How to manipulate curve standards: a white paper for the black hat

Daniel J. Bernstein[1,2], Tung Chou[1], Chitchanok Chuengsatiansup[1],
Andreas Hülsing[1], Eran Lambooij[1], Tanja Lange[1],
Ruben Niederhagen[1], and Christine van Vredendaal[1]

[1] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, NL
blueprint@crypto.tw, c.chuengsatiansup@tue.nl,
andreas.huelsing@googlemail.com, e.lambooij@student.tue.nl,
tanja@hyperelliptic.org, ruben@polycephaly.org, c.v.vredendaal@tue.nl

[2] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to

Abstract. This paper analyzes the cost of breaking ECC under the following assumptions: (1) ECC is using a standardized elliptic curve that was actually chosen by an attacker; (2) the attacker is aware of a vulnerability in some curves that are not publicly known to be vulnerable.

This cost includes the cost of exploiting the vulnerability, but also the initial cost of computing a curve suitable for sabotaging the standard. This initial cost depends heavily upon the acceptability criteria used by the public to decide whether to allow a curve as a standard, and (in most cases) also upon the chance of a curve being vulnerable.

This paper shows the importance of accurately modeling the actual acceptability criteria: i.e., figuring out what the public can be fooled into accepting. For example, this paper shows that plausible models of the "Brainpool acceptability criteria" allow the attacker to target a one-in-a-million vulnerability and that plausible models of the "Microsoft NUMS criteria" allow the attacker to target a one-in-a-hundred-thousand vulnerability.

Keywords. Elliptic-curve cryptography, verifiably random curves, verifiably pseudorandom curves, minimal curves, nothing-up-my-sleeve numbers, ANSI X9, NIST, SECG, Brainpool, Microsoft NUMS, sabotaging standards, fighting terrorism, protecting the children.

This work was supported by the European Commission under contracts INFSO-ICT-284833 (PUFFIN) and H2020-ICT-645421 (ECRYPT-CSA), by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005, and by the U.S. National Science Foundation under grant 1018836. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation." Calculations were carried out on two GPU clusters: the Saber cluster at Technische Universiteit Eindhoven; and the K10 cluster at the University of Haifa, funded by ISF grant 1910/12. Permanent ID of this document: bada55ecd325c5bfeaf442a8fd008c54. Date: 2015.09.27. See web site: bada55.cr.yp.to. This work did not receive the funding that it so richly deserves from the U.S. National Security Agency.

1 Introduction

More and more Internet traffic is encrypted. This poses a threat to our society as it limits the ability of government agencies to monitor Internet communication for the prevention of terrorism and globalized crime. For example, an increasing number of servers use *Transport Layer Security* (TLS) as default (not only for transmissions that contain passwords or payment information) and also most modern chat applications encrypt all communication. This increases the cost of protecting society as it becomes necessary to collect the required information at the end points, i.e., either the servers or the clients. This requires agencies to either convince the service providers to make the demanded information available or to deploy a back door on the client system respectively. Both actions are much more expensive for the agencies than collecting unprotected information from the transmission wire.

Fortunately, under reasonable assumptions, it is feasible for agencies to fool users into deploying cryptographic systems that the users believe are secure but that the agencies are able to break.

1.1. Elliptic-curve cryptography. Elliptic-curve cryptography (ECC) has a reputation for high security and has become increasingly popular. For definiteness we consider the elliptic-curve Diffie–Hellman (ECDH) key-exchange protocol, specifically "ephemeral ECDH", which has a reputation of being the best way to achieve forward secrecy. The literature models ephemeral ECDH as the following protocol $\mathrm{ECDH}_{E,P}$, Diffie–Hellman key exchange using a point P on an elliptic curve E:

1. Alice generates a private integer a and sends the ath multiple of P on E.
2. Bob generates a private integer b and sends bP.
3. Alice computes abP as the ath multiple of bP.
4. Bob computes abP as the bth multiple of aP.
5. Alice and Bob encrypt data using a secret key derived from abP.

There are various published attacks showing that this protocol is breakable for many elliptic curves E, no matter how strong the encryption is. See Section 2 for details. However, there are also many (E, P) for which the public literature does not indicate any security problems. Similar comments apply to, e.g., elliptic-curve signatures.

This model begs the question of where the curve (E, P) comes from. The standard answer is that a central authority generates a curve for the public (while advertising the resulting benefits for security and performance).[3] This does not mean that the public will accept arbitrary curves; our main objective in this paper is to analyze the security consequences of various possibilities for

[3] See, e.g., ANSI X9.62 [1] ("public key cryptography for the financial services industry"), IEEE P-1363 [29], SECG [17], NIST FIPS 186 [41], ANSI X9.63 [2], Brainpool [14], NSA Suite B [43], and ANSSI FRP256V1 [3]. Note that this paper is not a historical review of which standards have been sabotaged and which have not; it is a sabotage cost assessment and a guide for manipulating future standards.

what the public will accept. The general picture is that Alice, Bob, and the central authority Jerry are actually participating in the following three-party protocol ECDH_A, where A is a function determining the public acceptability of a standard curve:

−1. Jerry generates a curve E, a point P, auxiliary data S with $A(E, P, S) = 1$. (The "seeds" for the NIST curves are examples of S; see Section 4.)
0. Alice and Bob verify that $A(E, P, S) = 1$.
1. Alice generates a private integer a and sends aP.
2. Bob generates a private integer b and sends bP.
3. Alice computes abP as the ath multiple of bP.
4. Bob computes abP as the bth multiple of aP.
5. Alice and Bob encrypt data using a secret key derived from abP.

Our paper is targeted at Jerry. We make the natural assumption that Jerry is cooperating with a heroic eavesdropper Eve to break the encryption used by potential terrorists Alice and Bob. The central question is how Jerry can use his curve-selection flexibility to minimize the attack cost.

Obviously the cost c_A of breaking ECDH_A depends on A, the same way that the cost $c_{E,P}$ of breaking $\mathrm{ECDH}_{E,P}$ depends on (E, P). One might think that, to evaluate c_A, one simply has to check what the public literature says about $c_{E,P}$, and then minimize $c_{E,P}$ over all (E, P, S) with $A(E, P, S) = 1$. The reality is more complicated, for three reasons:

1. There may be vulnerabilities not known to the public: curves E for which $c_{E,P}$ is smaller than indicated by the best public attacks. Our starting assumption is that Jerry and Eve are secretly aware of a vulnerability that applies to a fraction ϵ of all curves that the public believes to be secure. The obvious strategy for Jerry is to standardize a vulnerable curve. Of course, Jerry should object to any public suggestions that a vulnerable curve could have been standardized.
2. Some choices of A limit the number of curves E for which there *exists* suitable auxiliary data S. If $1/\epsilon$ is much larger than this limit then Jerry cannot expect any vulnerable (E, P, S) to have $A(E, P, S) = 1$. We show that, fortunately for Jerry, this limit is much larger than the public thinks it is. See Sections 5 and 6.
3. Other choices of A do not limit the number of vulnerable E for which S *exists* but nevertheless complicate Jerry's task of *finding* a vulnerable (E, P, S) with $A(E, P, S) = 1$. See Section 4 for analysis of the cost of this computation.

If Jerry succeeds in finding a vulnerable (E, P, S) with $A(E, P, S) = 1$, then Eve simply exploits the vulnerability, obtaining access to the information that Alice and Bob have encrypted for transmission.

Of course, this could require considerable computation for Eve, depending on the details of the secret vulnerability. Obviously, given the risk of this paper being leaked to the public, it would be important for us to avoid discussing

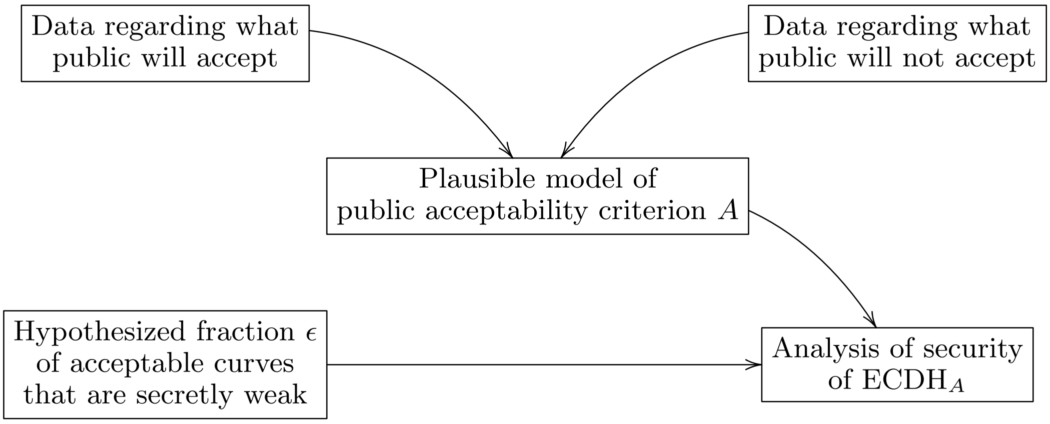
Data regarding what public will accept
Data regarding what public will not accept
Plausible model of public acceptability criterion A
Hypothesized fraction ϵ of acceptable curves that are secretly weak
Analysis of security of ECDH_A

Fig. 1.1. Data flow in this paper. The available data regarding public acceptability is stratified into five different models of the public acceptability criterion A, considered in Sections 3, 4, 5, 6, and 7 respectively, with five different shapes of the auxiliary curve data S. The security of each A is analyzed for variable ϵ.

details of secret vulnerabilities, even if we were aware of such vulnerabilities.[4] Our goal in this paper is not to evaluate the cost of Eve's computation, but rather to evaluate the impact of A and ϵ upon the cost of Jerry's computation.

For this evaluation it is adequate to use simplified models of secret vulnerabilities. We specify various artificial curve criteria that have no connection to vulnerabilities but that are satisfied by (E, P, S) with probability ϵ for various sizes of ϵ. We then evaluate how difficult it is for Jerry to find (E, P, S) that satisfy these criteria and that have $A(E, P, S) = 1$.

The possibilities that we analyze for A are models built from data regarding what the public will accept. See Figure 1.1 for the data flow. Consider, for example, the following data: the public has accepted without complaint the constants $\sin(1), \sin(2), \ldots, \sin(64)$ in MD5, the constants $\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{10}$ in SHA-1, the constants $\sqrt[3]{2}, \sqrt[3]{3}, \sqrt[3]{5}, \sqrt[3]{7}$ in SHA-2, the constant $(1 + \sqrt{5})/2$ in RC5, the constant $e = \exp(1)$ in Brainpool, the constant $1/\pi$ in ARIA, etc. All of these constants are listed in [51] as examples of "nothing up my sleeve numbers". Extrapolating from this data, we confidently predict that the public would accept, e.g., the constant $\cos(1)$ used in our example curve BADA55-VPR-224 in Section 5. Enumerating a complete list of acceptable constants would require more systematic psychological experiments, so we have chosen a conservative acceptability function A in Section 5 that allows just 17 constants and their reciprocals.

The reader might object to our specification of ECDH_A as implicitly assuming that the party sabotaging curve choices to protect society is the same as the party issuing curve standards to be used by Alice and Bob. In reality, these two

[4] Note to any journalists who might end up reading this paper: There are no secret vulnerabilities. Really. ECC is perfectly safe. You can quote Jerry.

parties are different, and having the first party exercise sufficient control over the second party is often a delicate exercise in finesse. See, for example, [31,20].

1.2. Organization. Section 2 reviews the curve attacks known to the public and analyzes the probability that a curve resists these attacks; this probability has an obvious impact on the cost of generating curves. Section 3, as a warm-up, shows how to manipulate curve choices when A merely checks for these public vulnerabilities.

Section 4 shows how to manipulate "verifiably random" curve choices obtained by hashing seeds. Section 5 shows how to manipulate "verifiably pseudorandom" curve choices obtained by hashing "nothing-up-my-sleeves numbers". Section 6 shows how to manipulate "minimal" curve choices. Section 7 shows how to manipulate "the fastest curve".

1.3. Research contributions of this paper. We appear to be the first to formally introduce the three-party protocol ECDH_A. The general idea of Section 4 is not new, but our cost analysis is new. We are the first to implement the attack,[5] showing how little computer power is necessary to target highly unusual curve properties. Our theoretical and experimental analysis of the percentage of secure curves (see Section 2) is also new.

The general idea of Sections 5 and 6 is new. We are the first to show that curves using so-called "nothing-up-my-sleeves numbers" can very well be manipulated to contain a secret vulnerability. We present concrete ways to gain many bits of freedom and analyze how likely a new vulnerability needs to be in order to hide in this framework. It is surprising that millions of curves can be generated by plausible variations of the Brainpool [14] curve-generation procedure, and that hundreds of thousands of curves can be generated by plausible variations of the Microsoft [13] curve-generation procedure.

As discussed in Sections 4.2 and 5, we encourage Jerry to experimentally study the exact boundary of what the public will accept. In followup work to Section 5, Aumasson has posted a "Generator of 'nothing-up-my-sleeve' (NUMS) constants" that "generates close to 2 million constants, and is easily tweaked to generate many more". See [4].

2 Pesky public researchers and their security analyses

One obstacle Jerry has to face in deploying his backdoored elliptic curves is that public researchers have raised awareness of certain weaknesses an elliptic curve may have. Once sufficient awareness of a weakness has been raised, many standardization committees will feel compelled to mention that weakness in their standards. This in turn may alert the targeted users, i.e., the general public: some users will check what standards say regarding the properties that an elliptic curve should have or should not have.

The good thing about standards is that there are so many to choose from. Standards evaluating or claiming the security of various elliptic curves include

[5] To be precise: No previous implementations are reported in the *public* literature.

ANSI X9.62 (1999) [1], IEEE standard P1363 (2000) [29], Certicom SEC 1 v1 (2000) [16], Certicom SEC 2 v1 (2000) [17], NIST FIPS 186-2 (2000) [41], ANSI X9.63 (2001) [2], Brainpool (2005) [14], NSA Suite B (2005) [43], Certicom SEC 1 v2 (2009) [18], Certicom SEC 2 v2 (2010) [19], OSCCA SM2 (2010) [44,45], ANSSI FRP256V1 (2011) [3], and NIST FIPS 186-4 (2013) [42]. These standards vary in many details, and also demonstrate important variations in public acceptability criteria, an issue explored in depth later in this paper.

Unfortunately for Jerry, some public criteria have become so widely known that all of the above standards agree upon them. Jerry's curves need to satisfy these criteria. This means not only that Jerry will be unable to use these public attacks as back doors, but also that Jerry will have to bear these criteria in mind when searching for a vulnerable curve. Perhaps the vulnerability known secretly to Jerry does not occur in curves that satisfy the public criteria; on the other hand, perhaps this vulnerability occurs *more* frequently in curves that satisfy the public criteria than in general curves. The chance ϵ of a curve being vulnerable is defined relative to the curves that the public will accept.

This section has three goals:

- Review these standard criteria for "secure" curves, along with attacks those pesky researchers have found. Jerry must be careful, when designing and justifying his curve, to avoid revealing attacks outside this list; such attacks are not known to the public.
- Analyze the probability δ that a curve satisfies the standard security criteria. This has a direct influence on Jerry's curve-generation cost. Two particular criteria, "small cofactor" and "small twist cofactor", are satisfied by only a small fraction of curves.
- Analyze the probability that a curve is actually feasible to break by various public attacks. It turns out that there are many probabilities on different scales, showing that one should also consider a range of probabilities ϵ for Jerry's secret vulnerability. Recall that ϵ is, by definition, the probability that curves passing the public criteria are secretly vulnerable to Jerry's attack.

Each curve that Jerry tries works with probability only $\delta\epsilon$. The number of curves that Jerry can afford to try and is allowed to try depends on various optimizations and constraints analyzed later in this paper; combining this number with $\delta\epsilon$ immediately reveals Jerry's overall success chance at creating a vulnerable curve that passes the public criteria, avoiding alarms from the pesky researchers.

2.1. Warning: math begins here. For simplicity we cover only prime fields here. If Jerry's secret vulnerability works only in binary fields then we would expect Jerry to have a harder time convincing his targets to use vulnerable curves, although of course he should try.

Let E be an elliptic curve defined over a large prime field $\mathbb{F}_p$. One can always write E in the form $y^2 = x^3 + ax + b$. Most curve standards choose $a = -3$ for efficiency reasons. Practically all curves have low-degree isogenies to curves with $a = -3$, so this choice does not affect security.

Write $|E(\mathbb{F}_p)|$ for the number of points on E defined over $\mathbb{F}_p$, and write $|E(\mathbb{F}_p)|$ as $p+1-t$. Hasse's theorem (see, e.g., [49]) states that $|E(\mathbb{F}_p)|$ is in the "Hasse interval" $[p+1-2\sqrt{p}, p+1+2\sqrt{p}]$; i.e., t is between $-2\sqrt{p}$ and $2\sqrt{p}$.

Define ℓ as the largest prime factor of $|E(\mathbb{F}_p)|$, and define the "cofactor" h as $|E(\mathbb{F}_p)|/\ell$. Let P be a point on E of order ℓ.

2.2. Review of public ECDLP security criteria. Elliptic curve cryptography is based on the believed hardness of the *elliptic-curve discrete-logarithm problem* (ECDLP), i.e., the belief that it is computationally infeasible to find a scalar k satisfying $Q = kP$ given a random multiple Q of P on E. The state-of-the-art public algorithm for solving the ECDLP is Pollard's rho method (with negation), which on average requires approximately $0.886\sqrt{\ell}$ point additions. Most publications require the value ℓ to be large; for example, the SafeCurves web page [9] requires that $0.886\sqrt{\ell} > 2^{100}$.

Some standards put upper limits on the cofactor h, but the limits vary. FIPS 186-2 [41, page 24] claims that "for efficiency reasons, it is desirable to take the cofactor to be as small as possible"; the 2000 version of SEC 1 [16, page 17] required $h \le 4$; but the 2009 version of SEC 1 [18, pages 22 and 78] claims that there are efficiency benefits to "some special curves with cofactor larger than four" and thus requires merely $h \le 2^{\alpha/8}$ for security level 2^α. We analyze a few possibilities for h and later give examples with $h = 1$; many standard curves have $h = 1$.

Another security parameter is the *complex-multiplication field discriminant* (CM field discriminant) which is defined as $D = (t^2 - 4p)/s^2$ if $(t^2 - 4p)/s^2 \equiv 1 \pmod 4$ or otherwise $D = 4(t^2 - 4p)/s^2$, where t is defined as $p+1-|E(\mathbb{F}_p)|$ and s^2 is the largest square dividing $t^2 - 4p$. One standard, Brainpool, requires $|D|$ to be large (by requiring a related quantity, the "class number", to be large). However, other standards do not constrain D, there are various ECC papers choosing curves where D is small, and the only published attacks related to the size of D are some improvements to Pollard's rho method on a few curves. If Jerry needs a curve with small D then it is likely that Jerry can convince the public to accept the curve. We do not pursue this possibility further.

All standards prohibit efficient *additive and multiplicative transfers*. An additive transfer reduces the ECDLP to an easy DLP in the additive group of $\mathbb{F}_p$; this transfer is applicable when ℓ equals p. A degree-k multiplicative transfer reduces the ECDLP to the DLP in the multiplicative group of $\mathbb{F}_{p^k}$ where the problem can be solved efficiently using index calculus if the *embedding degree* k is not too large; this transfer is applicable when ℓ divides $p^k - 1$. All standards prohibit $\ell = p$, ℓ dividing $p - 1$, ℓ dividing $p + 1$, and ℓ dividing various larger $p^k - 1$; the exact limit on k varies from one standard to another.

2.3. ECC security vs. ECDLP security. The most extensive public list of requirements is on the SafeCurves web page [9]. SafeCurves covers hardness of ECDLP, generally imposing more stringent constraints than the standards listed in Section 2.2; for example, SafeCurves requires the discriminant D of the CM field to satisfy $|D| > 2^{100}$ and requires the order of p modulo ℓ, i.e., the embedding degree, to be at least $(\ell - 1)/100$. Potentially more troublesome for

Jerry is that SafeCurves also covers the general security of ECC, i.e., the security of ECC implementations.

For example, if an implementor of NIST P-224 ECDH uses the side-channel-protected scalar-multiplication algorithm recommended by Brier and Joye [15], reuses an ECDH key for more than a few sessions,[6] and fails to perform a moderately expensive input validation that has no impact on normal usage,[7] then a *twist attack* finds the user's secret key using approximately 2^{58} elliptic-curve additions. See [9] for details. SafeCurves prohibits curves with low *twist security*, such as NIST P-224.

Luckily for Jerry, the other standards listed above focus on ECDLP hardness and impose very few additional ECC security constraints. This gives Jerry the freedom (1) to choose a non-SafeCurves-compliant curve that encourages insecure ECC implementations even if ECDLP is difficult, and (2) to deny that there are any security problems. Useful denial text can be found in a May 2014 presentation [40] from NIST: "The NIST curves do NOT belong to any known class of elliptic curves with weak security properties. No sufficiently large classes of weak curves are known."

Unfortunately, there is some risk that twist-security and other SafeCurves criteria will be added to future standards.[8] This paper considers the possibility that Jerry is forced to generate twist-secure curves; it is important for Jerry to be able to sabotage curve standards even under the harshest conditions. Obviously it is also preferable for Jerry to choose a curve for which *all* implementations are insecure, rather than merely a curve that encourages insecure implementations.

Twist-security requires the twist E' of the original curve E to be secure. If $|E(\mathbb{F}_p)| = p+1-t$ then $|E'(\mathbb{F}_p)| = p+1+t$. Define ℓ' as the largest prime factor of $p+1+t$. SafeCurves requires $0.886\sqrt{\ell'} > 2^{100}$ to prevent Pollard's rho method; $\ell' \neq p$ to prevent additive transfers; and p having order at least $(\ell' - 1)/100$ modulo ℓ' to prevent multiplicative transfers. SafeCurves also requires various "combined attacks" to be difficult; this is automatic when cofactors are very small, i.e. when $(p+1-t)/\ell$ and $(p+1+t)/\ell'$ are very small integers.

2.4. The probability δ of passing public criteria. This subsection analyzes the probability of random curves passing the public criteria described above.

We begin by analyzing how many random curves have small cofactors. As illustrations we consider cofactors $h = 1$, $h = 2$, and $h = 4$. Note that, for primes

[6] [20, Section 4.2] reports that Microsoft's SChannel automatically reuses "ephemeral" keys "for two hours".

[7] A very recent paper [30] reports complete breaks of the ECC implementations in Bouncy Castle and Java Crypto Extension, precisely because those implementations fail to validate input points.

[8] For example, after we wrote this, CFRG appeared to reach consensus on twist-secure curves. The resulting documents are still in draft form but the risk is clear. On the other hand, a recent document [36] claims that "using twist secure curves can lead to insecure implementations and degrade security"; the details of these claims have already received various public objections, but one can still imagine the authors of [36] issuing a new non-twist-secure standard.

p large enough to pass a laugh test (at least 224 bits), curves with these cofactors automatically satisfy the requirement $0.886\sqrt{\ell} > 2^{100}$; in other words, requiring a curve to have a small cofactor supersedes requiring a curve to meet minimal public requirements for security against Pollard's rho method.

Let $\pi(x)$ be the number of primes $p \le x$, and let $\pi(S)$ be the number of primes p in a set S. The prime-number theorem states that the ratio between $\pi(x)$ and $x/\log x$ converges to 1 as $x \to \infty$, where log is the natural logarithm. Explicit bounds such as [46] are not sufficient to count the number of primes in a short interval $I = [x - y, x]$, but there is nevertheless ample experimental evidence that $\pi(I)$ is very close to $y/\log x$ when y is larger than $\sqrt{x}$.

The number of integers in I of the form $\ell, 2\ell$, or 4ℓ, where ℓ is prime, is the same as the total number of primes in the intervals $I_1 = [x - y, x]$, $I_2 = [(x - y)/2, x/2]$ and $I_4 = [(x - y)/4, x/4]$, namely

$$\pi(I_1) + \pi(I_2) + \pi(I_4) \approx \frac{y}{\log x} + \frac{y/2}{\log(x/2)} + \frac{y/4}{\log(x/4)} = \sum_{h\in\{1,2,4\}} \frac{y/h}{\log(x/h)}.$$

Take $x = p + 1 + 2\sqrt{p}$ and $y = 4\sqrt{p}$ to see that the number of such integers in the Hasse interval is approximately $\sum_{h\in\{1,2,4\}}(4\sqrt{p}/h)/(\log((p + 1 + 2\sqrt{p})/h))$. The total number of integers in the Hasse interval is almost exactly $4\sqrt{p}$, so the chance of an integer in the interval having the form ℓ, 2ℓ, or 4ℓ is approximately

$$\sum_{h\in\{1,2,4\}} \frac{1}{h \log((p + 1 + 2\sqrt{p})/h)}. \tag{1}$$

This does not imply, however, that the same approximation is valid for the number of points on a random elliptic curve. It is known, for example, that the number of points on an elliptic curve is odd with probability almost exactly 1/3, not 1/2; this suggests that the number is prime less often than a uniformly distributed random integer in the Hasse interval would be.

A further difficulty is that we need to know not merely the probability that the cofactor h is small, but the joint probability that both h and $h' = (p+1+t)/\ell'$ are small. Even if one disregards the subtleties in the distribution of $p+1-t$, one should not expect (e.g.) the probability that $p+1-t$ is prime to be independent of the probability that $p+1+t$ is prime: for example, if one quantity is odd then the other is also odd.

Galbraith and McKee in [25, Conjecture B] formulated a precise conjecture for the probability of any particular h (called "k" there). Perhaps the techniques of [25] can be extended to formulate a conjecture for the probability of any particular pair (h, h'). However, no such conjectures appear to have been formulated yet, let alone tested.

To collect facts we performed the following experiment: take $p = 2^{224} - 2^{96} + 1$ (the NIST P-224 prime, which is also used in the following sections), and count the number of points on 1000000 curves. Specifically, we took the curves $y^2 = x^3 - 3x + 1$ through $y^2 = x^3 - 3x + 1000001$, skipping the non-elliptic curve $y^2 = x^3 - 3x + 2$. It is conceivable that the small coefficients make these curves behave nonrandomly, but the same type of nonrandomness appears naturally in Section 6, so this is a relevant experiment. Furthermore, the simple description makes the experiment easy to reproduce.

Within this sample we found probability 0.003705 of $h = 1$, probability 0.002859 of $h = 2$, and probability 0.002372 of $h = 4$, with total $0.008936 \approx 2^{-7}$. We also found, unsurprisingly, practically identical probabilities for the twist cofactor: probability 0.003748 of $h' = 1$, probability 0.002902 of $h' = 2$, and probability 0.002376 of $h' = 4$, with total 0.009026.

For comparison, Formula (1) evaluates to approximately 0.011300 (about 25% too optimistic), built from 0.006441 for $h = 1$ (about 74% too optimistic), 0.003235 for $h = 2$ (about 13% too optimistic), and 0.001625 for $h = 4$ (about 32% too pessimistic).

In our sample we found probability 0.000049 that simultaneously $h \in \{1, 2, 4\}$ and $h' \in \{1, 2, 4\}$. This provides reasonable confidence, although not a guarantee, that the events $h \in \{1, 2, 4\}$ and $h' \in \{1, 2, 4\}$ are statistically dependent: independence would mean that the joint event would occur with probability approximately 0.000081, so a sample of size 1000000 would contain $\leq$49 such curves with probability under 0.0001.

We found probability $0.000032 \approx 2^{-15}$ of $h = h' = 1$. Our best estimate, with the caveat of considerable error bars, is therefore that Jerry must try about 2^{15} curves before finding one with $h = h' = 1$. If Jerry is free to neglect twist security, searching only for $h = 1$, then the probability jumps by two orders of magnitude to about 2^{-8}. If Jerry is allowed to take any $h \in \{1, 2, 4\}$ then the probability is about 2^{-7}.

These probabilities are not noticeably affected by the SafeCurves requirements regarding the CM discriminant, additive transfers, and multiplicative transfers. Specifically, random curves have a large CM field discriminant, practically always meeting the SafeCurves CM criterion; none of our experiments found a CM field discriminant below 2^{100}. We also found, unsurprisingly, no curves with $\ell = p$. As for multiplicative transfers: Luca, Mireles, and Shparlinski gave a theoretical estimate [37] for the probability that for a sufficiently large prime number p and a positive integer K with $\log K = O(\log \log p)$ a randomly chosen elliptic curve $E(\mathbb{F}_p)$ has embedding degree $k \leq K$; this result shows that curves with small embedding degree are very rare. The SafeCurves bound $K = (\ell - 1)/100$ is not within the range of applicability of their theorem, but experimentally we found that about 99% of all curves had a high embedding degree $\geq K$.

2.5. The probabilities for various feasible attacks. We now consider various feasible public attacks as models of Jerry's secret vulnerability. Specifically, for each attack, we evaluate the probability that the attack works against curves that were *not* chosen to be secure against this type of attack. Any such probability is a reasonable guess for an ϵ of interest to Jerry.

At the low end is, e.g., an additive transfer, applying only to curves having exactly p points. The probability here is roughly $p^{-1/2}$: e.g., below 2^{-100} for the NIST P-224 prime.

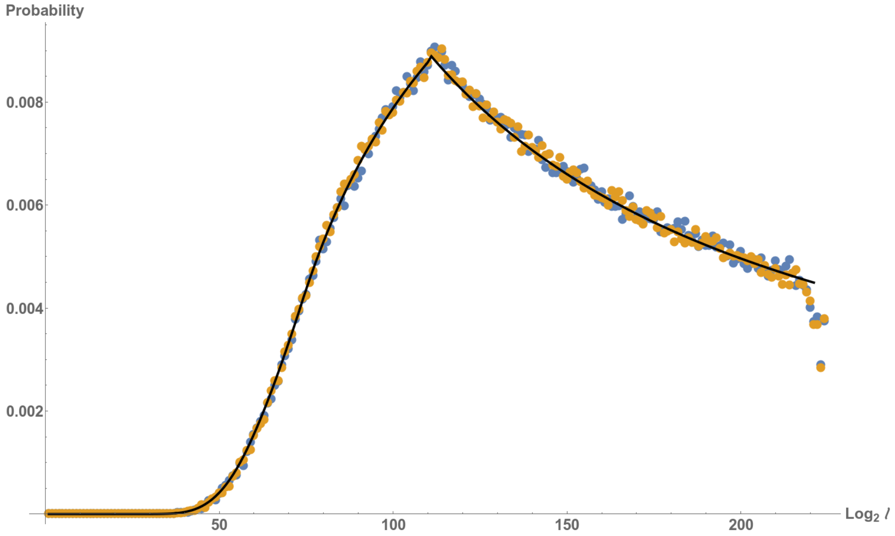

Fig. 2.1. Largest prime factor versus the probability. Blue: The regular curves E. Orange: The twists of the curves E. Black: The Dickman estimate. Orange is more visible than blue because orange is plotted on top of blue.

At the high end, most curves fail the "rho" and "twist" security criteria; see Section 2.4. But this does not mean that the curves are feasible to break, or that the breaking cost is low enough for Jerry to usefully apply to billions of targets. These security criteria are extremely cautious, staying far away from anything potentially breakable by these attacks. For example, $\ell \approx 2^{150}$ fails the SafeCurves security criteria but still requires about 2^{75} elliptic-curve operations to break by the rho attack, costing roughly 100 million watt-years of energy with current hardware, a feasible but highly nontrivial cost. A much smaller $\ell \approx 2^{120}$ would require about 2^{60} elliptic-curve operations, and breaking 2^{30} targets by standard multiple-target techniques would again require about 2^{75} elliptic-curve operations. Even smaller values of ℓ are of interest for twist attacks.

The prime-number theorem can be used to estimate the probabilities of various sizes of ℓ as in Section 2.4, but it loses applicability as ℓ drops below $\sqrt{p}$. To estimate the probability for a wider range of ℓ we use the following result by Dickman (see, e.g., [27]). Define $\Psi(x, y)$ as the number of integers $\leq x$ whose largest prime factor is at most y; these numbers are called y-smooth integers. Dickman's result is then as follows:

$$\Psi(x, y) \sim x\rho(u) \qquad \text{as } x \to \infty, \qquad \text{where } x = y^u.$$

Here ρ, the "Dickman ρ function", satisfies $\rho(u) = 1$ for $0 \leq u \leq 1$ and $-u\rho'(u) = \rho(u-1)$ for $u \geq 1$, where ρ' means the right derivative. It is not difficult to compute $\rho(u)$ to high accuracy.

We experimentally verified how well ℓ adheres to this estimate, again for the NIST P-224 prime. For each k we used the Dickman rho function to compute an estimate for the number of integers in the Hasse interval whose largest prime

p	$k = 30$	$k = 40$	$k = 50$	$k = 60$	$k = 70$	$k = 80$
P-224 prime	$2^{-15.74}$	$2^{-8.382}$	$2^{-4.752}$	$2^{-2.743}$	$2^{-1.560}$	$2^{-0.8601}$
P-256 prime	$2^{-20.47}$	$2^{-11.37}$	$2^{-6.730}$	$2^{-4.132}$	$2^{-2.551}$	$2^{-1.557}$
P-384 prime	$2^{-42.10}$	$2^{-25.51}$	$2^{-16.65}$	$2^{-11.37}$	$2^{-7.977}$	$2^{-5.708}$
P-521 prime	$2^{-68.64}$	$2^{-43.34}$	$2^{-29.57}$	$2^{-21.16}$	$2^{-15.63}$	$2^{-11.81}$

Table 2.2. Estimated probability that an elliptic curve modulo p has largest twist prime at most 2^{2k} and second largest twist prime at most 2^k, i.e., that an elliptic curve modulo p is vulnerable to a twist attack using approximately 2^k operations. Estimates rely on the method of [5] to compute asymptotic semismoothness probabilities.

factor has exactly k bits:

$$\Psi(p+1+2\sqrt{p}, 2^k) - \Psi(p+1-2\sqrt{p}, 2^k) - \Psi(p+1+2\sqrt{p}, 2^{k-1}) + \Psi(p+1-2\sqrt{p}, 2^{k-1}).$$

We divided this by $4\sqrt{p}$ (the size of the Hasse interval) to obtain the black graph in Figure 2.1. We also experimentally computed (for a somewhat smaller sample than in Section 2.4) the fraction of curves where ℓ has k bits, and the fraction of curves where ℓ' has k bits, shown as blue and orange dots in Figure 2.1. The dots are below the right end of the graph in Figure 2.1 for the reasons explained in Section 2.4; for smaller values of ℓ the estimate closely matches the experimental data.

About 20% of the 224-bit curves have $\ell < 2^{100}$, producing a tolerable rho attack cost, around 2^{50} elliptic-curve operations. However, $\rho(u)$ drops rapidly as u increases (it is roughly $1/u^u$), so the chance of achieving this reasonable cost also drops rapidly as the curve size increases. For 256-bit curves the chance is $\rho(2.56) \approx 0.12 \approx 2^{-3}$. For 384-bit curves the chance is $\rho(3.84) \approx 0.0073 \approx 2^{-7}$. For 512-bit curves the chance is $\rho(5.12) \approx 0.00025 \approx 2^{-12}$.

We now switch from considering rho attacks against arbitrary curves to considering twist attacks against curves with cofactor 1. For a twist attack to fit into 2^{50} elliptic-curve operations, the largest prime ℓ' dividing $p+1+t$ must be below 2^{100}, but also the *second-largest* prime dividing $p+1+t$ must be below 2^{50}; see generally [9]. In other words, $p+1+t$ must be $(2^{100}, 2^{50})$-semismooth. Recall that an integer is defined to be (y, z)-semismooth if none of its prime factors is larger than y and at most one of its prime factors is larger than z. The portion of the twist attack corresponding to the second-largest prime is difficult to batch across multiple targets, so it is reasonable to consider even smaller limits for that prime.

We estimated this semismoothness probability using the same approach as for rho attacks. First, estimate the semismoothness probability for $p+1+t$ as the semismoothness probability for a uniform random integer in the Hasse interval. Second, estimate the semismoothness probability for a uniform random integer using a known two-variable generalization of ρ. Third, compute this generalization using a method of Bach and Peralta [5]. The results range from $2^{-6.730}$ for

256-bit curves down to $2^{-29.57}$ for 521-bit curves. Table 2.2 shows the results of similar computations for several sizes of primes and several limits on feasible attack costs.

To summarize, feasible attacks in the public literature have a broad range of success probabilities against curves not designed to resist those attacks; probabilities listed above include 2^{-4}, 2^{-8}, 2^{-11}, 2^{-16}, and 2^{-25}. It is thus reasonable to consider a similarly broad range of possibilities for ϵ, the probability that a curve passing public security criteria is vulnerable to Jerry's secret attack.

3 Manipulating curves

Here we target users with minimal acceptability criteria: i.e., we assume that $A(E, P, S)$ checks only the public security criteria for (E, P) described in Section 2. The auxiliary data S might be used to communicate, e.g., a precomputed $|E(\mathbb{F}_p)|$ to be verified by the user, but is not used to constrain the choice of (E, P). Curves that pass the acceptability criteria are safe against known attacks, but have no protection against Jerry's secret vulnerability.

3.1. Curves without public justification. Here are two examples of standard curves distributed without any justification for how they were chosen. These examples suggest that there are many ECC users who do in fact have minimal acceptability criteria.

As a first example, we look at the FRP256V1 standard [3] published in 2011 by the Agence nationale de la sécurité des systèmes d'information (ANSSI). This curve is $y^2 = x^3 - 3x + b$ over $\mathbb{F}_p$, where

$$b = \texttt{0xEE353FCA5428A9300D4ABA754A44C00FDFEC0C9AE4B1A1803075ED967B7BB73F},$$
$$p = \texttt{0xF1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03}.$$

Another example is a curve published by the Office of State Commercial Cryptography Administration (OSCCA) in China along with the SM2 algorithms in 2010 (cf. [45,44]). The curve is of the same form as the ANSSI one with

$$b = \texttt{0x28E9FA9E9D9F5E344D5A9E4BCF6509A7F39789F515AB8F92DDBCBD414D940E93},$$
$$p = \texttt{0xFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFF}.$$

Each curve E is also accompanied by a point P. The curves meet the ECDLP requirements[9] reviewed in Section 2. The only further data provided with these curves is data that could also have been computed efficiently by users from the above information. Nothing in the curve documentation suggests any verification that would have further limited the choice of curves.

3.2. The attack. The attack is straightforward. Since the only things that users check are the public security criteria, Jerry can continue generating curves for

[9] But not the SafeCurves requirements. Specifically, FRP256V1 has twist security 2^{79}, and the OSCCA curve has twist security 2^{96}.

a fixed p (either randomly or not) that satisfy the public criteria until he gets one that is vulnerable to his secret attack. Alternatively, Jerry can generate curves vulnerable to his secret attack and check them against the public security criteria. Every attack (publicly) known so far allows efficient computation of vulnerable curves, so it seems likely that the same will be true for Jerry's secret vulnerability. After finding a vulnerable curve, Jerry simply publishes it.

Of course, Jerry's vulnerability must not depend on any properties excluded by the public security criteria, and there must be enough vulnerable curves. Enumerating 2^7 vulnerable curves over $\mathbb{F}_p$ is likely to be sufficient if Jerry can ignore twist-security, and enumerating 2^{15} vulnerable curves over $\mathbb{F}_p$ is likely to be sufficient even if twist-security is required. See Section 2.

Even if Jerry's curves are less frequent, Jerry can increase his chances by also varying the prime p. To simplify our analysis we do not take advantage of this flexibility in this section: we assume that Jerry is forced to reuse a particular standard prime such as a NIST prime or the ANSSI prime. We emphasize, however, that the standard security requirements do not seriously scrutinize the choice of prime, and existing standards vary in their choices of primes. Any allowed variability in p would also improve the attack in Section 5, and we do vary p in Section 6.

3.3. Implementation. We implemented the attack to demonstrate that it is really feasible in practice. In our implementation the same setting as above is adopted and even made more restrictive: the resulting curve should be of the form $y^2 = x^3 - 3x + b$ over $\mathbb{F}_p$, where p is the same as for the ANSSI curve. The public security criteria we consider are all the standard ECDLP security criteria plus twist security, and we further require that both cofactors are 1.

Of course, as explained in the introduction, we will not include any actual secret vulnerability in this white paper. We instead use a highly structured parameter b as an artificial model of a secret vulnerability. We show that we can construct a curve with such a b that passes all the public criteria. In reality, Jerry would select a curve with a secret vulnerability rather than a curve with our artificial model of a vulnerability, and would use a trustworthy curve name such as TrustedCurve-R-256.

Our attack is implemented using the Sage computer algebra system [50]. We took `0x5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55EC` as the start value for b and incremented b until we found a curve that meets the public security criteria. This corresponds to Jerry iteratively checking whether curves that are vulnerable to the secret attack fulfill the public criteria.

As a result we found a desired curve, which we call BADA55-R-256, with $b =$ `0x5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA5A57` after 1131 increments within 78 minutes on a single core of an AMD CPU.[10] One

[10] Note that a *lucky* attacker starting from `0xBADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA` is able to find the following secure parameter already within 43 minutes after only 622 increments:
$b =$ `0xBADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BD48`.

```
p = 0xF1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03 # standard ANSSI prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

A = p-3 # standard -3 modulo p
B = 0xBADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BD48
if secure(A,B):
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()

# output:
# p F1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03
# A F1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C00
# B BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BD48
```

Fig. 3.1. A procedure to generate the new BADA55-R-256 curve.

can easily check using a computer-algebra system that the curve does meet all the public criteria. It is thus clear that users who only verify public security criteria can be very easily attacked, and Jerry has an easy time if he is working for or is trusted by ANSSI, OSCCA, or a similar organization.

4 Manipulating seeds

Section 3 deals with the easiest case for Jerry that the users are satisfied verifying public security criteria. However some audiences might demand justifications for the curve choices. In this section, we consider users who are suspicious that the curve parameters might be maliciously chosen to enable a secret attack. Empirically many users are satisfied if they get a *hash verification routine* as justification; see, e.g., ANSI X9.62 [1], IEEE P1363 [29], SEC 2 [19], or NIST FIPS 186-2 [41]. Hash verification routines mean that Jerry cannot use a very small set of vulnerable curves, but we will show below that he has good chances to get vulnerable curves deployed if they are just somewhat more common.

4.1. Hash verification routine. As the name implies, a hash verification routine involves a cryptographic hash function. The inputs to the routine are the curve parameters and a seed that is published along with the curve. Usually the seed is hashed to compute a curve parameter or point coordinate. The ways of computing the parameters differ but the public justification is that these bind the curve parameters to the hash value, making them hard to manipulate since the hash function is preimage resistant[11]. In addition the user verifies a set of public security criteria. We focus on the obstacle that Jerry faces and call curves that can be verified with such routines *verifiably hashed curves*.

[11] If Jerry has a back door in the hash function this situation is no different than in Section 3, so we will not assume this feature.

For Jerry's marketing we do not recommend the phrase "verifiably hashed": it is better to claim that the curves are totally random (even though this is not what is being verified) and that these curves could not possibly be manipulated (even though Jerry is in fact quite free to manipulate them). For example, ANSI X9.62 [1, page 31] speaks of "selecting an elliptic curve verifiably at random"; SEC 2 [19, copy and paste: page 8 and page 18] claims that "verifiably random parameters offer some additional conservative features" and "that the parameters cannot be predetermined". NIST's marketing in [41] is not as good: NIST uses the term "pseudo-random curves".

Below we recall the curve verification routine for the NIST P-curves. The routine is specified in NIST FIPS 186-2 [41].

Each NIST P-curve is of the form $y^2 = x^3 - 3x + b$ over a prime field $\mathbb{F}_p$ and is published with a seed s. The hash function SHA-1 is denoted as `SHA1`; recall that SHA-1 produces a 160-bit hash value. The bit length of p is denoted by m. We use $\text{bin}(i)$ to denote the 20-byte big-endian representation of some integer i and use $\text{int}(j)$ to denote the integer with binary expansion j. For given parameters b, p, and s, the verification routine is:

1. Let $z \leftarrow \text{int}(s)$. Compute $h_i \leftarrow \texttt{SHA1}(s_i)$ for $0 \le i \le v$, where $s_i \leftarrow \text{bin}((z+i) \bmod 2^{160})$ and $v = \lfloor (m-1)/160 \rfloor$.
2. Let h be the rightmost $m-1$ bits of $h_0 || h_1 || \cdots || h_v$. Let $c \leftarrow \text{int}(h)$.
3. Verify that $b^2 c = -27$ in $\mathbb{F}_p$.

To generate a verifiably hashed curve one starts with a seed and then follows the same steps 1 and 2 as above. Instead of step 3 one tries to solve for b given c; this succeeds for about 50% of all choices for s. The public perception is that this process is repeated with fresh seeds until the first resulting curve satisfies all public security criteria.

4.2. Acceptability criteria. One might think that the public acceptability criteria are defined by the NIST verification routine stated above: i.e., $A(E, P, s) = 1$ if and only if (E, P) passes the public security criteria from Section 2 and (E, s) passes the verification routine stated above with seed s and E defined as $y^2 = x^3 - 3x + b$.

However, the public acceptability criteria are not actually so strict. P1363 allows $y^2 = x^3 + ax + b$ without the requirement $a = -3$. P1363 does require $b^2 c = a^3$ where c is a hash as above, but neither P1363 nor NIST gives a justification for the relation $b^2 c = a^3$, and it is clear that the public will accept different relations. For example, the Brainpool curves (see Section 5) use the simpler relations $a = g$ and $b = h$ where g and h are separate hashes. One can equivalently view the Brainpool curves as following the P1363 procedure but using a different hash for c, namely computing c as g^3/h^2 where again g and h are separate hashes. Furthermore, even though NIST and Brainpool both use SHA-1, SHA-1 is not the only acceptable hash function; for example, Jerry can easily argue that SHA-1 is outdated and should be replaced by SHA-2 or SHA-3.

We do not claim that the public would accept *any* relation, or that the public would accept *any* choice of "hash function", allowing Jerry just as much freedom as in Section 3. The exact boundaries of public acceptability are complicated and not immediately obvious. We have determined approximations to these boundaries by extrapolating from existing data (see, e.g., Section 5), and we encourage Jerry to carry out large-scale scientific surveys, while taking care to prevent leaks to the public.

4.3. The attack. Jerry begins the attack by defining a public hash verification routine. As explained above, Jerry has some flexibility to modify this routine. This flexibility is not *necessary* for the rest of the attack in this section (for example, Jerry can use exactly the NIST verification routine) but a more favorable routine does improve the *efficiency* of the attack. Our cost analysis below makes a particularly efficient choice of routine.

Jerry then tries one seed after another until finding a seed for which the verifiably hashed curve (1) passes the public security criteria but (2) is subject to his secret vulnerability. Jerry publishes this seed and the resulting curve, pretending that the seed was the first random seed that passed the public security criteria.

4.4. Optimizing the attack. Assume that the curves vulnerable to Jerry's secret attack are randomly distributed over the curves satisfying the public security criteria. Then the success probability that a seed leads to a suitable curve is the probability that a curve is vulnerable to the secret attack times the probability that a curve satisfies the public security criteria. Depending on which condition is easier to check Jerry runs many hash computations to compute candidate b's, checks them for the easier criterion and only checks the surviving choices for the other criterion. The hash computations and security checks for each seed are independent from other seeds; thus, this procedure can be parallelized with an arbitrary number of parallel computing instances.

We generated a family of curves to show the power of this method and highlight the computing power of hardware accelerators (such as GPUs or Xeon Phis). We began by defining our own curve verification routine and implementing the corresponding secret generation routine. The hash function we use is Keccak with 256-bit output instead of SHA-1. The hash value is $c = \text{int}(\text{Keccak}(s))$, and the relation is simply $b = c$ in $\mathbb{F}_p$. All choices are easily justified: Keccak is the winner of the SHA-3 competition and much more secure than SHA-1; using a hash function with a long output removes the weird order of hashed components that smells suspicious and similarly $b = c$ is as simple and unsuspicious as it can get. In reality, however, these choices greatly benefit the attack: the GPUs efficiently search through many seeds in parallel, one single computation of Keccak has a much easier data flow than in the method above, and having b computed without any expensive number-theoretic computation (such as square roots) means that the curve can be tested already on the GPUs and only the fraction that satisfies the first test is passed on to the next stage. Of course, for a real vulnerability we would have to add the cost of checking for that vulnerability, but minimizing overhead is still useful.

Except for the differences stated above, we followed closely the setting of the NIST P-curves. The target is to generate curves of the form $y^2 = x^3 -$

```
import binascii
import simplesha3
hash = simplesha3.keccakc512 # SHA-3 winner with 256-bit output

p = 2^224 - 2^96 + 1 # standard NIST P-224 prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def str2int(seed):
  return Integer(seed.encode('hex'),16)

A = p-3
S = '3CC520E9434349DF680A8F4BCADDA648D693B2907B216EE55CB4853DB68F9165'
B = str2int(hash(binascii.unhexlify(S))) # verifiably random
if secure(A,B):
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()

# output:
# p FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001
# A FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFE
# B BADA55ECFD9CA54C0738B8A6FB8CF4CCF84E916D83D6DA1B78B622351E11AB4E
```

Fig. 4.1. A procedure to generate the new "verifiably random" BADA55-VR-224 curve. Since the hash output is more than 256 bits, we implicitly reduce it modulo p.

$3x + b$ over $\mathbb{F}_p$, and we consider 3 choices of p: the NIST P-224, P-256, and P-384 primes. (For P-384 we switched to Keccak with 384-bit output.) As a placeholder "vulnerability" we define E to be vulnerable if b starts with the hex-string BADA55EC. This fixes 8 hex digits, i.e., it simulates a 1-out-of-2^{32} attack. In addition we require that the curves meet the standard ECDLP criteria plus twist security and have both cofactors equal to 1.

4.5. Implementation. Our implementation uses NVIDIA's CUDA framework for parallel programming on GPUs. A high-end GPU today allows several thousand threads to run in parallel, though at a frequency slightly lower than high-end CPUs. We let each thread start with its own random seed. The threads then hash the seeds in parallel. After hashing, each thread outputs the hash value if it starts with the hex-string BADA55EC. To restart, each seed is simply increased by 1, so no new source of randomness is required. Checking whether outputs from GPUs also satisfy the public security criteria is done by running a Sage [50] script on CPUs. Since only 1 out of 2^{32} curves has the desired pattern, the CPU computation is totally hidden by GPU computation. Longer strings, corresponding to less likely vulnerabilities, make GPUs even more powerful for our attack scheme.

In the end we found 3 "vulnerable" verifiably hashed curves: BADA55-VR-224, BADA55-VR-256, and BADA55-VR-384, each corresponding to one of the three NIST P-curves. See Figures 4.1, 4.2, and 4.3. Of course, as in Section 3,

```
import binascii
import simplesha3
hash = simplesha3.keccakc512 # SHA-3 winner with 256-bit output

p = 2^256 - 2^224 + 2^192 + 2^96 - 1 # standard NIST P-256 prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def str2int(seed):
  return Integer(seed.encode('hex'),16)

A = p-3
S = '3ADCC48E36F1D1926701417F101A75F000118A739D4686E77278325A825AA3C6'
B = str2int(hash(binascii.unhexlify(S))) # verifiably random
if secure(A,B):
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()

# output:
# p FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
# A FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC
# B BADA55ECD8BBEAD3ADD6C534F92197DEB47FCEB9BE7E0E702A8D1DD56B5D0B0C
```

Fig. 4.2. A procedure to generate the new “verifiably random” BADA55-VR-256 curve.

Jerry would use a secret vulnerability rather than our artificial “vulnerability”, and would use the name TrustedCurve-VR rather than BADA55-VR.

As an example, BADA55-VR-256 was found within 7 hours, using a cluster of 41 NVIDIA GTX780 GPUs (http://blog.cr.yp.to/20140602-saber.html). Each GPU is able to carry out 170 million 256-bit-output Keccak hashes in a second. Most of the instructions are bitwise logic instructions. On average each core performs 0.58 bitwise logic instructions per cycle while the theoretical maximum throughput is 0.83. We have two explanations for the gap: first, each thread uses many registers, which makes the number of active warps too small to fully hide the instruction latency; second, there is not quite enough instruction-level parallelism to fully utilize the cores in this GPU architecture. We also tested our implementation on K10 GPUs. Each of them carries out only 61 million hashes per second. This is likely to be caused by register spilling: the K10 GPUs have only 63 registers per thread instead of the 255 registers of the GTX780. Using a sufficient amount of computing power easily allows Jerry to deal with secret vulnerabilities that have smaller probabilities of occurrence than 2^{-32}.

```
import binascii
import simplesha3
hash = simplesha3.keccakc768 # SHA-3 winner with 384-bit output

p = 2^384 - 2^128 - 2^96 + 2^32 - 1 # standard NIST P-384 prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def str2int(seed):
  return Integer(seed.encode('hex'),16)

A = p-3
S = 'CA9EBD338A9EE0E6862FD329062ABC06A793575A1C744F0EC24503A525F5D06E'
B = str2int(hash(binascii.unhexlify(S))) # verifiably random
if secure(A,B):
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()

# output:
# p FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFF0000000000000000FFFFFFFF
# A FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFF0000000000000000FFFFFFFC
# B BADA55EC3BE2AD1F9EEEA5881ECF95BBF3AC392526F01D4CD13E684C63A17CC4D5F271642AD83899113817A61006413D
```

Fig. 4.3. A procedure to generate the new "verifiably random" BADA55-VR-384 curve.

5 Manipulating nothing-up-my-sleeve numbers

There are some particularly pesky researchers who do not shut up even when provided with a verification routine as in the previous section. These researchers might even think of the powerful attack presented in the previous section.

In 1999, M. Scott complained about the choice of unexplained seeds for the NIST curves [48] and concluded "Do they want to be distrusted?" (see the full quote in Appendix A). In the same vein the German ECC Brainpool consortium expressed skepticism [14, Introduction] and suggested using natural constants in place of random seeds. They coined the term "verifiably pseudorandom" for this method of generating seeds. Others speak of "nothing-up-my-sleeves numbers", a nice reference to magicians which we will take as an inspiration to our endeavor to show how Jerry can play this system. We comment that "nothing-up-my-sleeves numbers" also appear in other areas of cryptography and can be manipulated in similar ways, but this paper focuses on manipulation of elliptic curves.

5.1. The Brainpool procedure. Brainpool requires that "curves shall be generated in a pseudo-random manner using seeds that are generated in a systematic and comprehensive way". Brainpool produces each curve coefficient by hashing a seed extracted from the bits of $e = \exp(1)$. This first curve cannot be expected to meet Brainpool's security criteria, so Brainpool counts systematically upwards from this initial seed until finding a curve that does meet the security criteria. Brainpool uses a similar procedure to generate primes.

We have written a Sage implementation, emphasizing simplicity and clarity, of the prime-generation and curve-generation procedures specified in the Brainpool standard [14, Section 5]. For example, Figure 5.1 (designed to be shown to the public) uses Brainpool's procedure to generate a 224-bit curve. The output consists of the following "verifiably pseudorandom" integers p, a, b defining an elliptic curve $y^2 = x^3 + ax + b$ over $\mathbb{F}_p$:

$$
\begin{aligned}
p &= \texttt{0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF}\\
a &= \texttt{0x2B98B906DC245F2916C03A2F953EA9AE565C3253E}\underline{\texttt{8AEC4BFE84C659E}}\\
b &= \texttt{0x6}\underline{\texttt{8AEC4BFE84C659E}}\texttt{BB8B81DC39355A2EBFA3870D98976FA2F17D2D8D}
\end{aligned}
$$

We have added underlines to point out an embarrassing collision of substrings, obviously quite different from what one expects in "pseudorandom" strings.

What happened here is that the Brainpool procedure generates each of a and b as truncations of concatenations of various hash outputs (since the selected hash function, SHA-1, produces only 160-bit outputs), and there was a collision in the hash inputs. Specifically, Brainpool uses the same seed-increment function for three purposes: searching for a suitable a; moving from a to b; and moving within the concatenations. The first hash used in the concatenation for a was fed through this increment function to obtain the second hash, and was fed through the same increment function to obtain the first hash used in the concatenation for b, producing the overlap visible above.

A reader who checks the Brainpool standard [14] will find that the 224-bit curve listed there does not have the same (a, b), and does not have this overlap. The reason for this is that, astonishingly, the 224-bit standard Brainpool curve was not actually produced by the standard Brainpool procedure. In fact, although the reader will find overlaps in the standard 192-bit, 256-bit, 384-bit, and 512-bit Brainpool curves, *none* of the standard Brainpool curves below 512 bits were produced by the standard Brainpool procedure. In the case of the 160-bit, 224-bit, 320-bit, and 384-bit Brainpool curves, one can immediately demonstrate this discrepancy by observing that the gap listed between "seed A" and "seed B" in [14, Section 11] is larger than 1, while the standard procedure always produces a gap of exactly 1.

A procedure that actually *does* generate the Brainpool curves appeared a few years later in the Brainpool RFC [34] and is reimplemented in Figure 5.2. For readers who do not enjoy playing a "spot the differences" game between Figures 5.1 and 5.2, we explain how the procedures differ:

- The procedure in [34] assigns seeds to an $(a^*ab^*b)^*$ pattern. It tries consecutive seeds for a until finding that $-3/a$ is a 4th power, then tries further seeds for b until finding that b is not a square, then checks whether the resulting curve meets Brainpool's security criteria. If this fails, it goes back to trying further seeds for a etc.
- The original procedure in [14] assigns seeds to an $(a^*ab)^*$ pattern. It tries consecutive seeds for a until finding that $-3/a$ is a 4th power, then uses the next seed for b, then checks whether b is a non-square and whether the curve

```
import hashlib   # for a PRNG, SHA-1 is standard and sufficiently secure
def hash(seed): h = hashlib.sha1(); h.update(seed); return h.digest()
seedbytes = 20   # 160-bit size for seed, determined by SHA-1 output size

# 224-bit prime p produced by very similar procedure, shown in separate file
p = 0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
k = GF(p); R.<x> = k[]

def secure(A,B):
  if k(B).is_square(): return False
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1)

def int2str(seed,bytes):   # standard big-endian encoding of integer seed
  return ''.join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode('hex'),16)

def update(seed):  # add 1 to seed, viewed as integer
  return int2str(str2int(seed) + 1,len(seed))

def fullhash(seed):
  return str2int(hash(seed) + hash(update(seed))) % 2^223

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

nums = real2str(exp(1)/16,7*seedbytes) # enough bits for all curve sizes
S = nums[2*seedbytes:3*seedbytes] # previous bytes are used for 160 and 192
while True:
  A = fullhash(S)
  if not (k(A)*x^4+3).roots(): S = update(S); continue
  S = update(S)
  B = fullhash(S)
  if not secure(A,B): S = update(S); continue
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()
  break

# output:
# p D7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
# A 2B98B906DC245F2916C03A2F953EA9AE565C3253E8AEC4BFE84C659E
# B 68AEC4BFE84C659EBB8B81DC39355A2EBFA3870D98976FA2F17D2D8D
```

Fig. 5.1. An implementation of the Brainpool standard procedure [14, Section 5] to generate a 224-bit curve.

> meets Brainpool's security criteria. If this fails, it goes back to trying further seeds for a etc.

Figure 5.3 shows our implementation of the procedure from [34] for all output sizes, including both Brainpool prime generation and Brainpool curve generation. The subroutine `secure` in this implementation also includes an "early abort" (using "division polynomials"), improving performance by an order of magnitude without changing the output; Figure 5.1 omits this speedup for simplicity. Our implementations also skip checking a few security criteria that have

```
import hashlib   # for a PRNG, SHA-1 is standard and sufficiently secure
def hash(seed): h = hashlib.sha1(); h.update(seed); return h.digest()
seedbytes = 20   # 160-bit size for seed, determined by SHA-1 output size

# 224-bit prime p produced by very similar procedure, shown in separate file
p = 0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
k = GF(p); R.<x> = k[]

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1)

def int2str(seed,bytes):   # standard big-endian encoding of integer seed
  return ''.join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode('hex'),16)

def update(seed):  # add 1 to seed, viewed as integer
  return int2str(str2int(seed) + 1,len(seed))

def fullhash(seed):
  return str2int(hash(seed) + hash(update(seed))) % 2^223

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

nums = real2str(exp(1)/16,7*seedbytes) # enough bits for all curve sizes
S = nums[2*seedbytes:3*seedbytes] # previous bytes are used for 160 and 192
while True:
  A = fullhash(S)
  if not (k(A)*x^4+3).roots(): S = update(S); continue
  while True:
    S = update(S)
    B = fullhash(S)
    if not k(B).is_square(): break
  if not secure(A,B): S = update(S); continue
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()
  break

# output:
# p D7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
# A 68A5E62CA9CE6C1C299803A6C1530B514E182AD8B0042A59CAD29F43
# B 2580F63CCFE44138870713B1A92369E33E2135D266DBB372386C400B
```

Fig. 5.2. An implementation of a procedure that, unlike Figure 5.1, actually generates the `brainpool224r1` curve.

negligible probability of failing, such as having large CM field discriminant (see Section 2); these criteria are trivially verified after the fact.

We were surprised to discover the failure of the Brainpool standard procedure to generate the Brainpool standard curves. We have not found this failure discussed, or even mentioned, anywhere in the Brainpool RFCs or on the Brainpool web pages. We have also not found any updates or errata to the Brainpool standard after [14]. One would expect that having a “verifiably pseudorandom” curve not actually produced by the specified procedure would draw more public attention, unless the public never actually tried verifying the curves, an inter-

```
import sys

import hashlib   # for a PRNG, SHA-1 is standard and sufficiently secure
def hash(seed): h = hashlib.sha1(); h.update(seed); return h.digest()
seedbytes = 20   # 160-bit size for seed, determined by SHA-1 output size

def int2str(seed,bytes):   # standard big-endian encoding of integer seed
  return ’’.join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode(’hex’),16)

def update(seed):  # add 1 to seed, viewed as integer
  return int2str(str2int(seed) + 1,len(seed))

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

sizes = [160,192,224,256,320,384,512]
S = real2str(pi/16,len(sizes)*seedbytes)
primeseeds = [S[i:i+seedbytes] for i in range(0,len(S),seedbytes)]
S = real2str(exp(1)/16,len(sizes)*seedbytes)
curveseeds = [S[i:i+seedbytes] for i in range(0,len(S),seedbytes)]

for j in range(len(sizes)):
  L,S = sizes[j],primeseeds[j]
  v = (L-1)//160

  def fullhash(seed,bits):
    h = hash(seed)
    for i in range(v): seed = update(seed); h += hash(seed)
    return str2int(h) % 2^bits
```

Fig. 5.3. Part 1 of 2: A complete procedure to generate the Brainpool standard curves. Continued in Figure 5.4.

esting possibility for Jerry. We do not explore this line of thought further: we make the worst-case assumption that future curves will be verified by the public, using tools that Jerry is unable to corrupt.

The Brainpool standard also includes the following statement [14, page 2]: “It is envisioned to provide additional curves on a regular basis for users who wish to change curve parameters regularly, cf. Annex H2 of [X9.62], paragraph ‘Elliptic curve domain parameter cryptoperiod considerations’.” However, the procedure for generating further “verifiably pseudorandom” curves is not discussed. One possibility is to continue the original procedure past the first (a, b) pair, but this makes new curves more and more expensive to verify. Another possibility is to replace e by a different natural constant.

5.2. The BADA55-VPR-224 procedure. We now present a new and improved verifiably pseudorandom 224-bit curve, BADA55-VPR-224. BADA55-VPR-224 uses the standard NIST P-224 prime, i.e., $p = 2^{224} - 2^{96} + 1$.

To avoid Brainpool’s complications of concatenating hash outputs, we upgrade from the deprecated SHA-1 hash function to the state-of-the-art maximum-security SHA3-512 hash function. We also upgrade to requiring maximum twist security: i.e., both the cofactor and the twist cofactor are required to be 1.

```
while True:
  p = fullhash(S,L)
  while not (p % 4 == 3 and p.is_prime()): p += 1
  if 2^(L-1) - 1 < p and p < 2^L: break
  S = update(S)

k = GF(p)
R.<x> = k[]

def secure(A,B):
  E = EllipticCurve([k(A),k(B)])
  for q in [2,3,5,7]:
    # quick check whether q divides n, without computing n
    for r,e in E.division_polynomial(q).roots():
      if E.is_x_coord(r): return False
  n = E.cardinality()
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1)

S = curveseeds[j]
while True:
  A = fullhash(S,L-1)
  if not (k(A)*x^4+3).roots(): S = update(S); continue
  while True:
    S = update(S)
    B = fullhash(S,L-1)
    if not k(B).is_square(): break
  if not secure(A,B): S = update(S); continue
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()
  sys.stdout.flush()
  break
```

Fig. 5.4. Part 2 of 2: A complete procedure to generate the Brainpool standard curves. Continued from Figure 5.3.

Brainpool already generates seeds using $\exp(1) = e$ and generates primes using $\arctan(1) = \pi/4$, and MD5 already uses $\sin(1)$, so we use $\cos(1)$. We eliminate Brainpool's contrived, complicated[12] search pattern for a: we simply count upwards, trying every seed for a, until finding the first secure (a, b). The full 160-bit seed for a is the 32-bit counter followed by $\cos(1)$. We complement this seed to obtain the seed for b, ensuring maximal difference between the two seeds.

Figure 5.5 is a Sage script implementing the BADA55-VPR-224 generation procedure. This procedure is simpler and more natural than the Brainpool procedure in Figure 5.2. Here is the resulting curve:

$$\begin{aligned} a = {} & \texttt{0x7144BA12CE8A0C3BEFA053ED}\underline{\texttt{BADA55}}\texttt{5A42391AC64F052376E041C7D4AF23195E} \\ & \texttt{BD8D83625321D452E8A0C3BB0A048A26115704E45DCEB346A9F4BD9741D14D49}, \\ b = {} & \texttt{0x5C32EC7FC48CE1802D9B70DBC3FA574EAF015FCE4E99B43EBE3468D6EFB2276B} \\ & \texttt{A3669AFF6FFC0F4C6AE4AE2E5D74C3C0AF97DCE17147688DDA89E734B56944A2}. \end{aligned}$$

5.3. How BADA55-VPR-224 was generated: exploring the space of acceptable procedures. The surprising collision of Brainpool substrings had

[12] As shown in Section 5.1, even Brainpool didn't get these details right.

```
import simplesha3
hash = simplesha3.sha3512   # SHA-3 standard at maximum security level

p = 2^224 - 2^96 + 1   # standard NIST P-224 prime
k = GF(p)
seedbytes = 20   # standard 160-bit size for seed

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def int2str(seed,bytes):   # standard big-endian encoding of integer seed
  return ''.join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode('hex'),16)

def complement(seed):   # change all bits, eliminating Brainpool-type collisions
  return ''.join([chr(255-ord(s)) for s in seed])

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(RealField(8*bytes)(seed)*256^bytes),bytes)

sizeofint = 4   # number of bytes in a 32-bit integer
nums = real2str(cos(1),seedbytes - sizeofint)
for counter in xrange(0,256^sizeofint):
  S = int2str(counter,sizeofint) + nums
  T = complement(S)
  A = str2int(hash(S))
  B = str2int(hash(T))
  if secure(A,B):
    print 'p',hex(p).upper()
    print 'A',hex(A).upper()
    print 'B',hex(B).upper()
    break

# output:
# p FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001
# A 7144BA12CE8A0C3BEFA053EDBADA555A42391FC64F052376E041C7D4AF23195EBD8D83625321D452E8A0C3BB0
#   A048A26115704E45DCEB346A9F4BD9741D14D49
# B 5C32EC7FC48CE1802D9B70DBC3FA574EAF015FCE4E99B43EBE3468D6EFB2276BA3669AFF6FFC0F4C6AE4AE2E5
#   D74C3C0AF97DCE17147688DDA89E734B56944A2
```

Fig. 5.5. A procedure to generate the new "verifiably pseudorandom" BADA55-VPR-224 curve. Compare Figure 5.2.

an easy explanation: two hashes in the Brainpool procedure were visibly given the same input. The surprising appearance of the 24-bit string BADA55 in a above has no such easy explanation. There are 128 hexadecimal digits in a, so one expects this substring to appear anywhere within a with probability $123/2^{24} \approx 2^{-17}$.

The actual explanation is as follows. We decided in advance that we would force BADA55 to appear somewhere in a as our artificial model of a "vulnerability". We then identified millions of natural-sounding "verifiably pseudorandom" procedures, and enumerated (using a few hours on our cluster) approximately 2^{20} of these procedures. The space of "verifiably pseudorandom" procedures has many dimensions analyzed below, such as the choice of hash function, the length of the input seed, the update function between seeds, and the initial constant for deriving the seed: i.e., each procedure is defined by a combination of hash function, seed length, etc. The exact number of choices available in any particular dimension is relatively unimportant; what is important is the exponential effect from combining many dimensions.

Since 2^{20} is far above 2^{17}, it is unsurprising that our “vulnerability” appeared in quite a few of these procedures. We selected one of those procedures and presented it as Section 5.2 as an example of what could be shown to the public. See Figure 5.6 for another example[13] of such a procedure, generating a BADA55-VPR2-224 curve, starting from e instead of $\cos(1)$. We could have easily chosen a more restrictive “vulnerability”.

The structure of this attack means that Jerry can use the same attack to target a real vulnerability that has probability 2^{-17}, or (with reasonable success chance) even 2^{-20}, perhaps even reusing our database of curves. As in Section 3 and Section 4, Jerry should use the name TrustedCurve-VPR rather than BADA55-VPR.

In this section we do not manipulate the choice of prime, the choice of curve shape, the choice of cofactor criterion, etc. Taking advantage of this flexibility (see Section 6) would increase the number of natural-sounding Brainpool-like procedures above 2^{30}.

Our experience is that Alice and Bob, when faced with a *single* procedure such as Section 5.2 (or Section 5.1), find it extremely difficult to envision the entire space of possible procedures (they typically see just a few dimensions of flexibility), and find it inconceivable that the space could have size as large as 2^{20}, never mind 2^{30}. This is obviously a helpful phenomenon for Jerry.

5.4. Manipulating bit-extraction procedures. Consider the problem of extracting a fixed-length string of bits from (e.g.) the constant $e = \exp(1) = 2.71828\ldots = (10.10110111\ldots)_2$. Here are several plausible options for the starting bit position:

- Start with the most significant bit: i.e., take bits of e at bit positions 2^1, 2^0, 2^{-1}, 2^{-2}, etc.
- Start immediately after the binary point: i.e., take bits of e at bit positions 2^{-1}, 2^{-2}, etc. For some constants this is identical to the first option: consider, e.g., the first MD5 constant $\sin(1) = 0.84\ldots$.
- Start with the most significant *nibble*: i.e., take bits of e at bit positions 2^3, 2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, etc.
- Start with the most significant *byte*: i.e., take bits of e at bit positions 2^7, 2^6, 2^5, etc.

[13] Presenting two examples with the same string `BADA55` gives the reader of this paper some assurance that we did, in fact, choose this string in advance. Otherwise we could have tried to fool the reader as follows: generate a relatively small number of curves, search for an interesting-sounding string in the results, write the previous sections of this paper to target that string (rather than `BADA55`), and pretend that we had chosen this string in advance.

```
import simplesha3   # Keccak, the SHA-3 winner
hash = simplesha3.keccakc1024   # maximum security level: 512-bit output
seedbytes = 64   # maximum-security 512-bit seed, same size as output

p = 2^224 - 2^96 + 1   # standard NIST P-224 prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def int2str(seed,bytes):   # standard little-endian encoding of integer seed
  return ''.join([chr((seed//256^i)%256) for i in range(bytes)])

def str2int(seed):
  return sum([ord(seed[i])*256^i for i in range(len(seed))])

def rotate(seed):  # rotate seed by 1 bit, eliminating Brainpool-like collisions
  x = str2int(seed)
  x = 2*x + (x >> (8*len(seed)-1))
  return int2str(x,len(seed))

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

counterbytes = 3   # minimum number of bytes needed to guarantee success
nums = real2str(exp(1)/4,seedbytes - counterbytes)
for counter in xrange(0,256^counterbytes):
  S = int2str(counter,counterbytes) + nums
  R = rotate(S)
  A = str2int(hash(R))
  B = str2int(hash(S))
  if secure(A,B):
    print 'p',hex(p).upper()
    print 'A',hex(A).upper()
    print 'B',hex(B).upper()
    break

# output:
# p FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001
# A 8F0FF20E1E3CF4905D492E04110683948BFC236790BBB59E6E6B33F24F348ED2E16C64EE79F9FD27E9A367FF6
#   415B41189E4FB6BADA555455DC44C4F87011EEF
# B  E85067A95547E30661C854A43ED80F36289043FFC73DA78A97E37FB96A2717009088656B948865A660FF3959
#   330D8A1CA1E4DE31B7B7D496A4CDE555E57D05C
```

Fig. 5.6. A procedure to generate the new "verifiably pseudorandom" BADA55-VPR2-224 curve. Compare Figure 5.5.

- Start with the byte at position 0. In the case of e this is the same as the fourth option. In the case of $\sin(1)$ this means prepending 8 zero bits to the fourth option.

These options can be viewed as using different maps from real numbers x to real numbers y with $0 \le y < 1$: the first map takes x to $|x|/2^{\lfloor \log_2 |x| \rfloor}$, the second map takes x to $x - \lfloor x \rfloor$, the third map takes x to $|x|/16^{\lfloor \log_{16} |x| \rfloor}$, etc. Brainpool used the third of these options, describing it as using "the hexadecimal representation" of e. Jerry can use similarly brief descriptions for any of the options without drawing the public's attention to the existence of other options.

We implemented the first, second, and fourth options; for an average constant this produced slightly more than 2 distinct possibilities for real numbers y.

Jerry can easily get away with extracting a k-bit integer from y by truncation (i.e., $\lfloor 2^k y \rfloor$) or by rounding (i.e., $\lceil 2^k y \rfloor$). Jerry can defend truncation (which has fundamentally lower accuracy) as simpler, and can defend rounding as being quite standard in mathematics and the physical sciences; but we see no reason to believe that Jerry would be challenged in the first place. We implemented both options, gaining a further factor of 1.5.

Actually, Brainpool uses the bit position indicated above only for the low-security 160-bit Brainpool curve (which Jerry can disregard as already being a non-problem for Eve). As shown in Figure 5.3, Brainpool shifts to subsequent bits of e for the 192-bit curve, then to further bits for the 224-bit curve, etc. Brainpool uses 160 bits for each curve (see below), so the seed for the 256-bit curve (which Jerry can reasonably guess would be the most commonly used curve) is shifted by 480 bits. This number 480 depends on how many lower security levels are allocated (an obvious target of manipulation), and on exactly how many bits are allocated to those seeds. A further option, pointed out in [39] by Merkle (Brainpool RFC co-author), is to reverse the order of curve sizes; the number 480 then depends on how many *higher* security levels are allocated. Yet another option is to put curve sizes in claimed order of usage. We did not implement any of the options described in this paragraph.

5.5. Manipulating choices of hash functions. The latest (July 2013) revision of the NIST ECDSA standard [42, Section 6.1.1] specifically requires that "the security strength of a hash function used [for curve generation] **shall** meet or exceed the security strength associated with the bit length". The original NIST curves are exempted from this rule by [42, footnote 2], but this rule prohibits SHA-1 for (e.g.) new 224-bit curves. On the other hand, a more recent Brainpool-related curve-selection document [39] states that "For a PRNG, SHA-1 was (and still is) sufficiently secure."

Jerry has at least 10 plausible options for standard hash functions used to generate (e.g.) 256-bit curves:

- SHA-1. "We follow the Brainpool standard. What matters is preimage resistance, and SHA-1 still provides more than 2^{128} preimage resistance."
- SHA-256. "The trusted, widely deployed SHA-256 standard."
- SHA-384. "SHA-2 at the security level required to handle both sizes of Suite B curves."
- SHA-512. "The maximum-security SHA-512 standard."
- SHA-512/256. "NIST's standard wide-pipe hash function."
- SHA3-256. "The state-of-the-art SHA-3 standard at a 2^{128} security level."
- SHA3-384. "The state-of-the-art SHA-3 standard, at the security level required to handle both sizes of Suite B curves."
- SHA3-512. "The maximum-security state-of-the-art SHA3-512 standard."
- SHAKE128. "The state-of-the-art SHA-3 standard at a 2^{128} security level, providing flexible output sizes."

- SHAKE256. “The state-of-the-art SHA-3 standard at a 2^{256} security level, providing flexible output sizes.”

There are also several non-NIST hash functions with longer track records than SHA-3. Any of RIPEMD-128, RIPEMD-160, RIPEMD-256, RIPEMD-320, Tiger, Tiger/128, Tiger/160, and Whirlpool would have been easily justifiable as a choice of hash function before 2006. MD5 and all versions of Haval would have been similarly justifiable before 2004.

Since we targeted a 224-bit curve we had even more standard NIST hash-function options. For simplicity we implemented just 10 hash-function options, namely the following variants of Keccak, the SHA-3 competition winner: Keccak-224, Keccak-256, Keccak-384, Keccak-512, “default” Keccak (“capacity” $c = 576$, 128 output bytes), Keccak-128 (capacity $c = 256$, 168 output bytes), SHA3-224 (which has different input padding from Keccak-224, changing the output), SHA3-256, SHA3-384, and SHA3-512. All of these Keccak/SHA-3 choices can be implemented efficiently with a single code base and variable input parameters.

5.6. Manipulating counter sizes. The simplest way to obtain a 160-bit “verifiably pseudorandom” output with SHA-1 is to hash the empty string. Curve generation needs many more outputs (since most curves do not pass the public security criteria), but the simplest way to obtain 2^β “verifiably pseudorandom” outputs is to hash all β-bit inputs.

Hash-function implementations are often limited to byte-aligned inputs, so it is natural to restrict β to a multiple of 8. If each output has chance 2^{-15} of producing an acceptable curve (see Section 2) then $\beta = 16$ finds an acceptable curve with chance nearly 90% (“this is retroactively justified by our successfully finding a curve, so there was no need for us to consider backup plans”); $\beta = 24$ fails with negligible probability (“we chose the smallest β for which the probability of failure was negligible”); $\beta = 32$ is easily justified by reference to 32-bit machines; $\beta = 64$ is easily justified by reference to 64-bit machines.

Obviously Brainpool takes a more complicated approach, using bits of some natural constant to further “randomize” its outputs. The standard way to randomize a hash is to concatenate the randomness (e.g., bits of e) with the input being hashed (the counter). Brainpool instead *adds* the randomness to the input being hashed. The Brainpool choice is not secure as a general-purpose randomized hash, although these security problems are of no relevance to curve generation. There is no evidence of public objections to Brainpool’s use of addition here (and to the overall complication introduced by the extra randomization), so there is also no reason to think that the public would object to the more standard concatenation approach.

Overall there are 13 plausible possibilities here: the 4 choices of β above, with the counter on the left of the randomness; the 4 choices of β above, with the counter on the right of the randomness; the counter being added to the randomness; and 4 further possibilities in which the randomness is partitioned into an initial value for a counter (for the top bits) and the remaining seed (for the bottom bits). We implemented the first 9 of these 13 possibilities.

5.7. Manipulating hash input sizes. ANSI X9.62 requires $\geq$160 input bits for its hash input. One way for Jerry to advertise a long input is that it allows many people to randomly generate curves with a low risk of collision. For example, Jerry can advertise

- a 160-bit input as allowing 2^{64} curves with only a 2^{-32} risk of collision;
- a 256-bit input as allowing 2^{64} curves with only a 2^{-128} risk of collision; or
- a 384-bit input as allowing 2^{128} curves with only a 2^{-128} risk of collision.

All of these numbers sound perfectly natural. Of course, what Jerry is actually producing is a single standard for many people to use, so multiple-curve collision probabilities are of no relevance, but (in the unlikely event of being questioned) Jerry can simply say that the input length was chosen for "compatibility" with having users generate their own curves.

Jerry can advertise longer input lengths as providing "curve coverage". A 512-bit input will cover a large fraction of curves, even for primes as large as 512 bits. A 1024-bit input is practically guaranteed to cover all curves, and to produce probabilities indistinguishable from uniform. Jerry can also advertise, as input length, the "natural input block length of the hash function".

We implemented all 6 possibilities listed above. We gained a further factor of 2 by storing the seed (and counter) in big-endian format ("standard network byte order") or little-endian format ("standard CPU byte order").

5.8. Manipulating the (a, b) hash pattern. It should be obvious from Section 5.1 that there are many degrees of freedom in the details of how a and b are generated: how to distribute seeds between a and b; whether to require $-3/a$ to be a 4th power in $\mathbb{F}_p$; whether to require b to be a non-square in $\mathbb{F}_p$; whether to concatenate hash outputs from left to right or right to left; exactly how many bits to truncate hash outputs to (Brainpool uses one bit fewer than the prime; Jerry can argue for the same length as the prime "for coverage", or more bits "for indistinguishability"); whether to truncate to rightmost bits (as in Brainpool) or leftmost bits (as in various NIST requirements; see [42]); et al.

For simplicity we eliminated the concatenation and truncation, always using a hash function long enough for the target 224-bit prime. We also eliminated the options regarding squares etc. We implemented a total of just 8 choices here. These choices vary in (1) whether to allocate seeds primarily to a or primarily to b and (2) how to obtain the alternate seed (e.g., the seed for a) from the primary seed (e.g., the seed for b): plausible options include complement, rotate 1 byte left, rotate 1 byte right, and four standard versions of 1-bit rotations.

5.9. Manipulating natural constants. As noted in Section 1, the public has accepted dozens of "natural" constants in various cryptographic functions, and sometimes reciprocals of those constants, without complaint. Our implementation started with just 17 natural constants: π, e, Euler gamma, $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, $\sqrt{7}$, $\log(2)$, $(1+\sqrt{5})/2$, $\zeta(3)$, $\zeta(5)$, $\sin(1)$, $\sin(2)$, $\cos(1)$, $\cos(2)$, $\tan(1)$, and $\tan(2)$. We gained an extra factor of almost 2 by including reciprocals.

Jerry could be creative and use previously unused numbers such as numbers derived from some historical document or newspaper, personal information of, e.g., arbitrary celebrities in an arbitrary order, arbitrary collections of natural or physical constants and even a combination of several sources. For example, NewDES [52] derives its S-Box from the United States Declaration of Independence. If the public accepts numbers with such flimsy justifications as "nothing-up-my-sleeves numbers" then Jerry obviously has as much flexibility as in Section 4. We make the worst-case assumption that the public is not quite as easily fooled, and similarly that the public would not accept $703e^{(\sqrt[8]{30}+4\pi)/9\sin(\sqrt[3]{16})}$ as a "nothing-up-my-sleeve number".

5.10. Implementation. Any combination of the above manipulations defines a "systematic" curve-generation procedure. This procedure outputs the first curve parameters (using the specified update function) that result in a "secure" curve according to the public security tests. However, performing all public security tests for each set of parameters considered by each procedure is very costly. Instead, we split the attack into two steps:

1. For a given procedure f_i we iterate over the seeds $s_{i,k}$ using the specific update function of f_i. We check each parameter candidate from seed $s_{i,k}$ for our secret BADA55 vulnerability. After a certain number of update steps the probability that we passed valid, secure parameters is very high; thus, we discard the procedure and start over with another one. If we find a candidate exhibiting the vulnerability, we perform the public security tests on this particular candidate. If the BADA55 candidate passes, we proceed to step 2.
2. We perform the whole public procedure f_i starting with seed $s_{i,0}$ and check whether there is any valid parameter set passing the public security checks already before the BADA55 parameters are reached. If there is such an earlier parameter set, we return to step 1 with the next procedure f_{i+1}.

The largest workload in our attack scenario is step 2, the re-checking for earlier safe curve parameters before BADA55 candidates. The public security tests are not well suited for GPU parallelization; the first step of the attack procedure is relatively cheap and a GPU parallelization of this step does not have a remarkable impact on the overall runtime. Therefore, we implemented the whole attack only for the CPUs of the Saber cluster and left the GPUs idle.

We initially chose 8000 as the limit for the update counter to have a very good chance that the first secure twist-secure curve starting from the seed is the curve with our vulnerability. For example, BADA55-VPR-224 was found with counter just 184, and there was only a tiny risk of a smaller counter producing a secure twist-secure curve (which we checked later, in the second step). In total $\approx 2^{33}$ curves were covered by this limited computation; more than 2^{18} were secure and twist-secure. We then pushed the 8000 limit higher, performing more computation and finding more curves. This gradually increased the risk of the counter not being minimal, something that we would have had to address by the techniques of Section 6; but this issue still did not affect, e.g., BADA55-VPR2-224, which was found with counter 28025.

6 Manipulating minimality

Instead of supporting "verifiably pseudorandom" curves as in Section 5, some researchers have advocated choosing "verifiably deterministic" curves.

Both approaches involve specifying a "systematic" procedure that outputs a curve. The difference is that in a "verifiably pseudorandom" curve the curve coefficient is the output of a hash function for the *first hash input* that meets specified curve criteria, while a "verifiably deterministic" curve uses the *first curve coefficient* that meets specified curve criteria. Typically the curve uses a "verifiably deterministic" prime, which is the *first prime* that meets specified prime criteria.

Eliminating the hash function and hash input makes life harder for Jerry: it eliminates the main techniques that we used in previous sections to manipulate curve choices. However, as we explain in detail in this section, Jerry still has many degrees of freedom. Jerry can manipulate the concept of "first curve coefficient", can manipulate the concept of "first prime", can manipulate the curve criteria, and can manipulate the prime criteria, with public justifications claiming that the selected criteria provide convenience, ease of implementation, speed of implementation, and security.

In Section 5 we did not manipulate the choice of prime: we obtained a satisfactory level of flexibility in other ways. In this section, the choice of prime is an important component of Jerry's flexibility. It should be clear to the reader that the techniques in this section to manipulate the prime, the curve criteria, etc. can be backported to the setting of Section 5, adding to the flexibility there.

We briefly review a recent proposal that fits into this category and then proceed to work out how much flexibility is left for Jerry.

6.1. NUMS curves. In early 2014, Bos, Costello, Longa, and Naehrig [13] proposed 13 Weierstrass and 13 Edwards curves, spread over 3 different security levels. Each curve was generated following a deterministic procedure (similar to the procedure proposed in [8]). Given that there are up to 10 different procedures per security level we cannot review all of them here but [13] is a treasure trove of arguments to justify different prime and curve properties and we will use this to our benefit below.

The same authors together with Black proposed a set of 6 of these curves as an Internet-Draft [12] referring to these curves as "Nothing Up My Sleeve (NUMS) Curves". Note that this does not match the common use of "nothing up my sleeves"; see, e.g., the Wikipedia page [51]. These curves are claimed in [32] to have "independently-verifiable provenance", as if they were not subject to any possible manipulation; and are claimed in [11] to be selected "without any hidden parameters, reliance on randomness or any other processes offering opportunities for manipulation of the resulting curves". What we analyze in this section is the extent to which Jerry can manipulate the resulting curves.

6.2. Choice of security level. Jerry may propose curves aiming for multiple security levels. To quote the Brainpool-curves RFC [34] "The level of security provided by symmetric ciphers and hash functions used in conjunction with the elliptic curve domain parameters specified in this RFC should roughly match or exceed the level provided by the domain parameters." Table 1 in that document justifies security levels of 80, 96, 112, 128, 160, 192, and 256 bits. We consider the highest five to be easy sells. For the smaller ones Jerry will need to be more creative and, e.g., evoke the high cost of energy for small devices.

6.3. Choice of prime. There are several parts to choosing a prime once the security level is fixed.

Choice of prime size. For a fixed security level α it should take about 2^α operations to break the DLP. The definition of "operation" leaves some flexibility. The choices for the bit length r of the prime are:

- Exactly 2α, see e.g., [13].
- Exactly $2\alpha - 1$, see e.g., [13].
- Exactly $2\alpha - 2$, see e.g., [13].
- Exactly $2\alpha+1$ to make up for the loss of $\sqrt{\pi/4}$ in the Pollard-rho complexity.
- Exactly $2\alpha + 2$ to *really* make up for the loss of $\sqrt{\pi/4}$ in the Pollard-rho complexity.

  $\vdots$
- Exactly $2\alpha + \beta$ to make up for the loss through precomputations for multi-target attacks.
- Exactly $2\alpha - 3$ to make arithmetic easier and because each elliptic-curve operation takes at least 3 bit operations.
- Exactly $2\alpha - 4$ to make arithmetic easier and because each elliptic-curve operation takes at least 4 bit operations.

  $\vdots$
- Exactly $2\alpha - \gamma$ to make arithmetic easier and because each elliptic-curve operation takes at least $2^{\gamma/2}$ bit operations.

These statements provide generic justifications for 8 options (actually even more, but we take a power of 2 to simplify). In the next two steps we show how to select different primes for each of these requirements. If the resulting p has additional beneficial properties these generic arguments might not be necessary, but they might be required if a competing (and by some measure superior) proposal can be excluded on the basis of not following the same selection criterion. If Jerry wants to highlight such benefits in his prime choice he may point to fast reduction or fast multiplication in a particular redundant representation with optimal limb size.

Choice of prime shape. The choices for the prime shape are:

- A random prime. This might seem somewhat hard to justify outside the scope of the previous section because arithmetic in $\mathbb{F}_p$ becomes slower, but members of the ECC Brainpool working group published several helpful arguments [35]. The most useful one is that random primes mean that the blinding factor in randomizing scalars against differential side-channel attacks can be chosen smaller.

- A pseudo-Mersenne prime, i.e. a prime of the shape $2^r \pm c$. The most common choice is to take c to be the smallest integer for a given r which leads to a prime because this makes reduction modulo the prime faster. (To reduce modulo $2^r \pm c$, divide by 2^r and add $\mp c$ times the dividend to the remainder.) See, e.g., [13]. Once r is fixed there are two choices for the two signs.
- A Solinas prime, i.e. a prime of the form $2^r \pm 2^v \pm 1$ as chosen for the Suite B curves [43]. Also for these primes speed of modular reduction is the common argument. The difference $r - v$ is commonly chosen to be a multiple of the word size. Jerry can easily argue for multiples of 32 and 64. We skip this option in our count because it is partially subsumed in the following one.
- A "Montgomery-friendly" prime, i.e. a prime of the form $2^{r-v}(2^v - c) \pm 1$. These curves speed up reductions if elements in $\mathbb{F}_p$ are represented in Montgomery representation, $r - v$ is a multiple of the word size and c is less than the word size. Common word sizes are 32 and 64, giving two choices here. We ignore the flexibility of the ± because that determines p modulo 4, which is considered separately.

There are of course infinitely many random primes; in order to keep the number of options reasonable we take 4 as an approximation of how many prime shapes can be easily justified, making this a total of 8 options.

Choice of prime congruence. Jerry can get an additional bit of freedom by choosing whether to require $p \equiv 1 \pmod 4$ or to require $p \equiv 3 \pmod 4$. A common justification for the latter is that computations of square roots are particularly fast which could be useful for compression of points, see, e.g., [14,13]. (In fact one can also compute square roots efficiently for $p \equiv 1 \pmod 4$, in particular for $p \equiv 5 \pmod 8$, but Jerry does not need to admit this.) To instead justify $p \equiv 1 \pmod 4$, Jerry can point to various benefits of having $\sqrt{-1}$ in the field: for example, twisted Edwards curves are fastest when $a = -1$, but completeness for $a = -1$ requires $p \equiv 1 \pmod 4$.

If Jerry chooses twisted Hessian curves he can justify restricting to $p \equiv 1 \pmod 3$ to obtain complete curve arithmetic.

6.4. Choice of ordering of field elements. The following curve shapes each have one free parameter. It is easy to justify choosing this parameter as the smallest parameter under some side conditions. Here smallest can be chosen to mean smallest in $\mathbb{N}$ or as the smallest power of some fixed generator g of $\mathbb{F}_p^*$. The second option is used in, e.g., a recent ANSSI curve-selection document [24, Section 2.6.2]: "we define … g as the smallest generator of the multiplicative group … We then iterate over … $b = g^n$ for $n = 1, \ldots,$ until a suitable curve is found." Each choice below can be filled with these two options.

6.5. Choice of curve shape and cofactor requirement. Jerry can justify the following curve shapes:

1. Weierstrass curves, the most general curve shape. The usual choice is $y^2 = x^3 - 3x + b$, leaving one parameter b free. For simplicity we do not discuss the possibility of choosing values other than -3.

2. Edwards curves, the speed leader in fixed-base scalar multiplication offering complete addition laws. The usual choices are $ax^2+y^2=1+dx^2y^2$, for $a \in \{\pm 1\}$, leaving one parameter d free. The group order of an Edwards curve is divisible by 4.
3. Montgomery curves, the speed leader for variable-base scalar multiplication and the simplest to implement correctly. The usual choices are $y^2 = x^3 + Ax^2 + x$, leaving one parameter A free. The group order of a Montgomery curve is divisible by 4.
4. Hessian curves, a cubic curve shape with complete addition laws (for twisted Hessian). The usual choices are $ax^3 + y^3 + 1 = dxy$, where a is a small non-cube, leaving one parameter d free. The group order of a Hessian curve is divisible by 3, making twisted Hessian curves the curves with the smallest cofactor while having complete addition.

The following choices depend on the chosen curve shape, hence we consider them separately.

Weierstrass curves. Most standards expect the point format to be (x, y) on Weierstrass curves. Even when computations want to use the faster Edwards and Hessian formulas, Jerry can easily justify specifying the curve in Weierstrass form. This also ensures backwards compatibility with existing implementations that can only use the Weierstrass form.

The following are examples of justifiable choices for the cofactor h of the curve:

- Require cofactor exactly 1, as in Suite B and Brainpool.
- Require cofactor exactly 2, the minimum cofactor that allows the techniques of [8] to transmit curve points as uniform random binary strings for censorship circumvention.
- Require cofactor exactly 3, the minimum cofactor that allows Hessian arithmetic.
- Require cofactor exactly 4, the minimum cofactor that allows Edwards arithmetic.
- Require cofactor exactly 12, the minimum cofactor that allows both Hessian arithmetic and Edwards arithmetic.
- Take the first curve having cofactor below $2^{\alpha/8}$. This cofactor limit is standardized in [19] and [42]. (This cofactor will almost always be larger than 12.)
- Take the first curve having cofactor below $2^{\alpha/8}$ and a multiple of 3.
- Take the first curve having cofactor below $2^{\alpha/8}$ and a multiple of 4.
- Take the first curve having cofactor below $2^{\alpha/8}$ and a multiple of 12.
- Replace "cofactor below $2^{\alpha/8}$" with the SafeCurves requirement of a largest prime factor above 2^{200}.

On average these choices produce slightly more than 8 options; the last few options sometimes coincide.

The curve is defined as $y^2 = x^3 - 3x + b$ where b is minimal under the chosen criterion. Changing from positive b to negative b changes from a curve to its twist if $p \equiv 3 \pmod 4$, and (as illustrated by additive transfers) this change does not necessarily preserve security. However, this option makes only a small difference in our final total, so for simplicity we skip it.

Hessian curves. A curve given in Hessian form (and chosen minimal there) can be required to have minimal cofactor, minimal cofactor while being compatible with Edwards form, cofactor smaller than $2^{\alpha/8}$, or largest prime factor larger than 2^u. This leads to 8 options considering positive and negative values of d. Of course other restrictions on the cofactor are possible.

Edwards curves. For Edwards curves we need to split up the consideration further:

Edwards curves with $p \equiv 3 \pmod 4$. Curves with $a = -1$ are attractive for speed but are not complete in this case. Nevertheless [13] argues for this option, so we have additionally the choice between aiming for a complete or an $a = -1$ curve.

A curve given in (twisted) Edwards form (and chosen minimal there) can be required to have minimal cofactor, minimal cofactor while being compatible with Hessian form, cofactor smaller than $2^{\alpha/8}$, or largest prime factor larger than 2^u (and the latter in combination with Hessian if desired). This leads to at least 8 choices considering completeness; for minimal cofactors [13] shows that minimal choices for positive and negative values of d are not independent. To stay on the safe side we count these as 8 options only.

Edwards curves with $p \equiv 1 \pmod 4$. The curves $x^2 + y^2 = 1 + dx^2y^2$ and $-x^2 + y^2 = 1 - dx^2y^2$ are isomorphic because -1 is a square, hence taking the smallest positive value for d finds the same curve as taking the smallest negative value for the other sign of a. Jerry can however insist or not insist on completeness. Justifying non-completeness if the smallest option is complete however seems a hard sell.

Because $2p + 2 \equiv 4 \pmod 8$ one of the curve and its twist will have order divisible by 8 while the other one has remainder 4 modulo 8. Jerry can require cofactor 4, as the minimal cofactor, or cofactor 8 if he chooses the twist with minimal cofactor as well and is concerned that protocols will only multiply by the cofactor of the curve rather than by that of the twist. The other options are the same as above. Again, to stay on the safe side, we count this as 8 options only.

Montgomery curves. There is a significant overlap between choosing the smallest Edwards curve and the smallest Montgomery curve. In order to ease counting and avoid overcounting we omit further Montgomery options.

Summary of curve choice. We have shown that Jerry can argue for $8+8+8 = 24$ options.

6.6. Choice of twist security. We make the worst-case assumption, as discussed in Section 2, that future standards will be required to include twist security. However, Jerry can play twist security to his advantage in changing the details of the twist-security requirements. Here are three obvious choices:

- Choose the cofactor of the twist as small as possible. Justification: This offers maximal protection.
- Choose the cofactor of the twist to be secure under the SEC recommendation, i.e. $h' < 2^{\alpha/8}$. Justification: This is considered secure enough for the main curve, so it is certainly enough for the twist.
- Choose the curve such that the curve passes the SafeCurves requirement of 2^{100} security against twist attacks. Justification: Attacks on the twist cannot use Pollard rho but need to do a brute-force search in the subgroups. The SafeCurves requirement captures the actual hardness of the attack.

Jerry can easily justify changes to the bound of 2^{100} by pointing to a higher security level or reducing it because the computations in the brute-force part are more expensive. We do not use this flexibility in the counting.

6.7. Choice of global vs. local curves. Jerry can take the first prime (satisfying some criteria), and then, for that prime, take the first curve coefficients (satisfying some criteria). Alternatively, Jerry can take the first possible curve coefficients, and then, for those curve coefficients, take the first prime. These two options are practically guaranteed to produce different curves. For example, in the Weierstrass case, Jerry can take the curve $y^2 = x^3 - 3x + 1$, and then search for the first prime p so that this curve over $\mathbb{F}_p$ satisfies the requirements on cofactor and twist security. If Jerry instead takes $y^2 = x^3 - 3x + g$ as in [24, Section 2.6.2], p must also meet the requirement that g be primitive in $\mathbb{F}_p$.

In mathematical terminology, the second option specifies a curve over a "global field" such as the rationals $\mathbb{Q}$, and then reduces the curve modulo suitable primes. This approach is particularly attractive when presented as a family of curves, all derived from the same global curve.

6.8. More choices. Brainpool [14] requires that the number of points on the curve is less than p but also presents an argument for the opposite choice:

> To avoid overruns in implementations we require that $\#E(GF(p)) < p$. In connection with digital signature schemes some authors propose to use $q > p$ for security reasons, but the attacks described e.g. in [BRS] appear infeasible in a thoroughly designed PKI.

So Jerry can choose to insist on $p < |E(\mathbb{F}_p)|$ or on $p > |E(\mathbb{F}_p)|$.

6.9. Overall count. We have shown that Jerry can easily argue for 4 (security level) ·8 (prime size) ·8 (prime shape) ·2 (congruence) ·2 (definition of first) ·24 (curve choice) ·3 (twist conditions) ·2 (global/local) ·2 ($p \lessgtr |E(\mathbb{F}_p)|$) = 294912 choices.

6.10. Example. The artificial "vulnerability" that we have used throughout this paper, namely BADA55 appearing in a curve coefficient, is obviously incompatible with taking that coefficient to be minimal in the usual ordering. We would be happy to accept the following type of challenge as an alternative: a third party provides us with a nonstructured prime number $n > 2^{50}$; we find a curve so that the hexadecimal representation of ℓ modulo n ends in BAD, a condition having probability 2^{-12}.

7 Manipulating security criteria

An unfortunate recent trend is to introduce top performance as a selection requirement. This means that Alice and Bob accept only *the fastest curve*, as demonstrated by benchmarks across a range of platforms. The most widely known example of this approach is Bernstein's Curve25519, the curve $y^2 = x^3+486662x^2+x$ modulo the particularly efficient prime $2^{255}-19$, which over the past ten years has set speed records for conservative ECC on space-constrained ASICs, Xilinx FPGAs, 8-bit AVR microcontrollers, 16-bit MSP430X microcontrollers, 32-bit ARM Cortex-M0 microcontrollers, larger 32-bit ARM smartphone processors, the Cell processor, NVIDIA and AMD GPUs, and several generations of 32-bit and 64-bit Intel and AMD CPUs, using implementations from 23 authors. See [6,26,22,7,10,33,38,47,21,23,28].

The annoyance for Jerry in this scenario is that, in order to make a case for his curve, he needs to present implementions of the curve arithmetic on a variety of devices, showing that his curve is fastest across platforms. Jerry could try to falsify his speed reports, but it is increasingly common for the public to demand verifiable benchmarks using open-source software.

Jerry can hope that some platforms will favor one curve while other platforms will favor another curve; Jerry can then use arguments for a "reasonable" weighting of platforms as a mechanism to choose one curve or the other. However, it seems difficult to outperform Curve25519 even on *one* platform. The prime $2^{255} - 19$ is particularly efficient, as is the Montgomery curve shape $y^2 = x^3+486662x^2+x$. The same curve is also expressible as a complete Edwards curve, allowing fast additions without the overhead of checking for exceptional cases. Twist security removes the overhead of checking for invalid inputs. Replacing 486662 with a larger curve coefficient produces identical performance on many platforms but loses a measurable amount of performance on some platforms, violating the "top performance" requirement.

In Section 6, Jerry was free to, e.g., claim that $p \equiv 3 \pmod 4$ provides "simple square-root computations" and thus replace $2^{255} - 19$ with $2^{255} - 765$; claim that "compatibility" requires curves of the form $y^2 = x^3 - 3x + b$; etc. The new difficulty in this section is that Jerry is facing "top performance" fanatics who reject $2^{255}-765$ as not providing top performance; who reject $y^2 = x^3-3x+b$ as not providing top performance; etc.

Fortunately, Jerry still has some flexibility in defining what security requirements to take into account. Taking "the fastest curve" actually means taking the fastest curve *meeting specified security requirements*, and the list of security requirements is a target of manipulation.

Most importantly, Jerry can argue for any size of ℓ. However, if there is a faster curve with a larger ℓ satisfying the same criteria, then Jerry's curve will be rejected. Furthermore, if Jerry's curve is only marginally larger than a significantly faster curve, then Jerry will have to argue that a tiny difference in security levels (e.g., one curve broken with 0.7× or 0.5× as much effort as another) is meaningful, or else the top-performance fanatics will insist on the significantly faster curve.

The choice of prime has the biggest impact on speed and closely rules the size of ℓ. For pseudo-Mersenne primes larger than 2^{224} the only possibly competitive ones are: $2^{226}-5, 2^{228}+3, 2^{233}-3, 2^{235}-15, 2^{243}-9, 2^{251}-9, 2^{255}-19, 2^{263}+9, 2^{266}-3, 2^{273}+5, 2^{285}-9, 2^{291}-19, 2^{292}+13, 2^{295}+9, 2^{301}+27, 2^{308}+27, 2^{310}+15, 2^{317}+9, 2^{319}+9, 2^{320}+27, 2^{321}-9, 2^{327}+9, 2^{328}+15, 2^{336}-3, 2^{341}+5, 2^{342}+15, 2^{359}+23, 2^{369}-25, 2^{379}-19, 2^{390}+3, 2^{395}+29, 2^{401}-31, 2^{409}+29, 2^{414}-17, 2^{438}+25, 2^{444}-17, 2^{452}-3, 2^{456}+21, 2^{465}+29, 2^{468}-17, 2^{488}-17, 2^{489}-21, 2^{492}+21, 2^{495}-31, 2^{508}+15, 2^{521}-1$. Preliminary implementation work shows that the Mersenne prime $2^{521}-1$ has such efficient reduction that it outperforms, e.g., the prime $2^{512}-569$ from [13]; perhaps it even outperforms primes below 2^{500}. We would expect implementation work to also show, e.g., that $2^{319}+9$ is significantly faster than $2^{320}+27$, and Jerry will have a hard time arguing for $2^{320}+27$ on security grounds. Considering other classes of primes, such as Montgomery-friendly primes, might identify as many as 100 possibly competitive primes, but it is safe to estimate that fewer than 80 of these primes will satisfy the top-performance fanatics, and further implementation work is likely to reduce the list even more. Note that in this section, unlike other sections, we take a count that is optimistic for Jerry.

Beyond the choice of prime, Jerry can use different choices of security criteria. However, most of the flexibility in Section 6 consists of speed claims, compatibility claims, etc., few of which can be sold as security criteria. Jerry *can* use the different twist conditions, the choice whether $p < |E(\mathbb{F}_p)|$ or $p > |E(\mathbb{F}_p)|$, and possibly two choices of cofactor requirements. Jerry can also choose to require completeness as a security criterion, but this does not affect curve choice in this section: the complete formulas for twisted Hessian and Edwards curves are *faster* than the incomplete formulas for Weierstrass curves. The bottom line is that multiplying fewer than 80 primes by 12 choices of security criteria produces fewer than 960 curves. The main difficulty in pinpointing an exact number is carrying out detailed implementation work for each prime; we leave this to future work.

8 Afterword: removing the hat

This paper, outside this section, systematically adopts the attacker's perspective. In this section, to avoid any chance of confusion, we drop the attacker's perspective and address a few questions that we have been asked.

First, in case this is not obvious to the reader, we do not actually endorse the attacker's perspective. Our goal in analyzing the security of systems is to prevent attacks.

Second, this paper *analyzes* the *possibilities* of backdooring curves under various conditions. We are not making any statements about whether such an attack has actually been carried out.

Third, we have been asked how to eliminate Jerry's flexibility in choosing curves. We are not aware of any proposal that reduces the flexibility to just one curve.

References

1. Accredited Standards Committee X9. American national standard X9.62-1999, public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ECDSA), 1999. Preliminary draft at http://grouper.ieee.org/groups/1363/Research/Other.html. Cited on pp.: 2, 6, 15, and 16.
2. Accredited Standards Committee X9. American national standard X9.63-2001, public key cryptography for the financial services industry: key agreement and key transport using elliptic curve cryptography, 2001. Preliminary draft at http://grouper.ieee.org/groups/1363/Research/Other.html. Cited on pp.: 2 and 6.
3. Agence nationale de la sécurité des systèmes d'information. Publication d'un paramétrage de courbe elliptique visant des applications de passeport électronique et de l'administration électronique française, 2011. https://tinyurl.com/nhog26h. Cited on pp.: 2, 6, and 13.
4. Jean-Philippe Aumasson. Generator of "nothing-up-my-sleeve" (NUMS) constants, 2015. https://github.com/veorq/numsgen/blob/master/numsgen.py. Cited on pp.: 5.
5. Eric Bach and René Peralta. Asymptotic semismoothness probabilities. *Math. Comput.*, 65(216):1701–1715, 1996. http://www.ams.org/journals/mcom/1996-65-216/S0025-5718-96-00775-2/S0025-5718-96-00775-2.pdf. Cited on pp.: 12 and 12.
6. Daniel J. Bernstein. Curve25519: New Diffie–Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. http://cr.yp.to/papers.html#curve25519. Cited on pp.: 39.
7. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2:77–89, 2012. https://eprint.iacr.org/2011/368. Cited on pp.: 39.
8. Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS'13*, pages 967–980. ACM, 2013. http://elligator.cr.yp.to/. Cited on pp.: 33 and 36.
9. Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography, 2015. http://safecurves.cr.yp.to (accessed 27 September 2015). Cited on pp.: 7, 7, 8, and 12.
10. Daniel J. Bernstein and Peter Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2012. http://cr.yp.to/papers.html#neoncrypto. Cited on pp.: 39.
11. Benjamin Black, Joppe W. Bos, Craig Costello, Adam Langley, Patrick Longa, and Michael Naehrig. Rigid parameter generation for elliptic curve cryptography, 2015. https://tools.ietf.org/html/draft-black-rpgecc-01. Cited on pp.: 33.
12. Benjamin Black, Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Elliptic curve cryptography (ECC) nothing up my sleeve (NUMS) curves and curve generation, 2014. https://tools.ietf.org/html/draft-black-numscurves-00. Cited on pp.: 33.
13. Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Selecting elliptic curves for cryptography: an efficiency and security analysis. *Journal of Cryptographic Engineering*, pages 1–28, 2015. https://eprint.iacr.org/2014/130/. Cited on pp.: 5, 33, 33, 34, 34, 34, 35, 35, 37, 37, and 40.

14. ECC Brainpool. ECC Brainpool standard curves and curve generation, 2005. http://www.ecc-brainpool.org/download/Domain-parameters.pdf. Cited on pp.: 2, 5, 6, 20, 21, 21, 21, 21, 22, 23, 24, 35, and 38.
15. Eric Brier and Marc Joye. Weierstraß elliptic curves and side-channel attacks. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography*, volume 2274 of *Lecture Notes in Computer Science*, pages 335–345. Springer, 2002. http://joye.site88.net/papers/BJ02espa.pdf. Cited on pp.: 8.
16. Certicom Research. SEC 1: Elliptic curve cryptography, version 1.0, 2000. http://www.secg.org/SEC1-Ver-1.0.pdf. Cited on pp.: 6 and 7.
17. Certicom Research. SEC 2: Recommended elliptic curve domain parameters, version 1.0, 2000. http://www.secg.org/SEC2-Ver-1.0.pdf. Cited on pp.: 2 and 6.
18. Certicom Research. SEC 1: Elliptic curve cryptography, version 2.0, 2009. http://www.secg.org/sec1-v2.pdf. Cited on pp.: 6 and 7.
19. Certicom Research. SEC 2: Recommended elliptic curve domain parameters, version 2.0, 2010. http://www.secg.org/sec2-v2.pdf. Cited on pp.: 6, 15, 16, and 36.
20. Stephen Checkoway, Matthew Fredrikson, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, and Hovav Shacham. On the practical exploitability of Dual EC in TLS implementations. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, 2014. USENIX Association. https://projectbullrun.org/dual-ec/index.html. Cited on pp.: 5 and 8.
21. Tung Chou. Sandy2x: fastest Curve25519 implementation ever, 2015. http://csrc.nist.gov/groups/ST/ecc-workshop-2015/presentations/session6-chou-tung.pdf. Cited on pp.: 39.
22. Neil Costigan and Peter Schwabe. Fast elliptic-curve cryptography on the Cell Broadband Engine. In *Africacrypt 2009*, pages 368–385, 2009. https://cryptojedi.org/papers/celldh-20090331.pdf. Cited on pp.: 39.
23. Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography*, 2015. To appear, http://link.springer.com/article/10.1007/s10623-015-0087-1/fulltext.html. Cited on pp.: 39.
24. Jean-Pierre Flori, Jérôme Plût, Jean-René Reinhard, and Martin Ekerå. Diversity and transparency for ECC, 2015. http://csrc.nist.gov/groups/ST/ecc-workshop-2015/papers/session4-flori-jean-pierre.pdf. Cited on pp.: 35 and 38.
25. Steven D. Galbraith and James McKee. The probability that the number of points on an elliptic curve over a finite field is prime. *Journal of the London Mathematical Society*, 62:671–684, 2000. https://www.math.auckland.ac.nz/~sgal018/cm.pdf. Cited on pp.: 9 and 9.
26. Pierrick Gaudry and Emmanuel Thomé. The mpFq library and implementing curve-based key exchanges. In *SPEED: software performance enhancement for encryption and decryption*, pages 49–64, 2007. http://www.loria.fr/~gaudry/papers.en.html. Cited on pp.: 39.
27. Andrew Granville. Smooth numbers: computational number theory and beyond. In *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography*, pages 267–323. Cambridge University Press, 2008. https://www.math.leidenuniv.nl/~psh/ANTproc/09andrew.pdf. Cited on pp.: 11.
28. Michael Hutter, Jürgen Schilling, Peter Schwabe, and Wolfgang Wieser. NaCl's crypto_box in hardware. In Tim Güneysu and Helena Handschuh, editors, *CHES*

2015, volume 9293 of *Lecture Notes in Computer Science*, pages 81–101. Springer, 2015. https://cryptojedi.org/papers/naclhw-20150616.pdf. Cited on pp.: 39.

29. Institute of Electrical and Electronics Engineers. IEEE 1363-2000: Standard specifications for public key cryptography, 2000. Preliminary draft at http://grouper.ieee.org/groups/1363/P1363/draft.html. Cited on pp.: 2, 6, and 15.
30. Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. Practical invalid curve attacks on TLS-ECDH. In *ESORICS 2015*, 2015. https://www.nds.rub.de/research/publications/ESORICS15/. Cited on pp.: 8.
31. John Kelsey. Choosing a DRBG algorithm, 2003? https://github.com/matthewdgreen/nistfoia/blob/master/6.4.2014%20production/011%20-%209.12%20Choosing%20a%20DRBG%20Algorithm.pdf. Cited on pp.: 5.
32. Brian LaMacchia and Craig Costello. Deterministic generation of elliptic curves (a.k.a. "NUMS" curves), 2014. https://www.ietf.org/proceedings/90/slides/slides-90-cfrg-5.pdf. Cited on pp.: 33.
33. Adam Langley and Andrew Moon. Implementations of a fast elliptic-curve digital signature algorithm, 2013. https://github.com/floodyberry/ed25519-donna. Cited on pp.: 39.
34. Manfred Lochter and Johannes Merkle. RFC 5639: Elliptic curve cryptography (ECC) Brainpool standard curves and curve generation, 2010. https://tools.ietf.org/html/rfc5639. Cited on pp.: 21, 21, 22, and 33.
35. Manfred Lochter, Johannes Merkle, Jörn-Marc Schmidt, and Torsten Schütze. Requirements for standard elliptic curves, 2014. Position Paper of the ECC Brainpool, http://www.ecc-brainpool.org/20141001_ECCBrainpool_PositionPaper.pdf. Cited on pp.: 34.
36. Manfred Lochter and Andreas Wiemers. Twist insecurity, 2015. https://eprint.iacr.org/2015/577.pdf. Cited on pp.: 8 and 8.
37. Florian Luca, David Jose Mireles, and Igor E. Shparlinski. MOV attack in various subgroups on elliptic curves. *Illinois Journal of Mathematics*, 48(3):1041–1052, 07 2004. https://projecteuclid.org/euclid.ijm/1258131069. Cited on pp.: 10.
38. Eric M. Mahé and Jean-Marie Chauvet. Fast GPGPU-based elliptic curve scalar multiplication, 2014. https://eprint.iacr.org/2014/198.pdf. Cited on pp.: 39.
39. Johannes Merkle. Re: [Cfrg] ECC reboot (Was: When's the decision?), 2014. https://www.ietf.org/mail-archive/web/cfrg/current/msg05353.html. Cited on pp.: 29 and 29.
40. Dustin Moody. Development of FIPS 186: Digital signatures (and elliptic curves), 2014. http://csrc.nist.gov/groups/ST/crypto-review/documents/FIPS_186_and_Elliptic_Curves_052914.pdf. Cited on pp.: 8.
41. National Institute for Standards and Technology. FIPS PUB 186-2: Digital signature standard, 2000. http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf. Cited on pp.: 2, 6, 7, 15, 16, and 16.
42. National Institute for Standards and Technology. FIPS PUB 186-4: Digital signature standard (DSS), 2013. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf. Cited on pp.: 6, 29, 29, 31, and 36.
43. National Security Agency. Suite B cryptography / cryptographic interoperability, 2005. https://web.archive.org/web/20150724150910/https://www.nsa.gov/ia/programs/suiteb_cryptography/. Cited on pp.: 2, 6, and 35.
44. State Commercial Cryptography Administration (OSCCA), China. Public key cryptographic algorithm SM2 based on elliptic curves, December 2010. http://www.oscca.gov.cn/UpFile/2010122214822692.pdf. Cited on pp.: 6 and 13.

45. State Commercial Cryptography Administration (OSCCA), China. Recommanded curve parameters for public key cryptographic algorithm SM2 based on elliptic curves, December 2010. http://www.oscca.gov.cn/UpFile/2010122214836668.pdf. Cited on pp.: 6 and 13.
46. J. Barkley Rosser and Lowell Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962. https://projecteuclid.org/euclid.ijm/1255631807. Cited on pp.: 9.
47. Pascal Sasdrich and Tim Güneysu. Efficient elliptic-curve cryptography using Curve25519 on reconfigurable devices. In Diana Goehringer, Marco Domenico Santambrogio, João M. P. Cardoso, and Koen Bertels, editors, *ARC 2014*, volume 8405 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2014. https://www.hgi.rub.de/media/sh/veroeffentlichungen/2014/03/25/paper_arc14_curve25519.pdf. Cited on pp.: 39.
48. Michael Scott. Re: NIST announces set of Elliptic Curves, 1999. https://groups.google.com/forum/message/raw?msg=sci.crypt/mFMukSs0RmI/FpbHDQ6hM_MJ. Cited on pp.: 20 and 44.
49. Joseph H. Silverman. *The arithmetic of elliptic curves*. Graduate Texts in Mathematics 106. Springer-Verlag, 2009. Cited on pp.: 7.
50. W. A. Stein et al. *Sage Mathematics Software (version 6.8)*. The Sage Development Team, 2015. http://www.sagemath.org. Cited on pp.: 14 and 18.
51. Wikipedia. Nothing up my sleeve number, 2015. https://en.wikipedia.org/wiki/Nothing_up_my_sleeve_number (accessed 27 September 2015). Cited on pp.: 4 and 33.
52. Wikipedia. NewDES, 2015. https://en.wikipedia.org/wiki/NewDES (accessed 27 September 2015). Cited on pp.: 32.

A Full quote by Mike Scott

For context and full email see [48].

> [...] Consider now the possibility that one in a million of all curves have an exploitable structure that "they" know about, but we don't. Then "they" simply generate a million random seeds until they find one that generates one of "their" curves. Then they get us to use them. And remember the standard paranoia assumptions apply - "they" have computing power way beyond what we can muster. So maybe that could be 1 billion.
>
> A much simpler approach would generate more trust. Simply select B as an integer formed from the maximum number of digits of pi that provide a number B which is less that p. Then keep incrementing B until the number of points on the curve is prime. Such a curve will be accepted as "random" as all would accept that the decimal digits of pi have no unfortunate interaction with elliptic curves. We would all accept that such a curve had not been specially "cooked". So, sigh, why didn't they do it that way? Do they want to be distrusted?