In search of CurveSwap: Measuring elliptic curve implementations in the wild

Luke Valenta*, Nick Sullivan†, Antonio Sanso‡, Nadia Heninger*
*University of Pennsylvania, †Cloudflare, Inc., ‡Adobe Systems

Abstract—**We survey elliptic curve implementations from several vantage points. We perform internet-wide scans for TLS on a large number of ports, as well as SSH and IPsec to measure elliptic curve support and implementation behaviors, and collect passive measurements of client curve support for TLS. We also perform active measurements to estimate server vulnerability to known attacks against elliptic curve implementations, including support for weak curves, invalid curve attacks, and curve twist attacks. We estimate that 0.77% of HTTPS hosts, 0.04% of SSH hosts, and 4.04% of IKEv2 hosts that support elliptic curves do not perform curve validity checks as specified in elliptic curve standards. We describe how such vulnerabilities could be used to construct an elliptic curve parameter downgrade attack called CurveSwap for TLS, and observe that there do not appear to be combinations of weak behaviors we examined enabling a feasible CurveSwap attack in the wild. We also analyze source code for elliptic curve implementations, and find that a number of libraries fail to perform point validation for JSON Web Encryption, and find a flaw in the Java and NSS multiplication algorithms.**

1. Introduction

In 2015, Nick Sullivan outlined a theoretical parameter downgrade attack against TLS versions 1.0–1.2 which he named CurveSwap [45]. The main observation behind CurveSwap is that in the TLS handshake, the client's list of supported elliptic curves is not authenticated until the client finished message, and is authenticated only by the negotiated Diffie-Hellman secret. Thus if a man-in-the-middle attacker were able to precompute or solve an elliptic curve discrete log online for some curve, they could downgrade the connection to use that weak curve, allowing them to decrypt or modify the encrypted communications. The attack was inspired by the FREAK [11] and Logjam [6] cipher suite downgrade attacks against TLS.

In his 31C3 presentation, Sullivan concluded that the weakest commonly supported curve was sect163k, supported by 4.3% of sampled clients and 0.13% of the Alexa top 100,000 web sites. Since a 160-bit elliptic curve discrete log has yet to be publicly demonstrated, let alone computed within a TLS handshake timeout, the attack appeared to remain theoretical.

In this paper, we evaluate the feasibility of a practical CurveSwap attack by exploring the protocol-level and implementation-level attack surface of elliptic curve usage in TLS, IPsec, SSH, and JSON Web Encryption (JWE). There are a number of potential vulnerabilities in elliptic curve implementations that taken in combination could enable a CurveSwap attack, including support for curves of small order, point validation failures, and twist insecurity. We performed extensive passive and active measurements of these behaviors and implementation choices among clients and servers. Among our scans, we found populations of servers that accept invalid curve points years after flaws have been publicly disclosed and patched in common libraries, little vulnerability to twist attacks, and significant populations of hosts that repeat public key exchange values both across IP addresses and across multiple scans. However, these behaviors were not present in combinations that would lead to an effective attack for vulnerable curves. Ultimately we conclude that TLS, IPsec, and SSH do not appear to be vulnerable on any significant scale to a feasible CurveSwap attack based on the vectors we evaluated.

Some protocol designs are much more resistant to CurveSwap-style downgrade attacks than others. We observe that the design of SSH and TLS 1.3, where the server uses their long-term authentication key to sign the entire handshake, are much more resistant to parameter downgrade attacks like CurveSwap than earlier versions of TLS.

Our survey of elliptic curve support for TLS, IPsec, and SSH gives a snapshot of elliptic curve deployments in 2017. The NIST-standardized curve secp256r1 is the most widely supported curve in our measurements, while support for other curves in our data was in general lower, with a long tail of more unusual standardized curves. Curve support varied wildly by protocol. We found small but nontrivial support for a number of 160-bit curves that only offer 80 bits of security, although only a negligible number of clients or servers preferred these curves over stronger curves. We were surprised to discover that very few hosts supported secp224r1 on any protocol, many hosts failed to respect a client's selection of elliptic curves, and that essentially no TLS hosts servers supported custom curves.

We also extensively examined source code, and discovered several vulnerabilities. The JWE protocol standard fails to mention that implementations need to perform curve validity checks, and we discovered a number of JWE libraries that were vulnerable to a classic invalid curve attack allowing an attacker to recover the private key, including Cisco's node-jose, jose2go, Nimbus JOSE+JWT and jose4j. We also discovered flaws in NSS and Java's scalar point multiplication routines that could cause them to output incorrect results given certain inputs, although these flaws do not appear to be exploitable.

1.1. Our Contributions

In this paper, we perform a broad survey of elliptic curve cryptography on the public Internet. The maze of different standards, curves, and implementation choices for elliptic curve cryptography makes a holistic evaluation of our cryptographic infrastructure quite challenging. We measure the landscape of elliptic curve implementations on the Internet with passive and active measurements, describe known and new attack vectors against ECC, and examine source code to find implementation vulnerabilities.

- **Active Measurements** We perform Internet-wide scans of TLS, SSH, and IPsec servers to measure elliptic curve support and implementation behaviors.
- **Passive Measurements** We measure TLS client support and preferences for elliptic curves.
- **Protocol Analysis** We explore analogues of CurveSwap for IPsec and SSH. We also survey attacks against elliptic curves and evaluate their impact on the CurveSwap attack for TLS.
- **Source Code Analysis** We extensively examined source code and found widespread invalid curve vulnerabilities in JWE libraries, as well as flawed scalar multiplication routines in Java and NSS.

Although some elliptic curve implementations have fallen victim to known implementation pitfalls, for TLS, SSH, and IPsec, most hosts appear to resist known attacks. We conclude that protocol designers should continue to build in defense in depth.

1.2. Disclosure and Mitigations

In February 2017 we submitted bug reports to the developers of several libraries implementing JSON Web Encryption (JWE, RFC 7516) that were vulnerable to invalid curve attacks, including Cisco's node-jose, jose2go, Nimbus JOSE+JWT and jose4j. They have all acknowledged the issue and released a patch. We also described the nature of the invalid curve attack applied to JWE in a blog post [38]. We reported the NSS vulnerability to Mozilla in March 2017. NSS fixed the issue in the 3.31 release. We reported the Java vulnerability to Oracle in March 2017. Oracle issued a patch that fixes the issue on July 18, 2017. We also disclosed these vulnerabilities to the public in a blog post [39].

2. Preliminaries

Elliptic curve cryptography can be used for key exchange, asymmetric encryption, or for signatures. Among widely implemented public key primitives, elliptic curves offer the best resistance to cryptanalytic attacks on classical computers, and as a result can be used with smaller key sizes than RSA or finite field based discrete logarithm schemes. In this paper, we focus on elliptic curve Diffie-Hellman key exchange.

2.1. Elliptic Curve Cryptography

A number of standards exist defining elliptic curves for use in cryptography. In 2000, the Certicom SECG published the SEC 2 specification [40] giving parameters for 33 elliptic curves of varying sizes and properties. Several of these curves were later standardized by NIST, ISO, and ANSI under different names. Other proposals for curves include the Oakley elliptic curve groups [37], the Brainpool curves [33], and more recent constructions such as Curve25519 [8], Curve41417 [9], and Curve448 [24].

2.1.1. Prime curves. An elliptic curve $E(\mathbb{F}_p)$ over a prime finite field $\mathbb{F}_p$ with $p \neq 2$ is the set of points $P = (x, y) \in \mathbb{F}_p^2$ that are solutions to some equation E over $\mathbb{F}_p$, together with an extra point $\mathcal{O}$, the point at infinity. It is possible to define an addition law, so that these points form a group.

Such curves are often specified in Weierstrass form $E : y^2 = x^3 + ax + b \pmod p$ where $a, b \in \mathbb{F}_p$ are domain parameters that define the curve. Every elliptic curve over a finite field $\mathbb{F}_p$ of a prime order can be converted to this form. Some widely-used examples of prime curves are the NIST curves from FIPS 186-4 [30] and the Brainpool curves [33].

Cryptographic applications typically work within a cyclic subgroup of prime order n. This group will be generated by a base point $G \in E(\mathbb{F}_p)$.

One can compute an element kG of this group using a scalar-by-point multiplication algorithm. The underlying hardness assumption in most elliptic curve cryptography is the elliptic curve discrete logarithm problem: given an elliptic curve $E(\mathbb{F}_p)$, a generator G, and a point P it is hard to find a k satisfying $P = kG$. The best known algorithms for solving the elliptic curve discrete logarithm problem run in square root time in the order of the subgroup generated by the elliptic curve's generator.

2.1.2. Binary curves. Elliptic curves over characteristic 2 finite fields $\mathbb{F}_{2^m}$ are specified as the set of points $P = (x, y) \in \mathbb{F}_{2^m}^2$ that are solutions to the equation $E : y^2 + xy = x^3 + ax^2 + b$ in $\mathbb{F}_{2^m}$.

Recent progress on the elliptic curve discrete logarithm problem for small-characteristic fields has raised concern about the security of binary curves, although there are not yet any subexponential time attacks against curves standardized for use in the network protocols we study in this paper [23], [42].

The SEC 2 standard [40] includes parameters for a number of binary curves. The Oakley elliptic curve groups [37] are also binary curves.

2.1.3. Domain parameters. An elliptic curve group is defined by a set of domain parameters which consist of the following values: q, an integer that defines the order of the finite field $\mathbb{F}_q$ of the curve; a and b, the coefficients of the curve equation; G, a generator of a subgroup of prime order on the curve; n, the order of the subgroup that G generates; and h, the cofactor, which is equal to the number of curve points w divided by n.

2.2. ECDH Key Exchange

In this paper, we are primarily interested in elliptic curve Diffie-Hellman key exchange. To negotiate a shared secret using ECDH, Alice generates a random private key k_a, generates her public value $Q_a = k_a G$, and sends Q_a to Bob. Bob generates a random private key k_b, generates his public value $Q_b = k_b G$, and sends Q_b to Alice. Alice can then compute the shared secret as $P = k_a Q_b$ and Bob can compute it as $P = k_b Q_a$. Real-world protocols then use P to derive symmetric keys that Alice and Bob use to establish an authenticated and encrypted communication channel.

2.2.1. Scalar-by-point multiplication algorithms. The most important operation on elliptic curves for the cryptographic algorithms we study in this paper is scalar-by-point multiplication. That is, given a point P on an elliptic curve and an integer k, compute the curve point kP.

Point representation. Elliptic curve points can be represented in many different forms. The canonical representation uses *affine coordinates*, where a point on the curve is represented by a pair of integers (x, y) that satisfy the curve equation. This is called *uncompressed* point format. However, this representation requires an expensive field inversion operation to add two elliptic curve points.

Most applications of elliptic curves use only the x-coordinate of a point. A valid x-coordinate could correspond to two possible y coordinates of points on the curve, the point (x, y) or the point $(x, -y)$; these can be recovered from x using the curve equation. Thus a point can be uniquely represented by sending only the x-coordinate and the sign of the y-coordinate; this is called *compressed* format.

Double and add. The simplest algorithm to compute scalar-by-point multiplication is double-and-add. This algorithm iteratively applies the group addition law and a doubling procedure. There are a number of variants of this algorithm, such as sliding windows. However, this algorithm has the drawback that it is not secure against side channel attacks. It also requires both the x and y coordinates of the input points.

Montgomery ladder. Some elliptic curves can also be specified in Montgomery form [34]: $E : By^2 = x^3 + Ax^2 + x$. An advantage of this form is that it allows a very fast algorithm for scalar-by-point multiplication using only the x coordinate, the Montgomery ladder.

The single-coordinate version of the Montgomery ladder algorithm for scalar-by-point multiplication requires fewer arithmetic operations than standard Weierstrass scalar-by-point multiplication methods and offers better side channel resistance [29], [36]. Curve25519, introduced by [8], is specified in Montgomery form, as are Curve41417 [9] and Curve448 [24] (the Goldilocks curve).

Brier-Joye. It is possible to compute an x-coordinate only scalar multiplication for Weierstrass-form elliptic curves using the Brier-Joye ladder [15]. This algorithm is constant time and has good side channel resistance. Unfortunately, it is slow.

2.3. Invalid Point Attacks

For most curves, ECDH implementations must validate that the public key exchange messages they receive are valid points on the correct elliptic curve, otherwise they may be vulnerable to a variety of attacks.

2.3.1. Small subgroup attacks. Small subgroup attacks against prime-field Diffie-Hellman were described by Lim and Lee [32]. In this type of attack, the cryptographic domain parameters specify a subgroup within a larger group. If the cofactor of the order of the correct subgroup has small prime factors p_i, an adversary could send a key exchange that lies in a subgroup of order p_i instead of the correct subgroup and use the victim's response to deduce the victim's secret modulo p_i. The attacker can then repeat this attack for different primes and use the Chinese remainder theorem to reconstruct the victim's secret modulo the product of these primes.

Elliptic curves that are standardized for cryptographic use are typically chosen to have small cofactors to limit the number of elements of small order on the curve and to limit the checks required to protect against these small subgroup attacks [8]. NIST recommends a maximum cofactor for various curve sizes [30]. The NIST curves specified in FIPS 186-4 have cofactor 1, 2, or 4. Curves in Montgomery form always have a cofactor that is a multiple of 4 [34].

One can also protect against this type of attack by checking that a received point P has the correct group order by checking that $nP = \mathcal{O}$. Alternatively, one can use ECDH with cofactor multiplication, in which both parties multiply their Diffie-Hellman shared secret by h [41].

2.3.2. Invalid curve attacks. A double-and-add-based implementation of scalar multiplication that does not validate key exchange values is vulnerable to a much more severe invalid curve attack. In an invalid curve attack, the attacker sends an elliptic curve point of small order that lies on a *different curve*. This attack is due to Antipa et al. [7].

In a Weierstrass-form curve, textbook double-and-add algorithms are independent of the curve parameter b, so an attacker can search for values b' such that a curve $E' : y^2 = x^3 + ax + b'$ has points $P_i = (x_i, y_i)$ of small order q_i and send them to the victim. If the victim does not verify that the received key exchange value and computed shared secret are on the correct curve and has the correct order, the victim's response may allow the attacker to compute the victim's secret key modulo q_i.

In contrast to the Lim-Lee attack for prime-field Diffie-Hellman where an attacker is limited to the prime factors of the cofactor of the correct subgroup, the attacker in this elliptic curve scenario has much more leeway in choosing curves that have points of suitably small coprime order.

This attack can be prevented if an implementation validates that the points it receives lie on the correct curve. This attack is also somewhat mitigated by scalar-by-point multiplication algorithms that use only the x-coordinate, although these may be vulnerable to twist attacks, described below.

2.3.3. Curve twist attacks. A Weierstrass curve of the form $E : y^2 = x^3 + ax + b \bmod p$ is related to a twisted curve, $E' : dy^2 = x^3 + ax + b$.

Any x-coordinate has an associated pair of y coordinates that are either on the original curve or some twisted curve. If d is a quadratic residue, i.e., if there is a w with $w^2 = d \bmod p$, then E and E' are isomorphic mod p and thus have equivalent security. If d is a quadratic non-residue, E' is not isomorphic to E and the curve orders satisfy $|E| + |E'| = p + 2$. This is called a *nontrivial quadratic twist*.

An implementation that uses a single-coordinate ladder such as the Montgomery ladder might be vulnerable to a form of invalid curve attacks in which the attacker sends an x-coordinate that lies on a weak twist of the correct curve. This type of attack is due to Foque, Lercier, Réal, and Valette [21].

The NIST-standardized curves secp192r1 and secp224r1 have weak twists that reduce the cost of such an attack to 2^{48} and 2^{59}, respectively [10], [21]. The binary curves ec2n_155 and ec2n_185 also have weak twists which reduce the attack cost to 2^{33} and 2^{47}, respectively. secp256r1 and secp384r1 have secure twists. Recent curve constructions such as Curve25519 were explicitly designed to have strong twists and not require an additional validation step. Otherwise, implementations must verify that the received coordinate lies on the correct curve.

2.3.4. Curve downgrade attacks. In a curve downgrade attack, a man-in-the-middle adversary interferes with a connection to cause the communicating parties to choose a weaker curve than they would otherwise negotiate. In Section 5, we present the CurveSwap attack against TLS, and study the feasibility of similar curve downgrade attacks against SSH and IPsec.

2.4. ECC in TLS

Elliptic curve use in TLS versions 1.2 and earlier is specified by RFC 4492 [13]. Elliptic curves can be used in static elliptic curve Diffie-Hellman (ECDH) and ephemeral ECDH (ECDHE) key exchange, and ECDSA signatures. In this paper, we focus on ECDHE key exchange.

Clients declare support for elliptic curves by including ECHD(E) cipher suites in their list of supported cipher suites and via the supported elliptic curves and the supported points format extensions in the client hello message. This message consists of a list of supported elliptic curves sorted by client preference, and a list of the point formats that the client can parse. The list of supported elliptic curves can include 25 of the named curves specified in SEC 2 [40], and can also indicate support for arbitrary explicit prime or binary curves.

If the server chooses an ECDHE cipher suite, the server key exchange message includes an indication of the server's chosen curve (either named or a set of parameters for an explicit curve), the server's public key exchange value given as the encoding type and a byte string representing an elliptic curve point, and a digital signature on these two values using the server's certificate key. Servers typically select the most secure elliptic curve supported by the client, but may be configured to respect client preference. If the server has a preferred list of curves and the client supports an overlapping set of curves, any connection between the two will use the preferred curve of the server.

The client key exchange message includes the client's public key exchange value on the negotiated curve, which specifies the encoding type and a byte string representing an elliptic curve point.

The premaster secret is computed as the x-coordinate of the ECDH shared secret elliptic curve point. The premaster secret is then used to derive a set of encryption and authentication keys. The client uses the derived keys to authenticate the entire handshake in the client finished message, and the server does the same in the server finished message.

In TLS 1.3, only (EC)DHE key exchange methods are allowed, the keying material is derived from the hash of the entire transcript of the handshake as described in RFC 7627 [12], and the server signs the hash of the transcript with its certificate key, which prevents any type of downgrade attack other than a full man-in-the-middle attack by an attacker who has compromised the server's private certificate key.

2.5. ECC in SSH

Elliptic curve use in SSH is specified by RFC 5656 [44]. Elliptic curves can be used in ECDH or ECMQV key exchange and ECDSA for digital signatures. In the SSH handshake, both client and server send a list of supported encryption algorithms in their KEXINIT message, and negotiate an algorithm from among the algorithms both support. Supported curves are listed as separate cipher choices for key exchange and signature algorithms. RFC 5656 specifies that SSH implementations must support secp256r1 (nistp256), secp384r1 (nistp384), and secp521r1 (nistp521), and lists 9 additional curves from NIST and SEC2 standards as recommended. Point compression is optional.

If client and server negotiate an ECDH key exchange with a specific curve, the client sends its public key exchange value first. The server then responds with its long-term public host key, its public ECDH key exchange value, and a digital signature using the server's host key over the client and server KEXINIT messages, the server's public host key, the client and server key exchange messages, and the negotiated shared secret. SSH uses ECDH with cofactor multiplication to derive the shared secret.

2.6. ECC in IPsec

IPsec uses the Internet Key Exchange (IKE) protocol to negotiate an encrypted and authenticated session. There are two versions of the IKE protocol, IKEv1 and IKEv2. Both rely on Diffie-Hellman key exchange over a set of fixed, standardized groups to negotiate a shared secret. Cremers [17] carried out an automated analysis of the key

				Number of hosts that support…						
Proto	**Port**	**Date**	**BASE**	**ECDHE**	**secp224r1**	**secp256r1**	**secp384r1**	**secp521r1**	**x25519**	**b-pool256r1**
TLS	443	11/2016	38.6M	24.8M	643.4K (2.6%)	24.1M (97.0%)	5.7M (22.9%)	2.5M (10.2%)	0 (0.0%)	980.1K (3.9%)
	443	08/2017	41.0M	28.8M	811.6K (2.8%)	25.0M (86.9%)	9.1M (31.6%)	2.2M (7.7%)	740.7K (2.6%)	2.4M (8.4%)
SSH	22	11/2016	14.5M	7.9M	0 (0.0%)	7.7M (97.8%)	7.5M (95.6%)	7.5M (95.4%)	6.1M (77.2%)	0 (0.0%)
IKEv1	500	11/2016	1.1M	215.4K	143.8K (66.8%)	211.8K (98.3%)	206.8K (96.0%)	152.8K (71.0%)	0 (0.0%)	0 (0.0%)
IKEv2	500	11/2016	1.2M	101.1K	4.1K (4.1%)	98.2K (97.1%)	98.0K (96.9%)	240 (0.2%)	0 (0.0%)	0 (0.0%)

TABLE 1. **Server Supported Curves**—*BASE* gives the number of hosts that we were able to negotiate any key exchange with and *ECDHE* gives the number that support ECDHE key exchange. Percentage support for each curve is with respect to *ECDHE*.

agreement protocols in IKEv1 and IKEv2 and found a number of vulnerabilities.

The original IKEv1 protocol specified two optional binary curves, ec2n_155 (Oakley Group 3), a 155-bit binary curve, and ec2n_185 (Oakley Group 4), a 185-bit binary curve, among the four groups for Diffie-Hellman key exchange. (The other two were 768-bit and 1024-bit primes for prime field Diffie-Hellman.) Additional optional binary and prime curves, including the curves from SEC 2, NIST, and Brainpool, have been registered with IANA for IKEv1 and IKEv2 over the course of several RFCs, including RFC 5903 [22], RFC 5114 [31], and RFC 6932 [25].

RFC 2409 specifies that the key exchange value for Oakley groups 3 and 4 consists of the x-coordinate, and the y-coordinate is derived as necessary and not used to derive the shared key. However, RFC 4753 specifies that implementations should send both x and y as the Diffie-Hellman public value and use both in the shared secret.

2.6.1. IKEv1.

IKEv1 is specified in RFC 2409. There are two types of handshakes, Main Mode, which requires six messages to establish the connection, and Aggressive mode, which requires three. In main mode, the initiator sends a Security Association (SA) payload, which specifies a collection of cipher suites and Diffie-Hellman groups they support. The responder sends its own SA payload containing its selected cipher suite. The initiator and responder then send key exchange messages for the chosen group. Both parties are then able to compute shared key material, called SKEYID. The computation of SKEYID depends on the authentication method. When signatures are used for authentication, SKEYID = $\mathrm{prf}(N_i|N_r, k_ik_rP)$ where k_ik_rP is the negotiated Diffie-Hellman secret. For the other two authentication methods, public-key encryption and pre-shared key, SKEYID does not depend on the negotiated Diffie-Hellman shared secret, and instead is derived from the cookie or the pre-shared key respectively. Each party authenticates itself by sending an authentication message (AUTH) derived from a hash of SKEYID, the public Diffie-Hellman key exchange messages, the cookies, the initiator's security association, and initiator and responder IDs. In main mode, these authentication messages are encrypted and authenticated using keys derived from the negotiated Diffie-Hellman secret.

In aggressive mode, it is not possible to negotiate the group for Diffie-Hellman. The initiator sends SA and KE messages together, and the responder sends its SA, KE, and AUTH messages together. The initiator finally responds with its AUTH message. The authentication messages are not encrypted.

2.6.2. IKEv2.

IKEv2 combines the SA and KE messages into a single message. The initiator provides a best guess ciphersuite for the KE message. If the responder accepts that proposal and chooses not to renegotiate, the responder replies with a single message containing both SA and KE payloads. Both parties then send and verify AUTH messages, starting with the initiator. The authentication messages are encrypted using session keys derived from the SKEYSEED value which is derived from the negotiated Diffie-Hellman shared secret. The standard authentication modes use public-key signatures over the handshake values.

3. Related Work

Bos et al. [14] surveyed elliptic curve adoption rates in 2014, and found that approximately 10% of TLS and SSH hosts supported elliptic curve cipher suites. The ICSI Certificate Notary [27] publishes ongoing statistics on observed SSL/TLS ciphersuites in connections originating from ten research institutes, and reports that at least 88% of connections used ECDHE key exchange in July/August 2017.

Jager, Schwenk, and Somorovsky [28] manually examined ECDH implementations in eight popular TLS libraries in 2015, and found that three of them failed to validate elliptic curve points, leading to full private key recovery for Oracle's default Java JSSE TLS implementation and BouncyCastle. Their analysis was only performed in local test environments. We are unaware of prior work measuring elliptic curve point validation.

Valenta et al. [46] studied prime-field Diffie-Hellman implementations in TLS, SSH, and IPsec in 2016 using both internet-wide scans and source code examination, and found that most examined implementations did not validate subgroup order. Springall, Durumeric, and Halderman [43] measured DHE and ECDHE key exchange reuse among Alexa Top 1 Million domains and found that 1.5% of HTTPS domains supporting ECDHE repeated the same key exchange value in multiple scans, and noted one service that repeated the same key exchange value for 61 days.

4. Elliptic Curve Measurements

In this section, we present our measurements of elliptic curve implementations for TLS, SSH, and IPsec.

4.1. Server Curve Support and Preferences

The popularity of different curves varies depending on the protocol. In this section, we describe measurements we performed to understand server curve support for various common ports and protocols, to give a snapshot of elliptic curve deployments.

4.1.1. Server scanning methodology. We performed our scans between November 2016 and August 2017 from the University of Pennsylvania. We used the Zmap [20] Internet-wide scanning tool to perform 10% scans of the IPv4 address space. We extended the Zgrab protocol parser for TLS and SSH to include support for the numerous curves we tested, and used our own Zgrab module for IKEv1 and IKEv2.

For most of our measurements, we scanned a random 10% sample of the public IPv4 Internet on a selection of common ports for TLS, SSH, and IPsec. Unless otherwise specified, the results we present in this paper are extrapolations of our 10% scans to the full IPv4 space, to simplify comparison with other measurements.

For each scan, we first perform a Zmap scan of a randomly selected set of hosts to detect whether a particular port was live. Then, we perform repeated scans of the set of responding hosts using the Zgrab protocol module to detect fine-grained behaviors and support for various cryptographic parameters.

In a TLS and IKE ECDH key exchange, a curve can only be negotiated if it is supported by both the client and the server. To measure support for the elliptic curves shown in Table 1 for TLS, we use Zgrab to perform multiple TLS handshakes, each only offering a single curve at a time in the supported curves extension. For IKE, we offer a security association that includes a curve together with a variety of popular cipher proposal options. In SSH, both the client and server send the list of curves they support, so we can gather curve support from a single scan.

In order to get a baseline measure of support for each protocol, we used scans offering a variety of parameters. The Censys project [18] performs regular 100% TLS and SSH scans using Zmap, so we used their scans from November 2016 and August 2017 as a baseline for support for those protocols. We performed our own 100% IKEv1 and IKEv2 baseline scans.

4.1.2. Server measurement limitations. The survey of Durumeric et al. [19] provides a view of Internet-wide scanning, documenting both the advantages and limitations of the approach. In short, scanning does not allow us to measure hosts that are behind firewalls or are otherwise configured to reject scanning attempts, or hosts whose network operators have requested to be excluded from our scans. Our scans are further restricted to IPv4 hosts, as scanning the IPv6 space efficiently remains an open problem. Despite these limitations, Internet scanning remains an invaluable tool for network operators and defensive security research.

Due to the large number of scans required to measure the selected combinations of server behaviors for our study, we chose to limit each scan to only 10% of the public IPv4 space instead of performing full IPv4 scans. However, we do not expect this to limit the statistical accuracy of our measurements, although we may occasionally miss rare server behaviors.

4.1.3. Server curve support. ECDH is widely supported by TLS and SSH hosts. We find that 64% of HTTPS hosts and 54% of SSH hosts support ECDH key exchange. As a comparison, Bos et al. [14] report that 7.2% of 30 million HTTPS hosts and 13.8% of 12 million SSH hosts that responded to a ZMap scan in October 2013 supported some form of ECDH key exchange. Adoption of ECDH using common curves for IKE appears to be significantly slower.

Table 1 shows the result of 10% scans extrapolated to full IPv4 scans. We omitted Curve25519 from the November 2016 TLS and IPsec scans since support for this curve was not standardized at the time of the scans. However, we performed additional TLS scans in August 2017 to provide up-to-date numbers on Curve25519 deployment.

The NIST curves secp256r1, secp384r1, and secp521r1 were the most commonly supported curves among servers, but support for each curve varies widely by protocol. secp256r1 was the most popular curve among TLS on port 443, SSH, and IPsec. Support for secp224r1 was surprisingly rare, except for IKEv1. There is a long tail of curve support for other curves in the IANA registries for each protocol; in Section 6.1.2 we give measurements for a number of weak curves.

We performed 10% IKEv1 and IKEv2 scans offering the binary curves ec2n_155 and ec2n_185, but did not detect any hosts that were willing to negotiate these curves. We found two IKE implementations that documented support for these Oakley groups for backwards-compatibility: MikroTik [4] and OpenBSD's iked [5]. We verified that OpenBSD's implementation does indeed support these binary curves by running our scans against an OpenBSD 6.12 instance running in a VM.

TLS also allows servers to specify a custom curve using arbitrary_explicit_prime_curves and arbitrary_explicit_char2_curves. We also performed 10% TLS scans requesting these custom curves, and received no responses on any of the tested ports.

We give full scan data in Appendix B for additional TLS ports.

4.2. Client Curve Support and Preferences

4.2.1. Client data methodology. We study client preferences using a sample of client hellos provided by Cloudflare, a popular web performance and security service.

Cloudflare acts as a reverse proxy for web services: when a client connects to a site that uses Cloudflare, a TLS

Supported Curves	User Agents	Operating Systems	Count
23,24,25	FireFox/46.0-49.0, FitbitMobile/2.28, IE/11.0, uservoice-android-1.2.4, Safari/9.0, Tinder/63105	Win7, Win8, Win10, iOS	1.5M (35.9%)
29,23,24	Chrome/50.0-54.0	Win7, Win8, Win10, Mac OSX, Chrome OS	909.0K (21.7%)
23,24	IE/11, Edge/13.0, Chrome/47.0-51.0	WinVista, Win7, Win8, Win10	661.7K (15.8%)
14,13,25,11,12,24,9,10,22,23,8,6, 7,20,21,4,5,18,19,1,2,3,15,16,17	uservoice-android-1.2.4, Picsart/3.0, okhttp/3.2.0, Playstation/4, Netscape/4.0, Python-urllib/2.7	Win7, Win10, Other	621.3K (14.8%)
25,24,23	SamsungBrowser/2.0-2.1, Wget/1.12	Android, Other	184.4K (4.4%)
23,25,28,27,24,26, 22,14,13,11,12,9,10	Chrome/47.0-53.0, Deluge/1.3.12, Plex Music Agent/1.0, qBittorrent/3.3.7, Transmission/2.84	Win7, Win8, Win10, Other	40.1K (1.0%)
empty	libhttp/3.50, libhttp/4.01, Chrome/25.0	Linux, PlayStation/4	24.9K (0.6%)

TABLE 2. **Client Supported Curves Extensions with User Agents**—We show the ranked list of the most common supported curves lists along with the user agents and operating systems of the clients for a sample of 4,187,201 client hellos collected from Cloudflare. The mapping of curve IDs in the supported curves list to curve names is maintained by IANA [26].

connection is established with a geographically proximal server operated by Cloudflare. This server handles incoming HTTP requests from the client. If a request is for a resource that is cached by Cloudflare, that resource is returned to the client in the response; if the resource is not cached, the Cloudflare server forwards the request to the origin server to obtain a response, which is then returned to the client.

We examined the contents of the TLS client hello together with the client's HTTP user agent string from a uniform sample of incoming HTTPS connections to Cloudflare servers around the world over an approximately 5 minute period on October 17, 2016. 99.4% of the 4.2M client hellos in the sampled traffic included the supported curves extension. At the time of the measurement, Cloudflare was used as an HTTP/HTTPS reverse proxy for over six million domains.

4.2.2. Client measurement limitations. The client dataset that we gathered, while insightful, has multiple limitations. First, the request samples are skewed toward users who were awake and active during the collection period. Collection over a longer period of time might produce a distribution that is more representative of all users. Second, since our data is a raw sample of Cloudflare requests, popular Cloudflare customers are overrepresented in our dataset. Thus, the composition of the data is likely not representative of the web as a whole. We were unable to obtain captured requests from other data sources at the same scale for comparison. Finally, a nontrivial number of requests are from non-browser traffic, including requests from API clients, automated scripts, mobile applications, crawlers, and other bots. This adds depth to the dataset, but means that the dataset does not necessarily reflect the stereotypical view of web traffic as coming exclusively from human-controlled web browsers.

4.2.3. Client curve support. Table 2 summarizes several of the most common orderings of the supported curves list among sampled clients, using the IANA IDs for each curve. We used Browscap [1] to map software versions to the provided user agent strings. The most common curve preference ordering requests the NIST curves secp256r1, secp384r1, secp521r1 in increasing order of strength, which was provided by a variety of clients. The second most common curve preference ordering in our sample preferred Curve25519, from recent versions of Chrome. The next most common client curve preference ordering in our sample, apparently requested by various APIs, requests most of the curves from SEC 2 in decreasing order of strength.

4.3. Repeated Key Exchange Values

For performance reasons, a common behavior among servers is to reuse the same key exchange value for multiple connections, to avoid the need to recompute this value for each client. To detect this behavior, we scan each server twice in rapid succession and check if the key exchange value changes. In Table 3, we offer secp256r1 as the key exchange value, and collect the key exchange values in the server responses.

22% of hosts on TLS port 443 (primarily HTTPS) repeated the same key exchange value in successive scans. 2.6% of TLS port 443 hosts served a non-unique key exchange value that was shared by at least one other host in the same scan. This could be due to shared hosting providers configured with ephemeral-static key exchange, or random number generation issues.

4.4. Other Observations

Our scans uncovered some other interesting server behaviors.

4.4.1. TLS servers ignoring client supported curves. We found that some TLS servers appear to ignore the curves

			Repeats...	
Proto	**Port**	**secp256r1**	**Across Hosts**	**By Host**
TLS	443	24.0M	638.7K (2.7%)	5.5M (22.9%)
SSH	22	7.5M	0 (0.0%)	0 (0.0%)
IKEv1	500	168.5K	210 (0.1%)	540 (0.3%)
IKEv2	500	95.1K	800 (0.8%)	1.9K (1.9%)

TABLE 3. **Repeated Key Exchanges**—In November 2016, we scanned a randomly selected 10% of IPv4 addresses twice in rapid succession, offering curve secp256r1. *Across Hosts* gives the number of hosts that sent the same key exchange value as another host within a single scan, and *By Host* shows the number of hosts that sent the same key exchange value in both scans.

Client Supported Curve	**Server Key Exchange**	**Hosts**
brainpoolp256r1	secp256r1	849.4K
brainpoolp256r1	secp384r1	428
brainpoolp256r1	secp521r1	47
secp224r1	secp256r1	850.0K
secp224r1	secp384r1	474
secp224r1	secp521r1	46
secp256r1	secp384r1	506
secp256r1	secp521r1	49
secp384r1	secp256r1	849.9K
secp384r1	secp521r1	45
secp521r1	secp256r1	849.7K
secp521r1	secp384r1	429

TABLE 4. **Servers Ignoring Client Supported Curves**—In our scans, we found that some servers responded with the same curve regardless of client's list of supported curves. RFC 4492 states that a server must not negotiate the use of an ECC cipher suite if it is not able to complete an ECC handshake with the parameters offered by the client [13].

sent in the client supported curves extension, and instead reply with the same curve regardless of whether or not the client indicated support. Across all of the TLS scans we performed in November 2016, we found that 25%, or 8.5M distinct hosts out of 34.6M total hosts returned a server key exchange value specifying a curve that was not present in the client supported curves extension. In Table 4, we show the number of hosts that responded to our scans with an unsupported curve. It appears that these hosts always attempt to negotiate either secp256r1, secp384r1, or secp521r1 rather than terminate the connection when the client offers a curve that they do not support.

In order to understand whether this might be a vulnerability, we experimentally compared responses when our scanner client offered a point on secp256r1 versus the curve that was originally specified by the client. No servers who sent a point on an incorrect curve accepted a point on the curve that the client originally requested.

4.4.2. Scalar multiplication algorithms. We also performed scans offering points on the twist of the curve. As discussed in Section 2.2.1, TLS implementations do not appear to use single-coordinate ladders for point multiplication, and thus reject points on the twist of the curve. We suspect that hosts that accept invalid curve points but do not accept points on the twist as the client key exchange value are using a mixed-Jacobian scalar-by-point multiplication algorithm, which would cause points on the twist to fail with an arithmetic error but would succeed for points on an invalid curve. However, as shown in Table 7, small numbers of SSH and IKE hosts accepted key exchange values on the twist, suggesting that they may use single-coordinate ladders.

4.4.3. Echo servers. In our IPsec scans, we found that some of the repeated server key exchange values that we observed could be attributed to servers that simply echoed back the same static key exchange value and nonce that we offered in the scan. There were 30 IKEv1 hosts and 25 IKEv2 hosts that exhibited this behavior. These hosts appear to simply echo back an identical copy of any data that they receive. We omit these hosts from the results presented in Table 3.

5. CurveSwap Attack

The CurveSwap attack was introduced by Nick Sullivan in 2015 [45]. It is a theoretical attack targeting the curve negotiation to be performed against TLS deployments. Similar in spirit to the FREAK [11] and Logjam [6] attacks, CurveSwap allows a man-in-the-middle to trigger a downgrade attack to force a connection to use the weakest elliptic curve that both parties support. The CurveSwap attack is a parameter negotiation downgrade attack, and can be performed if both client and server support an elliptic curve for which an attacker can break ECDH, either by solving the discrete log or other means. The existence of this attack reduces the overall security of a connection to the security of the weakest elliptic curve supported by both parties.

5.1. CurveSwap for TLS

As explained in Section 2.4, a TLS client and server use the supported curves extension [13] to specify which curves each party supports in order to negotiate an elliptic curve group for use in key establishment. The CurveSwap attack demonstrates that in TLS 1.2 and earlier, a man-in-the-middle that can break ECDH for the weakest curve supported by both parties can compromise a connection.

In Figure 1, we depict the CurveSwap attack in a TLS handshake. To mount a CurveSwap attack, the attacker needs to be in a position to man in the middle a connection. When the client sends its client hello message to the server, the attacker replaces it with a client hello message where the client cipher suite list contains only ECDHE cipher suites, and the supported curves extension only contains weak curves.

The server will then reply with its ECDHE public key exchange value on the attacker's chosen weak curve. The attacker passes this message back to the client without

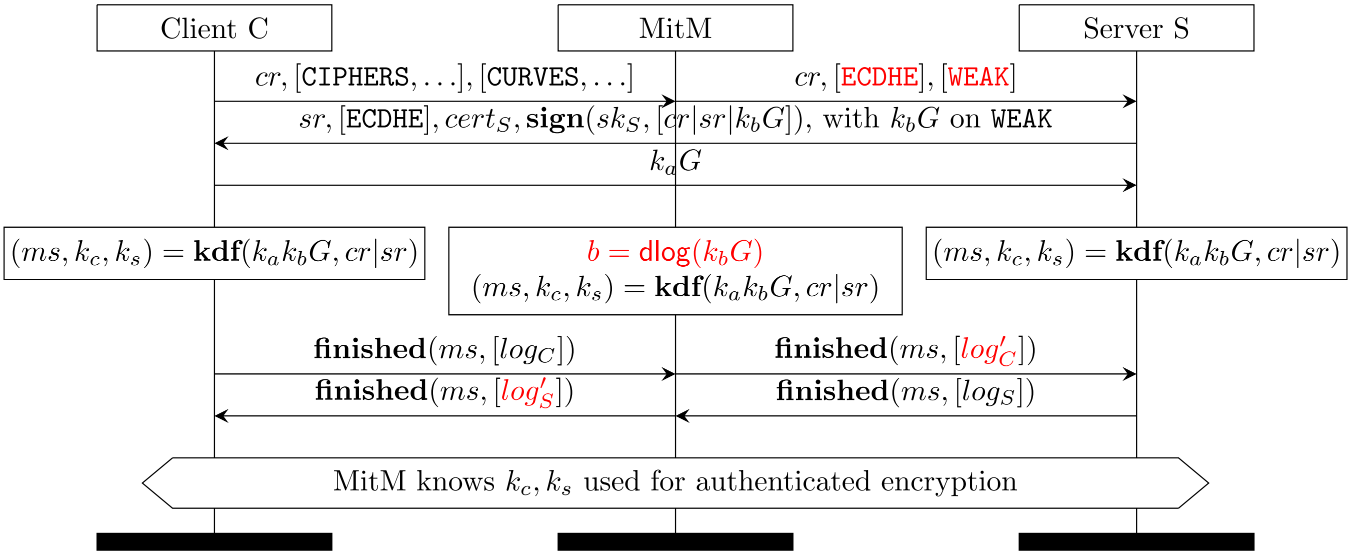
Client C
MitM
Server S
cr, [CIPHERS, . . .], [CURVES, . . .]
cr, [ECDHE], [WEAK]
sr, [ECDHE], $cert_S$, $\mathbf{sign}(sk_S, [cr|sr|k_bG])$, with k_bG on WEAK
k_aG
$(ms, k_c, k_s) = \mathbf{kdf}(k_ak_bG, cr|sr)$
$b = \text{dlog}(k_bG)$
$(ms, k_c, k_s) = \mathbf{kdf}(k_ak_bG, cr|sr)$
$(ms, k_c, k_s) = \mathbf{kdf}(k_ak_bG, cr|sr)$
$\mathbf{finished}(ms, [log_C])$
$\mathbf{finished}(ms, [log'_C])$
$\mathbf{finished}(ms, [log'_S])$
$\mathbf{finished}(ms, [log_S])$
MitM knows k_c, k_s used for authenticated encryption

Figure 1. **The CurveSwap attack.** A man-in-the-middle can force TLS clients to use the weakest curve that both the client and server support. Then, by computing the discrete log on the weak curve, the attacker can learn the session key and arbitrarily read or modify message contents.

modification. The client then replies with its key exchange value on the weak curve. The attacker then computes the elliptic curve discrete log of either the client or server's key exchange message to compute the client or server's ephemeral private key. At this point, all parties, including the attacker, can then compute the master secret and the session keys. The attacker then intercepts the client and server finished messages and replaces them with finished messages corresponding to the other party's view of the handshake. After the compromised handshake, the client and server have a set of shared session keys that are known to the attacker, allowing the attacker to arbitrarily read and modify messages.

CurveSwap is a vulnerability in the TLS protocol itself, and affects TLS 1.0, 1.1 and 1.2. For these TLS versions, this vulnerability is mitigated somewhat by the TLS Session Hash and Extended Master Secret Extension, described in RFC 7627 [12]. RFC 7267 specifies that the premaster secret is computed from the entire transcript of the handshake, so in the case of an attempted parameter downgrade attack of this form, the attacker would be forced to man in the middle the entire connection instead of merely downgrading it. The CurveSwap attack is mitigated entirely in TLS 1.3, because the server sends a certificate verify message that includes a signature of the entire handshake transcript hash. In order to downgrade the connection, the attacker would need to forge this signature.

5.2. CurveSwap for SSH

In SSH, the server uses its long-term host key to sign the entire handshake, including both client and server lists of cipher suites and the negotiated Diffie-Hellman shared secret. Thus a CurveSwap-style attack would require the attacker to compromise the server's host key *and* learn the Diffie-Hellman shared secret. Such a powerful attack does not seem to have any advantage over a complete man-in-the-middle attack.

5.3. CurveSwap for IKE

In IKEv1 aggressive mode, it is not possible for the parties to negotiate the Diffie-Hellman group, so a group downgrade attack using aggressive mode is not possible. We note that for the pre-shared key and public-key encryption authentication methods, however, the AUTH messages in aggressive mode do not depend on the negotiated Diffie-Hellman shared secret.

In IKEv1 main mode, both the initiator and responder include the initiator's security association (but not the responder's security association) in their AUTH messages, which are encrypted using the negotiated Diffie-Hellman shared secret. An attacker would thus need to learn the Diffie-Hellman shared secret online in addition to compromising the authentication methods used by both parties. There are offline brute-force attacks against pre-shared keys in aggressive mode; documents leaked by Edward Snowden also reference attacks allowing the NSA to learn pre-shared keys in some situations [47], [48], [49].

In IKEv2, authentication is done by having each party sign or MAC their own security association and key exchange messages together with each party's nonces. The initiator and responder's authentication messages are both encrypted and authenticated using the Diffie-Hellman shared secret. Thus a CurveSwap-style downgrade attack would require the attacker to learn the initiator's authentication secret and to learn the Diffie-Hellman shared secret in order to forge the initiator's authentication message online.

6. Vulnerability Measurements

We performed a number of large-scale measurements of elliptic curve deployments with a focus on insecure implementation choices that might leave clients or servers vulnerable to CurveSwap.

6.1. Brute-forcing Small Curves

The Internet Assigned Numbers Authority (IANA) maintains a registry of valid curves for TLS, which includes several curves at the 80-bit security level [26].

6.1.1. CurveSwap via small curves. The CurveSwap attack allows a man in the middle to downgrade a TLS handshake to use the weakest curve that both the client and the server support. 2^{80} computational work is likely within range for advanced government-level adversaries. However, this amount of computation is quite significant, and is unlikely to be feasible within the timeout of a live TLS handshake in the near future.

However, the widespread use of static-ephemeral key exchange by servers means that a server might reuse its key exchange value for a long enough period to allow an attacker to pre-compute the server's secret exponent for a weak curve. The attacker could then use its knowledge of the server's secret exponent for this particular curve to downgrade any clients who support this curve, even if they would normally not prefer it, to this weak curve, and thus be able to decrypt or modify messages during the session.

6.1.2. Weak curve and ephemeral-static measurements. Table 5 shows support statistics for several weak curves, with the number of servers that repeat key exchange values when scanned twice in rapid succession.

We performed additional scans of hosts that initially repeated key exchange values to test the lifespan of ephemeral-static keys. Scanning with curve secp160k1, only 5 hosts responded with the same key exchange value as they did initially after five hours, and only 2 hosts returned the same key exchange value after 25 hours.

We also measure client implementations, and find that a significant number of clients offer weak curves in the supported curves extension. In Table 6, we show that in a sample of over 4 million client hellos collected from Cloudflare, over 16% indicate support for a curve with 80-bit security, opening up these clients to potential CurveSwap attacks. The user agents of these clients indicate that they are mostly API clients rather than browsers.

		Repeats...	
CurveID	**Support**	**Across Hosts**	**By Host**
ECDHE Hosts	41.0M	–	–
sect163k1	271.7K	2.1K (0.8%)	9.7K (3.6%)
sect163r1	267.8K	230 (0.1%)	7.1K (2.6%)
sect163r2	271.8K	2.1K (0.8%)	10.1K (3.7%)
secp160k1	274.9K	250 (0.1%)	7.7K (2.8%)
secp160r1	276.2K	290 (0.1%)	8.1K (2.9%)
secp160r2	266.9K	360 (0.1%)	7.2K (2.7%)

TABLE 5. **TLS Server Support for Weak Curves**—In August 2017, we scanned a randomly selected 10% of TLS hosts to measure support for weak curves. We scanned each host twice for each curve to detect servers using ephemeral-static keys. The baseline scan shows the number of hosts with which we were able to negotiate any curve. The repeat percentages are with respect to the support scans for each curve.

CurveID	**Support**
sect163k1	685.6K (16.4%)
sect163r1	682.1K (16.3%)
sect163r2	682.1K (16.3%)
secp160k1	682.6K (16.3%)
secp160r1	682.6K (16.3%)
secp160r2	682.6K (16.3%)

TABLE 6. **TLS Client Support for Weak Curves**—From a sample of 4,187,201 client hellos collected from Cloudflare in October 2016, over 16% offer weak curves in the client hello supported curves extension.

6.2. Invalid Curve Attacks

6.2.1. CurveSwap via an invalid curve attack. We now consider the scenario in which a man-in-the-middle attempts to learn the server secret through an invalid curve attack before initiating a CurveSwap attack. In this case, a CurveSwap attack would allow the attacker to force a connection to use a curve for which it already knows the server's ephemeral-static key. Servers are vulnerable to invalid curve attacks when they both fail to validate key exchange parameters and reuse the same ephemeral-static key for multiple connections. If a victim supports a variety of curves, some which are vulnerable to invalid curve attacks, and some which are not, this attack would allow the attacker to downgrade the victim to a vulnerable curve for which they can learn the server's secret.

6.2.2. Measuring invalid curve attacks. We performed extensive measurements to measure the prevalence of implementations vulnerable to invalid curve attacks, and present the results in Table 7. In the end, our scans found evidence of key exchange validation failure and of key reuse, but no hosts that both failed to validate and repeated keys either across hosts or across scans. Thus we do not find evidence of servers vulnerable to invalid curve key recovery attacks.

To test if servers properly validate received client key exchange values, we performed a key exchange using an element of order 5 on an invalid curve for secp256r1. We give the coordinates of this point and the equation of the generator in Appendix A. Table 7 shows the number of hosts that appeared to accept invalid curve points for the protocols that we scanned.

Since we send an invalid curve point of order 5, the shared secret for the session will be limited to one of five curve elements: (x_1, y_1), $(x_1, -y_1)$, (x_2, y_2), $(x_2, -y_2)$, and infinity. For TLS, SSH, and IKE, only the x-coordinate of the curve element is used as the shared secret for computing the session MAC, so a client sending an invalid point on this curve would have a $2/5$ chance of guessing the value correctly by choosing x_1 or x_2 as the shared secret.

In TLS, a client can reach the end of the handshake without authenticating, so in our scans we counted the number of hosts that accepted our client finished message and responded with a server finished message. Thus, we expect the number of hosts that are not properly validating to be $5/2$ times as large as the number of hosts that respond with a server finished message. Since Table 7 indicates that 0.31% of HTTPS hosts on port 443 accepted our guessed client finished with our invalid curve point, we estimate that 0.77% of HTTPS hosts fail to perform proper validation.

For SSH and IKE, our scanning methodology does not allow us to reach the end of the handshake without authenticating as a valid client, so we count the number of servers that fail to immediately indicate an error upon receipt of an invalid key exchange value. This does not require us to correctly guess the shared secret, so there is no need to scale the results as for TLS. This also does not account for hosts that perform validation checks later in the handshake, so the numbers presented are an overestimate. In the case of the SSH scans, we show the number of hosts that respond with an ssh_key_exchange_ecdh_reply message after receiving the invalid client public value. All of the SSH hosts that responded to these scans had a protocol banner indicating either "Cerberus", "VShell", or "SshServer". Manually installing CerberusFTPServer_8.0, we were able to replicate this behavior, and found that the server correctly logged an invalid key exchange value in its server logs. This appears to be in violation to RFC 5656, which specifies that the server should validate the client key exchange before sending its own key exchange value.

6.3. Twist Attacks

6.3.1. CurveSwap via twist attacks. We now investigate an attack vector that exploits the fact that there are several standardized curves with weak twist security. For example, an invalid curve attack using the twist for secp224r1 can be used to recover the secret key in only $2^{58.4}$ work, compared to its expected 112-bit security level [10].

Consider a server that uses a single-coordinate ladder for scalar-by-point multiplication, such as the Montgomery or Brier-Joye ladders. Single-coordinate ladders operate on only the x-coordinate of the key exchange value, making it impossible to specify a point on an invalid curve [15], [34]. However, an attacker can send an x-coordinate that does not correspond to a point on the negotiated curve, but does lie on the twist of the curve. If the server employs a single-coordinate ladder for scalar-by-point multiplication, then the server will compute the shared secret as a point on the twist of the curve. For curves with a weak twist, the attacker can send low-order points on the twist, and carry out a small subgroup attack to reconstruct the server's ephemeral-static key. To prevent this attack, an additional check is required to ensure that the specified x-coordinate lies on the curve, and not the twist of the curve.

Proto	Port	Twist	Invalid	InvalidRepeat
TLS	25	0 (0.0%)	40 (0.0%)	0 (0.0%)
	110	0 (0.0%)	20 (0.0%)	0 (0.0%)
	143	0 (0.0%)	0 (0.0%)	0 (0.0%)
	443	0 (0.0%)	75.5K (0.3%)	0 (0.0%)
	465	0 (0.0%)	260 (0.0%)	0 (0.0%)
	563	0 (0.0%)	10 (0.0%)	0 (0.0%)
	587	0 (0.0%)	0 (0.0%)	0 (0.0%)
	636	0 (0.0%)	150 (0.1%)	0 (0.0%)
	853	0 (0.0%)	20 (1.1%)	0 (0.0%)
	989	0 (0.0%)	0 (0.0%)	0 (0.0%)
	990	0 (0.0%)	230 (0.1%)	0 (0.0%)
	992	0 (0.0%)	10 (0.0%)	0 (0.0%)
	993	0 (0.0%)	8.1K (0.3%)	0 (0.0%)
	994	0 (0.0%)	10 (0.4%)	0 (0.0%)
	995	0 (0.0%)	6.7K (0.2%)	0 (0.0%)
	8443	0 (0.0%)	19.2K (1.5%)	0 (0.0%)
SSH	22	4.1K (0.1%)	3.3K (0.0%)	0 (0.0%)
IKEv1	500	530 (0.2%)	500 (0.2%)	0 (0.0%)
IKEv2	500	4.1K (4.0%)	4.1K (4.0%)	0 (0.0%)

TABLE 7. Invalid Key Exchanges—In November 2016, we scanned a randomly selected 10% of IPv4 addresses offering order 5 points on an invalid curve and on the twist of curve secp256r1. We show the number of hosts for which handshake negotiation is successful. As described in Section 6.2.2, we estimate that the number of vulnerable TLS hosts is $5/2$ times larger than the numbers reported in the table. For SSH and IKE, these numbers are an upper bound on the number of vulnerable hosts.

There are a number of curves with weak twists that bring twist attacks into feasible range [10], [21]. Notably, in addition to the NIST-standardized secp224r1, brainpoolp256t1 also has a weak twist, with an attack cost of 2^{44}. secp256r1 is secure against twist attacks with an attack cost of 2^{120}.

6.3.2. Measuring twist attacks. To test for this behavior, we perform scans sending a point in the subgroup of order 5 on the twist of secp256r1 as the client key exchange value. We chose secp256r1 because it has the highest support among the protocols we studied. We give the point coordinates and the twist equation in Appendix A. The scan results, presented in Table 7, indicate that no hosts accepted points on the twist of the curve. To test if point compression influenced server behavior, we performed an additional 10% scan of TLS on port 443 sending a compressed point of order 5 on the twist of secp256r1, and found that no hosts accepted this key exchange value.

We suspect that hosts accepting invalid curve points but not accepting points on the twist as the client key exchange value are using a mixed-Jacobian scalar-by-point multiplication algorithm, which would cause points on the twist to fail with an arithmetic error but would succeed for points on an invalid curve.

Library	Language	ECDH Support	Status
cjose	C/C++	No	–
jose-jwt	Haskell	No	–
jose4j	Java	Yes	fixed v0.5.5
Nimbus JOSE+JWT	Java	Yes	fixed v4.34.2
Apache CXF	Java	Yes	not vuln.
json-jwt	Ruby	No	–
phpOIDC	PHP	No	–
jose-php	PHP	No	–
js-jose	Javascript	No	–
go-jose	Go	Yes	fixed v1.0.4
jose2go	Go	Yes	fixed v1.3
node-jose	node.js	Yes	fixed v0.9.3

TABLE 8. **JWE Libraries**—We manually inspected the source code of several libraries implementing JSON Web Encryption, and found that many were vulnerable to a classic invalid curve attack.

7. Source Code Analysis

We examined a number of libraries to understand their elliptic curve implementations, and found multiple vulnerabilities. We also described our findings in a blog post [38].

7.1. Failure to Validate in JSON Web Encryption Standards and Implementation

We examined the source code of many libraries implementing RFC 7516, JSON Web Encryption (JWE), focusing on the Key Agreement with Elliptic Curve Diffie-Hellman Ephemeral Static (ECDH-ES) algorithms. The complete list of libraries that we examined is available in Table 8. We found that many of these libraries were vulnerable to a classic invalid curve attack as described in Section 2.3.2. This would allow an attacker in the role of a sender to completely recover the secret key of the receiver. Almost all the implementations we examined failed to validate that the received public key, contained in the `JWE Protected Header`, is on the curve. Although they did not validate the recieved public key before performing the scalar multiplication, some of the libraries that we examined (Nimbus JOSE+JWT, jose4j) were protected from the invalid curve attack by Java's BouncyCastle or up-to-date Java Sun JCA elliptic curve library, which includes a check that the result of the scalar multiplication is on the curve. However, libraries implemented in languages without this additional check, such as Cisco's node-jose and jose2go, were completely vulnerable. As shown in Table 8, we reported the vulnerabilities to library maintainers to ensure that implementations included the check that incoming public keys are on the agreed-upon curve. The go-jose vulnerability was found and reported by Nguyen [35].

7.2. Bug in NSS/Java in Elliptic Curve Addition

Both NSS and Java use the 5-bit window NAF method for scalar-by-point multiplication from [16]. Both implementations missed a critical `if/else` statement that lead the calculations to produce incorrect results on some inputs. In particular, there exist values of the scalar for which the algorithm would yield the point at infinity as a result while the actual correct result should be a finite value. We were unable to figure out a way to exploit this flaw.

We disclosed these flaws to Mozilla and Oracle in March 2017. The flaw was patched by including the missing `if/else` statement [2], [3].

8. Discussion

Although we found some vulnerable, buggy, and non-compliant elliptic curve behavior in most of the protocols we measured, the fact that these behaviors do not appear to lead to a full CurveSwap attack is good news. (The exception is JWE, where the invalid curve attacks are devastating and do not require a parameter downgrade.)

8.1. Complexity of Curve Support

We observe that there are a large number of curves that are supported in the protocols we studied, some of them dating from much earlier in the study of elliptic curves before different varieties of implementation attacks were as well understood. While having many curve sizes or parameter types would seem to give protocols and implementations room to adapt their speed and security needs, support for many of these curves risks becoming a liability if attacks on some classes improve enough to allow a feasible CurveSwap attack in TLS or other protocols. In addition, enumerating the current state of different attacks on each curve is quite complex. [10]

While recent curve constructions such as Curve25519 are designed to be as resistant to implementation mistakes as possible, the move to "new" algorithms such as single-coordinate ladders, which appear from our data not to be widely implemented for most curves, will likely result in the discovery of new bugs of the type we discovered in NSS/Java.

8.2. Protocol Security

The recent spate of cipher downgrade, transcript mismatch, and message forwarding attacks against TLS has highlighted the need for protocol-level protections against these types of man-in-the-middle attacks. Fortunately, TLS 1.3 includes multiple layers of handshake downgrade protection, including client and server authentication of the entire transcript hash using long-term secrets when possible, and computing session keys from the entire transcript. We note that the SSH protocol builds in such protection by having the server sign the entire transcript, as does IKE when using signature authentication. We hope that the community's improved understanding of protocol security means that downgrade attacks are a thing of the past.

Acknowledgments

We are grateful to Douglas Stebila for collaboration throughout this project, and to the University of Pennsylvania's information security and network admin staff including Chris Rogers, Kris Varhus, and Josh Beeman for their support in our scanning and network research. We also thank Juraj Somorovsky for valuable corrections. This material is based upon work supported by the National Science Foundation under Grants No. 1408734, 1505799, 1513671, and 1651344, and a gift from Cisco.

References

[1] "Browser capabilities project," 2017. [Online]. Available: https://browscap.org/ua-lookup

[2] "Mozilla foundation security advisory [8th Aug 2017]," Aug. 2017, https://www.mozilla.org/en-US/security/advisories/mfsa2017-18/#CVE-2017-7781.

[3] "Oracle critical patch update advisory [18th Jul 2017]," Jul. 2017, http://www.oracle.com/technetwork/security-advisory/cpujul2017-3236622.html.

[4] "MikroTik Manual:IP/IPsec," Feb. 2018, https://wiki.mikrotik.com/wiki/Manual:IP/IPsec.

[5] "OpenBSD iked.conf," Jan. 2018, https://man.openbsd.org/iked.conf.5.

[6] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[7] A. Antipa, D. Brown, A. Menezes, R. Struik, and S. Vanstone, "Validation of elliptic curve public keys," in *International Workshop on Public Key Cryptography*. Springer, 2003, pp. 211–223.

[8] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *PKC*, New-York, NY, US, Apr. 2006, pp. 207–228.

[9] D. J. Bernstein, C. Chuengsatiansup, and T. Lange, "Curve41417: Karatsuba revisited," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2014, pp. 316–334.

[10] D. J. Bernstein and T. Lange, "SafeCurves: choosing safe curves for elliptic-curve cryptography," Jan. 2014, https://safecurves.cr.yp.to/.

[11] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of tls," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 535–552.

[12] K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension," IETF RFC 7627, Sep. 2015.

[13] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)," IETF RFC 4492, May 2006.

[14] J. W. Bos, J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, and E. Wustrow, "Elliptic curve cryptography in practice," in *International Conference on Financial Cryptography and Data Security*. Springer, 2014, pp. 157–175.

[15] É. Brier and M. Joye, *Weierstraß Elliptic Curves and Side-Channel Attacks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 335–345. [Online]. Available: https://doi.org/10.1007/3-540-45664-3_24

[16] M. Brown, D. Hankerson, J. Lopez, and A. Menezes, "Software implementation of the nist elliptic curves over prime fields," in *TOPICS IN CRYPTOLOGY CT-RSA 2001, VOLUME 2020 OF LNCS*. Springer, 2001, pp. 250–265.

[17] C. Cremers, "Key exchange in ipsec revisited: Formal analysis of ikev1 and ikev2," *Computer Security–ESORICS 2011*, pp. 315–334, 2011.

[18] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman, "A search engine backed by Internet-wide scanning," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[19] Z. Durumeric, M. Bailey, and J. A. Halderman, "An internet-wide view of internet-wide scanning," in *Proceedings of the 23rd USENIX Security Symposium*, Aug. 2014.

[20] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast Internet-wide scanning and its security applications," in *Proceedings of the 22nd USENIX Security Symposium*, Aug. 2013.

[21] P. A. Fouque, R. Lercier, D. Ral, and F. Valette, "Fault attack on elliptic curve montgomery ladder implementation," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, Aug 2008, pp. 92–98.

[22] D. Fu and J. Solinas, "Elliptic curve groups modulo a prime (ecp groups) for ike and ikev2," 2010.

[23] S. D. Galbraith and P. Gaudry, "Recent progress on the elliptic curve discrete logarithm problem," *Des. Codes Cryptography*, vol. 78, no. 1, pp. 51–72, Jan. 2016. [Online]. Available: http://dx.doi.org/10.1007/s10623-015-0146-7

[24] M. Hamburg, "Ed448-Goldilocks, a new elliptic curve." *IACR Cryptology ePrint Archive*, vol. 2015, p. 625, 2015.

[25] D. Harkins, "Brainpool elliptic curves for the internet key exchange (ike) group description registry," 2013.

[26] IANA, "Transport Layer Security (TLS) Parameters," http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml, May 2017.

[27] International Computer Science Institute, "The ICSI certificate notary." [Online]. Available: https://notary.icsi.berkeley.edu/

[28] T. Jager, J. Schwenk, and J. Somorovsky, "Practical invalid curve attacks on TLS-ECDH," in *Proceedings of the 20th European Symposium on Research in Computer Security*, 2015.

[29] M. Joye and S. Yen, "The Montgomery powering ladder," in *Cryptographic Hardware and Embedded Systems (CHES) 2002*. Springer, 2002, pp. 291–302.

[30] C. F. Kerry and C. R. Director, "NIST SP 186-4: Digital signature standard (DSS)," 2013.

[31] M. Lepinski and S. Kent, "Additional Diffie-Hellman groups for use with ietf standards," IETF RFC 5114, 2008.

[32] C. H. Lim and P. J. Lee, "A key recovery attack on discrete log-based schemes using a prime order subgroup," in *Proceedings of the 17th International Cryptology Conference*, 1997.

[33] J. Merkle and M. Lochter, "Elliptic curve cryptography (ECC) Brainpool standard curves and curve generation," RFC 5639, Internet Engineering Task Force, Mar. 2010.

[34] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, Jan. 1987.

[35] Q. Nguyen, "Practical cryptanalysis of json web token and galois counter mode's implementations," in *Real World Crypto Conference 2017*, 2017, http://www.realworldcrypto.com/rwc2017. [Online]. Available: https://rwc.iacr.org/2017/Slides/nguyen.quan.pdf

[36] K. Okeya, H. Kurumatani, and K. Sakurai, "Elliptic curves with the Montgomery-form and their cryptographic applications," in *Public Key Cryptography (PKC) 2000*. Springer, 2000, pp. 238–257.

[37] H. Orman, "The Oakley key determination protocol," IETF RFC 2412, Nov. 1998.

[38] A. Sanso, "Critical vulnerability in JSON Web Encryption (JWE) - RFC 7516 ," Mar. 2017, http://blog.intothesymmetry.com/2017/03/critical-vulnerability-in-json-web.html.

[39] ——, "CVE-2017-7781/CVE-2017-10176: Issue with elliptic curve addition in mixed Jacobian-affine coordinates in Firefox/Java," Aug. 2017, http://blog.intothesymmetry.com/2017/08/cve-2017-7781cve-2017-10176-issue-with.html.

[40] S. SEC, "SEC 2: Recommended elliptic curve domain parameters," *Standards for Efficient Cryptography Group, Certicom Corp*, 2000.

[41] ——, "SEC 1: Elliptic curve cryptography," *Standards for Efficient Cryptography Group, Certicom Corp*, 2009.

[42] I. Semaev, "New algorithm for the discrete logarithm problem on elliptic curves," Cryptology ePrint Archive, Report 2015/310, 2015, http://eprint.iacr.org/2015/310.

[43] D. Springall, Z. Durumeric, and J. A. Halderman, "Measuring the security harm of tls crypto shortcuts," in *Proceedings of the 2016 Internet Measurement Conference*, ser. IMC '16. New York, NY, USA: ACM, 2016, pp. 33–47. [Online]. Available: http://doi.acm.org/10.1145/2987443.2987480

[44] D. Stebila and J. Green, "Elliptic-curve algorithm integration in the secure shell transport layer," 2009.

[45] N. Sullivan, "goto fail; exploring two decades of transport layer security," https://media.ccc.de/v/32c3-7438-goto_fail, Dec. 2015.

[46] L. Valenta, D. Adrian, A. Sanso, S. Cohney, J. Fried, M. Hastings, J. A. Halderman, and N. Heninger, "Measuring small subgroup attacks against diffie-hellman." in *Networks and Distributed Systems Security Symposium*. Internet Society, 2017.

[47] "Intro to the VPN exploitation process," Media leak, Sep. 2010, http://www.spiegel.de/media/media-35515.pdf.

[48] "VPN SigDev basics," Media leak, http://www.spiegel.de/media/media-35520.pdf.

[49] "What your mother never told you about SIGDEV analysis," Media leak, http://www.spiegel.de/media/media-35551.pdf.

Appendix A. Invalid Curve and Twist Points

We tested for curve validation in secp256r1 by using a generator of a subgroup of order 5 on the curve $y^2 = x^3 + ax + (b-1)$ with a and b as specified in [40] for secp256r1. The coordinates of our generator were

$$x = \texttt{BFD3 5739 ED4B 4D93 8C91 E835 7C7E C4C4 1DE9 FDFC} \\ \texttt{1669 88EB D1DF A09C 7959 6661}$$

$$y = \texttt{8949 2141 E9E8 1674 9798 62D9 FC62 21C4 A672 B890} \\ \texttt{33E0 7B86 DA40 D67D 5C0F 53E3}$$

We tested for twist validation in secp256r1 by sending a point of order 5 on the twist $y^2 = x^3 + a'x + b'$ with

$$a' = \texttt{8EB0 E29E C8A5 CCCB 65B9 936F B5B2 67E6 57D4 83DB} \\ \texttt{CDC0 2A88 8A7F 72E8 935B B316}$$

$$b' = \texttt{2F9B 5262 887E 1766 8BBA F58E 54B8 2E42 C72E D167} \\ \texttt{21BD 3325 DEB7 9B62 ADE7 4BD6}$$

The coordinates of our generator were

$$x = \texttt{8FB5 0654 3387 E96C D244 8468 9BF6 CC0C F383 4F33} \\ \texttt{D8CD 6442 4B11 7D3B ECA1 E0B5}$$

$$y = \texttt{E042 260E 3A00 30A5 5B46 8D2A DEBA D3D4 B613 373C} \\ \texttt{0C38 FCD8 5434 C2B8 B7F7 C1EA}$$

Appendix B. Extended Scans on Multiple Ports

We extended our scans to a variety of ports where TLS is used to secure services such as IMAP, POP3, SMTP, LDAP, and more.

			Repeats...	
Proto	**Port**	**secp256r1**	**Across Hosts**	**By Host**
TLS	25	375.8K	590 (0.2%)	21.9K (5.8%)
	110	126.3K	190 (0.2%)	290 (0.2%)
	143	120	0 (0.0%)	0 (0.0%)
	443	24.0M	638.7K (2.7%)	5.5M (22.9%)
	465	2.6M	156.1K (6.1%)	60.3K (2.3%)
	563	45.5K	36.4K (79.8%)	37.7K (82.7%)
	587	310.4K	160 (0.1%)	160 (0.1%)
	636	119.5K	39.1K (32.7%)	77.7K (65.0%)
	853	1.7K	40 (2.4%)	840 (50.6%)
	989	1.8K	80 (4.4%)	1.1K (61.3%)
	990	245.6K	24.1K (9.8%)	39.3K (16.0%)
	992	28.4K	40 (0.1%)	980 (3.5%)
	993	771.3K	55.0K (7.1%)	83.2K (10.8%)
	994	2.2K	100 (4.5%)	1.0K (45.9%)
	995	717.1K	57.4K (8.0%)	79.6K (11.1%)
	8443	1.3M	49.3K (3.9%)	274.6K (21.7%)
SSH	22	7.5M	0 (0.0%)	0 (0.0%)
IKEv1	500	168.5K	210 (0.1%)	540 (0.3%)
IKEv2	500	95.1K	800 (0.8%)	1.9K (1.9%)

TABLE 9. **Repeated Key Exchanges**—See Table 3.

Proto	**Port**	**Twist**	**Invalid**	**InvalidRepeat**
TLS	25	0 (0.0%)	40 (0.0%)	0 (0.0%)
	110	0 (0.0%)	20 (0.0%)	0 (0.0%)
	143	0 (0.0%)	0 (0.0%)	0 (0.0%)
	443	0 (0.0%)	75.5K (0.3%)	0 (0.0%)
	465	0 (0.0%)	260 (0.0%)	0 (0.0%)
	563	0 (0.0%)	10 (0.0%)	0 (0.0%)
	587	0 (0.0%)	0 (0.0%)	0 (0.0%)
	636	0 (0.0%)	150 (0.1%)	0 (0.0%)
	853	0 (0.0%)	20 (1.1%)	0 (0.0%)
	989	0 (0.0%)	0 (0.0%)	0 (0.0%)
	990	0 (0.0%)	230 (0.1%)	0 (0.0%)
	992	0 (0.0%)	10 (0.0%)	0 (0.0%)
	993	0 (0.0%)	8.1K (0.3%)	0 (0.0%)
	994	0 (0.0%)	10 (0.4%)	0 (0.0%)
	995	0 (0.0%)	6.7K (0.2%)	0 (0.0%)
	8443	0 (0.0%)	19.2K (1.5%)	0 (0.0%)
SSH	22	4.1K (0.1%)	3.3K (0.0%)	0 (0.0%)
IKEv1	500	530 (0.2%)	500 (0.2%)	0 (0.0%)
IKEv2	500	4.1K (4.0%)	4.1K (4.0%)	0 (0.0%)

TABLE 10. **Invalid Key Exchanges**—See Table 7.

				Number of hosts that support…						
Proto	**Port**	**Date**	**BASE**	**ECDHE**	**secp224r1**	**secp256r1**	**secp384r1**	**secp521r1**	**x25519**	**b-pool256r1**
TLS	25	11/2016	–	1.0M	420 (0.0%)	1.0M (99.7%)	3.1K (0.3%)	220 (0.0%)	0 (0.0%)	0 (0.0%)
	110	11/2016	–	182.7K	270 (0.1%)	176.7K (96.7%)	125.3K (68.6%)	113.6K (62.2%)	0 (0.0%)	580 (0.3%)
	143	11/2016	–	130	0 (0.0%)	130 (100.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	443	11/2016	38.6M	24.8M	643.4K (2.6%)	24.1M (97.0%)	5.7M (22.9%)	2.5M (10.2%)	0 (0.0%)	980.1K (3.9%)
	443	08/2017	41.0M	28.8M	811.6K (2.8%)	25.0M (86.9%)	9.1M (31.6%)	2.2M (7.7%)	740.7K (2.6%)	2.4M (8.4%)
	465	11/2016	–	2.7M	21.6K (0.8%)	2.7M (99.9%)	230.4K (8.4%)	213.2K (7.8%)	0 (0.0%)	2.0K (0.1%)
	563	11/2016	–	45.7K	60 (0.1%)	45.7K (99.9%)	2.9K (6.3%)	1.6K (3.6%)	0 (0.0%)	280 (0.6%)
	587	11/2016	–	836.9K	20 (0.0%)	836.6K (100.0%)	330 (0.0%)	40 (0.0%)	0 (0.0%)	0 (0.0%)
	636	11/2016	–	121.0K	2.8K (2.3%)	120.8K (99.8%)	43.5K (36.0%)	10.7K (8.8%)	0 (0.0%)	1.1K (0.9%)
	853	11/2016	–	1.8K	60 (3.4%)	1.7K (97.2%)	1.2K (66.5%)	400 (22.7%)	0 (0.0%)	240 (13.6%)
	989	11/2016	–	1.9K	30 (1.6%)	1.8K (98.9%)	1.3K (69.9%)	280 (15.1%)	0 (0.0%)	140 (7.5%)
	990	11/2016	–	246.4K	1.3K (0.5%)	243.7K (98.9%)	202.1K (82.0%)	184.1K (74.7%)	0 (0.0%)	690 (0.3%)
	992	11/2016	–	28.5K	300 (1.1%)	28.5K (99.8%)	27.7K (97.1%)	26.8K (93.9%)	0 (0.0%)	300 (1.1%)
	993	11/2016	–	2.9M	31.8K (1.1%)	772.8K (26.5%)	2.6M (89.0%)	380.2K (13.0%)	0 (0.0%)	97.9K (3.4%)
	994	11/2016	–	2.5K	100 (4.0%)	2.3K (94.3%)	1.6K (63.2%)	510 (20.6%)	0 (0.0%)	260 (10.5%)
	995	11/2016	–	2.8M	24.5K (0.9%)	717.9K (25.9%)	2.5M (89.0%)	359.5K (13.0%)	0 (0.0%)	88.6K (3.2%)
	8443	11/2016	–	1.3M	102.4K (7.9%)	1.3M (98.9%)	406.9K (31.5%)	159.5K (12.4%)	0 (0.0%)	22.1K (1.7%)
SSH	22	11/2016	14.5M	7.9M	0 (0.0%)	7.7M (97.8%)	7.5M (95.6%)	7.5M (95.4%)	6.1M (77.2%)	0 (0.0%)
IKEv1	500	11/2016	1.1M	215.4K	143.8K (66.8%)	211.8K (98.3%)	206.8K (96.0%)	152.8K (71.0%)	0 (0.0%)	0 (0.0%)
IKEv2	500	11/2016	1.2M	101.1K	4.1K (4.1%)	98.2K (97.1%)	98.0K (96.9%)	240 (0.2%)	0 (0.0%)	0 (0.0%)

TABLE 11. **SERVER SUPPORTED CURVES**—SEE TABLE 1.