a cube root of unity. Presumably, p was chosen in a way to take this into account.

Unfortunately, SEC2 does not fully explain how p was derived. If p chosen to be as close as possible to 2^{256}, so that it has a prime size, when one would it to be much closer to 2^{256}. Basically, the term 2^{32} does not have a clear role. Perhaps, under some sophisticated analysis, this term aids efficiency.

So this p seems to have a decimal exponential complexity of 14 symbols. But it seems quite likely that a prime of decimal exponential complexity of 9 or 10 (by dropping the term -2^{32} and adjusting the term 977 accordingly), would serve just as well. Of course, there are far more 256-bit DEC-14 primes than 256-bit DEC-9 primes, but to be fair, the secp256k1 p has some special structure among those DEC-14 primes.

B A simplified implementation strategy

This section applies some well-known algorithmic implementation techniques to the to the field size of $p = 8^{91} + 5$. It uses a simplified model of the computer hardware.

This strategy is not necessarily optimal in efficiency or security.

B.1 Simplified computer hardware model

Consider computer hardware that uses 64 bits to represent any integer x with $|x| < 2^{63}$. Call this a 64-bit signed integer. Two such integers x and y can be added by the hardware, provided that the result is also representable: if $|x + y| < 2^{63}$. When $|x + y| \geq 2^{63}$, the computer hardware addition is said to overflow. Overflow might generated an error, or may be handled using arithmetic modulo 2^{64}. Either way, our implementation is strategy is use algorithms guaranteed to avoid overflow.

Subtraction of 64-bit signed integers is similar.

We further presume that the computer hardware has an ability to multiply two 64-bit signed integers, yielding a 128-bit signed integer, with no possibility of an overflow. The 128-bit signed integers may be internally represented as two 64-bit integers.

We further assume that that computer hardware, or the programming language of the implementation can realize addition and subtraction of these 128-bit signed integer (in the same way as 64-bit signed integers.).

For example, a C compiler might implement 64-bit signed integers as type `long` and 128-bit signed integers as type `long long`, but most use `long long` for 64-bit integers. The compiler GCC (version 4.6 and up) uses type `__int128` for 128-bit integers.

Alternative computer hardware models may be more appropriate, or superior, depending on the computer hardware. For example:

- Unsigned integers may be better than signed integers for some hardware.
- 32-bit (or smaller) integers may be the only hardware option on hardware, or may offer greater efficiency due to better parallelism[2].
- Floating point values may be better than the integers for some hardware.

An unconfirmed guess was that these alternatives do not work well for field size $8^{91} + 5$.

B.2 Field element representation

Map $\mathbb{Z}^5$ to $\mathbb{F}_p$ by sending vector x to the dot product

$$x \cdot (1, 2^{55}, 2^{110}, 2^{165}, 2^{220}) \mod p. \tag{18}$$

In other words, use a radix (base) 2^{55}. (Side note: I tried a mixed radix implementation, but it was slightly slower and more complicated.)

By linearity of the representation, conventional vector operations correspond to field operations, including addition, subtraction and scaling (by integers).

Field elements are usually represented as 5-tuples of 65-bit signed integers. An exception is that during multiplication of field elements, an intermediate format 5-tuples of 128-bit signed integers will be used.

B.3 Multiplication

For multiplication using the fact that $4p = (2^{55})^5 + 20$ seems helpful. (I also tried a mixed-radix multiplication, which was slightly slower and more complicated.)

[2]Tung Chou (NIST workshop on ECC, 2015) discusses using 32-bit vector instruction for field size $2^{255} - 19$ on 64-bit hardware.

B.3.1 Cyclic schoolbook multiplication

One way to multiply vectors x and y is with a cyclic convolution variant of schoolbook multiplication, where $z = xy$ has coordinates:

$$
\begin{aligned}
z_0 &= x_0y_0 & -20x_1y_4 && -20x_2y_3 && -20x_3y_2 && -20x_4y_1, && (19)\\
z_1 &= x_0y_1 & +x_1y_0 && -20x_2y_4 && -20x_3y_3 && -20x_4y_2, && (20)\\
z_2 &= x_0y_2 & +x_1y_1 && +x_2y_0 && -20x_3y_4 && -20x_4y_3, && (21)\\
z_3 &= x_0y_3 & +x_1y_2 && +x_2y_1 && +x_3y_0 && -20x_4y_4, && (22)\\
z_4 &= x_0y_4 & +x_1y_3 && +x_2y_2 && +x_3y_1 && +x_4y_0. && (23)
\end{aligned}
$$

The following lemmas establish some conditions under which overflow does not occur when using the cyclic schoolbook formulas above.

Lemma 2. *If $|x_i|, |y_j| < 2^{60}$ for all i and j, then $|z_k| < 2^{127}$ for all k.*

Proof. If $|x_i|, |y_j| < 2^{60}$ for all i, j, then $|x_iy_j| < 2^{60}2^{60} = 2^{120}$ for all i, j. Then $|z_k| < (1 + 20 + 20 + 20 + 20)2^{120} < 2^7 2^{120} = 2^{127}$. □

The next lemma may be helpful to optimize the multiplication of field elements by pre-multiplying $20x_i$, in other words computing $20x_iy_j$ as $(20x_i)y_j$. This may help optimize if $20x_i$ can be computed entirely with 64-bit integer operations.

Lemma 3. *If $|x_i| < 2^{58}$ for all i, then $|20x_i| < 2^{63}$ for all i.*

Proof. If $|x_i| < 2^{58}$, then $|20x_i| < |2^5 2^{58}| = 2^{63}$. □

B.4 Reduction

The arithmetic operations described above tend to increase the magnitudes of the coordinates of the tuples. If this operations were iterated repeatedly, eventually the entries in the tuples would overflow the computer hardware integers.

So some form of reduction operation is needed to avoid overflow.

A second type of reduction helps to give each field element a unique representation. These unique representations are vital at the final stages of ECC calculations to ensure interoperability, but they are not necessary for the intermediate calculations.

B.4.1 Partial reduction

In partial reduction, we only try to avoid overflow, and forgo uniqueness.

Suppose $|z_i| < 2^{127}$. Let $z_i = 2^{110}q_i + 2^{55}u_i + r_i$, with $|r_i| < 2^{55}$ and $|u_i| < 2^{55}$, and $|q_i| < 2^{17}$. Let v be the vector with coordinates:

$$v_0 = r_0 \quad -20u_4 \quad -20q_3, \tag{24}$$
$$v_1 = r_1 \quad +u_0 \quad -20q_4, \tag{25}$$
$$v_2 = r_2 \quad +u_1 \quad +q_0, \tag{26}$$
$$v_3 = r_3 \quad +u_2 \quad +q_1, \tag{27}$$
$$v_4 = r_4 \quad +u_3 \quad +q_2. \tag{28}$$

One might hope that the computer hardware is able do some of the additions above in parallel since the additions used to compute v_i do not depend on the results of the additions used to compute v_j.

The following lemma shows the reduced vectors v is ready to be used in various arithmetic operations without overflow.

Lemma 4. *If $|z_i| < 2^{127}$, then $|v_i| < 2^{60}$.*

Proof. Now $|v_i| \leq |r_i| + 20|u_{i-1}| + 20|q_{i-2}| < 2^{55} + 20(2^{55}) + 20(2^{17}) < (22)2^{55} < 2^5 2^{55} = 2^{60}$. □

Unfortunately, this lemma is not ideal, for two reasons:

- It does not ensure that adding partially reduced numbers are multiplication-ready.
- It does not ensure that 5-tuple v is ready for the pre-multiplication optimization.

In other words, this form of partial reduction might lead to errors.

A caution: I must admit that I preliminarily implemented a version of elliptic Diffie–Hellman using the Montgomery with partial reduction as above. Such a preliminary implementation is likely vulnerable to an implementation-fault attack. An adversary could generate field values that would cause an overflow, which in turn would leak bits of a secret key. Therefore, a case can be made that field size $8^{91} + 5$ is susceptible to implementation faults.

It seems that we should do a further reduction of v_0, to ensure a better readiness for follow-up operations. To that end, let $v_0 = 2^{55}s_0 + w_0$ with

$|w_0| < 2^{55}$, and $|s_0| < 2^8$. Let:

$$w_0 = w_0, \tag{29}$$
$$w_1 = v_1 + s_0, \tag{30}$$
$$w_2 = v_2, \tag{31}$$
$$w_3 = v_3, \tag{32}$$
$$w_4 = v_4, \tag{33}$$

Lemma 5. *If $|z_i| < 2^{127}$, then $|w_i| < 2^{57}$.*

Proof. By definition $|w_0| < 2^{55} < 2^{57}$.

For $i \in 2, 3, 4$, we have $|w_i| = |v_i| \leq |r_i| + |u_i| + |q_i| < 3(2^{55}) < 2^{57}$.

Finally, $|w_1| = |r_1 + u_0 - 20q_4 + s_0| \leq |r_1| + |u_0| + 20|q_i| + |s_0| < 4(2^{55}) = 2^{57}$. □

B.4.2 Full reduction (finalization)

A fully reduced vector w has $w_i \geq 0$, but each w_i as small as possible. A unique representation is generally needed so that Alice and Bob can interoperate.

If we assume a computer hardware model that compute non-negative remainders of negative integers modulo powers of two, then a straightforward method is to compute quotients and remainders starting from the least significant limb. The quotient from the most significant limb is scaled by -5 before being added to the least significant limb.

One cycle of these limb reductions can result in the first (least signficant) limb (limb zero in an array implementation) either negative, or exceeding its positional radix 2^{55}. A simple fix is to do a second cycle. After the second cycle, exceeding the radix is impossible, because the last limb was reduced in the first cycle, and can now only cause a negative carry to the first limb.

If this least significant limb is negative, then one undoes this last step. This looks correct to me now, but I must admit that it took me few tries to get this right.

B.5 Inversion and square roots

Inversion in finite fields is needed for ECC operations. For public values, some variant of extended Euclidean algorithm is often the fastest, but this algorithms are variable-time. A variance in time depending on the secrets is a potential weakness. A commonly proposed countermeasure is to use Fermat inversion, compute x^{-1} as x^{p-2}.

Fermat inversion for our field, has $p - 2 = 8^{91} + 3$. Computing x^{p-2} is therefore very fast. Just compute:

$$x^2, x^4, x^8, \ldots, x^{2^{273}}, x^{2^{273}+2}, x^{p-2} \tag{34}$$

where ellipsis indicates repeated squaring. This takes 273 squarings and 2 multiplications, which is nearly optimally efficient for modulus of this size.

To compute square root, the usual method for primes with $p \equiv 5 \bmod 8$ can be used. The first step is raise the field element to the power of $(p + 5)/8 = 2^{270} + 1$. This simple and fast, like inversion, mostly involving repeated squarings. As usual for primes $p \equiv 5 \bmod 8$, one may need to adjust this power by multiplication by $\sqrt{-1}$ if the the power obtained is not a square root of the input. Finally, on inputs that do not have square roots at all, the square root procedure should indicated a failure result.

C Sample code

The sample code described below follows the algorithmic strategies discussed earlier in this paper. It aims for simplicity, like the field size $8^{91} + 5$ itself. It is naively written, and unlikely to be optimal in any respect. Using the sample code may require a license.

The sample code is not hardened against various pitfalls to secure ECC implementation: I do not claim accurate knowledge of these pitfalls. Indeed, the code has not even been subjected to any quality control, so may not even be fully functional, let alone secure.

The sample code uses a non-portable variant of the C programming language, making it more portable than machine-specific assembler code, yet still quite efficient (because C instructions often closely correspond to machine instructions). Two non-portable features of the C sample code are:

- an 128-bit signed integer type `__int128` (this type is available in GCC, version 4.6 and up, but is not part of standard C), and
- negative integer operands for the C operators bit-wise operators `>>` and `<<` and `&` (strictly speaking standard C says these operators undefined on negative integers, but two's complement and arithmetic shifts are now very common).

The sample code uses (without permission or understanding) some, but not all, of the C coding techniques in Bernstein and team's TweetNaCl library and in M. Scott's implementation of NIST P-521.

```
/*
  smpl_8^91+5.c

  A compact field for ECC.

  (c) 2016, Dan Brown, Certicom/BlackBerry

  Sample code only: unfit for real-world use.

  Needs gcc 4.6 (or later).
*/

typedef signed long long int i     ;
typedef                      i f[5];

# define FUN    inline void
# define FOR(S) {i j;for(j=0;j<5;j+=1){S;}}

FUN add(f z, f x, f y){ FOR (z[j]=x[j]+y[j]); }

FUN sub(f z, f x, f y){ FOR (z[j]=x[j]-y[j]); }

FUN mal(f z, i s, f y){ FOR (z[j]=s*y[j]); }

typedef __int128 ii      ;  /* gcc 4.6+ */
typedef          ii ff[5];

static FUN med(f z, ff zz)
{
# define QUA(x) (x       >>55)
# define MAD(x) (((((i)1)<<55)-1)&x)
# define Q(j)    QUA(QUA(zz[j]))
# define U(j)    MAD(QUA(zz[j]))
# define R(j)        MAD(zz[j])
  z[0] = R(0) - 20*U(4) - 20*Q(3);
  z[1] = R(1) +    U(0) - 20*Q(4);
  z[2] = R(2) +    U(1) +    Q(0);
  z[3] = R(3) +    U(2) +    Q(1);
  z[4] = R(4) +    U(3) +    Q(2);
  z[1] += QUA(z[0]);
  z[0]  = MAD(z[0]);
}

FUN fix(f x)
{
  i q, j ;
# define FIX(j,r,k)       \
  q           = x[j] >> r;\
  x[j]       -= q    << r;\
  x[(j+1)%5] += q    *  k;
  for(j=0 ; j<2; j+=1) {
    FIX(0, 55,  1); FIX(1, 55,  1);
    FIX(2, 55,  1); FIX(3, 55,  1);
    FIX(4, 53, -5);
  }
  q     = (x[0]<0) ;
  x[0] += q  * 5   ;
  x[4] += q << 53  ;
}

FUN mil(f z, i s, f y)
{
  ff zz ;
  FOR( zz[j] = s * (ii)y[j] );
  med(z,zz) ;
}

# define CYC(M)\
  ff zz ;\
  zz[0] = M(0,0) - 20*M(1,4) - 20*M(2,3) - 20*M(3,2) - 20*M(4,1);\
  zz[1] = M(0,1) +    M(1,0) - 20*M(2,4) - 20*M(3,3) - 20*M(4,2);\
  zz[2] = M(0,2) +    M(1,1) +    M(2,0) - 20*M(3,4) - 20*M(4,3);\
  zz[3] = M(0,3) +    M(1,2) +    M(2,1) +    M(3,0) - 20*M(4,4);\
  zz[4] = M(0,4) +    M(1,3) +    M(2,2) +    M(3,1) +    M(4,0);\
  med(z,zz);

FUN mul(f z, f x, f y)
{
# define MUL(j,k) x[j] * (ii)y[k]
  CYC(MUL) ;
}

FUN squ(f z, f x)
{
# define SQR(j,k) x[j] * (ii)x[k]
# define SQU(j,k) SQR(j>k?j:k, j<k?j:k)
  CYC(SQU);
}

void inv(f y, f x)
{
  f z ; i j;
  fix(x);
  squ(z,x) ;
  mul(y,x,z) ;
  for(j=2; j<=273; j+=1){squ(z,z);}
  mul(y,z,y) ;
}
```

D Example curves

The field of size $8^{91} + 5$ can be used with any equation defining a secure elliptic curve.

D.1 A curve with a compact equation and an efficient endomorphism

The curve with equation:

$$2y^2 = x^3 + x \tag{35}$$

has size $72q$ for a prime q, with $q \approx 2^{266.8}$.

This curve equation is quite compact, expressible as `2*y^2=x^3+x` in 11 symbols.

The factorization of $q - 1$ seems to be

$$\begin{aligned}&2^3 \times 101203 \times 23810182454264420359 \\ &\times 1093478435746377647334249806229996592595611508697699265 7\end{aligned} \tag{36}$$

(obtained from Sage). It seems that $p^{(q-1)/u} \not\equiv 1 \bmod q$ for $u = 4$ and each of the odd prime factors of $q-1$, which means that the embedding degree of the order q subgroup is $(q-1)/2$. This renders the MOV attack infeasible (and also any pairings).

This curve equation permits the use Montgomery's efficient differential addition formulas. Of course, any curve with order divisible by four can be transformed into a similar shape. Also, this curve, and any other with co-factor divisible by four, can be transformed into an Edwards curve if needed.

Using Cornacchia's algorithm finds $p = u^2 + v^2$ with

$$u = 1043033027901133467787029269772887051447 69 \tag{37}$$

$$v = 655585368017578752283604058587318062815 06. \tag{38}$$

Then $72q = (u+1)^2 + v^2 = p + 2u + 1$.

D.1.1 Conjectured attack?

It is commonly conjectured that an attack exists against curves with efficient endomorphisms (or, more generally, complex multiplication by low discriminant, which in this example 4). On the other hand, efficient-endomorphism curves

- were introduced by Miller at the same time as he introduced elliptic curve cryptography, and
- have been used in BitCoin.

No major attacks have been discovered since then. So, the fact that the remain unbroken despite their early proposal and deployment large-incentive target suggests that the common conjecture is perhaps too fearful.

To be completed.

D.1.2 Not ideal for static Diffie–Hellman

The elliptic curve $2y^2 = x^3 + x$ is not ideal for use with static keys, for the reasons given further below. In other words, it is best used only for ephemeral static Diffie–Hellman key exchange.

Twist security The twist of the curve has order with prime factorization:

$$2^2 \times 5 \times 1526119141 \times 788069478421 \times 182758084524062861993 \\ \times 3452464930451677330036005252040328546941. \quad (39)$$

The prime factorization of the twist affects certain types of security. The main implementations that are affected are those that

1. use static Diffie–Hellman key exchange,
2. without public key validation,
3. use the Montgomery ladder (or similar x-only coordinate algorithm) for scalar multiplication.
4. use the Diffie–Hellman shared secret key before verifying proof-of-possession of the other entity's key (such as decrypted a ciphertext without checking the associated message authentication code).

Such an implementation potentially allows an attacker to send the static DH victim just five invalid public keys, and then do a computation of about 2^{68} group operations to the extract the static private key. If the implementation also exposes the raw shared secret, which is a requirement of some esoteric protocols, then the attacker's work is further reduced to about 2^{65} steps. If the static Diffie–Hellman implementation requires proof-of-posssion of the other parties' private key, then it seems that the attack is resisted, unless that

Cheon security The factorization of $q+1$ is

$$\begin{aligned}&2 \times 3 \times 11 \times 21577 \times 54829 \times 392473 \times 854041\\ &\quad \times 805420153081115125375393663558120685638177971145156481304 1\end{aligned} \tag{40}$$

which suggests that the curve is a little more vulnerable to Cheon's attack than is typical for a random curve, because $q+1$ has many small factors. Cheon's attack is only relevant for static Diffie–Hellman keys. Cheon's attack is thwarted by secure key derivation functions, but it is nonetheless desirable to have curves that better resist Cheon's attack on their own, without relying on the key derivation function.

den Boer security The presence of smallish prime factors in $q+1$ leads to a variant of den Boer's reduction (see [Bro14] for some details).

Let D be the minimum cost of the solving Diffie–Hellman problem in the order q subgroup of the elliptic curve, and let L be the minimum cost of solving the discrete logarithm problem in the same subgroup. Let M be cost of a scalar muliplication in the elliptic curve group. Then the approximate inequality:

$$D \geq \frac{L - 2^{100}M}{2^{13}} \tag{41}$$

seem to hold, for the reasons outlined below (following a variant of den Boer's reduction).

Convert the elliptic curve subgroup into the field $\mathbb{F}_q$ by using a Diffie–Hellman solver to implement multiplication, and the elliptic curve group operation for addition. In other words, if G is the base point over which Diffie–Hellman problem is defined, then aG represents the field element with standard integer representation a. Solving the discrete logarithm is therefore equivalent to determine the standard integer representation from the elliptic curve point representation of a field element. The opposite direction is much cheaper, just corresponding to scalar multiplication.

Sometimes, multiplication in $\mathbb{F}_q$ can be done without using a Diffie–Hellman solver: if one of the elements has a known standard integer represention, in other words, its logarithm to the base G is known, then we can use a convential scalar multiplication to compute the product in $\mathbb{F}_q$.

Implement the field $\mathbb{F}_{q^2}$ reprepresenting field elements in $\mathbb{F}_{q^2}$ as pairs of field elements in F_q. Implement $\mathbb{F}_{q^2}$ addition using two $\mathbb{F}_q$ additions. Implement $\mathbb{F}_{q^2}$ using Karatsuba's algorithm, so with three $\mathbb{F}_q$ multiplications and some number of additions. Note that the multiplicative group $\mathbb{F}^*_{q^2}$ has order $q^2 - 1$, and in particular, has a subgroup of order $q + 1$. Let H be this group.

Given a field element as a point P, the discrete logarithm problem is find an integer d such that $P = dG$. The strategy to these is to solve a discrete logarithm depending on d in the subgroup of size $q + 1$ in $\mathbb{F}_{q^2}$. Doing so, allows us to determine a standard integer representation of d in terms of F_{q^2} discrete logarithm found.

Solving discrete logs in H, can done as follows. Let h be the target (a value depending on P), and let g be the base (not depending on P). We aim to find l such that $h = g^l$. For each of the seven small prime factor p of $q + 1$, do the following. Compute $h_p = h^{(q+1)/p}$, by solving about about $3 \times 1.5 \times 273 \approx 1228$ Diffie–Hellman problems in the elliptic curve group. Now h_p belongs to an order p subgroup, so its discrete log (to base $g_p = g^{(q+1)/p}$) can be found using baby-step giant-step in about $2\sqrt{p}$ group operations in H. Each such group operation in H can be implemented as a scalar mulitplication in the elliptic curve group, so does not require the use of a Diffie–Hellman solver. The log of h_p determines $l \bmod p$. Doing these for each the seven primes, requires a total of about 2^{13} Diffie–Hellman solvers.

Use the Chinese remainder theorem to solve for l modulo P where P is the product of the small primes. Now $hg^{-l \bmod P}$ is an element of H with order $(q+1)/P$, which is a large prime of size approximately 2^{192}. As above, one can use baby-step giant-step to solve to find its discrete log, using on sclar muliplication. This uses about 2^{100} scalar muliplications.

To illustrate what (41) implies consider the following situations:

- If $L \geq 2^{124}M$, approximately, which we would expect[3] if Pollard rho is the best algorithm to solve the elliptic curve discrete logarithm, then $D \geq 2^{110}M$.

- If we only have the much weaker bound $L \geq 2^{108}M$, because of some surprise attack faster than Pollard rho, then we still have a decent bound of $D \geq 2^{96}$.

Of course, it is possible to just simply conjecture that $D \geq L$. The point of using reductionist security, such as the den Boer arguments, is to avoid just assuming security. One can still make that case, the conjecture $D \geq L$ is old and studied enough (it has aegis), to render the den Boer reduction redundant.

[3]Roughly, Pollard rho should $\sqrt{q} \approx 2^{133}$ group operations, while Scalar multiplication should take approximately 2^9 group operations, giving Pollard rho a cost of $2^{124}M$.

D.1.3 Hashing into the curve (elligator)

Preliminary—to be verified.

Bernstein, Hamburg, Krasnova and Lange [BHKL13] define a map, which they call Elligator 2, which transforms finite field elements into elliptic curve points. Their map works for any field, any curve with a point of order two, unless it has j-invariant 1728. Unfortunately, their map does not work for the curve $2y^2 = x^3 + x$.

So, briefly, here is an alternative map. It is a minor modification of the Elligator 2 map, so for lack of a better name, call the resulting map Elligator i. Let i be a fourth root of one modulo p. Let r be any field element. Let:

$$x = \frac{r^2 - 2i}{1 - ir^2} = i - \frac{3i}{1 - ir^2}. \tag{42}$$

The key property of this x is that $x + 2i = ir^2(x - i)$.

If there is no solution y to $2y^2 = x^3 + x$ for the x computed above, then add i to x and try again. There is guaranteed to be a solution for y on the second try because:

$$(x+i)^3 + (x+i) = (x+2i)(x+i)x = ir^2(x-i)(x+i)x = ir^2(x^3+x). \tag{43}$$

Note that, in Elligator 2, x is replaced by $-a - x$ instead of $x + i$ (and the curve equation has the form $y^2 = x^3 + ax^2 + bx$ instead of $2y^2 = x^3 + x$). The step of choosing x or $x + i$ can, of course, be expressed as an algebraic equation rather than a branching statement, by computing the Legendre symbol of $x^3 + x$, just the way it is done for Elligator 2.

For the y-coordinate, just set $y = \pm\sqrt{(x^3 + x)/2}$, using some canonical square root function, and choosing the sign according to whether one had to the step of shifting x and $x + i$. Retaining this sign ensures that r maps to x and stays there, while r' maps to $x - i$ and then shifts to $(x - i) + i = x$, that the resulting y-coordinates will be different.

It seems that the benefits of Elligator 2 can also be achieved with Elligator i, because of the similarity of the two maps. The map ψ above from r to x has the property that $\psi(r) = \psi(-r)$, and $\psi^{-1}(\psi(r)) = \{r, -r\}$.

The map $\iota : \{0, 1, \ldots, 255\}^{34} \to X$ first maps a byte string b of length 34 into a field element r of $\mathbb{F}_p$ such that $r < 2^{272} < p/2$, then it applies map ϕ above a curve point. The function ι is injective and easily inverted, much like Elligator 2 map.

The map ι is not surjective. Its image has about $p/2$ points, which is only about half the points on the curve.

To be verified.

D.1.4 Using an efficient endomorphism in Bernstein's ladder

This curve has an efficient endomorphisms. For example, complex multiplication by i, given by:

$$[i](x, y) \mapsto (-x, iy) \tag{44}$$

This efficient endomorphism is potentially useful to forms of simultaneous multiplication. For example, Bernstein's two-dimensional differential addition chains [Ber06], an enhancement of Montgomery's ladder. Of course, sometimes the overhead of an optimization strategy outweighs its benefits. The task then becomes to minimize the overhead.

Rather than conventionally starting from the notion of integer multiplier, we observe that ephemeral Diffie–Hellman can be achieved without explicitly representing the integer multipliers. Instead, it is sufficient to have a sequence of elliptic curve operations that can be applied at least twice: the first time to generate the public keys, the second time to generate the shared secret. Of course, these sequence of operations do indeed implicitly correspond to a scalar multiplication by some integer, and one has to be sure that the integer is sufficiently large.

To this end, we first review a variant Bernstein's ladder, using a rather abstract perspective of linear algebra on the multipliers. The aim of this theoretical preparation is to identify some simple bit operations that can minimize the overhead of the ladder. Define square matrices:

$$D = \begin{pmatrix} 1 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix}, \qquad T_{0,0} = \begin{pmatrix} 2 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}, \qquad T_{0,1} = \begin{pmatrix} 0 & 2 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}, \tag{45}$$

$$T_{1,0} = \begin{pmatrix} 0 & 0 & 2 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}, \qquad T_{1,1} = \begin{pmatrix} 0 & 2 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}. \tag{46}$$

The matrices $T_{u,v}$ represent an elliptic curve double and two elliptic curve differential additions. The matrix D is used to determine the differences needed in the elliptic curve differential additions.

Our ladder will apply an arbitrary sequence of the four matrices. It is hoped that the implicit corresponding scalar multiplication by an integer has the integer with sufficient security.

Define column matrices (thought of as vectors):

$$A = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \qquad E_{0,0} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad E_{0,1} = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}, \tag{47}$$

$$C_0 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \qquad C_1 = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}, \qquad E_{1,0} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \qquad E_{1,1} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}. \tag{48}$$

Bernstein's ladder will act on two column matrices simultaneously, by applying the matrices $T_{u,v}$ and D as needed. But first we focus on the action of the square matrices single column. The action of the square matrices on A is simple:

$$DA = 0, \qquad T_{u,v}A = 2A, \tag{49}$$

for all index pairs (u, v). Using modulo two arithmetic on the indices [in brackets], we can summarize the action of the square matrices $T_{u,v}$ on the elementary column matrices $E_{f,g}$ as:

$$T_{u,v}E_{f,g} = E_{[f+u],\ [g+v(f+u+1)]} \quad + \quad (-1)^g[f+u+1]A. \tag{50}$$

Consequently, column matrices of the form $nA + E_{f,g}$ are closed under the action of the matrices $T_{u,v}$. More precisely:

$$T_{u,v}(nA + E_{f,g}) = (2n + h')A + E_{f',g'} \tag{51}$$

where $h' \in -1, 0, 1$ and f', g', h' are determined from (50), as $f' = [f + u]$ and $g' = [g + v(f + u + 1)]$ and $h' = (-1)^g[f + u + 1]$.

The action of the square matrix D on the elementary matrices of the form $nA + E_{f,g}$ is given by:

$$D(nA + E_{f,g}) = (-1)^{(f+g)}(E_{f,g} + C_f) \tag{52}$$

There are only four values of the column matrices distinct column matrices $E_{f,g} + C_g$. The result of applying D only depends on the bit indices f, g, and these can be tracked simply using (50).

Now consider a two-column matrix $B = (b_{i,j})$ with first column $n_0A + E_{f_0,g_0}$ and second column $n_1A + E_{f_1,g_1}$. We insist that $[f_0 + f_1] = 1$, so f_0 and f_1 have opposite parity.

Suppose we have a triple (x_0, x_1, x_2) of x-coordinates of three elliptic curve points R_0, R_1 and R_2 corresponding to the three rows of the matrix B, in the sense $R_i = b_{i,0}P_0 + b_{0,1}P_1$ for some points P_0 and P_1.

Let $B' = T_{u,v}B$. Our task is to compute the corresponding triple of triple (x_0', x_1', x_2') of x-coordinates of the points points R_i' corresponding to the rows of B'.

For convenience, now write $F = f_0 + 2f_1$ and $G = g_0 + 2g_1$, and furthermore $(U, V) = (3u, 3v)$. The purpose of this is to use bit-wise operations on integers to perform parallel bit operations. Of course, since f_0 and f_1 have opposite parity, we must have $F \in \{1, 2\}$. We also write $E^+_{F,G}$ for the two-column matrix whose columns are E_{f_0,g_0} and E_{f_1,g_1}.

After application of a matrix $T_{u,v}$, an updated (F', G') is derived using the bit operations from (50) above, now using bit-wise integer operations.

Define two functions, x-coordinate doubling and x-coordinate differential addition such that:

$$\delta : x(P) \qquad \mapsto x(2P) \tag{53}$$

$$\alpha : (x(P), x(Q), x(P - Q)) \qquad \mapsto x(P + Q) \tag{54}$$

for any points P and Q, where $x(P)$ means the x-coordinate. In practice, one use projective coordinates, to avoid division. Then we have:

$$x_0' = \delta(x_{v+2u(1-v)}), \tag{55}$$

$$x_1' = \alpha(x_1, \qquad x_{2u}, \qquad y), \tag{56}$$

$$x_2' = \alpha(x_2, \qquad x_0, \qquad z), \tag{57}$$

where the index arithmetic is now *not* done modulo two. The values y and z are field elements extracted from a small pre-computed table as determined based on the current value of the pairs (u, v) and (F, G).

We first treat z. It corresponds to the middle row of the matrix $DE^+_{F,G}$. Just crunching through the tedious calculations, it seems the pattern is determined by the modulo sum of the bits of $F + 4G$. If the sum is zero, then $z = x(P_0 + P_1)$ and otherwise it is $z = x(P_0 - P_1)$. Since $F \in \{1, 2\}$, we may write this as

$$z = x(\ P_0 - (-1)^{|G|} P_1\ ), \tag{58}$$

where $|G|$ is the Hamming parity of G.

Finally, y is to be determined. If $u = 0$, then y is determined from the top row of the matrix $DE^+_{F,G}$. If $y = 1$, then it is the bottom row. The tedious calculations of all eight possible matrices $DE^+_{F,G}$, leads to the following rule:

$$y = x(P_{[F+u]}). \tag{59}$$

For the difference table, if we take $P_0 = P$ and $P_1 = [i](P)$, and set $x = x(P)$. Then $x(P_0) = x$ and $x(P_1) = -x$, and:

$$x(P_0 + P_1) = x([1+i]P) = \frac{-i(x^2+1)}{2x} = -\frac{i}{2}\left(x + \frac{1}{x}\right) \tag{60}$$

which we would represent in projective coordinates. Finally, $x(P_0 - P_1) = x([1-i]P) = -x([1+i]P)$ is just the negative of the above.

Initialize the triple (x_0, x_1, x_2) and the pair (F, G) as follows. Put $(F, G) = (2, 0)$. Initialize B as

$$B = \begin{pmatrix} 2 & 0 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{pmatrix} + E^{+}_{2,0} \tag{61}$$

This means that x_0 should be the x-coordinate of $x([2]P) = \delta(x)$. I think that we cannot have the first row of B be all zeros, because the usual differential addition laws are not well-defined if one of the inputs is the point-at-infinity.

D.1.5 Sample code

The sample code below uses the strategy above. I have not tested it substantially. In particular, I have not proved that overflow is avoided.

The sample code uses secret array indices, which potentially results cache-timing attacks. Therefore, the sample code should be hardened to avoid this problem.

I bench-marked this code to similar code for the standard Montgomery ladder. This version seems slightly faster.

```
/* ecdh_bern_8^91+5.c */

#include "smpl_8^91+5.c"

// A Montomgery & Gallant--Lambert--Vanstone curve: 2*y^2 = x^3 + x.
// Montgomery differential addition laws [hyperelliptic.org/EFD]
// Bernstein uniform differential addition chain (ladder)

typedef f p[2] ;  /* XZ coordinates (projective) */

FUN double_xz (p p2, p p1)
{
  f a,aa,b,bb,c,d ;
  add(a,p1[0],p1[1]);
  sub(b,p1[0],p1[1]);
  squ(aa,a);
  squ(bb,b);
  sub(c,aa,bb);
  add(d,bb,aa);
  mul(p2[0],aa,bb);
  mal(p2[0],2,p2[0]); /* mil? */
  mul(p2[1],c,d);
}

FUN diff_add (p p5, p p3, p p2, p p1)
{
# define DIFF_ADD_PREP()\
  f a,b,c,d,da,cb ;\
  add(a,    p2[0], p2[1]);\
  sub(b,    p2[0], p2[1]);\
  add(c,    p3[0], p3[1]);\
  sub(d,    p3[0], p3[1]);\
  mul(da,   d,     a    );\
  mul(cb,   c,     b    );\
  add(p5[0],da,    cb   );\
  sub(p5[1],da,    cb   );\
  squ(p5[0],p5[0]       );\
  squ(p5[1],p5[1]       );

  DIFF_ADD_PREP();
  mul(p5[0],p5[0],p1[1]);
  mul(p5[1],p5[1],p1[0]);
}

FUN diff_add_1 (p p5, p p3, p p2, p p1)
{
  DIFF_ADD_PREP();
  mul(p5[1],p5[1],p1[0]);
}

# define MAKE(I) i j;\
  static f I =   {2,0,0,0,0};\
  static i have_I = 0;\
  if (! have_I) {\
        for(j=1; j<=271; j+=1){squ(I,I);}\
        mal(I,2,I); fix(I);\
        have_I = 1;\
  }

void endo_1i(p p1i, f px)
{
  f xx ;
  MAKE(I);
  squ(xx,px);
  add(xx,(f){1},xx);
  mul(p1i[0], I,xx);
  mal(p1i[1],-2,px); /* mil? */
}

typedef struct rung {i x0; i x1; i y; i z;}  k[137] ;

void bern (f ps, k sk, f px)
{
  i h; i j ;
  p w[3], x[3], y[2]={{{},{1}}}, z[2] ;

  fix(px); mal(y[0][0],1,px); endo_1i(z[0],px);

# define COPY(q,r) mal(q[0], 1,r[0]); mal(q[1],1,r[1]);
# define ENDO(q,r) mal(q[0],-1,r[0]); mal(q[1],1,r[1]);

  ENDO(y[1],y[0]); ENDO(z[1],z[0]);
  COPY(x[1],y[0]); COPY(x[2],z[0]);
  double_xz(x[0],y[0]);

  for (j=0 ; j<137; j+=1){

# define S sk[j]

  /* CACHE-UNSAFE: Replace secret array indexing by cache-safe
         selection */

        double_xz (w[0],        x[S.x0] );
        diff_add_1(w[1], x[1], x[S.x1],  y[S.y]);
        diff_add  (w[2], x[2], x[0],     z[S.z]);

        for(h=0; h<3; h+=1) { COPY(x[h],w[h]);}
  }
  inv(ps,x[1][1]);
  mul(ps,x[1][0],ps);
  fix(ps);
}

void weak_key (k sk, unsigned char * s)
{
  i j,u,v;
  i b=1, f=2, g=0;
  for (j=0 ; j<137; j+=1){
        u = !! (s[j/4]&((1<<(j%4))  ));
        v = !! (s[j/4]&((1<<(j%4))+1));
        b *= ('\0' != s[j/4]) ;
        u *= b; v *= b;
        sk[j].x0 = v + 2*u*(1-v);
        sk[j].x1 = 2*u;
        sk[j].y  = (f+u)%2;
        sk[j].z  = (g==0)||(g==3) ;
        u *= 3; v *= 3;
        g ^= v & (f^u^3) ;
        f ^= u;
  }
}
```

References

[Ber06] D. J. Bernstein. *Differential addition chains.* http://cr.yp.to/papers.html#diffchain, 2006.

[Ber14] ———. *Curve25519, Curve41417, E-521*. Presentation to Cryptographers Forum Research Group at Internet Engineering Task Force 90, https://www.ietf.org/proceedings/90/slides/slides-90-cfrg-4.pdf, 2014.

[Bro14] D. R. L. Brown. *CM55: special prime-field elliptic curves almost optimizing den boer's reduction between Diffie–Hellman and discrete logs*. Cryptology ePrint Archive, Report 2014/877, 2014. http://ia.cr/2014/877.

[BCC$^+$14] D. J. Bernstein, T. Chou, C. Chuengsatiansup, A. Hülsing, T. Lange, R. Niederhagen and C. van Vredendaal. *How to manipulate curve standards: a white paper for the black hat*. Cryptology ePrint Archive, Report 2014/571, 2014. http://ia.cr/2014/571.

[BHKL13] D. J. Bernstein, M. Hamburg, A. Krasnova and T. Lange. *Elligator: Elliptic-curve points indistinguishable from uniform random strings*. In *ACM Conference on Computer and Communications Security*. 2013. http://cr.yp.to/papers.html#elligator.

[BL13] D. J. Bernstein and T. Lange. *Safecurves: choosing safe curves for elliptic-curve cryptography: Rigidity*. http://safecurves.cr.yp.to/rigid.html, 2013.

[IT03] T. Izu and T. Takagi. *Exceptional procedure attack on elliptic curve cryptosystems*. In Y. G. Desmedt (ed.), *Public key cryptography – PKC 2003*, no. 2567 in LNCS, pp. 224–239. Springer, 2003.