Fooling primality tests on smartcards

Vladimir Sedlacek[1,2], Jan Jancar[3], and Petr Svenda[4]

[1] Masaryk University, vlada.sedlacek@mail.muni.cz
[2] Ca' Foscari University of Venice,
[3] Masaryk University, j08ny@mail.muni.cz
[4] Masaryk University, svenda@fi.muni.cz

Abstract. We analyse whether the smartcards of the JavaCard platform correctly validate primality of domain parameters. The work is inspired by Albrecht et al. [1], where the authors analysed many open-source libraries and constructed pseudoprimes fooling the primality testing functions. However, in the case of smartcards, often there is no way to invoke the primality test directly, so we trigger it by replacing (EC)DSA and (EC)DH prime domain parameters by adversarial composites. Such a replacement results in vulnerability to Pohlig-Hellman [30] style attacks, leading to private key recovery.
Out of nine smartcards (produced by five major manufacturers) we tested[5], all but one have no primality test in parameter validation. As the JavaCard platform provides no public primality testing API, the problem cannot be fixed by an extra parameter check, making it difficult to mitigate in already deployed smartcards.

Keywords: pseudoprimes · primality testing · JavaCard · (EC)DSA · (EC)DH

1 Introduction

Many public key cryptosystems crucially rely on prime numbers for their security. Yet for performance reasons (especially on constrained devices such as smartcards), most widely used primality tests, such as the Miller-Rabin (MR) test [23, 32], are only probabilistic [9, 1]. Thus there exist *pseudoprimes*, i.e., composite numbers passing these tests. When implemented correctly, probabilistic tests still provide a sufficient assurance of primality. However, carefully crafted pseudoprimes [4] can fool an implementation that is not utilizing enough randomness [1]. In (EC)DH and (EC)DSA, this can lead to private key recovery, using Pohlig-Hellman [30] style attacks.

JavaCard [29] is a popular platform for building systems based on programmable smart cards. It offers a Java-like environment on which multiple applications, *applets*, can be installed. Thanks to Javacard's rich cryptographic API (supporting (EC)DSA, (EC)DH and much more [33]), these applets include electronic passports and IDs, EMV applets for credit-cards, key managers, cryptocurrency wallets or applets for two-factor authentication. While the API is defined by an open standard, the implementation of the platform itself is almost always proprietary, with manufacturers releasing very little information about the code used in a particular family of cards. This black-box nature

[5] See https://crocs.fi.muni.cz/papers/primality_esorics20 for more information.

makes the public assessment of implementation security more difficult, but nevertheless, security problems have been discovered in the past [25].

In this paper, we test the robustness of present primality tests in JavaCards by replacing (EC)DSA and (EC)DH prime parameters with MR pseudoprimes and other composites. In contrast to [1], we do not have access to the code inside the smartcard and are not able to call a primality testing function on its own. Instead, we resort to performing standard operations (such as signature generation) using modified parameters (which still need to have specific properties), and observe any deviations from the expected behaviour. This is further complicated by the fact that the smartcards do not act deterministically, do not have debugging functionality, and are prone to many errors.

The main contributions of this paper are:

- We open the topic of fooling primality tests on black-box devices and propose a method for a systematic review of primality tests (and the relevant domain parameter validation) in black-box devices that use (EC)DSA/(EC)DH.
- We develop new ways in which parameters can be replaced with pseudoprimes in (EC)DH and (EC)DSA, along with practical attacks against these parameters. In particular, the attack against composite p in ECDSA is new to the best of our knowledge.
- We examine the implementation security of ECDH and ECDSA in nine smartcards from five major manufacturers, showing that all cards but one are vulnerable due to insufficient primality testing of domain parameters. Issues found were responsibly disclosed to affected vendors.
- We systematically survey the relevant attacker scenarios and types of attacks with possible real-world impact and propose defence mechanisms.

We review the previous work on attacking primality tests in Section 2. Section 3 analyses the attack scenarios and briefly presents possible attacks. The methodology for testing the cards is given in Section 4, along with a basic explanation of the used domain parameters. Readers interested only in practical security should feel free to skip this section, while still grasping most of the contents of Section 5 that analyses the testing results, and Section 7 that follows up with a discussion of proposed defences. Section 6 provides technical details about the full parameter generation and possible attacks and Section 8 concludes our paper. Finally, the appendices contain an overview of the MR test (Appendix 1), the pseudoprime construction (Appendix 2), datasets of generated domain parameters (Appendix 2.1) and example implementations of concrete attacks (Appendix 3).

2 Previous work

The idea of breaking a cryptographic protocol by fooling primality tests was first mentioned in [9]. In [1], the authors analysed primality tests in open-source libraries, and fooled many of them with carefully crafted pseudoprimes. Their construction (extending the one in [4] and briefly described in Appendix 2) relies on the assumption that the implementation of the MR test uses only a small number of bases that are either fixed or chosen from a relatively small set. This was indeed the case for many libraries.

Note that all the inspected libraries had a dedicated function for primality testing whose source code was accessible. In contrast, the situation for black-box devices where the code is not known, and the primality test (if present) cannot be separated from the rest of the program, has not been studied before to the best of our knowledge.

Furthermore, somewhat practical examples of attacks against various (EC)DH implementations with insufficient primality tests, including the case when pseudoprimes are included in elliptic curve domain parameters, were described in [16].

3 Attack scenarios

As in [1], we assume a setting where the attacker can control or affect the cryptosystem domain parameters used by the applet – so that primes can be replaced by composites – and wants to break the confidentiality of (EC)DH or unforgeability in (EC)DSA. We also assume that the attacker knows the factorization of the injected parameters, as he most likely crafted them himself.

However, with the exception of primality testing, we still expect that all of parameter validation is implemented properly (with the exception of a cofactor check of an elliptic curve, as the cards lack the performance to do it).

In our scenario an *applet* developed by an *applet developer* uses the functions of the JavaCard API on a card supplied by a *manufacturer* to perform some cryptographic operations while allowing untrusted parameters provided by the attacker to be used.

3.1 Rationale for the attack scenarios

To explain the rationale behind such a scenario, we consider the specifics of the JavaCard environment as well as existing cryptographic protocols and standards. Note that physical access (as is commonly relevant for the smartcard usage domain) is often not required.

A JavaCard applet developer might use untrusted domain parameters, because:

- The API functions that set parameter values, like `ECPrivateKey.setFieldFP()`, place no limitation (except bit-sizes) on the parameters, which are provided as sequences of bytes and are interpreted as unsigned integers.
- The API documentation contains no security notice that the set parameters should be trusted or a warning of what are the consequences of setting domain parameters that are untrusted or otherwise invalid [29].
- The API contains no functionality for direct primality testing or domain parameter validation for (EC)DSA or (EC)DH and no way to implement it efficiently. Thus the developer might (understandably) assume that the validation is performed implicitly.

Multiple protocols allow to transmit the domain parameters and thus force a party to either authenticate or validate them:

- TLS, up to version 1.2 [8] and prior to RFC8422 [26], allowed explicit (EC)DH parameters to be sent from the server to the client, although authenticated by the server public key.

- The certificate format specified in the X.509 standard allows public keys to hold full domain parameters for (EC)DH or (EC)DSA [31]. Using this format in a JavaCard applet (e.g., for interoperability reasons) might lead to untrusted parameters being used.
- The ICAO document 9303 [18] specifying the security requirements for machine-readable travel documents allows transmitting the (EC)DH domain parameters in the Chip Authentication and PACE protocols. The specification warns that insecure domain parameters will cause leaks of secret data and that parameters should not be used unless explicitly known to be secure (without further elaboration). As the card transmits the parameters to the reader, it is the one responsible for the validation.

All relevant (EC)DH and (EC)DSA standards specify procedures for validating the domain parameters and allow the use of untrusted domain parameters provided the validation succeeds. For (EC)DSA, two standards specify the validation requirements:

- FIPS 186-4 [14] refers to the NIST Special publication 800-89 [27] that in turn requires the primes used in the domain parameters to be accompanied by a seed and verifies they were generated using the specified verifiably random method.
- ANSI X9.62 [2] requires a primality test of the prime domain parameters, using the MR test with the number of rounds equal or larger than 50, using random bases. The IEEE P1363 [19] standard for (EC)DH has exactly the same requirement.

The strong requirements for primality testing and domain parameter validation in the above standards might lead the applet developer to believe that an appropriate validation is performed by the card and that the use of given parameters is secure. As the detailed implementation guidance is not provided by JavaCard specifications and recommendations from standards like IEEE P1363 and x9.62 are not explicitly mentioned, the platform vendor is left with decision what level of checks to implement.

We also consider another scenario where primality testing and domain parameter validation make a significant difference in security. TLS is an open system where communicating parties are likely to be realised by different software vendors. In the case of closed systems like dedicated network line encryption boxes, the same entity configures both communicating endpoints, which may be based on the commodity cards. A *platform integrator* (not the same as the card *manufacturer*) supplies the software responsible for setting the domain parameters on both ends. These two endpoints are designed to communicate with each other and to establish a secure channel using (EC)DH (and potentially (EC)DSA for authentication). Without robust primality testing and domain parameter validation on the card, the domain parameters supplied to cards at both ends can contain pseudoprimes or composites and be weak to a passive eavesdropping attacker. These parameters can even be authenticated by the *platform integrator*, yet without proper validation and primality testing, the card will accept them. The *platform integrator* could then also claim some plausible deniability, by blaming the weak parameters on a bug in the customised curve generation codebase or arguing the pseudoprime in the parameters passed their primality tests. A similar case happened in the Juniper Dual EC incident [11], where the exploitable weakness was a result of a series of small coding errors, seemingly unintentional.

One example of a vulnerability, where attacker-controlled domain parameters were used, was the Microsoft CryptoAPI ECDSA verification vulnerability (CVE-2020-0601) [28]. It was due to a faulty certificate verification mechanism, which matched certificates provided to the trusted ones by comparing the public key. This allowed an attacker to supply a certificate with modified domain parameters, which would be trusted.

Even when not directly using untrusted parameters, the adversarial setting makes sense when we account for the physical nature of cards and, thus, for fault injection attacks. These could be mounted to manipulate any trusted parameters [7, 34] that the applet will use (e.g., in (EC)DH).

3.2 Attacks overview

We focus on attacks theoretically applicable to all implementations accepting composite parameters, instead of those stemming from specific behaviour of any one implementation. We present four different attacks, based on the cryptosystem and the injected parameter. In all four cases, it is possible to efficiently recover the private key for suitable injected parameters. The details will be given in Section 6.

When the group order n is composite in ECDSA/ECDH or DSA/DH, it is well known that the discrete logarithm problem (DLP) in the group can be decomposed into DLPs in its quotient groups of prime-power order, which are much easier [30]. Thus for sufficiently smooth injected group orders, the discrete logarithm can be computed.

A similar decomposition and DLP difficulty reduction occurs when injecting a composite in place of the prime defining the full multiplicative group in DH/DSA [12].

We use yet another decomposition when injecting a composite in place of the prime defining the finite field for ECDSA/ECDH. As far as we know, this is a new result.

4 Methodology for assessing primality tests

In this section, we describe the method we used to analyse primality testing in cards of the JavaCard platform. Throughout the remaining text, the term *pseudoprime* will always mean a composite number that passes the MR test with respect to several small bases (the first t primes in our case).

In [1], the library functions for testing primality are ready to be called directly, and the source code can be analysed to see for what purpose and with what parameters they are invoked. In contrast, we cannot even be sure if such functions exist in the closed-source implementation of the JavaCard platform. Hence we need to guess where they could be likely present and invoked (e.g., during domain parameter validation or key generation) and what parts of the algorithm could behave problematically if a prime input was replaced with a composite one. Also, unlike in [1], we only have a very limited amount of pseudoprime bit lengths to choose from.

JavaCard specifies five main cryptographic algorithms involving prime numbers or domain parameters: RSA, DSA, ECDSA, DH and ECDH (though not all cards support all of them). We analysed all the relevant functions from the JavaCard specification and found no way to invoke primality testing in the RSA API with user-provided inputs. Also, the primes used there constitute the private key, and a scenario with them being replaced

with pseudoprimes does not trigger a primality test. As a result, only the methods of the (EC)DH and (EC)DSA algorithms are applicable. Additionally, we restricted the testing focus on the ECDSA and ECDH algorithms only, as none of the tested cards support DH and only one supports DSA. However, we still analyse the theoretical aspects of using DSA/DH parameters.

The practical analysis of primality testing consists of three steps:

1. Constructing pseudoprimes and other composites (Section 4.2 and Section 4.3).
2. Generating (EC)DSA and (EC)DH parameters with primes replaced with the numbers crafted in the previous step (Section 6).
3. Triggering the card's primality test with the modified parameters as input, e.g., key generation, signing, verification in case of (EC)DSA or key agreement in case of (EC)DH. (The rest of this section.)

In the last step, for any operation we perform on the card, the card only returns a response (output or error value) and the duration of the computation, which is often insufficient to understand exactly what happened due to implementations being closed-source. By the behaviour of the card under test, we mean such a response to our calls of API functions. To gain more information, we could also observe the card's power consumption or EM emissions during computation, but we do not consider these here. We use three types of basic operations in sequence to observe the behaviour:

3a) *Parameter setting.* Individual (EC)DSA or (EC)DH parameters are set on a `Key` object as byte arrays, interpreted as unsigned integers.
3b) *Key generation.* After setting all parameters, a `Keypair` can be generated. Note that the JavaCard does not differentiate between an ECDSA and ECDH keypair. In our tests, we skip this operation if it fails and continue with a manually generated private key, to also test the scenario where a keypair to be used is imported to the card.
3c) *Signing and verification* or *Key agreement.* After a `Keypair` object is successfully generated, it can be used to initialise a `Signature` or a `KeyAgreement` object and perform the operation. We supplied random data for signing and performed the key agreement between two keypairs generated on the card if possible. If the key generation failed, we instead substituted the private key and performed key agreement between it and the generator point on the curve.

To perform these operations, we developed and released our `ECTester` tool [21], which accesses the public JavaCard API and is generic to all cards.

4.1 Domain parameters

In this section, we examine the requirements on domain parameters used in (EC)DSA and (EC)DH, specifically primality requirements and show what requirements need to be fulfilled while replacing a prime with a composite. Since the parameters and the corresponding implementation checks for the finite field case and for the elliptic curve case differ significantly, we study them separately.

The DSA/DH case In DSA/DH, there are three domain parameters [14]:

- p is the prime defining the multiplicative group $\mathbb{Z}_p^*$ in which we compute,
- g is an element of $\mathbb{Z}_p^*$,
- q is the order of g in $\mathbb{Z}_p^*$.

Note that the above already implies $g^q \equiv 1 \pmod{p}$, $q \mid p-1$ and $g \neq 1$ (unless $q = 1$) and we can expect that these conditions could be checked by the implementation.

The supported sizes include $\{(1024, 160), (2048, 224), (2048, 256)\}$ bits for p, q respectively. Classically, q is required to be prime, as the running time of the Pohlig-Hellman algorithm [30] depends on the size of the largest factor of q. Also, the random nonce k, which is generated during signing, needs to be invertible mod q. Thus for testing, we could replace either p or q with a pseudoprime. However, this replacement is non-trivial, as the conditions above are quite easy to satisfy when computing p and g from q, but somewhat hard if given p, as one needs to factor $p-1$ and hope it has a prime factor q of the correct size. We discuss this in Section 4.7.

In DH on the JavaCard platform, the domain parameters are the same as in DSA, but the q parameter is optional [29]. This means that either no checks related to q are performed, or that p is assumed to be a safe-prime, i.e. $p = 2q + 1$. We do not consider the case when the safe prime condition is assumed in the remainder of this paper and instead refer the reader to [16]. Similarly, we do not consider the case where there are no checks related to q present, as it is straightforward to subvert the parameters in such a system (for example, q can be very small).

Note that we did not test actual DSA/DH parameter sets, as mentioned earlier in Section 4, due to lack of support in the tested cards.

The ECDSA/ECDH case This case is a little more complicated. The JavaCard API supports curves in the short Weierstrass form either over prime fields $\mathbb{F}_p$ or binary fields $\mathbb{F}_{2^m}$. We do not work with the binary field case, as most cards at our disposal do not support it. The prime field case then requires the inputs p, a, b, G_x, G_y, n, h, where:

- p is the prime defining the field $\mathbb{F}_p$ over which we will work,
- a, b are the coefficients of the elliptic curve E in short Weierstrass form over $\mathbb{F}_p$,
- G_x, G_y are the affine coordinates of the generator point $G \in E(\mathbb{F}_p)$,
- n is the order of G,
- h is the cofactor, equal to the order of $E(\mathbb{F}_p)$ divided by n.

As for supported sizes, p should have either $160, 192, 224, 256, 384, 512$ or 521 bits. Computing the group order or n is prohibitively expensive for the card, so it is reasonable to assume that only the condition $[n]G = \infty$ will be checked, possibly together with the size of n (by Hasse's theorem, $n \cdot h$ should be roughly the same size as p). Again, for ECDSA/ECDH, n should be prime for the same reasons as q in DSA/DH. Thus for testing, we could replace either p or n with a pseudoprime and tested (the case of pseudoprime n was discussed in [16]). For the replacement, we need to either construct an elliptic curve with a prescribed number of points (we used our tool ecgen [20] that supports the complex multiplication method [10]) when n is replaced, or to construct an "elliptic curve" over $\mathbb{Z}_p$ (with composite p) and correctly compute its order.

For each card and each bit-size in $\{160, 192, 224, 256, 384, 512, 521\}$, we test the card's behaviour for ECDSA and ECDH with parameter sets described in Table 2. The rest of this section shows how we generated p and n, while Section 6 explains how we constructed the malicious parameters from them. The full parameters used for testing in this paper are included in Appendix 2.1.

4.2 Generating pseudoprimes

As we are considering only the MR primality test, we use a slightly tweaked version of Arnault's method with three pseudoprime factors, described in Appendix 2. We construct numbers that are pseudoprime to t smallest primes taken as bases, assuming the resource constrained smartcard will choose its bases from a set of small primes. The only limitation is that the bit-size of the pseudoprime must be one of the supported ones, as discussed in Section 4. To achieve this, we must try many combinations of t, k_2, k_3 to arrive precisely at the supported bit-sizes, while also trying to maximise t (Table 1). For each bit-size, the pseudoprime generation process took at most a few minutes on an ordinary laptop (using the precomputed values of t, k_2 and k_3).

bit-size	t	k_2	k_3
160	11	73	101
192	13	61	101
224	14	197	257
256	16	233	101
384	23	137	157
512	30	137	157
521	30	137	157
1024	52	241	281

Table 1: Parameters for constructing pseudoprimes by tweaked Arnault's method [3, 1].

4.3 Generating special composites

To systematically compare the card behaviour, we also used random composites with controlled numbers of factors or varying levels of smoothness, to get finer granularity. In this way, we can detect if the primality test is present at all (though possibly faulty).

Composites with a given number of factors. To generate a composite number of a given bit-size with a given number of factors, we use a greedy approach. In each step, we generate a random prime number of size b/r, where b is the number of remaining bits, and r is the number of remaining factors to be generated.

Composites with a given smoothness level. For the smooth case, we employ a similar greedy algorithm that randomly chooses prime factors up to the smoothness bound and retries until a number with the right bit-size is constructed.

4.4 Generating complete domain parameters

In this section, we explain how to generate complete parameters for ECDSA/ECDH and DSA/DH, based on the pseudoprime and other composite inputs generated in Section 4.2

and Section 4.3. In the ECDSA/ECDH case, these are exactly the parameters we used for testing the cards.

The challenge in embedding composites into the domain parameters lies in the fact that the card might check many properties of the parameters, while the only thing we are currently interested in is the compositeness of some of them. Thus the parameters should be as close to correct parameters as possible. The properties of the parameters that the card might verify are listed in the standards specifying domain parameter validation algorithms [2, 19] and we listed them in Section 4.1. For each scenario, we also list the corresponding attack.

4.5 ECDSA/ECDH: prime p, composite n.

The approach, in this case, is almost the same as the one described in [16]. We use the complex multiplication method (described in [10], realised by our tool `ecgen` [20]), which is able to construct a curve over a prime field in short Weierstrass form with a given number of points. We need to take into account that the structure of $E(\mathbb{F}_p)$ is either cyclic or a product of two cyclic groups. This poses an issue because the JavaCard platform limits the size of the cofactor to an unsigned short integer, so just 16 bits. In the curves generated by two points, often the cofactor does not fit into 16 bits, even if we pick a large subgroup. Thankfully, the cards do not perform validation of the cofactor, as it is an optional input, so we just pick the generator with the largest order and set the cofactor to 1. Given the composite n, generating a suitable 256-bit curve took just a few minutes on an ordinary laptop.

One of the forms of composite n we tried to generate was that of an appropriately sized primorial (i.e., the product of all the primes up to some bound). However, the complex multiplication method, as implemented in the ecgen tool, was unable to generate them, even after a significant time spent on the task (e.g., a week on a single curve). The method searches for the curves by enumerating values of their complex multiplication discriminant, starting from 1, until a suitable curve and prime field is found. This points to an absence of prime field curves with primorial order and a small complex multiplication discriminant, which is an interesting observation.

4.6 ECDSA/ECDH: composite p, arbitrary n.

Here we assume for simplicity that p is square-free and has no small factors (up to some bound, we chose 50). We want to find a curve whose order has no small divisors; otherwise, the card might reject the curve for a wrong reason, as we have observed before.

For each prime factor p_i of p, we iterate over all possible curves over $\mathbb{Z}_{p_i}$ until we find one whose order is prime (this will minimise the number of prime factors of the resulting curve over $\mathbb{Z}_p$). We also prefer if the order of the curve is never repeated for different p_i's, but this is easily satisfied in practice. When such a curve is found for each p_i, we create the desired curve modulo p just by using the CRT on the Weierstrass coefficients a, b of the individual curves. Since p has no small prime divisors, we can expect the same to be true for the order of the final curve as well, thanks to the construction, as the resulting order is the product of the individual orders.

To obtain a generator point of the resulting curve, we simply pick a generator point of each curve, and we use the CRT again on their coordinates. Since each curve over $\mathbb{Z}_{p_i}$ was cyclic and their orders were distinct, the final curve is cyclic as well, so we can set the cofactor to be 1. This whole process takes just seconds for the 3- and 10-factor 256-bit composites used in this paper.

4.7 DSA/DH: prime p, composite q.

This is the easiest scenario, as it almost completely follows the way ordinary DSA parameters are generated. We first pick a composite or pseudoprime q, then choose random properly sized integers k until $p = kq + 1$ becomes a prime. Then we repeatedly pick a random $r \in \mathbb{Z}_p^*$ until we get a generator of $\mathbb{Z}_p^*$ and compute $g = r^{(p-1)/q}$. In this way, we ensure that g has order q modulo p. This generation process is very fast, and takes just seconds to generate 1024-bit parameters.

4.8 DSA/DH: composite p, prime q.

This case is more problematic to construct than the above one. First, let us assume that p is a Carmichael number (as is the case for the pseudoprimes we are constructing (Appendix 2). We assume that either of the conditions

$$q \mid p-1, \quad g^q \equiv 1 \pmod{p} \quad \text{and } g \neq 1$$

could be checked, so we will want to satisfy all of them.

These conditions imply that $g^q \equiv 1 \pmod{p_i}$ for all prime factors p_i of p, hence $g^{\gcd(q,p_i-1)} \equiv 1 \pmod{p_i}$. Since q is a prime and $g \neq 1 \pmod{p_i}$ for some i (otherwise $g = 1$), this implies $q \mid p_i - 1$ for some i.

Thus we need $p - 1$ to have a prime factor q of a size corresponding to the size of p (e.g., if p has 1024 bits, then we need q to have 160 bits). Given a specially constructed p, this means factoring $p - 1$ and hoping for a factor of the correct size. This is exactly what we did for generating the DSA parameters, even though it was only practical for the 1024-bit parameters, given that factoring larger than 1024-bit random integers and hoping for a factor of a correct bit-size is computationally hard for our computation cluster. Finding an appropriate 1024-bit pseudoprime p such that $p - 1$ has a 160-bit factor took a few days on an equivalent of an ordinary laptop.

Once we have p and q, we can again loop through random r from $\mathbb{Z}_p^*$ and compute g as $g = r^{(p-1)/q}$ until $g \neq 1$. This will imply that $g \not\equiv 1 \pmod{p_i}$ for at least one i, so that the primality of q together with the congruence $g^q \equiv r^{p-1} \equiv 1 \pmod{p}$ (as p is a Carmichael number) will imply that the order of g modulo p_i is q, hence $q \mid p_i - 1$.

Note that it is possible that no such g exists, even if p is a pseudoprime - for example for the Carmichael number $p = 7 \cdot 19 \cdot 67$ and $q = 5$, we have that $q \mid p-1$, but $q \nmid p_i - 1$ for any i, so there is no element of order q modulo p. However, it can be empirically seen that is unlikely to happen when p and q are large enough.

It seems hard to adapt this strategy of generating parameters for a fixed composite non-Carmichael p (which instead has a given number of factors or is smooth). One would have to simultaneously force $q \mid p - 1$ and $q \mid p_i - 1$ for some prime factor p_i of

p, which is equivalent to $q \mid \gcd(p-1, p_i - 1)$. But unlike in the Carmichael case (where $\gcd(p-1, p_i - 1) = p_i - 1$), heuristics show that we cannot expect $\gcd(p-1, p_i - 1)$ to have a large prime factor for most composite p, let alone a factor of an exactly given size. Thus we do not consider this case further, but we stress that its significance is mostly limited to testing of black-box devices. A motivated attacker would use pseudoprime (or just Carmichael) p, as it has a much better chance to bypass potential primality tests, while making the generation of the other parameters easier.

5 Practical results

The analysis was performed on cards with ECC support that we were able to obtain in small quantities and covers most major vendors (except for Gemalto and Idemia). The cards were fabricated in the period between 2012 and 2018. Note that due to lengthily and costly certification processes, the pace of software changes in the smartcard environment is significantly slower than for standard software development. As a result, the products by the same vendor tend to reuse the same existing codebase (as visible from results for the NXP cards), and our findings are likely valid for the newer product versions as well. The results are summarized in Tables 2a and 2b.

The main result of our testing is that most manufacturers, apart from *Athena* and *Infineon*, seem to lack primality tests of the p and n parameters for ECDSA and ECDH. This follows from the same observed card behaviour for the tests with pseudoprime parameters (Section 4.2) as for the tests with general composite parameters (Section 4.3). Missing primality testing invites Pohlig-Hellman style attacks mentioned in Section 3. Due to the non-deterministic nature of ECDSA key and nonce generation, we had to run the tests many times to get representative results. The different bit-sizes of the curves used, ranging from 160 bits to 521, do not impact the results in an unexpected way.

We may have passed the primality test using a pseudoprime curve order in the case of the *Infineon CJTOP 80k* card, as the key generation and ECDSA signing and verification worked in a few rare cases, even though the card rejected the parameters most of the time. We observed this in roughly 3 out of 1000 tries on a 192-bit pseudoprime order curve. Our hypothesis is that the implementation is choosing small MR bases, which occasionally lie in the set of liars for our provided pseudoprime.

We were not able to pass the primality test present on the *Athena IDProtect* card, perhaps because it uses random MR bases or some other primality test.

We also observed that cards occasionally went mute and did not respond to the command, often upon invoking key generation. This behaviour is outside of the PCSC specification[6] and results in a PCSC error being raised by the reader's driver. It could also mean that the cards perform some kind of a self-test during the operation and stop responding as a security measure if the test fails. The presence of such self-tests is well documented in cards. In ECDSA, this error might stem from the card generating a nonce k that is non-invertible modulo n, which the system might not expect.

In the ECDSA case, several cards occasionally produced invalid signatures. This is possibly due to the modular inversion algorithm assuming a prime modulus. We did not

[6] The PCSC specification specifies the general communication protocol between the card and the reader device.

Card	*p*			*n*				
	prime	pseudo	3f	pseudo	3f	10f	11s odd	11s even
Athena IDProtect	OK	IL	IL	IL	IL	IL	CYC	EXC
G&D SmartCafe 6.0	OK	OK	OK	OK	OK	OK	CYC	EXC
G&D SmartCafe 7.0	OK	OK/MUT	OK/MUT	OK	OK	OK	MUT	EXC
Infineon CJTOP 80k	OK	IL	IL	IL/OK	IL	IL	EXC	EXC
NXP JCOP v2.4.1	OK	OK/VRF	OK/VRF	OK	OK	OK	IL	IL
NXP JCOP CJ2A081	OK	OK	OK	OK	OK	OK	IL	IL
NXP JCOP v2.4.2 J2E145G	OK	OK/VRF	OK/VRF	OK	OK	OK	IL	IL
NXP JCOP J3H145	OK	OK/MUT	OK/VRF/MUT	OK	OK	OK	EXC	EXC
TaiSYS SIMoME VAULT	OK	OK/MUT	IL/MUT*	OK	OK	OK	EXC	EXC

(a) ECDSA results.

Card	*p*			*n*				
	prime	pseudo	3f	pseudo	3f	10f	11s odd	11s even
Athena IDProtect	OK	IL	IL	IL	IL	IL	CYC	EXC
G&D SmartCafe 6.0	OK	MUT	MUT	MUT	MUT	MUT	CYC	EXC
G&D SmartCafe 7.0	OK	OK	OK	OK	OK	OK	MUT	EXC
Infineon CJTOP 80k	OK	IL	IL	IL	IL	IL	EXC	EXC
NXP JCOP v2.4.1	OK	OK	OK	OK	OK	OK	IL	IL
NXP JCOP CJ2A081	OK	OK	OK	OK	OK	OK	IL	IL
NXP JCOP v2.4.2 J2E145G	OK	OK	OK	OK	OK	OK	IL	IL
NXP JCOP J3H145	OK	OK/MUT	OK/MUT	OK	OK	OK	EXC	EXC
TaiSYS SIMoME VAULT	OK	OK	OK	OK	OK	OK	EXC	EXC

(b) ECDH results.

Table 2: Results of domain parameters validation using on-card primality testing by nine different cards from five major manufacturers. Multiple values separated with a slash indicate that multiple results are present with decreasing occurrence from left to right. *IL (see below) happens on verification, key generation and signing works.

Result types

OK	Operation without error
IL	`ILLEGAL_VALUE` exception
VRF	Failed to verify signature
EXC	Unexpected exception
CYC	Card cycles indefinitely
MUT	Card does not respond

Parameter names

prime	standard parameters
pseudo *p*	pseudoprime *p*
3f *p*	3-factor composite *p*
pseudo *n*	pseudoprime *n*
3f *n*	3-factor composite *n*
10f *n*	10-factor composite *n*
11s odd *n*	11-smooth odd *n*
11s even *n*	11-smooth even *n*

- Green background signifies tests with the expected result, i.e. the card correctly computed with the parameters or the card correctly rejected them.
- Yellow background marks tests where the card exhibits unexpected behaviour, but are not vulnerabilities, and are not exploitable by the attacks from Section 3.
- Red background marks tests where the card accepted parameters it should have rejected, and is thus vulnerable to attacks from Section 3.

investigate this matter further, but these invalid signatures might leak information about the private key or the used nonce, which might be abused by a lattice attack.

The behaviour of the cards also differs for smooth n and for 10-factor n. We think this is due to some unknown checks failing when such a smooth order is given, not due to a primality test. Furthermore, two cards (*Athena IDProtect*, *G&D SmartCafe 6.0*) cycle indefinitely on key generation on a curve with smooth odd order, we do not have any explanation for this behaviour.

Algorithms used during the operations, such as the modular multiplicative inverse or the modular square root, may be implemented to rely on the modulus being prime. Thus we were surprised to see the cards mostly working for composite p.

6 The attacks in detail

In this section, we discuss the attack details in each of the four scenarios we consider.

6.1 Attack on ECDSA/ECDH with prime p and composite n

Using the classical Pohlig-Hellman algorithm [30], the DLP asymptotically becomes only as hard as the DLP in a subgroup of order l, where l is the largest prime factor of the group order n. There it can be solved by the Pollard ρ algorithm, which costs roughly $\sqrt{\frac{\pi}{4}l} \approx 0.886\sqrt{l}$ point additions [5]. Thus for example, when using a 256-bit curve and n has three factors of roughly the same size, the total computation cost of the DLP is approximately $3 \times 0.886 \times \sqrt{2^{86}} \approx 2^{44}$, which is already practical (and can be much cheaper for a larger number of factors). Compare this with a case of using the Pollard ρ algorithm to solve DLP on a standard 256-bit curve, where one gets the cost of $0.886 \times \sqrt{2^{256}} \approx 2^{128}$. An example of this attack is given in Appendix 3.

6.2 Attack on ECDSA/ECDH with composite p, and arbitrary n

When a composite p is a product of distinct primes $p_1, \ldots, p_e$ in ECDSA or ECDH, we are working with an "elliptic curve" over $\mathbb{Z}_p$ (see [37] for a proper definition and basic properties), whose group can be thought of as a direct sum of groups of the same elliptic curve regarded over $\mathbb{Z}_{p_i}$, i.e., $E(\mathbb{Z}_p) \cong \bigoplus_{i=1}^{e} E(\mathbb{Z}_{p_i})$. The isomorphism is essentially realised by the CRT applied to point coordinates. Thus the DLP on $E(\mathbb{Z}_p)$ again asymptotically becomes only as hard as the hardest DLP on some $E(\mathbb{Z}_{p_i})$ (since after solving the DLP in all individual groups, we can use the CRT to obtain the desired discrete logarithm). Since the order of $E(\mathbb{Z}_{p_i})$ is roughly p_i, the situation is very similar to the one for composite n in ECDSA/ECDH. An example of this attack is given in Appendix 3.

6.3 The attack on DSA/DH with prime p and composite q

The Pohlig-Hellman algorithm is applicable in an exact analogy to the composite n case in ECDSA/ECDH. Note that the sub-exponential index calculus algorithm could also be used to solve the individual DLPs, but we expect it to perform worse than Pollard ρ (whose cost is asymptotically the same as for ECDSA/ECDH), as it cannot efficiently use the extra information about the factorisation of q.

6.4 The attack on DSA/DH with composite p and prime q

In this case, we know the value g^x modulo p, where $0 < x < q$ and $q \mid p_i - 1$ for some prime factor p_i of p (this follows from the construction described in Section 4.8). Thus we also know the value g^x modulo p_i and finding x modulo p_i gives us x directly, since $x < q \leq p_i - 1$. Therefore it is sufficient to solve the DLP modulo p_i. Note that Pollard ρ does not have an advantage compared with the case with a real prime p, as the group order is still q. On the other hand, the complexity of an index calculus algorithm only depends on p_i, which can be much lower than p. Hence the security level will be lower than it should be and might lead to a private key recovery for small enough p_i. The practicality of this approach is demonstrated in Appendix 3.

7 Proposed defences

Without a robust primality test, a card cannot properly validate domain parameters. As the public JavaCard API lacks primality testing functionality, we cannot expect the developers to perform the validation either. Thus applications that allow the setting of custom domain parameters may result in a vulnerable applet.

Furthermore, the absence of primality testing functionality hinders the development of more complex cryptographic applications. For example, the vulnerability in the RSA key generation presented in the ROCA attack [25] could have been mitigated by applets generating the primes for their RSA keypairs themselves, thus avoiding full firmware fixes of the affected devices (which are often impossible in the case of cards). The lack of solid number-theoretic functionality in the JavaCard API prevented this though.

Fortunately, most of the protocols and implementations use standard named curves such as NIST P-256 or Curve25519. This seems to limit the current real-world impact of the aforementioned absence of primality testing in domain parameter validation.

We analysed an extensive list of open-source implementations of JavaCard applets [13] and found none that would use unauthenticated domain parameters in (EC)DSA or (EC)DH. Most used a fixed standard curve, with a few using domain parameters supplied in a command, but those were either authenticated or it was apparent from the context that they were provided by a trusted party, for example during the setup of the applet. However, one should keep in mind the possibilities of an untrusted setup described in Section 3, as well as the possibility of fault injection attacks. We also note that open-source JavaCard development comprises only a very small part of deployed JavaCards and that most applets are closed-source.

The recent trends in cryptography head towards misuse-resistance, the property of protocols and APIs that makes it hard for the developers to use and implement them incorrectly. Protocols and cryptosystems should allow simple implementations, as those are more likely to be correct and secure. Furthermore, the simple and fast implementation should always be a secure one. Examples of this include the nonce-misuse resistant authenticated encryption modes such as the SIV [17] or libraries with a very simple API such as libsodium or NaCl [6]. With this direction in mind, the missing domain parameter validation steers the developers to misuse the API and undermine the security of their applets.

We thus propose several changes to the JavaCard specification:

- Require full domain parameter validation, for example as specified in ANSI X9.62 [2] and IEEE P1363 [19], which includes primality tests of prime parameters.
- Add API that supports using a set of named curves and allow manufacturers to only support this API. Consider perhaps deprecating or discouraging explicit domain parameter setting.
- Add a primality test to the public API.

Validating elliptic curve domain parameters consists of more than primality testing and general sanity checks on the parameters. It contains tests on certain algebraic properties of the curves that might make the DLP easier (e.g., by allowing transfers into weaker groups). Luckily, these are all specified in the aforementioned standards.

The modification of JavaCard API to accept only named curves instead of the full specification of curve parameters limits flexibility for the future inclusion of new curves as it might not be possible to update the list after card deployment. On the other hand, strict usage of only named curves prevents attacks similar to the recent attack on the Microsoft CryptoAPI library (CVE-2020-0601) [28], which cannot be prevented only by domain parameter validation.

The Miller-Rabin with random bases or Baillie-PSW primality tests should allow a robust and reasonably efficient (even on limited smartcard chips) implementation of primality testing. For an example of a performant and misuse-resistant primality test, see Massimo and Paterson [22].

8 Summary

We have explored the robustness of primality testing in domain parameter validation by smartcards of the JavaCard platform. Due to unavailability of primality testing functionality in the public JavaCard API, we tried to trigger the tests indirectly by using specially crafted composite domain parameters for ECDSA and ECDH operations.

We analysed nine different smartcards from five major manufacturers and found that all but one failed to properly verify the primality of the provided ECDSA and ECDH domain parameters, not even requiring pseudoprimes to fool them, just composites. This results in a vulnerability to Pohlig-Hellman [30] style attacks, allowing the extraction of the private key. Our approach is generic to all black-box devices performing ECDSA and ECDH and the tooling can be reused.

Furthermore, the vulnerability is not easily mitigated for the already deployed smartcards. The code responsible for the domain parameter validation is often stored in a read-only memory without the possibility for an update. In addition, the on-card verification of the provided domain parameters by the developer cannot be efficiently performed due to a lack of a primality testing functionality in the public JavaCard API.

Acknowledgements. The authors would like to thank K.G. Paterson, M. Sys, V. Matyas and anonymous reviewers for their helpful comments. J. Jancar was supported by the grant MUNI/C/1701/2018, V. Sedlacek by the Czech Science Foundation project GA20-03426S. Some of the tools used and P. Svenda were supported by the CyberSec4Europe Competence Network. Computational resources were supplied by the project e-INFRA LM2018140.

Appendix

1 The Miller-Rabin primality test

The MR test [23, 32] was one of the first practical primality tests and to this day remains very popular because of its simplicity and efficiency. In particular, we believe that if a low-resource device such as a smartcard (shortened as *card* for the rest of text) uses a primality test, MR is the most probable choice (perhaps followed by the Lucas test, which does not seem to be that widespread, and a Ballie-PSW test, which is a combination of these two), as most other tests are too resource-heavy.

However, the MR test cannot be used to prove that a number is prime; only compositeness can be proven. It relies on the fact that there exist no nontrivial roots of unity modulo a prime. More precisely, let n be the number we want to test for primality and let $n - 1 = 2^s d$, where d is odd. If n is prime, Fermat's Little Theorem implies that for any $1 \leq a < n$, we have either $a^d \equiv 1 \pmod{n}$ or $a^{2^i d} \equiv -1 \pmod{n}$ for some $0 \leq i < s$. By taking the contrapositive, if there is some $1 \leq a < n$ such that none of these congruences hold, then n is composite (and a is called a *witness of compositeness* for n). However, if at least one of the congruences holds, then we say that n is *pseudoprime with respect to base a* (or that a is a *non-witness of compositeness* for n, or also a *liar* for n). There is the Monier-Rabin bound [24] for the number $S(n)$ of such bases (that are less than n): $S(n) \leq \frac{\varphi(n)}{4}$, where φ is the Euler totient function.

Since $\varphi(n) \approx n$ for large n, we get a practical upper bound for the number of inputs that pass the test for a given a. Thus if we repeat the test t times for random a's, the probability of fooling the MR test will be at most $(\frac{1}{4})^t$.

The fact that the a's were picked randomly is crucial for the guarantees above. If the bases are fixed and known in advance (as in [1]), it is possible to construct a pseudoprime (see Appendix 2), i.e., a number that passes the test with respect to these bases.

2 Constructing pseudoprimes

We will briefly describe how to generate pseudoprimes having 3 prime factors with respect to given distinct prime bases $a_1, \ldots, a_t$ according to [1] and [3], where more details can be found. The whole method can be summarised as follows:

1. Choose t odd prime bases $a_1 < \cdots < a_t$ (we always choose the first t smallest primes) and let $A := \{a_1, \ldots, a_t\}$.
2. Let $k_1 = 1$ and choose distinct coprime $k_2, k_3 \in \mathbb{Z}$, $k_2, k_3 > a_t$ (see Table 10.
3. For each $a \in A$, compute the set S_a of primes p reduced modulo $4a$ s.t. $\left(\frac{a}{p}\right) = -1$. This can be done constructively by looping over values $x \in \{1, 2, \ldots, 4a - 1\}$ and adding x to S_a iff $\left(\frac{x}{a}\right)(-1)^{(x-1)(a-1)/4} = -1$ (using quadratic reciprocity).
4. For each $a \in A$, compute the intersection $R_a := \bigcap_{j=1}^{3} k_j^{-1}(S_a + k_j - 1)$, where $k_j^{-1}(S_a + k_j - 1)$ denotes the set $\{k_j^{-1}(s + k_j - 1) \mod 4a \mid s \in S_a\}$ for each $a \in A$. If any are empty, go back to step 2.
5. For each $a \in A$, randomly pick an element $r_a \in R_a$.

6. Using the Chinese Remainder Theorem, find p_1 such that
$$p_1 \equiv k_3^{-1} \pmod{k_2}, p_1 \equiv k_2^{-1} \pmod{k_3} \text{ and } p_1 \equiv r_a \pmod{4a} \text{ for all } a \in A.$$
7. Compute $p_2 = k_2(p_1 - 1) + 1$ and $p_3 = k_3(p_1 - 1) + 1$. If all p_1, p_2, p_3 are primes, then $p_1 p_2 p_3$ is pseudoprime with respect to all bases $a \in A$. Otherwise, go back to step 4 (or even 2 or 1 after a certain amount of time has passed).

If we take $a_1 = 2$ and enforce the condition $p_1 \equiv 3 \pmod 8$ (by slightly tweaking some steps above), the constructed pseudoprimes will meet the Monier-Rabin bound (maximizing the probability of passing the test for a random base choice) and will also pass the MR test for any composite base with no prime divisors greater than a_t [1].

Recall that Carmichael numbers are composite n that divide $a^{n-1} - 1$ for all $a \in \mathbb{Z}$ coprime to n. Equivalently, a composite integer n is a Carmichael number if and only if n is square-free, and $p - 1 \mid n - 1$ for all prime divisors p of n [24]. The pseudoprimes generated in this way are automatically Carmichael numbers [1] and we are using this fact in Section 4.8.

2.1 Generated domain parameters

The generated domain parameters and scripts used to generate them and produce our results are available at `https://crocs.fi.muni.cz/papers/primality_esorics20`.

3 Examples of attacks

3.1 ECDSA/ECDH: Composite *n*

This case uses the 10-factor n parameters as specified in Appendix 2.1. Such a smooth order of the curve allows for a direct application of the Pohlig-Hellman algorithm for computing discrete logarithms to obtain the private key.

The SAGE [36] code (embedded) recovered the private key on a 256-bit curve in just about 7 seconds on an ordinary laptop. Computing such a discrete logarithm on a standard 256-bit curve is currently computationally infeasible.

3.2 ECDSA/ECDH: Composite *p*

This case uses the 10-factor p parameters as specified in Appendix 2.1. Such a curve with composite p can be decomposed into ten much smaller curves modulo the prime divisors of p. On these curves, it is trivial to compute the discrete logarithm of the public key. The resulting discrete logarithm (and the private key) is then recovered via the CRT.

The SAGE code (embedded) recovered the private key on a 256-bit curve in about 9 seconds on an ordinary laptop.

3.3 DSA/DH: Composite *q*

In case of composite q in DSA/DH, the Pohlig-Hellman algorithm for computing discrete logarithms applies again. The SAGE code (embedded) computed the private key of a public key using the 1024 bit DSA/DH parameters given in Appendix 2.1 in 35 minutes on one Intel Xeon X7560 @ 2.26 GHz processor.

3.4 DSA/DH: Composite p

We have used the CADO-NFS [35] implementation of the Number Field Sieve, to demonstrate the ease of computing the discrete logarithm of a public key using the 1024 bit DSA/DH parameters given in Appendix 2.1. We computed the discrete logarithm in the order q subgroup of $\mathbb{Z}^*_{p_1}$ as it defined the smallest group of only 336 bits.

The computation took 70 minutes to recover the private key on three Intel Xeon X7560 @ 2.26 GHz processors (24 cores total), with total CPU time of 22 hours. Furthermore, this computation is generic for all public keys using the given domain parameters. The per-key computation is trivial and takes a few minutes at most.

Only one computation of the discrete logarithm on prime 1024 bit DSA/DH parameters is publicly known [15]. It used the fact that the prime was trapdoored and ran much faster than random parameters. Even then, it took two months on a large computation cluster, with a total CPU time of 385 CPU years.

References

1. Albrecht, M.R., Massimo, J., Paterson, K.G., and Somorovsky, J.: Prime and Prejudice: Primality Testing Under Adversarial Conditions. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 281–298. ACM, New York, NY, USA (2018). DOI: 10.1145/3243734.3243787
2. American National Standard X9.62-1998, Public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ECDSA). Preliminary draft, Accredited Standards Committee X9 (1998)
3. Arnault, F.: Constructing Carmichael numbers which are strong pseudoprimes to several bases. Journal of Symbolic Computation 20(2), 151–161 (1995). DOI: 10.1006/jsco.1995.1042
4. Arnault, F.: Rabin-Miller primality test: composite numbers which pass it. Mathematics of Computation 64(209), 355–361 (1995). DOI: 10.1090/S0025-5718-1995-1260124-2
5. Bernstein, D.J., and Lange, T.: SafeCurves: choosing safe curves for elliptic-curve cryptography, (2017). https://safecurves.cr.yp.to/
6. Bernstein, D.J., Lange, T., and Schwabe, P.: The Security Impact of a New Cryptographic Library. In: Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings, pp. 159–176 (2012). DOI: 10.1007/978-3-642-33481-8_9
7. Biehl, I., Meyer, B., and Müller, V.: Differential Fault Attacks on Elliptic Curve Cryptosystems. In: Proceedings of the 20th Annual International Cryptology Conference. CRYPTO '00, pp. 131–146. Springer, Berlin, Heidelberg (2000). DOI: 10.1007/3-540-44598-6_8
8. Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and Moeller, B.: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492, pp. 1–35. RFC Editor (2006)
9. Bleichenbacher, D.: Breaking a Cryptographic Protocol with Pseudoprimes. In: Public Key Cryptography - PKC 2005, 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005, Proceedings, pp. 9–15 (2005). DOI: 10.1007/978-3-540-30580-4_2
10. Bröker, R.: Constructing elliptic curves of prescribed order. Thomas Stieltjes Institute for Mathematics (2006).

11. Checkoway, S., Maskiewicz, J., Garman, C., Fried, J., Cohney, S., Green, M., Heninger, N., Weinmann, R., Rescorla, E., and Shacham, H.: A Systematic Analysis of the Juniper Dual EC Incident. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016, pp. 468–479 (2016). DOI: 10.1145/2976749.2978395
12. Dorey, K., Chang-Fong, N., and Essex, A.: *Indiscreet Logs: Persistent Diffie-Hellman Backdoors in TLS*, https://eprint.iacr.org/2016/999. (2016)
13. EnigmaBridge: Curated list of JavaCard applications, (2019). https://github.com/EnigmaBridge/javacard-curated-list (visited on 03/17/2020)
14. FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION 186-4 Digital Signature Standard (DSS). Standard, National Institute for Standards and Technology (2013)
15. Fried, J., Gaudry, P., Heninger, N., and Thomé, E.: A Kilobit Hidden SNFS Discrete Logarithm Computation. In: Advances in Cryptology - EUROCRYPT 2017, Paris, France, April 30 - May 4, 2017, Proceedings, Part I, pp. 202–231 (2017). DOI: 10.1007/978-3-319-56620-7_8
16. Galbraith, S.D., Massimo, J., and Paterson, K.G.: Safety in Numbers: On the Need for Robust Diffie-Hellman Parameter Validation. In: Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II, pp. 379–407 (2019). DOI: 10.1007/978-3-030-17259-6_13
17. Harkins, D.: Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES). RFC 5297, pp. 1–26. RFC Editor (2008)
18. Doc 9303 - Machine Readable Travel Documents. Document, International Civil Aviation Organization (2015)
19. IEEE Standard - Specifications for Public-Key Cryptography. Standard, IEEE Std 1363-2000 Working Group (2000)
20. Jancar, J.: ecgen, (2019). https://github.com/J08nY/ecgen
21. Jancar, J., and Svenda, P.: ECTester, (2019). https://crocs-muni.github.io/ECTester/
22. Massimo, J., and Paterson, K.G.: *A Performant, Misuse-Resistant API for Primality Testing*, (2020). https://eprint.iacr.org/2020/065
23. Miller, G.L.: Riemann's Hypothesis and Tests for Primality. In: Proceedings of the Seventh Annual ACM Symposium on Theory of Computing. STOC '75, pp. 234–239. ACM, Albuquerque, New Mexico, USA (1975). DOI: 10.1145/800116.803773
24. Monier, L.: Evaluation and comparison of two efficient probabilistic primality testing algorithms. Theoretical Computer Science 12(1), 97–108 (1980). DOI: 10.1016/0304-3975(80)90007-9
25. Nemec, M., Sys, M., Svenda, P., Klinec, D., and Matyas, V.: The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli. In: 24th ACM Conference on Computer and Communications Security (CCS'2017), pp. 1631–1648. ACM, New York, NY, USA (2017). DOI: 10.1145/3133956.3133969
26. Nir, Y., Josefsson, S., and Pegourie-Gonnard, M.: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier. RFC 8422, pp. 1–34. RFC Editor (2018)
27. Special Publication 800-89: Recommendation for Obtaining Assurances for Digital Signature Applications. Standard, National Institute for Standards and Technology (2006)
28. NSA: Windows CryptoAPI Spoofing Vulnerability (CVE-2020-0601), (2020). https://nvd.nist.gov/vuln/detail/CVE-2020-0601 (visited on 03/17/2020)
29. Oracle: Java Card API 3.0.5, Classic Edition, (2019). https://docs.oracle.com/javacard/3.0.5/api/index.html (visited on 03/17/2020)

30. Pohlig, S., and Hellman, M.: An Improved Algorithm for Computing Logarithms over $GF(p)$ and Its Cryptographic Significance. IEEE Transactions on Information Theory 24(1), 106–110 (1978). DOI: 10.1109/TIT.1978.1055817
31. Polk, T., Housley, R., and Bassham, L.: Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3279, pp. 1–27. RFC Editor (2002)
32. Rabin, M.O.: Probabilistic algorithm for testing primality. Journal of Number Theory 12, 128–138 (1980). DOI: 10.1016/0022-314X(80)90084-0
33. Svenda, P.: JCAlgTest: Detailed analysis of cryptographic smart cards running with JavaCard platform, (2019). https://www.fi.muni.cz/~xsvenda/jcalgtest/ (visited on 03/17/2020)
34. Takahashi, A., and Tibouchi, M.: *Degenerate Fault Attacks on Elliptic Curve Parameters in OpenSSL*, (2019). https://eprint.iacr.org/2019/400
35. The CADO-NFS Development Team: *CADO-NFS, An Implementation of the Number Field Sieve Algorithm*. Release 2.3.0. 2017. http://cado-nfs.gforge.inria.fr/.
36. The Sage Developers: *SageMath, the Sage Mathematics Software System (Version 8.9)*. 2019. https://www.sagemath.org.
37. Washington, L.C.: Elliptic Curves: Number Theory and Cryptography, Second Edition. Chapman & Hall/CRC, Boca Raton, FL, USA (2008)