Prime and Prejudice: Primality Testing Under Adversarial Conditions

Martin R. Albrecht[1], Jake Massimo[1], Kenneth G. Paterson[1], and Juraj Somorovsky[2]

[1] Royal Holloway, University of London
[2] Ruhr University Bochum, Germany
martin.albrecht@rhul.ac.uk, jake.massimo.2015@rhul.ac.uk, kenny.paterson@rhul.ac.uk, juraj.somorovsky@rub.de

Abstract. This work provides a systematic analysis of primality testing under adversarial conditions, where the numbers being tested for primality are not generated randomly, but instead provided by a possibly malicious party. Such a situation can arise in secure messaging protocols where a server supplies Diffie-Hellman parameters to the peers, or in a secure communications protocol like TLS where a developer can insert such a number to be able to later passively spy on client-server data. We study a broad range of cryptographic libraries and assess their performance in this adversarial setting. As examples of our findings, we are able to construct 2048-bit composites that are declared prime with probability 1/16 by OpenSSL's primality testing in its default configuration; the advertised performance is 2^{-80}. We can also construct 1024-bit composites that *always* pass the primality testing routine in GNU GMP when configured with the recommended minimum number of rounds. And, for a number of libraries (Cryptlib, LibTomCrypt, JavaScript Big Number, WolfSSL), we can construct composites that *always* pass the supplied primality tests. We explore the implications of these security failures in applications, focusing on the construction of malicious Diffie-Hellman parameters. We show that, unless careful primality testing is performed, an adversary can supply parameters (p, q, g) which on the surface look secure, but where the discrete logarithm problem in the subgroup of order q generated by g is easy. We close by making recommendations for users and developers. In particular, we promote the Baillie-PSW primality test which is both efficient and conjectured to be robust even in the adversarial setting for numbers up to a few thousand bits.

1 Introduction

Many cryptographic primitives rely on prime numbers, with RSA being the most famous example. However, even in constructions that do not rely on the difficulty of factoring integers into prime factors, primality is often relied upon to prevent an adversary from applying a divide-and-conquer approach (e.g. in the Pohlig-Hellman algorithm or in a Lim-Lee small subgroup attack [VAS+17]) or to prevent the existence of degenerate cases such as zero divisors (which may complicate security proofs or reduce output entropy).

One approach to obtaining prime numbers in instantiations of these cryptographic primitives is to produce such numbers as they are needed on whatever device requires them. This is accomplished by sampling random integers and checking for primality. This process can be computationally intensive to the point of being prohibitively so. The high cost of producing prime numbers led implementations to seek ways to reduce this cost and, as demonstrated in [NSS+17], these performance improvements may then lead to devastating attacks.

If the required prime numbers are public, an alternative approach is possible: (low-power) devices are provisioned with prime numbers from a server or a standard. For example, the popular Telegram messenger [LLC18] uses Diffie-Hellman (DH) parameters provided *by the server* to establish end-to-end encryption *between peers*. If the peers do not validate the correctness of the supplied DH parameters,[3] the Telegram server can provide malicious DH parameters with composite group orders and thereby passively obtain the established secrets. As a concrete and

[3] We stress that they *do* perform validation in the default implementation.

closely related example, Bleichenbacher [Ble05] showed how the reliance on a small and fixed set of bases in Miller-Rabin primality testing in GNU Crypto 1.1.0 could be exploited to fool GNU Crypto into accepting malicious DH parameters. In particular, this led to an attack on the GNU Crypto implementation of SRP enabling users' passwords to be recovered.

Another example is the Transport Layer Security protocol [DR08] which can use Diffie-Hellman key exchange to establish master secrets in the handshake protocol. The DH parameters are generated by the TLS server and sent to the client during each TLS handshake.[4] It is clear that the TLS server provider does not gain any advantage by sending malicious DH parameters to the client since it knows the established master key. However, we can consider an adversarial developer who implements a malicious sever with backdoored DH parameter generation, cf. [Won16,FGHT17]. If such parameters are accepted by TLS clients and used in the DH key exchange, a passive adversary can observe the traffic and obtain the master key. Here, weak DH parameters that still pass tests by trusted tools offer a sense of plausible deniability. Moreover, if an application simply silently rejects bad parameters then any countermeasures could be overcome by repeatedly sending malicious parameter sets having a reasonable probability of fooling those countermeasures, until the target client accepts them.

In recent years we have seen several backdoors in cryptographic implementations. For example, NIST standardised the Dual EC pseudorandom number generator (PRNG) which allows an adversary to predict generated random values if it can select a generator point Q whose discrete logarithm is known and collect enough PRNG output [CNE+14]. In 2016 it was shown that Juniper had implemented this PRNG in such a way as to enable an adversary to passively decrypt VPN sessions [CMG+16].

A notable example of a potential backdoor involving a composite number is the security advisory [Rie16] pushed by command-line data transfer utility `socat`, which is popular with security professionals such as penetration testers. There, the DH prime p parameter was replaced with a new 2048 bit value because "*the hard coded* 1024 *bit DH p parameter was not prime*". The advisory goes on to state "*since there is no indication of how these parameters were chosen, the existence of a trapdoor that makes possible for an eavesdropper to recover the shared secret from a key exchange that uses them cannot be ruled out*", which highlights a real world application of this attack model. Similarly, the prime group parameter p given by Group 23 of RFC5114 [LK08] for use in DH key exchanges has been found to be partially vulnerable to small subgroup attacks [VAS+17]. It might seem that code reviews and the availability of rigorous primality testing (in, say, mathematical software packages, cf. Appendix K) impose high rates of detectability for malicious parameter sets in code or standards, but as these examples highlight, such sets still occur in practice.

Given these incidents we can assume a motivated adversary who is able to implement software serving maliciously generated primes and/or DH parameters. Thus, there is a need for cryptographic applications that rely on third-party primes to perform primality testing. Indeed, many cryptographic libraries incorporate primality testing facilities and thus it appears this requirement is easy to satisfy. However, the primary application of these tests is to check primality (or, more precisely, compositeness) for locally-generated, random inputs during prime generation. Thus, it is a natural question to ask whether these libraries are robust against malicious inputs, i.e. inputs designed to fool the library into accepting a composite number as prime. We refer to this setting as *primality testing under adversarial conditions*.

1.1 Overview of Primality Testing

One of the most widely used primality tests is the Miller-Rabin [Mil75,Rab80] test. Based upon modular exponentiation by repeated squaring, Miller-Rabin is an efficient polynomial-time algorithm with complexity $O(t \log^3 n)$ where t is the number of trials performed. Yet due to its probabilistic nature, it is well known that a t-trial Miller-Rabin test is only accurate in declaring a given composite number to be composite with probability at least $1 - (1/4)^t$. Arnault [Arn95], Pomerance [PSW80] and Narayanan [Nar14] all explore methods of producing Miller-Rabin pseudoprimes, that is,

[4] Up to version 1.2 (inclusive) of the protocol.

Table 1. Results of our analysis of cryptographic libraries. This shows how the number of rounds of Miller-Rabin used is determined, whether a Baillie-PSW test is implemented, the documented failure rate of the primality test (that is, the probability that it wrongly declares a composite to be prime), and our highest achieved failure rate for composite input.

Library	Rounds of MR testing	Baillie-PSW?	Documented Failure Rate	Our Highest Failure Rate
OpenSSL 1.1.1-pre6	Default bit-size based	No	$< 2^{-80}$	1/16
GNU GMP 6.1.2	User-defined t	No	$(1/4)^t$	100% for $t \leq 15$
GNU Mini-GMP 6.1.2	User-defined t	No	$(1/4)^t$	100% for $t \leq 101$
Java 10	User-defined t	Yes (≥ 100 bits)	$< (1/2)^t$	0% for ≥ 100 bits
JSBN 1.4	User-defined t	No	$< (1/2)^t$	100%
Libgcrypt 1.8.2	User-defined t	No	Not given	1/1024[†]
Cryptlib 3.4.4	User-defined $t \leq 100$	No	Not given	100%
LibTomMath 1.0.1	User-defined $t \leq 256$	No	$(1/4)^t$	100%
LibTomCrypt 1.18.1	User-defined $t \leq 256$	No	$(1/4)^t$	100%
WolfSSL 3.13.0	User-defined $t \leq 256$	No	$(1/4)^t$	100%
Bouncy Castle C# 1.8.2	User-defined t	No	$(1/4)^t$	$(1/4)^t$
Botan 2.6.0	User-defined t	No	$\leq (1/2)^t$	$(1/4)^t$
Crypto++ 7.0	2 or 12	Yes	Not given	0%
GoLang 1.10.3	User-defined t	Yes	$< (1/4)^t$	0%
GoLang pre-1.8	User-defined t	No	$< (1/4)^t$	100% for $t \leq 13$

[†] When calling the `check_prime` function as opposed to `gcry_prime_check` (or calling `gcry_prime_check` in versions prior to 1.3.0).

composite numbers that when tested by Miller-Rabin, achieve the highest probability of $(1/4)^t$ of being wrongly classified as "probably prime".

Another common choice is the Lucas test [BW80], and its more stringent variant the strong Lucas probable prime test. Similarly to the Miller-Rabin test, t trials of a strong Lucas test will declare a given composite number as being composite with probability at least $1 - (4/15)^t$ and as being prime with probability at most $(4/15)^t$ [Arn97]. As with the Miller-Rabin test, there are known methods for constructing strong Lucas pseudoprimes [Arn95].

The Lucas test (strong or standard) can be combined with a single Miller-Rabin test (on base 2) to form what is known as the Baillie-PSW test [Pom84]. Due to slightly longer running times, this test is often only adopted for use in mathematical software packages and seen less in cryptographic libraries. Unlike the Miller-Rabin and Lucas tests when performed alone, there are no known pseudoprimes for the Baillie-PSW test (yet there is no proof that they cannot exist).

Clearly, when conducting a Miller-Rabin or Lucas test, the choice of the parameter t (the number of trials) is critical. Many cryptographic libraries, for example OpenSSL [OP18b], use test parameters originating from [DLP93] as popularised in the *Handbook of Applied Cryptography* [MVOV96]. These give the number of iterations of Miller-Rabin needed for an error rate less than 2^{-80}, when testing a *random* input n. A main result of [DLP93] is that if n is a randomly selected b-bit odd integer, then t independent rounds of Miller-Rabin testing to give an error probability:

$$P(X|Y_t) < b^{3/2} 2^t t^{-1/2} 4^{2-\sqrt{tb}} \quad \text{for } 3 \leq t \leq b/9 \text{ and } b \geq 21,$$

where X denotes the event that n is composite, and Y_t the event that t rounds of Miller-Rabin declares n to be prime. This bound enables the computation of the minimum value t needed to obtain $P(X|Y_t) \leq 2^{-80}$ for a range of bit-sizes b; see Table 2.

However, these error estimates are for primality testing with Miller-Rabin on randomly generated n. In the *adversarial* setting, we are actually concerned with the probability that t trials of Miller-Rabin (or some other test) declare a *given* n to be prime, given that it is composite. This probability is independent of bit-size, and is at most $(1/4)^t$ if random bases are used in Miller-Rabin tests. Similar remarks apply for both variants of the Lucas test.

Technical guideline documents such as Cryptographic Mechanisms BSI TR–02102–1 [fSidI17] and publicly announced standards such as the Digital Signature Standard (DSS) FIPS 186–4 C.3.2 [Pub13] and the International Standard for Prime Number Generation ISO/IEC 18032 [Sta05] provide formal guidance and suggestions on parameter choices. BSI TR–02102–1 suggests that in the worst case, 50 rounds of random base selection Miller-Rabin must be performed, and in the average case it, like ISO/IEC 18032, references the method proposed by Damgård [DLP93] and the Handbook of Applied Cryptography [MVOV96] as described above. BSI TR–02102–1 also references the guidance given in FIPS 186–4, who give much more conservative rounds of iterations ($t = 40$ for 1024 and $t = 56$ for 2048 bit n) for DSA parameter generation, as well as a detailed justification. FIPS 186–4 advocates the use of an additional Lucas primality test (cf. Section 2.3) and also gives an elaboration of the distinction between the two conditional probabilities described above, in its Appendix F.2. The standard ISO/IEC 18032 correctly states that the worst case failure probability is indeed $(1/4)^t$, but does not make the distinction between the two conditional probabilities as clear.

Many libraries, for example GNU GMP [Gt18], provide primality testing functions to be deployed in applications such as mathematical software packages that require arbitrary precision arithmetic. These functions often obligate the user to choose the 'certainty' or accuracy of the primality test performed. Since these parameters are often hidden from the end user, this then forces the responsibility of choosing suitable parameters on the developer of the application using the library. The only resulting guidance that is filtered through from the standards are then found in the documentation of the library, which are often brief and informal.

1.2 Contributions & Outline

We investigate the implementation landscape of primality testing in both cryptographic libraries and mathematical software packages, and measure the security impact of the widespread failure of implementations to achieve robust primality testing in the adversarial setting.

We review primality testing in Section 2. In Section 3, we then review known techniques for constructing pseudoprimes and extend them with our target applications in mind. In Section 4, we then survey primality testing in cryptographic libraries and mathematical software, evaluating their performance in the adversarial setting. We propose techniques to defeat their tests where we can. Overall, our finding is that most libraries are not robust in the adversarial setting. Our main results in this direction are summarised in Table 1.

As one highlight of our results, we find that OpenSSL with its default primality testing routine will declare certain composites n of cryptographic size to be prime with probability $1/16$, while the documented failure rate is 2^{-80}. This arises from OpenSSL's reliance on Table 2 to compute the number of rounds of Miller-Rabin testing required, and this number decreases as the size of n increases. As another highlight, we construct a 1024-bit composite that is guaranteed to be declared prime by the GNU GMP library [Gt18] for anything up to and including 15 rounds of testing (the recommended minimum by GMP). This is as a result of GNU GMP initialising its PRNG to a static state and consequently using bases in its Miller-Rabin testing that depend only on n, the number being tested. We also show how base selection by randomly sampling from a fixed list of primes, as in Cryptlib, LibTomCrypt, JavaScript Big Number (JSBN) and WolfSSL, can be subverted: we construct composites n of cryptographic size that are guaranteed to be declared prime by these libraries regardless of how many rounds of testing are performed.

We go on to examine the implications of our findings for applications, focussing on DH parameter testing. The good news is that OpenSSL is not impacted because of its insistence on safe primes for use in DH; that is, it requires DH parameters (p, q, g) for which $q = (p-1)/2$ and both p, q are tested for primality. Our current techniques cannot produce malicious parameters in this case. On the other hand, when more liberal choices of parameter are permitted, as is the case in Bouncy Castle and Botan, we are able to construct malicious DH parameter sets which pass the libraries' testing but for which the discrete logarithm problem in the subgroup generated by g is easy.

We close by discussing avenues for improving the robustness of primality testing in the adversarial setting in Section 6.

1.3 Disclosure and Mitigations

We reported our findings and suggested suitable mitigations based on the outcome of our analysis to OpenSSL, GMP, JSBN, Cryptlib, LibTomMath, LibTomCrypt, WolfSSL, Bouncy Castle and Botan. We give a short review of the outcome of these discussions.

When we reached out to the OpenSSL developers, they were in the process of amending their primality testing code to make it FIPS-complaint [OP18a]. However, these changes do not consider the adversarial scenario on which our paper focuses, and the default settings in OpenSSL remain weak in that scenario. Thus, it is left to the user to choose parameters suitable for this scenario. LibTomMath and LibTomCrypt developers are also in the process of adjusting the primality testing functions within their library. They plan to remove the fixed base Miller-Rabin testing and replace the function with a Baillie-PSW test in accordance with our recommendations [Lib18]. WolfSSL have made several adaptations in an upcoming release [Wol18a] to their primality testing in response to our findings. This includes now performing Miller-Rabin with pseudorandom bases, not overriding the user's choice of iterations and increasing the number of rounds performed on prime parameters in DH and DSA check functions. Bouncy Castle have also made changes based upon our findings, by removing the DH verification function and replacing it with a whitelisting approach in upcoming release 1.8.3. They are also looking into performing Baillie-PSW in future versions as per our suggestion. Botan version 2.7.0 [Llo18b] has increased the number of rounds of Miller-Rabin performed in DH verification and includes the addition of the Lucas test to perform Baillie-PSW as per our suggestions. GNU GMP, Mini-GMP and Cryptlib all remain unchanged, but the authors of Cryptlib pointed out a code comment that indicates the limitations of their primality test. We received no correspondence from JSBN.

2 Background on Primality Testing

A primality test is an algorithm used to determine whether or not a given number is prime. These primality tests come in two different varieties; deterministic and probabilistic. Deterministic primality testing algorithms prove conclusively that a number is prime, but they tend to be slow and are not widely used in practice. A famous example is the AKS test [AKS04]. We do not discuss such tests further in this paper, except where they arise in certain mathematical software.

Probabilistic primality tests make use of arithmetic conditions that all primes must satisfy, and test these conditions for the number n of interest. If the condition *does not* hold, we learn that n must be composite. However, if it does hold we may only infer that n is *probably* prime, since some composite numbers may also pass the test. By making repeated tests, the probability that n is composite conditioned on it having passed some number t of tests can be made sufficiently small for cryptographic applications. A typical target probability is 2^{-80}, cf. [MVOV96, 4.49]. A critical consideration here is whether n was generated adversarially or not, since the bounds that can be inferred on probability may be radically different in the two cases; more on this below.

We now discuss three widely-used tests: the Fermat, Miller-Rabin, and Lucas tests.

2.1 Fermat Test

The Fermat primality test is based upon the following theorem.

Theorem 1 (Fermat's Little Theorem). *If p is prime and a is not divisible by p, then*

$$a^{p-1} \equiv 1 \pmod{p}.$$

To test n for primality, one simply chooses a base a and computes $a^{n-1} \pmod{n}$. If $a^{n-1} \not\equiv 1 \pmod{n}$, then we can be certain that n is composite. If after testing a variety of bases a_i, we find that that they all satisfy $a_i^{n-1} \equiv 1 \pmod{n}$, we may conclude that n is probably prime.

It is well known that there exists composite numbers that satisfy $a^{n-1} \equiv 1 \pmod{n}$ for all integers a that are not divisible by n. These numbers completely thwart the Fermat test, and are known as Carmichael numbers. These will be of relevance in the sequel. The following result is fundamental in the construction of Carmichael numbers.

Theorem 2 (Korselt's Criterion). *A positive composite integer n is a Carmic-hael number if and only if n is square-free, and $p-1 \mid n-1$ for all prime divisors p of n.*

2.2 Miller-Rabin Test

The Miller-Rabin [Mil75,Rab80] primality test is based upon the fact that there are no non-trivial roots of unity modulo a prime. Let $n > 1$ be an odd integer to be tested and write $n = 2^e d + 1$ where d is odd. If n is prime, then for any integer a with $1 \leq a < n$, we have:

$$a^d \equiv 1 \mod n \quad \text{or} \quad a^{2^i d} \equiv -1 \bmod n \text{ for some } 0 \leq i < e.$$

The Miller-Rabin test then consists of checking the above conditions, declaring a number to be (probably) prime if one of the two conditions holds, and to be composite if both fail. If one condition holds, then we say n is a *pseudoprime to base a*, or that a is a *non-witness to the compositeness of n* (since n may be composite, but a does not demonstrate this fact).

For a composite n, let $S(n)$ denote the number of non-witnesses $a \in [1, n-1]$. An upper-bound on $S(n)$ is given by results of [Mon80,Rab80]:

Theorem 3 (Monier-Rabin Bound). *Let $n \neq 9$ be odd and composite. Then*

$$S(n) \leq \frac{\varphi(n)}{4}$$

where φ denotes the Euler totient function.

This bound will be critical in determining the probability that an adversarially generated n passes the Miller-Rabin test. Since for large n, we have $\varphi(n) \approx n$, it indicates that no composite n can pass the Miller-Rabin test for t random bases with probability greater than $(1/4)^t$. Hence achieving a target probability of 2^{-80} requires $t \geq 40$. The test is commonly implemented using either (a) a set of fixed bases (e.g. JSBN) or (b) randomly chosen bases (e.g. OpenSSL). Of course, the $(1/4)^t$ bound only holds in the case of randomly chosen bases.

2.3 Lucas Test

The Lucas primality test [BW80] makes use of Lucas sequences, defined as follows:

Definition 1 (Lucas sequence [Arn97]). *Let P and Q be integers and $D = P^2 - 4Q$. Then the Lucas sequences (U_k) and (V_k) (with $k \geq 0$) are defined recursively by:*

$$\begin{aligned} U_{k+2} &= PU_{k+1} - QU_k \quad \textit{where,} & U_0 &= 0,\ U_1 = 1, \\ V_{k+2} &= PV_{k+1} - QV_k & V_0 &= 2,\ V_1 = P. \end{aligned}$$

The Lucas probable prime test then relies on the following theorem (in which $\left(\frac{x}{p}\right)$ denotes the Legendre symbol, with value 1 if x is a square modulo p and value -1 otherwise):

Theorem 4 ([CP06]). *Let P, Q and D and the Lucas sequences $(U_k), (V_k)$ be defined as above. If p is a prime with $\gcd(p, 2QD) = 1$, then*

$$U_{p-\left(\frac{x}{p}\right)} \equiv 0 \pmod{p}. \tag{1}$$

The Lucas probable prime test repeatedly tests property (1) for different pairs (P, Q). This leads to the notion of a Lucas pseudoprime with respect to such a pair.

Definition 2 (Lucas pseudoprime). *Let n be a composite number such that $\gcd(n, 2QD) = 1$. If $U_{n-\left(\frac{x}{n}\right)} \equiv 0 \pmod{n}$, then n is called a* Lucas pseudoprime *with respect to parameters (P, Q).*

We can now introduce the notion of a strong Lucas probable prime and strong Lucas pseudoprime with respect to parameters (P, Q) by the following theorem.

Theorem 5 ([Arn97]). *Let p be a prime number not dividing $2QD$. Set $p - \left(\frac{D}{p}\right) = 2^k q$ with q odd. Then one of the following conditions is satisfied:*

$$p \mid U_q \quad or \quad \exists i \text{ such that } 0 \leq i < k \text{ and } p \mid V_{2^i q}. \tag{2}$$

The strong Lucas probable prime test repeatedly tests property (2) for different pairs (P, Q). This leads to the definition of a strong Lucas pseudoprime with respect to parameters (P, Q) as follows.

Definition 3 (strong Lucas pseudoprime). *Let n be a composite number such that $\gcd(n, 2QD) = 1$. Set $n - \left(\frac{D}{n}\right) = 2^k q$ with q odd. Suppose that:*

$$n \mid U_q \quad or \quad \exists i \text{ such that } 0 \leq i < k \text{ and } n \mid V_{2^i q}.$$

Then n is called a strong Lucas pseudoprime *with respect to parameters (P, Q).*

A strong Lucas pseudoprime is also a Lucas pseudoprime (for the same (P, Q) pair), but the converse is not necessarily true. The strong version of the test is therefore seen as the more stringent option.

Remark 1. The Lucas pseudoprime and strong Lucas pseudoprime tests are also known as a Lucas-Selfridge test and a strong Lucas-Selfridge test respectively, specifically when used with Selfridge's parameters $P = 1$, $Q = -1$.

Analogously to the Monier-Rabin theorem for pseudoprimes for the Miller-Rabin primality test, Arnault [Arn97] showed that for an integer D and n a composite with $\gcd(D, n) = 1$ and $n \neq 9$, the number of pairs (P, Q) with $0 \leq P, Q < n$, $\gcd(Q, n) = 1$, $P^2 - 4Q \equiv D \pmod{n}$ such that n is strong Lucas pseudoprime with respect to (P, Q) is at most $4n/15$. There is an exception to this result for certain forms of twin primes (we omit the details here), but Arnualt goes on to prove that even these particular forms of twin prime n have at most $n/2$ pairs (P, Q) such that n is a strong Lucas pseudoprime with respect to (P, Q). From this, we can infer that t applications of the strong Lucas test would declare a composite n to be probably prime with a probability at most $(4/15)^t$.

2.4 Baillie-PSW

The Baillie-PSW [Pom84] test is a probabilistic primality test formed by combining a single Miller-Rabin test with base 2 with either a Lucas or strong Lucas pseudoprime test. The idea of this test is that the two components are "orthogonal" and so it is very unlikely that a number n will pass both parts. Indeed, there are no known composite n that pass the Baille-PSW test. Gilchrist [Gil13] confirmed that there are no Baillie-PSW pseudoprimes less than 2^{64}. PRIMO [Mar16] is an elliptic curve based primality proving program that uses the Baillie-PSW test to check all intermediate probable primes. If any of these values were indeed composite, the final certification would necessarily have failed. Since this has never occurred during its use, PRIMO's author Martin estimates [Wei18] that there are no Baillie-PSW pseudoprimes with less than about 10000 digits. This empirical evidence suggests that numbers of cryptographic size for use in Diffie-Hellman and RSA are unlikely to be Baillie-PSW pseudoprimes. However, Pomerance gives a heuristic argument in [Pom84] that there are in-fact infinitely many Baillie-PSW pseudoprimes. The construction of a single example is a significant open problem in number theory.

3 Constructing Pseudoprimes

In this section, we review known methods of constructing pseudoprimes for the Miller-Rabin and Lucas tests. We also provide variations on these methods. We will use the results of this section in the next one, where we study the robustness of cryptographic libraries for primality testing in the adversarial setting.

3.1 Miller-Rabin Pseudoprimes

The exact number of non-witnesses $S(n)$ for any composite number n can be computed given the factorisation of n [CP06]. Generating composites n that have large numbers of non-witnesses is not so straightforward. In empirical work, Pomerance *et al.* [PSW80] showed that many composite numbers that pass a Miller-Rabin primality test have the form $n = (k+1)(rk+1)$ where r is small and $k+1$ is prime. More recently, Höglund [Hö16] and Nicely [Nic16] used the Miller-Rabin primality test as implemented in GNU GMP to test randomly generated numbers of this form for various values of r and for various different sizes of k. Their results support the claims made by [PSW80].

We now consider existing methods for producing composites which have many non-witnesses, for two forms of the Miller-Rabin test: firstly where the bases are chosen randomly and secondly where a fixed set of bases is used.

3.1.1 Random Bases. For random bases, we are interested in constructing composite n that have large numbers of non-witnesses, i.e. for which $S(n)$ is large. Such numbers will pass the Miller-Rabin test with probability $S(n)/n$ per trial; of course, this probability is bounded by $\varphi(n)/4n \approx 1/4$ by the Monier-Rabin theorem, but we are interested in how close to this bound we can get. We rely on the following:

Theorem 6 ([Nar14]). *Consider an odd composite integer n with m distinct prime factors $p_1, \ldots, p_m$. Suppose that $n = 2^e \cdot d + 1$ where d is odd. Also suppose that $n = \prod_{i=1}^{m} p_i^{t_i}$ where each p_i can be expressed as $2^{e_i} \cdot d_i + 1$ with each d_i odd. Then*

$$S(n) = \prod_{i=1}^{m} \gcd(d, d_i) \cdot \left(\frac{2^{\min(e_i) \cdot m} - 1}{2^m - 1} + 1 \right). \tag{3}$$

Note how the bound in this theorem does not depend on the exponents t_i, indicating that square-free numbers will have relatively large $S(n)$. Also note the dependence on the terms $\gcd(d, d_i)$, indicating that ensuring that the odd part of each prime factor p_i has a large gcd with the odd part of n is necessary for large n. As an easy corollary of this theorem, we obtain:

Corollary 1 ([Nar14]). *Let x be an odd integer such that $2x+1$ and $4x+1$ are both prime. Then $n = (2x+1)(4x+1)$ has $\varphi(n) = 8x^2$ and achieves the Monier-Rabin bound, i.e. it satisfies $S(n) = \varphi(n)/4$.*

The proof of this corollary follows easily on observing that we may take $m = 2$ and $d = d_1 = d_2 = x$ in the preceding theorem. Narayanan [Nar14] also showed that if n is a Carmichael number of the form $p_1 p_2 p_3$, where each p_i is a distinct prime with $p_i \equiv 3 \pmod 4$, then $S(n)$ achieves the Monier-Rabin bound. He also gave further results showing that these two forms for n are the only ones achieving the Monier-Rabin bound, with all other n satisfying $S(n) \leq \varphi(n)/6$.

3.1.2 Fixed Bases. Some implementations of the Miller-Rabin primality test select bases from a fixed list (often of primes), rather than choosing them at random. For example, until 2010, the PyCrypto 2.1.0 (2009) [Lit09] primality test `isPrime()` performed 7 rounds of Miller-Rabin using the first 7 primes as bases, while LibTomMath chooses the first t entries from a hard-coded list of primes as bases.

Arnault [Arn95] presented a method for producing composite numbers $n = p_1 p_2 \cdots p_h$ that are guaranteed to be declared prime by Miller-Rabin for any fixed set of prime bases $A = \{a_1, a_2, \ldots, a_t\}$. We give an overview and examples of Arnault's method in Appendix A.

Since fixed base Miller-Rabin tests are relatively uncommon in implementations, it might seem that Arnault's method would not be very useful. We shall however see that this method is particularly helpful when an implementation chooses bases randomly from a large fixed list of possibilities. For example, an implementation might select prime bases randomly from a list of

primes below 1000; since Arnault's method scales well (we simply need to solve more congruences simultaneously with the CRT) we can use this method to produce a composite n such that all primes below 1000 are non-witnesses for n. We shall see applications of this approach for different libraries in Sections 4.3, 4.5, 4.7, 4.8, 4.9 and 4.10.

3.1.3 Hybrid Technique. The method above produces composites that are in fact always Carmichael numbers. We know from Section 3.1.1 that if n is a Carmichael number with 3 distinct prime factors all congruent to 3 (mod 4), then n has the maximum number of non-witnesses, $\varphi(n)/4$. We can set $h = 3$ in Arnault's method and tweak it slightly to ensure that, as well as producing n with a specified set A of non-witnesses, it produces an n meeting the Monier-Rabin bound, so that random base Miller-Rabin tests will also pass with the maximum probability. The tweak is very simple: we ensure that $2 \in A$; this forces $p_1 \equiv 3$ or $5 \pmod 8$; we then select $p_1 \equiv 3 \pmod 8$ so that $p_1 \equiv 3 \pmod 4$. Arnault's method sets $p_i = k_i(p_1 - 1) + 1$ where the k_i are co-prime to all the elements of A. Since $2 \in A$, the k_i must all be odd; it is easy to see that this forces $p_i \equiv 3 \pmod 4$ too.

We will give an application of this technique in Section 4.6.

3.1.4 Extension For Composite Fixed Bases. The method of Arnault [Arn95] works (as presented) only for prime bases, and not for *composite* bases. Although less common, some implementations use both prime and composite bases in their Miller-Rabin testing. By setting $n \equiv 3 \pmod 4$, we know that $e = 1$ when writing $n = 2^e \cdot d + 1$ for d odd. In this case, the conditions to pass the Miller-Rabin test simply become $a^{(n-1)/2} \equiv \pm 1 \pmod n$. Hence, if $n \equiv 3 \pmod 4$ is pseudoprime to some set of bases $\{a_1, a_2, \ldots a_t\}$, then n is also pseudoprime for any base b arising as a product $b = a_1^{e_1} \cdot a_2^{e_2} \cdot \cdots \cdot a_t^{e_t} \pmod n$ (for any set of indices $e_i \in \mathbb{Z}$). Therefore we can construct a composite n that is pseudoprime with respect to any list of bases $\{b_1, \ldots, b_t\}$ (of which any number can be composite) by using the hybrid method described in Section 3.1.3, but with set A in that method being the complete set of prime factors arising in the b_i. Note that in this method, n is of the form $n = p_1p_2p_3$ where each $p_i \equiv 3 \pmod 4$, so we have $n \equiv 3 \pmod 4$ as needed. Moreover, because of the form of n, the composites generated in this manner will also meet the Monier-Rabin bound.

We will give an application of this technique in Section 4.3, where we study Mini-GMP [Gt18] which uses Euler's polynomial to generate Miller-Rabin bases.

3.2 Lucas Pseudoprimes

Like Miller-Rabin pseudoprimes, Lucas pseudoprimes are with respect to some choice of test parameters. Throughout this work we follow Selfridge's Method A [BW80] of parameter selection, which is summarised as follows:

Definition 4 (Selfridge's Method A [BW80]). *Let D be the first element of the sequence $5, -7, 9, -11, 13, \ldots$ for which $\left(\frac{D}{n}\right) = -1$. Then set $P = 1$ and $Q = (1 - D)/4$.*

There are two reasons for studying this particular method for setting parameters. The first is that it is the parameter choice used when performing the Lucas part of the Baillie-PSW primality test [PSW80,BW80]. The second is that this is the method that both Java [Cor18] and Crypto++ [Dai18] libraries that we study use in their implementation of the Lucas test.

The Lucas and strong Lucas-probable prime tests with this parameter choice are commonly referred to in the literature as Lucas and strong Lucas-Selfridge probable prime tests. Pseudoprimes for this parameter choice are well-documented. The OEIS sequence A217120 [Bai13a] presents a small list of them, referring to a table of all Lucas pseudoprimes below $10^{14} \approx 2^{47}$ compiled by Jacobsen [Jac15]. There is an equivalent sequence A217255 [Bai13b] for strong Lucas pseudoprimes. Any pseudoprime for the strong Lucas probable prime test with respect to some parameter set (P, Q), is also a pseudoprime for the Lucas probable prime test.

Arnault [Arn95] also presented a scalable method that takes as input a set of parameter choices $\{(P_1, Q_1, D_1), (P_2, Q_2, D_2), \ldots, (P_t, Q_t, D_t)\}$ and returns a composite n of the form $n = p_1 p_2 \cdots p_h$ that is a strong Lucas pseudoprime to the parameters (P_i, Q_i, D_i) for all $1 \leq i \leq t$. The method is similar to that for constructing Miller-Rabin pseudoprimes for fixed bases, but differs in its details. In particular, the two construction methods are sufficiently different that it seems hard to derive a single method producing n that are pseudoprimes for both the Miller-Rabin and Lucas tests.

3.2.1 A specialisation of Arnault [Arn95] for Selfridge's Method A For Selfridge's Method A, we know that if we take an n such that $\left(\frac{5}{n}\right) = -1$, then a single test on n with parameter set $(P, Q, D) = (1, -1, 5)$ will be performed. We next show how to specialise Arnault's construction [Arn95] so that it will produce composites n that are guaranteed to be declared prime by a strong Lucas test for this parameter set.

Following Arnault's construction, we consider n of the form $n = p_1 p_2 p_3$ where $p_i = k_i(p_1 + 1) - 1$ for $i \in \{2, 3\}$, with k_2 and k_3 odd integers.

We first note that the p_i must satisfy certain conditions with respect to Legendre symbols (see [Arn95, Lemmas 6.1 and 6.2]):

$$\left(\frac{D}{p_i}\right) = \left(\frac{Q}{p_i}\right) = -1 \qquad \text{for all } i \text{ such that } 1 \leq i \leq 3.$$

With our single parameter set $(P, Q, D) = (1, -1, 5)$, this becomes:

$$\left(\frac{-1}{p_i}\right) = \left(\frac{5}{p_i}\right) = -1 \qquad \text{for all } i \text{ such that } 1 \leq i \leq 3. \tag{4}$$

Now $\left(\frac{-1}{p_i}\right) = -1 \Leftrightarrow p_i \equiv 3 \pmod 4$. Since $p_i = k_i(p_1 + 1) - 1$ for $i \in \{2, 3\}$, and the k_i are odd, then it is easy to show that if $p_1 \equiv 3 \pmod 4$ then it follows that $p_i \equiv 3 \pmod 4$ for $i = 2, 3$ as well. We also have that $\left(\frac{5}{p_i}\right) = -1 \Leftrightarrow p_i \equiv 2 \text{ or } 3 \pmod 5$. Therefore condition (4) is satisfied when $p_1 \equiv 3 \text{ or } 7 \pmod{20}$ (by the CRT) and $p_i \equiv 2 \text{ or } 3 \pmod 5$ for $i \geq 2$.

At this point we must choose k_2, k_3 and add conditions that ensure the coefficients in [Arn95, Lemma 6.1] are indeed integers. These conditions are simple:

$$p_1 \equiv k_3^{-1} \pmod{k_2} \qquad \text{and} \qquad p_1 \equiv k_2^{-1} \pmod{k_3}.$$

We choose to fix $p_1 \equiv 7 \pmod{20}$ and select $(k_2, k_3) = (31, 43)$. This produces our final congruence that prime p_1 must satisfy: $p_1 \equiv 6647 \pmod{26660}$. We now search for a prime p_1 that satisfies this congruence, and such that p_2 and p_3 satisfying $p_i = k_i(p_1 + 1) - 1$ for $i = 2, 3$ are also primes with $p_2 \equiv p_3 \equiv 2 \text{ or } 3 \pmod 5$.

The smallest solution is the following:

$$p_1 = 486527, p_2 = 15082367, p_3 = 20920703$$

This yields a 68-bit $n = 153515674455111174527$ which indeed does pass the strong Lucas test using Selfridge's Method A for parameter selection. Of course, we can take any (p_1, p_2, p_3) satisfying the above conditions (which are not too onerous to satisfy), and in this sense the method scales well to numbers n of cryptographically interesting size. For example, Appendix B shows a 2050-bit example generated using the above procedure.

This generation technique is also versatile, as we can simply include additional parameters in our set dependent on which parameter selection methods a particular test uses. This allows us to generate composites that are declared prime by a variety of strong Lucas tests, at the small cost of solving a few more simultaneous congruences with the CRT.

4 Cryptographic Libraries and Mathematics Packages

Many cryptographic libraries offering implementations of common cryptographic protocols also provide a toolkit for handling arbitrary-precision integer arithmetic, including primality testing. These functions would be used, for example, for testing the primality of Diffie-Hellman parameters.

This section provides a survey of primality testing in a broad and representative range of cryptographic libraries (OpenSSL, GNU GMP and Mini-GMP, Java, JavaScript Big Number (JSBN), Libgcrypt, Cryptlib, LibTomMath, LibTomCrypt, WolfSSL, Bouncy Castle, Botan, Crypto++ and GoLang). For each library, we first describe how it implements primality testing. We then tailor a composite likely to be declared prime by each particular library, and quantify the probability that our composite passes the library's primality test (so that the primality test fails). Our findings are summarised in Table 1. Throughout, we will refer to the number of rounds of Miller-Rabin testing as t.

4.1 OpenSSL

OpenSSL is the most widely used open source cryptographic library and TLS implementation. Throughout, we consider OpenSSL 1.1.1-pre6 [OP18b], although the components studied are largely stable across releases and remain similar to that of the early releases (version 0.9.6 of Sept. 2000).

Analysis. The primality tests in OpenSSL reside in the crypto library, which also houses a wide range of implementations of cryptographic algorithms. The services provided by the crypto library are used by the OpenSSL implementations of SSL, TLS and S/MIME, and have also been used to implement SSH, OpenPGP, and other cryptographic standards.

The functions called upon to perform primality testing in the OpenSSL BIGNUM library are `BN_is_prime_ex` and `BN_is_prime_fasttest_ex` found in `bn_prime.c`. The bulk of the primality testing algorithm is done in `BN_is_prime_fasttest_ex` where $t =$`checks` rounds of Miller-Rabin are performed, each with a randomly chosen base. The `checks` variable is provided as a parameter to the primality verification function. The function `BN_is_prime_ex` simply calls `BN_is_prime_fasttest_ex` without doing any trial divisions. The composites n that we produce have factors much larger than those in the trial divisions that OpenSSL performs. This means that, for our purposes, the result of calling either function is equivalent. Therefore we will focus only on `BN_is_prime_fasttest_ex`.

Number of Miller-Rabin rounds. Both primality testing functions allow the user to determine the rounds of Miller-Rabin performed. The documentation indicates that if the user sets the value of `checks` to the variable `BN_prime_checks`, then the number of Miller-Rabin iterations t is chosen such that the probability of a Miller-Rabin test declaring a *random* composite number n as prime is less than 2^{-80}. The number of rounds performed is then based on the bit-size b of the number n being tested. The relationship between these two values is shown in Table 2. The entries here are based on average case error estimates taken from the Handbook of Applied Cryptography [MVOV96], which in turn references [DLP93].

Base Selection. OpenSSL chooses the Miller-Rabin bases it uses in a pseudorandom manner, by using OpenSSL's function `BN_rand_range()` with an optional flag set to `PRIVATE`. This then calls `bnrand` to generate a pseudorandom base a in the range $1 \leq a < n$ using a cryptographically strong pseudorandom number generator with entropy inputs gathered from the operating system, cf. [Str16] for details on OpenSSL's random number generation.

Pseudoprimes. As mentioned in Section 1, the average case estimates from [DLP93] are designed only to be used on testing numbers during prime generation. Indeed, OpenSSL correctly applies primality testing as outlined above in this situation. However, we found nothing in the documentation to warn about the adversarial setting. Instead it appears to be left up to the user to decide how

Table 2. The rounds t of Miller-Rabin performed chosen by OpenSSL when testing b-bit integers with `checks = BN_prime_checks`.

b	t	b	t
$b \geq 1300$	2	$400 > b \geq 350$	8
$1300 > b \geq 850$	3	$350 > b \geq 300$	9
$850 > b \geq 650$	4	$300 > b \geq 250$	12
$650 > b \geq 550$	5	$250 > b \geq 200$	15
$550 > b \geq 450$	6	$200 > b \geq 150$	18
$450 > b \geq 400$	7	$150 > b$	27

many rounds of testing are needed, and if they set `checks = BN_prime_checks` then Table 2 would dictate how many rounds are applied. In this setting, we are able to undermine OpenSSL's guarantees by producing composite numbers using the methods described in Section 3.1.1. That is, we can easily construct numbers of the form $n = (2x + 1)(4x + 1)$ with x odd and $2x + 1, 4x + 1$ prime, and be sure that n will pass random-base Miller-Rabin tests with probability roughly $1/4$ per test. For example, for n having $b = 2048$ bits, OpenSSL will apply $t = 2$ tests, and we have a $1/16$ chance of our composite n deceiving OpenSSL.

4.2 GNU GMP

The GNU Multiple Precision Arithmetic Library [Gt18], GNU GMP or simply GMP, is a popular open source arbitrary precision integer library that is widely deployed in mathematical software packages. We consider the latest version GMP 6.1.2 throughout.

Analysis. GMP provides its own datatype to handle big integers known as `mpz_t`. GMP's primality test is implemented in `mpz_probab_prime_p(mpz_t n, int reps)`. On input n, this function performs some trial divisions, then a fixed-base Fermat test with base $210 = 2 \cdot 3 \cdot 5 \cdot 7$, and finally $t =$ `reps` rounds of Miller-Rabin; the latter is implemented in function `mpz_millerrabin`. The value of `reps` is selected by the caller. The documentation gives assurance that a composite number will be identified as being prime with a probability of less than $(1/4)^{\texttt{reps}}$ and states that "reasonable values of `reps` are between 15 and 50".

Base Selection. GMP uses a pseudorandom number generator (PRNG) to choose the base used for each Miller-Rabin test. The PRNG's state is initialised in the function `mpz_millerrabin` by calling `gmp_randinit_default(rstate)`, which uses the Mersenne Twister algorithm. This initial seed state is then used as a source of randomness in `mpz_urandomm(a, rstate, n)` to generate a uniform random integer base a between 2 and $n - 2$ inclusive.

While GMP offers to seed PRNGs and to explicitly pass them to functions requiring access to pseudorandom numbers, this option is not available for primality testing, i.e. each call to `mpz_millerrabin` will work with an identical PRNG state. Thus, since the initial seed state is constant, the resulting sequence of a values chosen by `mpz_urandomm` for a fixed n is also constant. Note, though, that different a may be chosen for different n, since the bases a are sampled uniformly in a range depending on n. This, in effect, means that the bases chosen when testing n are defined as a function of n. Therefore the result of `mpz_probab_prime_p(mpz_t n, int reps)` for fixed values of n and t is deterministic.[5]

Pseudoprimes. For integers n, t, let $(a_1, a_2, \ldots, a_t)$ denote the deterministic list of bases used by GMP, where $t =$ `reps`. By setting $n = (2x + 1)(4x + 1)$ with x odd and $2x + 1, 4x + 1$ both prime,

[5] We note that the same sequence of a_i may still be produced even for different n when n is only slightly smaller than a power of two. This is due to the application of rejection sampling by comparison with n to sample in a range up to n.

we will obtain a number for which random base MR tests will pass with probability roughly 1/4. Since $(a_1, a_2, \ldots, a_t)$ is pseudorandom, we may expect that an n constructed in this way would pass the MR tests in GMP with probability $(1/4)^t$. Thus, for example, for the minimum recommended value of $t = 15$, it might be feasible to construct a suitable n which would always be declared prime by just trying sufficiently many random values of x.

However, recall that we need $2x + 1$ and $4x + 1$ to be simultaneously prime, and we must also pass the base 210 Fermat test. This makes the cost of constructing n prohibitively high with this direct approach, since the probability that random x will give prime pairs $(2x + 1, 4x + 1)$ is approximately $(2/\ln x)^2$, and the special form of n means that a Fermat test will pass with probability roughly 1/2 (see Appendix C), while passing t rounds of MR testing will happen with probability only $(1/4)^t$. Putting this together, each x would pass with probability about $1/2^{2t-1}(\ln x)^2$; for a 99% chance of success in finding a good x with $\ln x = s$, we would need about $5 \cdot 2^{2t-1}s^2$ trials, each trial involving at least a primality test on $2x+1$. For a 1024-bit n and $t = 15$ trials (the minimum recommended by GMP), roughly 2^{47} trials would be needed, each involving at least a 512-bit primality test.

Instead, and partly inspired by the ROCA attack [NSS+17] and the form of the primes exploited there [JPV00,JP06], we consider x of the special form $x = kM + 189$ where M is a product of the first ℓ primes from in the set $\mathcal{P} = \{2, 3, \ldots, 373\}$ and k is a randomly chosen integer of a size to make $n = (2x + 1)(4x + 1)$ have a desired target size (say, 1024 bits). The selection of x of this form ensures that $2x + 1 = 2kM + 379$ and $4x + 1 = 4kM + 757$ are *not* divisible by the first ℓ primes in $\mathcal{P}$, boosting the chances that $2x + 1$ and $4x + 1$ are both prime (the form of x essentially ensures that $2x + 1, 4x + 1$ pass trial divisions for the first ℓ primes in $\mathcal{P}$; here we rely on the fact that 379 and 757 are both prime and larger than 373). The offset of 189 is specially chosen so that the Fermat test on n to base 210 will always pass for n of the chosen form. This follows from a bespoke mathematical analysis that is deferred to Appendix C.

Our code for constructing x (and n) of this special form first picks a target bit-size for n, then selects ℓ as large as possible so that there are enough choices for k for there to be sufficiently many candidates that one suitable x will result. For each resulting x, our code tests $2x + 1$ and then $4x + 1$ for primality, and (if these tests pass) applies the GMP primality test for the desired number of t rounds of MR testing.

For n of 1024 bits, we set $\ell = 69$, taking M as the product of the primes up to 349, and leaving a 51-bit value for k. The choice of M increases the probability that both of $2x + 1$ and $4x + 1$ are prime by a factor of roughly 25, and the form of x ensures that the Fermat test always passes, giving another factor of 2 improvement. Using a total of 33,885 core-hours (3.87 core-years) of computation in parallel on 872 cores running at 2.4GHz (kindly donated by CloudFlare), we found the following 1024-bit example passing GMP's primality test with $t = 15$ rounds of MR testing:

$$
\begin{aligned}
n = 2^{960} &\cdot \texttt{0x000000000000000000000000000000081d564fbdd20b406} \\
+ 2^{768} &\cdot \texttt{0x750af7bd334dcf547b131a1d8f8235fd603dba44e22e0775} \\
+ 2^{576} &\cdot \texttt{0x0ecf755051d33cb8895413f5d69f5a3df701889e3a69f92e} \\
+ 2^{384} &\cdot \texttt{0xdd3f5f36662521877231ba4753a3e7185a89ddb0b2d73a35} \\
+ 2^{192} &\cdot \texttt{0x9e976a9bcfeae1a7c026d74bc7515a5010f3cd62c69fa9ad} \\
+ 2^{0} &\cdot \texttt{0x7b699f40e7a85192e1a4aa95537363fcb93d789aee32bbbf}.
\end{aligned}
$$

We recall that this n will always pass GMP's primality testing for 15 MR rounds because the generation of the MR bases depends deterministically on n.

4.3 Mini-GMP

Mini-GMP is a small implementation of a subset of GMP's `mpn` and `mpz` interfaces included within GMP 6.1.2 [Gt18]. This library includes its own miniature implementation of `mpz_probab_prime_p(n, reps)`. The most significant change compared to GMP is that Miller-Rabin is performed explicitly with a deterministic sequence of t bases obtained by evaluating Euler's polynomial $a(x) = x^2 + x + 41$ at $x = 0, 1, 2, \ldots, t - 1$. It also omits GMP's Fermat test.

Pseudoprimes. The use of a sequence of deterministic bases in Mini-GMP enables us to predict the bases that will be chosen for any particular value t of `reps`. The bases are not all prime (though Euler's polynomial famously does produce many primes), so we cannot directly use Arnault's method from Section 3.1.2. Instead, we use our extension for composite, fixed bases method in Section 3.1.4.

Using this approach, we constructed a 2960-bit composite $n = p_1p_2p_3$ that passes up to $t = 101$ rounds of Mini-GMP's Miller-Rabin testing. Of the 101 bases produced by Euler's polynomial, 86 were already primes and the remaining 15 bases all factorised into various combinations of the four primes $163, 167, 179$ and 199. The combined list of 90 unique primes was then used with the method described in Section 3.1.4 to produce n. This n is given in Appendix D. We note that the documentation for Mini-GMP is shared with the main GMP library, implying to a user that 15 to 50 rounds of MR testing would be reasonable.

4.4 Java

Java implementations provide their own methods for arbitrary precision arithmetic, including primality tests, as seen in `java.math.BigInteger`. We consider OpenJDK10 [Cor18], although there seems to be no significant changes to this section of the code in older versions such as JDK8.

Analysis. The primality testing function `isProbablePrime` is passed a single parameter `certainty`. This is a value chosen by the user and is described in the documentation as: "a measure of the uncertainty that the caller is willing to tolerate: if the call returns true the probability that this BigInteger is prime *exceeds* $(1-1/2^{\texttt{certainty}})$." The `certainty` parameter is then used to determine how many rounds of testing will be performed. This is done by calling the function `primeToCertainty` which is shown in Appendix E. This function first sets a variable `n` as $(\texttt{certainty}+1)/2$. This would produce a non-integer result when `certainty` is even, yet the result is cast to an integer, implicitly flooring the result.[6]

This function also takes into consideration the bit-size of the number being tested; if it is less than 100, then Miller-Rabin is performed with at most 50 rounds; if it is greater than 100, then both Miller-Rabin and a Lucas probable prime test with Selfridge's parameters are performed, as described in Section 3.2. In the latter case, the maximum number of rounds of Miller-Rabin is determined based on the bit-size of the tested number, similarly to OpenSSL. In both cases, the user's choice of `certainty` will determine the actual number of rounds of Miller-Rabin performed only if it is *less* than the internally-specified number for that bit-size.

Pseudoprimes. For numbers of cryptographically interesting size, Java performs both Miller-Rabin and Lucas probable prime tests. Using the method outlined in Section 3.2 we could produce composites that are guaranteed to be declared prime by the Lucas test. However, the resulting forms do not fit into any of the known families of composites having high numbers of Miller-Rabin non-witnesses. Hence, we are unable to construct any numbers passing Java's primality test with high probability using our current techniques.

4.5 JavaScript Big Number (JSBN)

The Java Script Big Number (JSBN) library written by Tom Wu [Wu17] provides a small cryptographic toolkit for Java Script applications. Here we study the most recent release JSBN 1.4 from 2013. According to its homepage the library has been used in a variety of applications, including: Forge (a pure JavaScript implementation of SSL/TLS), Google's V8 benchmark suite version 6, the JavaScript Cryptography Toolkit and the RSA-Sign JavaScript library.

[6] Because of the role that `n` plays in determining the number of rounds of Miller-Rabin to be performed, the result is that there is no difference in testing `isProbablePrime(k)` and `isProbablePrime(k+1)` when k is odd. This has an effect on the assurance given to the user — the guarantee of $1 - 1/2^{\texttt{certainty}}$ is no longer accurate for half of the values of `certainty`.

Analysis. The library offers the primality test `bnIsProbablePrime(t)` where the parameter `t` defines the number of rounds of Miller-Rabin the user wishes to perform. The code documentation states that this function will "test primality with certainty $\geq 1 - .5^t$". The function pseudorandomly chooses a base `a` for each round of Miller-Rabin from a hard-coded list of all primes below 1000 called `lowprimes`.

Pseudoprimes. We can consider this implementation as performing tests with fixed bases, where the bases chosen are all primes between 2 and 1000. We can then use Arnault's method (Section 3.1.2) to construct composite numbers n that pass JSBN's primality test no matter how many rounds of testing t the user wishes to perform. For example, we used SageMath 7.6 [S+17] to obtain a 4279-bit composite n having 3 prime factors, see Appendix F for the details.

4.6 Libgcrypt

Libgcrypt [Koc18] is a general purpose cryptographic library originally based on code from GnuPG. The library provides various cryptographic functions, including public key algorithms, large integer functions and primality testing. We analyse the current stable version 1.8.2, released in December 2017

Analysis. The documentation for Libgcrypt states that the function used for checking the primality of primes is `gcry_prime_check` which is found in `primegen.c`. This function then calls `check_prime` in which the actual testing performed. This function `check_prime` performs three testing steps. The first step is trial division with all primes up to 4999. The second step is a Fermat test with base $a = 2$. The last step comprises t rounds of Miller-Rabin where the bases are pseudorandomly chosen. We note that t is user defined, but cannot be set to less than 5. The default for checking the numbers produced in the prime generation algorithm is set to 5, but when a user calls `gcry_prime_check` the choice of t is hard-coded to 64.

Pseudoprimes. Following Section 3.1, beating steps 1 and 2 of the testing performed in `check_prime` is trivial if we choose n as a Carmichael number of the form $n = pqr$ where $p, q, r > 4999$. By using the hybrid technique in Section 3.1.3, we can create a Carmichael number that also has the maximum number of randomly distributed non-witnesses. We then need only to overcome the t Miller-Rabin tests with pseudorandom bases. This happens with probability $(1/4)^t$. If the user calls `gcry_prime_check` then the probability with which we can fool this test would be only 2^{-128}. Yet performing 64 rounds of Miller-Rabin is quite time consuming, and a user may be tempted to bypass `gcry_prime_check` and call `check_prime` with fewer rounds. In this hypothetical situation, or in versions of Libgcrypt prior to 1.3.0 (2007) [Koc05] (where `gcry_prime_check` would call $t = 5$ rounds by default) the best we could achieve is passing the test with probability 1/1024 (for $t = 5$).

4.7 Cryptlib

Cryptlib 3.4.3 [Gut18] is an open source security toolkit library developed by Peter Gutmann. It provides a variety of services including: public key algorithms, various cryptographic functions and primality testing.

Analysis. The primalty test in Cryptlib is the function `primeProbable` found in `kg_prime.c` and is composed of t rounds of Miller-Rabin, where the value of t must be between 1 and 100 (inclusive) and is chosen by the user upon calling. The function then chooses the base for each test incrementally from the start of a fixed list of primes. This is either a list of the first 54 primes (2 to 251) or the first 2048 primes (2 to 17863), depending on the preprocessor directive `CONFIG_CONSERVE_MEMORY`.

Pseudoprimes. Since $t \leq 100$, we will at most only ever test using the primes between 2 and 541 (the hundredth prime) as bases. We can therefore generate numbers that are guaranteed to be declared prime by this test for any valid input t, simply by using Arnault's method to generate a composite n that has the first 100 primes as non-witnesses. Indeed, using the method described in Section 3.1.2 we can produce a 2329-bit composite that is pseudoprime to all prime bases up to and including 541. See Appendix G for details.

4.8 LibTomMath

LibTomMath v1.0.1 [Den18b] is an open source multiple-precision integer library with a number theoretic toolkit.

Analysis. LibTomMath includes several methods for primality testing in the form of trial division, Fermat tests, and Miller-Rabin tests. The latter two take a single base a and a number n to test as arguments and return whether or not a is a witness or non-witness. The main primality test is defined by the function `mp_prime_is_prime`, which takes arguments n (the number to be tested), and integer t with $1 \leq t \leq 256$. It then performs some trial divisions (on a default of the first 256 primes) and then t rounds of Miller-Rabin. The selection of bases to be used is made similarly as in Cryptlib: it simply picks incrementally from a list of hard-coded primes (but this time a list of 256 primes up to 1619 are used).

The documentation of LibTomMath (`bn.pdf`) discusses the number of rounds of Miller-Rabin required with the statement: *"Generally to ensure a number is very likely to be prime you have to perform the Miller-Rabin with at least a half-dozen or so unique bases."* This is complemented with a function `mp_prime_rabin_miller_trials` that gives the number of rounds needed to achieve an error rate less than 2^{-96} based on the bit-size of the number tested (similar to that in OpenSSL and [DLP93]) and a comment in the header file `tommath.h` above `mp_prime_rabin_miller_trials` that states the probability of a false classification is no more than $(1/4)^t$.

Pseudoprimes. Since the bases are chosen deterministically based on the value of t, we can achieve a failure rate of 100% simply by using the method of Section 3.1.2 to produce a composite n that has the first 256 primes as non-witnesses; such an n is guaranteed to be declared prime by `mp_prime_is_prime`, for any value of t (including the t chosen by `mp_prime_rabin_miller_trials` that describes an error rate less than 2^{-96}). Appendix H provides a 7023-bit example of such an n. Much smaller examples can be obtained if smaller values of t are guaranteed to be used; in particular, we can easily obtain a 1024-bit example for $t \leq 40$ (see also Appendix H).

4.9 LibTomCrypt

LibTomCrypt v1.18.1 [Den18a] is an additional cryptographic toolkit that shares many resources with LibTomMath.

Analysis. The primality test in LibTomCrypt is called as `isprime(n,t,result)`. It takes as arguments an n to test and carries out t rounds of Miller-Rabin. The documentation of LibTomCrypt advises that each round of Miller-Rabin reduces the probability of n being a pseudoprime by a factor of 4, and therefore deduces that the overall error is at most $(1/4)^t$. LibTomCrypt supports selection from three different big integer libraries at runtime.

If LibTomMath is chosen then `isprime` will call `mp_prime_is_prime` as described in Section 4.8, passing on parameters n and t. If TomsFastMath [Den18c] is chosen then `isprime` will call `fp_isprime_ex`, a function defined in the math library TomsFastMath that performs equivalent testing as LibTomMath's `mp_prime_is_prime`. If GMP is selected then `isprime` will call `mpz_probab_prime_p` as described in Section 4.2. The value of t used by any of the three choices is inherited from the original call to `isprime`, however if $t = 0$ the value is overwritten to $t = 40$.

Pseudoprimes. If either LibTomMath or TomsFastMath are selected, the pseudoprimes described in Section 4.8 (see Appendix H) will always be declared prime by the primality test. If GMP is selected we can apply the analysis in Section 4.2 to generate pseudoprimes (see Appendix C).

4.10 WolfSSL

WolfSSL 3.13.0 [Wol18b] (formerly CyaSSL) is a small SSL/TLS library targeted for use in embedded systems. WolfSSL provides primality testing tools based on public domain TomsFastMath 0.10 [Den18c] and LibTomMath 0.38 [Den18b] functions.

Analysis. The primality test in WolfSSL is the function `mp_prime_is_prime` which takes a number n to be tested and the rounds of testing t as parameters. This function is directly taken from an older version of LibTomMath, namely 0.38 [Den18b]. WolfSSL will use LibTomMath by default, but can optionally be compiled to use TomsFastMath 0.10 [Den18c] at runtime. The primality test in LibTomMath 0.38 is unchanged from that analysed in version 1.0.1 in Section 4.8. When using TomsFastMath, `mp_prime_is_prime` calls `fp_isprime` which strips the user's choice of t and simply calls `fp_isprime_ex` with the hard-coded value of $t = 8$. The function `fp_isprime_ex` then performs trial division (on a default of the first 256 primes) and then does 8 rounds of Miller-Rabin using the first 8 primes as bases. It thus acts equivalently to `mp_prime_is_prime` in LibTomMath, but with $t = 8$.

Pseudoprimes. Since the testing in WolfSSL is in effect the same as that performed in LibTomMath (but using only 8 rounds of Miller-Rabin when using TomsFastMath), the composite examples given in Appendix H are also declared prime with 100% success.

4.11 Bouncy Castle

Bouncy Castle is a cryptographic library written in Java and C# [otBCI18]. The primality test in Bouncy Castle Java is based on the `BigInteger` class from JDK as described in Section 4.4. Bouncy Castle C# implements its own primality tests. We analyse Bouncy Castle C# version 1.8.2.

Analysis. The relevant function responsible for primality tests is located in the class `BigInteger`. This class provides method `IsProbablePrime` which accepts `certainty` as a parameter. The method then uses Miller-Rabin tests with t rounds, where t is computed as $t = ((\texttt{certainty} - 1)/2) + 1$. In each round the base is selected using a secure random number generator (`SecureRandom`) which is provided by the Bouncy Castle library.

The `certainty` parameter must always be provided to invocation of the `IsProbablePrime` method. Therefore, a user choice completely determines how many Miller-Rabin rounds are performed. For example, this method is directly used in the `TlsDHUtilities` class, which provides Diffie-Hellman operations for TLS. When validating the incoming DH parameters, the `ValidateDHParameters` method invokes `isProbablePrime` with `certainty` = 2. This results in only a single Miller-Rabin test being carried out.

Pseudoprimes. We can produce composites n using any of the methods in Section 3.1; such n meet the Monier-Rabin bound and so will pass Bouncy Castle's primality testing with probability $(1/4)^t$ with t as derived from `certainty`. Although there is no formal documentation, a comment above the primality testing code indicates that the failure rate of this testing function should be $(1/2)^{\texttt{certainty}}$, and so the user's choice of `certainty` *is* achieved.

4.12 Botan

Botan is a cryptographic library written in C++11 [Llo18a]. In addition to the crypto functionality it offers a TLS client and server implementation. We analyse Botan 2.6.0.

Analysis. The relevant primality test implementation can be found in `numthry.cpp`, which contains function `is_prime`. This function first evaluates whether a tested number is divisible by small primes up to 65521. It then performs Miller-Rabin primality tests with randomly chosen bases. The source of randomness and the number of Miller-Rabin rounds are based on parameters passed to the `is_prime` function. The number of rounds is computed based on parameter `prob` and t is set as $(\texttt{prob}+2)/2$. Botan's documentation is very clear on the distinction between testing numbers of random and possibly adversarial origin. To distinguish the source, the function `is_prime` contains a boolean flag `is_random`. If set, then the code uses [DLP93] to assign t based on the bit-size of the number being tested, with a target failure rate less than 2^{-80}.

Pseudoprimes. As with Bouncy Castle, we can produce composite n using any of the methods in Section 3.1; such n meet the Monier-Rabin bound and will pass Botan's primality test with the highest probability of $(1/4)^t$ where t is from the user's choice of `prob` via $t = (\texttt{prob}+2)/2$. In this sense, the test's guarantees match the user's expectations.

4.13 Crypto++

Crypto++ 7.0 is an open source C++ cryptography library originally written by Wei Dai [Dai18]. Crypto++ has a variety of primality testing algorithms in `nbtheory.cpp`. These consist of trial division, Fermat, Miller-Rabin and both strong and standard Lucas probable prime tests. Crypto++'s primality testing function `isprime` is performing both Miller-Rabin and strong Lucas tests. Thus, to fool it, we would need to find Baillie-PSW pseudoprimes (though the Miller-Rabin test is a random base test, unlike that performed in Baillie-PSW). We do not currently know any such pseudoprimes.

4.14 GoLang

The Go programming language (GoLang) 1.10.3 [Goo18] created at Google in 2009 is an open source project including arbitrary-precision arithmetic and cryptographic functionality.

Analysis The relevant primality test implementation can be found in `int.go`, which contains function `ProbablyPrime(t)`. The parameter `t` defines the number of rounds of Miller-Rabin the user wishes to perform. The function first performs trial division with a series of small primes, then `t` rounds of Miller-Rabin (where one base is forced to be 2 and all other bases are chosen pseudorandomly), and finally a Lucas probable prime test. Therefore the function is performing a Baillie-PSW test. Before version 1.8, Go's `ProbablyPrime(t)` function applied only the Miller-Rabin tests. The documentation provided by GoLang makes it clear that the probability of the function declaring a randomly chosen composite input to be prime is at most $(1/4)^t$. It also states that "`ProbablyPrime(t)` is not suitable for judging primes that an adversary may have crafted to fool the test".

From an attack perspective it is interesting that the pseudorandom number generator used in this primality test is seeded with the tested number n. Thus, an attacker can reliably predict the pseudorandomly generated Miller-Rabin bases.

Pseudoprimes Since a Baillie-PSW test is being performed, we know of no composites that are incorrectly declared prime by GoLang. However, for versions prior to 1.8 released in 2017, we are able to exploit the insecure nature of the Miller-Rabin base selection to produce composite numbers that are guaranteed to be declared prime with respect to a parameter t. Since this is the same method GNU GMP uses to choose bases for Miller-Rabin, we can use the method described in Section 4.2 to produce such composites. We give an example of a composite n that is always declared prime for $t \leq 13$ in Appendix I.

4.15 Mathematics Software Packages

We have also examined primality tests found in popular mathematics software packages and computer algebra systems, namely: Magma, Maple, Maxima, SageMath, SymPy and Wolfram Mathematica. We include these in our analysis since they might be relied upon by developers when manually checking values in standards or code. Some of the libraries use deterministic tests for proving primality, though most still rely on probabilistic methods when testing candidates larger than 64 bits in size. Maple, Maxima and SymPy have dependencies on GMP and therefore inherit the same issues with its primality test as discussed in Section 4.2; however they all also perform Lucas tests in their latest versions, so this "cross contamination" does not result in exploitable weaknesses. Full details are provided in Appendix K.

5 Application to Diffie-Hellman

Validating the correctness of Diffie-Hellman (DH) parameters is a vital step for verifying the integrity of the key exchange. As mentioned in the introduction, since the DH parameter set (p, q, g), with $g \in \mathbb{Z}_p$ generating a group of order q, is public, they can originate from third-party sources such as a server or a standard. An adept DH parameter validation function should check that p, q are both prime and that $p = kq + 1$ for some integer k. It would also test that the given generator g generates the subgroup of order q and that any received DH values lie in the correct subgroup. A common choice is to set $k = 2$, and thus p is a safe-prime. For p that are not safe primes, the group order q can be much smaller than p, offering performance improvements. The security level is then based upon the bit-size of q, which must still be large enough to thwart the Pohlig-Hellman algorithm for solving the Discrete Logarithm Problem (DLP), which for prime q runs in time $O(\sqrt{q})$. A common parameter choice is a 160-bit q with a 1024-bit p or a 256-bit q with a 2048-bit p.

More precisely, the Pohlig-Hellman algorithm runs in time $O(\sqrt{t})$ where t is the largest prime factor of q. Thus, an attacker armed with the ability to fool a primality test can supply a sufficiently smooth composite q such that $p = kq+1$ is still prime. For example, if q is of the form $(2x+1)(4x+1)$ this leads to an attack on DLP with complexity 2^{40} resp. 2^{64} for the sizes mentioned above.

We stress, though, that none of the constructions for malicious composites in this work pose a risk to protocols such as Telegram that insist on $k = 2$, i.e. which check both $q = (p-1)/2$ and p for compositeness. For example, the construction of Section 3.1.1 would set $q = (2x + 1)(4x + 1)$ and yield p that is always divisible by 3; moreover q would not be smooth enough for Pohlig-Hellman to pose a threat for parameters of cryptographically appropriate size. It is an interesting open question to find a large, sufficiently smooth composite q passing a primality test with high probability such that $p = 2q + 1$ is prime or passes a primality test, too.

We now discuss DH verification functions in various libraries. For each library, we apply the analysis from Section 4 to check how robust these libraries are to attack. We note that the other libraries discussed in Section 4 do not implement a higher-level function for verification of DH parameters. Of course, this does not prevent an application from using these libraries to realise its own verification function. Such an application would inherit the weaknesses and strengths of the underlying library (when $k \neq 2$ is permitted). We give an example of this scenario for the GMP library below. We close with a discussion of the important use case of SSL/TLS.

OpenSSL The file `dh_check.c` contains the functions `DH_check_params` and `DH_check`. The former is a lightweight check that just confirms that p and g are 'likely enough' to be valid, by testing to see if p is odd and $1 < g < p - 1$. The latter function is more thorough and calls `BN_is_prime_ex` to test the primality of both p and $q = (p - 1)/2$. These primality tests are called with `checks = BN_prime_checks`, therefore the rounds of Miller-Rabin are determined by Table 2. This means for example that they will declare as prime with probability $1/16$ composites n of the special form $n = (2x + 1)(4x + 1)$, for x odd and $2x + 1, 4x + 1$ prime, when n has more than 1300 bits. Since no private data is required, this testing function's most likely use-case is checking Diffie-Hellman

parameters that have been generated by someone else (perhaps from an untrusted server or an unknown origin) and therefore clearly misuses OpenSSL's own primality testing functions.

However, since OpenSSL restricts parameter sets (p, q, g) to safe-primes p, efficient attacks are not feasible. Using our current techniques, we *cannot* generate a set that will, with high probability, pass primality testing on both p and q simultaneously and allow efficient solving of the DLP.

Bouncy Castle The validation of DH parameters in `ValidateDHParameters` extracts p, g from a DH parameter set and then only checks the primality of p with 1 round of Miller-Rabin. We can therefore produce composites that are accepted as DH moduli with probability $1/4$. More seriously, q is not given to the check function, so even with a prime p, the value of g can be chosen so that it has small order, making Pohlig-Hellman as easy as desired. Even if g had large prime order, the flexibility in choosing parameters would allow Lim-Lee small subgroup attacks, as explored in [VAS+17].

Botan The Botan function `is_prime` is used in the class `DL_Group` (located in `dl_group.cpp`) which is also used for verifying DH parameters. This class contains the `verify_group` function, which can be invoked with boolean parameter `strong`. If `strong` is set to `true`, the `is_prime` function is invoked with `prob=128`. This results in $t = 65$ Miller-Rabin computations. Otherwise, `prob=10` and 6 Miller-Rabin computations are performed. This test is performed for both p and q; the code also checks that $q|(p-1)$ but does not insist on p being a safe prime.

Using the methods described in Section 3.1 we can find a q of 160-bits that passes 6 rounds of MR testing with probability $1/4096$ such that q has 2 or 3 prime factors. Then we can construct 1024-bit prime p as $p = kq + 1$ by using the flexibility in k, and a g that generates the subgroup of size q. Since this p is indeed prime and $q|(p-1)$, all of Botan's tests on the parameter set (p, q, g) will pass with probability $1/4096$ if `strong` is set to `false`. We can subsequently use the Pohlig-Hellman algorithm to solve the DLP in the subgroup generated by g and break DH with about 2^{28} effort. See Appendix J for an example of such a parameter set.

GNU GMP The 256-bit integer $q = (2x + 1)(4x + 1)$ with

$$x = \texttt{0x400286bac15132db85b1c936709f369b}$$

passes 15 rounds of GMP's primality test `mpz_is_probab_prime_p`; picking $k = 2^{1792} + 1254$ produce the 2048 bit prime $p = kq + 1$. The resulting parameter set (p, q, g) would pass even fully adept DH validation with certainty if the underlying primality testing was based on GNU GMP's code with the minimum recommended number of rounds of Miller-Rabin.

SSL/TLS We close by commenting on the situation for DH parameter testing in SSL/TLS. Here, the server chooses parameters but only sends (p, g) to the client. There is no requirement that p be a safe prime. This makes it difficult for clients to validate the DH parameters (they would need to factor $p - 1$ and then try different divisors to determine the order of q) or to perform group membership tests on received DH values. Consequently most clients perform only simple sanity checks, e.g. checking that $g \notin \{0, \pm 1\}$. This makes SSL/TLS vulnerable to a variety of malicious DH parameter attacks, cf. [Won16,VAS+17], and in view of these, exhibiting composite primes p that fool primality tests would be overkill for the SSL/TLS standards in their present form. However, our work shows that even if clients tried to validate DH parameters by factoring $p - 1$, finding the order of g and then testing it for primality, they could still fall foul of malicious DH parameters. And if the SSL/TLS protocol were amended so that the server provides full DH parameters, careful checks would still be needed. Finally we note that only a small number of fixed, safe prime DH parameter sets are permitted in TLS 1.3. These were recently standardised in RFC 7919 [Gil16], alleviating these issues for future versions of the protocol.

6 Conclusion and Recommendations

Our work has explored primality testing in the adversarial setting and its impact for Diffie-Hellman parameter testing. Our main finding is that leading libraries are not designed for this setting, and therefore often vulnerable to accepting as prime composite inputs that are maliciously chosen, see Table 1.

The need for careful distinction between non-adversarial (or random) and adversarial primality testing is of course well understood in the cryptographic research community. However, this distinction is not necessarily reflected and implemented in cryptographic libraries and their documentation. As such, we can generally classify the underlying cause of the failure in prime classification accuracy as non-consideration of the adversarial setting. More explicitly, we can categorise most failures in terms of how the bases for Miller-Rabin are chosen, i.e. fixed base, predictable bases, insufficient number of bases. Mini-GMP, JSBN, Cryptlib, LibTomMath, LibTomCrypt and WolfSSL all fail due to the selection of bases from a fixed list, whereas GNU GMP and GoLang pre 1.8 both suffer from predictable bases. OpenSSL, Libgcrypt, Botan and Bouncy Castle C# all have options to run as many rounds of Miller-Rabin as the user desires, but either default to, or call the test (elsewhere in the library) with too few rounds.

Based on our analysis, we make the following recommendations:

- In the absence of known pseudoprimes, we recommend that libraries switch to using the Baillie-PSW primality test wherever possible. The negative impact on performance is moderate, and the positive impact on security is significant. An existing benchmark for such a trade-off is found in the documentation of the computer algebra system PARI/GP [The18b] (on which Sage bases its primality testing functions). PARI/GP implements both a Miller-Rabin test with user-defined t and a Baillie-PSW test and indicate [The18a] that their Baillie-PSW test is about as fast as their Miller-Rabin test with $t = 3$.
- Libraries that wish to continue to use Miller-Rabin only (for example, to maintain a small codebase) should use pseudorandom bases, cf. Cryptlib, LibTomCrypt, JavaScript Big Number, WolfSSL. In particular, the bases should not depend only on n, cf. GNU GMP.
- We also recommend to default to worst-case bounds when picking the number of iterations and only assume average-case behaviour when explicitly instructed to by the user. This may require changes to interfaces to primality testing code.
- Designers of new protocols should avoid the pitfalls made in SSL/TLS, where DH parameter validation is made impractical for clients. TLS 1.3 does so by fixing and requiring use of a small collection of parameter sets.

Definitions in the cryptographic literature routinely start with "Let p be a prime ..." whereas our work highlights that many implementations do not necessarily provide strong guarantees for this assumption to hold. It is thus an interesting open question which other seemingly innocuous assumptions concerning domain parameters in the literature can be undermined in a similar fashion.

Acknowledgements

Albrecht was supported by EPSRC grant EP/P009417/1. Massimo was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/K035584/1). Paterson was supported by EPSRC grants EP/M013472/1, EP/K035584/1, and EP/P009301/1. Somorovsky was supported by the Horizon 2020 program under project number 700542 (FutureTrust).

We thank Christian Elsholtz for initial guidance on the mathematical literature and Ian Miers for assistance with our analysis of OpenSSL. We are grateful to Brendan McMillion and Nick Sullivan from Cloudflare for their generous provision of computing resources which enabled us to find the examples in Section 4.2.

References

AKS04. Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of mathematics*, pages 781–793, 2004.

AM93. A Oliver L Atkin and François Morain. Elliptic curves and primality proving. *Mathematics of computation*, 61(203):29–68, 1993.

Arn95. François Arnault. Constructing Carmichael numbers which are strong pseudoprimes to several bases. *Journal of Symbolic Computation*, 20(2):151–161, 1995.

Arn97. François Arnault. The Rabin-Monier theorem for Lucas pseudoprimes. *Mathematics of Computation of the American Mathematical Society*, 66(218):869–881, 1997.

Bai13a. Robert Baillie. OEIS A217120: Lucas pseudoprimes. `https://oeis.org/A217120`, March 2013.

Bai13b. Robert Baillie. OEIS A217255: Strong Lucas pseudoprimes. `https://oeis.org/A217255`, March 2013.

BCP97. Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. *J. Symbolic Comput.*, 24, 1997. Computational algebra and number theory (London, 1993).

Ble05. Daniel Bleichenbacher. Breaking a cryptographic protocol with pseudoprimes. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 9–15. Springer, Heidelberg, January 2005.

BW80. Robert Baillie and Samuel S Wagstaff. Lucas pseudoprimes. *Mathematics of Computation*, 35(152):1391–1417, 1980.

CMG+16. Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the juniper dual EC incident. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 468–479. ACM Press, October 2016.

CNE+14. Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of dual EC in TLS implementations. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 319–335, San Diego, CA, 2014. USENIX Association.

Cor18. Oracle Corporation. *OpenJDK 10 Open Java Development Kit*, 2018. `openjdk.java.net`.

CP06. Richard Crandall and Carl Pomerance. *Prime numbers: a computational perspective*, volume 182. Springer Science & Business Media, 2006. pp.136-140.

Dai18. Wei Dai. Crypto++. `https://www.cryptopp.com/`, April 2018.

Den18a. Tom St Denis. LibTomCrypt. `http://www.libtom.net/LibTomCrypt/`, April 2018.

Den18b. Tom St Denis. LibTomMath. `http://www.libtom.net/LibTomMath/`, April 2018.

Den18c. Tom St Denis. TomsFastMath. `http://www.libtom.net/TomsFastMath/`, April 2018.

DLP93. Ivan Damgård, Peter Landrock, and Carl Pomerance. Average case error estimates for the strong probable prime test. *Mathematics of Computation*, 61(203):177–194, 1993.

DR08. T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919.

FGHT17. Joshua Fried, Pierrick Gaudry, Nadia Heninger, and Emmanuel Thomé. A kilobit hidden SNFS discrete logarithm computation. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 202–231. Springer, Heidelberg, April / May 2017.

fSidI17. Bundesamt für Sicherheit in der Informationstechnik. BSI TR-02102-1 Cryptographic Mechanisms: Recommendations and Key Lengths. Technical guideline, Federal Office for Information Security, January 2017.

Gil13. Jeff Gilchrist. Pseudoprime enumeration with probabilistic primality tests. `http://gilchrist.ca/jeff/factoring/pseudoprimes.html`, August 2013.

Gil16. D. Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919 (Proposed Standard), August 2016.

Goo18. Google. The Go Programming Language. `https://golang.org`, July 2018.

Gt18. Torbjorn Granlund and the GMP development team. GNU MP: The GNU Multiple Precision Arithmetic Library. `https://gmplib.org`, April 2018.

Gut18. Peter Gutmann. CryptLib. `http://www.cryptlib.com/`, April 2018.

Hö16. Andreas Höglund. MPZ_SPSP's under GMP 5.0.1. http://www.hoegge.dk/gmp/gmp501.htm, 2016. Last accessed 2016-10-31.

Jac15. Dana Jacobsen. Pseudoprime Statistics, Tables, and Data. http://ntheory.org/pseudoprimes.html, August 2015.

Jae93. Gerhard Jaeschke. On strong pseudoprimes to several bases. *Mathematics of Computation*, 61(204):915–926, 1993.

JP06. Marc Joye and Pascal Paillier. Fast generation of prime numbers on portable devices: An update. In Louis Goubin and Mitsuru Matsui, editors, *CHES 2006*, volume 4249 of *LNCS*, pages 160–173. Springer, Heidelberg, October 2006.

JPV00. Marc Joye, Pascal Paillier, and Serge Vaudenay. Efficient generation of prime numbers. In Çetin Kaya Koç and Christof Paar, editors, *CHES 2000*, volume 1965 of *LNCS*, pages 340–354. Springer, Heidelberg, August 2000.

Koc05. Werner Koch. Github - libgcrypt changes to default primality test. https://github.com/gpg/libgcrypt/commit/78a84338cb36748f17cc444b17ab7033ce384c34#diff-96a06fc4d0080caec00d423ca08a6c86, April 2005.

Koc18. Werner Koch. Libgcrypt. https://gnupg.org/software/libgcrypt/index.html, April 2018.

Lib18. LibTomMath. Pull request - added Fips 186.4 compliance, an additional strong Lucas-Selfridge (for BPSW). https://github.com/libtom/libtommath/pull/113, August 2018.

Lit09. Dwayne C. Litzenberger. PyCrypto 2.1.0. https://pypi.python.org/pypi/pycrypto/2.1.0, December 2009.

LK08. M. Lepinski and S. Kent. Additional Diffie-Hellman Groups for Use with IETF Standards. RFC 5114 (Informational), January 2008.

LLC18. Telegram FZ LLC. Telegram Messenger. https://telegram.org, 2018.

Llo18a. Jack Lloyd. Botan. https://github.com/randombit/botan, August 2018.

Llo18b. Jack Lloyd. Botan pull request - add Lucas test from FIPS 186-4. https://github.com/randombit/botan/pull/1636, August 2018.

Mac17. Macsyma group. *Maxima 5.41.0, Documentation*, 2017. Available at http://maxima.sourceforge.net/docs/manual/maxima_10.html.

Mac18. Macsyma group. *Maxima 5.41.0, a Computer Algebra System*, 2018. Available at http://maxima.sourceforge.net/index.html.

Mar16. Marcel Martin. *PRIMO–Primality Proving*, 2016. https://www.ellipsa.eu.

McC97. Jud McCranie. OEIS A014233: Smallest odd number for which Miller-Rabin primality test on bases less than or equal to the n-th prime does not reveal compositeness. https://oeis.org/A014233, Feb 1997.

Mil75. Gary L Miller. Riemann's hypothesis and tests for primality. In *Proceedings of seventh annual ACM symposium on Theory of computing*, pages 234–239. ACM, 1975.

Mon80. Louis Monier. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoretical Computer Science*, 12(1):97–108, 1980.

MVOV96. Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

Nar14. Shyam Narayanan. *Improving the Speed and Accuracy of the Miller-Rabin Primality Test*. MIT PRIMES-USA, 2014. https://math.mit.edu/research/highschool/primes/materials/2014/Narayanan.pdf.

Nic16. Thomas Nicely. GNU GMP mpz_probab_prime_p pseudoprimes. http://www.trnicely.net/misc/mpzspsp.html, 2016. Last accessed 2016-10-31.

NSS+17. Matús Nemec, Marek Sýs, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of coppersmith's attack: Practical factorization of widely used RSA moduli. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 1631–1648. ACM Press, October / November 2017.

OP18a. GitHub The OpenSSL Project. Pull request - Increase number of MR tests for RSA prime generation #6075. https://github.com/openssl/openssl/pull/6075, August 2018.

OP18b. The OpenSSL Project. OpenSSL: The Open Source toolkit for SSL/TLS. www.openssl.org, May 2018.

otBCI18. Legion of the Bouncy Castle Inc. *The Bouncy Castle Crypto Package For C Sharp*, 2018. https://github.com/bcgit/bc-csharp.

Pom84. Carl Pomerance. Are there counter-examples to the Baillie-PSW primality test. Dopo Le Parole aangeboden aan Dr. A. K. Lenstra., 1984.

PSW80. Carl Pomerance, John L Selfridge, and Samuel S Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026, 1980.

Pub13. Federal Information Processing Standards Publication. FIPS PUB 186-4 Digital Signature Standard (DSS). Standard, National Institute of Standards and Technology, Gaithersburg, MD, July 2013.

Rab80. Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of number theory*, 12(1):128–138, 1980.
RI18. Wolfram Research, Inc. Mathematica, Version 11.3, 2018. Champaign, IL, 2018.
Rie16. Gerhard Rieger. *Socat security advisory 7 - Openwall oss-security mailing list*, 2016. `http://www.openwall.com/lists/oss-security/2016/02/01/4`.
S+17. William Stein et al. *Sage Mathematics Software Version 8.2*. The Sage Development Team, 2017. Available at `http://www.sagemath.org`.
Sta05. British Standards. ISO/IEC 18032:2005 Information technology – Security techniques – Prime number generation. Standard, International Organization for Standardization, January 2005.
Str16. Falko Strenzke. An analysis of OpenSSL's random number generator. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 644–669. Springer, Heidelberg, May 2016.
Sym17a. SymPy. SymPy GitHub repository. Available at `https://github.com/sympy/sympy/commit/9e35a94eceaff73b350794dcc70b4a412dc2f6e6#diff-e20bc128d13486b598a04fce77584900`, 2017.
Sym17b. SymPy Development Team. *SymPy: Python library for symbolic mathematics*, 2017.
The18a. The PARI Group, Univ. Bordeaux. *PARI/GP Frequently Asked Questions*, 2018. available from `http://pari.math.u-bordeaux.fr/faq.html#primetest`.
The18b. The PARI Group, Univ. Bordeaux. *PARI/GP version* `2.9.0`, 2018. available from `http://pari.math.u-bordeaux.fr/`.
VAS+17. Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *NDSS 2017*. The Internet Society, February / March 2017.
Wat17. Waterloo Maple (Maplesoft). *Maple Version 2017*, 2017. Available at `https://www.maplesoft.com/products/Maple/`.
Wei18. Eric W. Weisstein. Baillie-PSW primality test from MathWorld- a Wolfram web resource. `http://mathworld.wolfram.com/Baillie-PSWPrimalityTest.html`, April 2018.
Wol18a. WolfSSL Inc. Pull request - Prime Number Testing. `https://github.com/wolfSSL/wolfssl/pull/1665`, August 2018.
Wol18b. WolfSSL Inc. WolfSSL. `https://www.wolfssl.com/wolfSSL/Home.html`, April 2018.
Won16. David Wong. How to Backdoor Diffie-Hellman. Cryptology ePrint Archive, Report 2016/644, 2016. `https://eprint.iacr.org/2016/644`.
Wu17. Tom Wu. JSBN: RSA and ECC in JavaScript. `http://www-cs-students.stanford.edu/~tjw/jsbn/`, April 2017.

A An Overview of Arnault's Method

Arnault's method generates n of the form $n = p_1 p_2 \dots p_h$ where the p_i are distinct odd primes such that n is pseudoprime to a set of t prime bases $\{a_1, a_2, \dots, a_t\}$. By [Arn95, Lemma 3.2] we know that if $\gcd(a, n) = 1$ and $\left(\frac{a}{p_i}\right) = -1$ for all $1 \leq i \leq h$, then a will be a Miller-Rabin non-witness with respect to n (this set of conditions is sufficient but not necessary for a to be a Miller-Rabin non-witness with respect to n).

Now, by Gauss's law of quadratic reciprocity, we know that, for any prime p, $\left(\frac{a}{p}\right)$ can be determined from $\left(\frac{p}{a}\right)$ and the values of a and p taken modulo 4. This in turn means that, for each a, we can compute the set S_a of possible non-residues mod $4a$ of potential primes p. That is, we can compute the set S_a satisfying

$$\left(\frac{a}{p}\right) = -1 \iff p \bmod 4a \in S_a.$$

Arnault's method selects p_1 and then determines the other p_i from equations of the form $p_i = k_i(p_1 - 1) + 1$ where the k_i are values also chosen as part of the method (with $k_1 = 1$). This is done so as to ensure that the resulting $n = p_1 p_2 \dots p_h$ is a Carmichael number. But the conditions $\left(\frac{a}{p_i}\right) = -1$ for all $1 \leq i \leq h$ imply that, for each $a \in A$ and each $1 \leq i \leq h$ we have

$k_i(p_1 - 1) + 1 \in S_a$. Rewriting this, we obtain that:

$$p_1 \bmod 4a \in \bigcap_{i=1}^{h} k_i^{-1}(S_a + k_i - 1), \tag{5}$$

where $k_i^{-1}(S_a + k_i - 1)$ denotes the set $\{k_i^{-1}(s + k_i - 1) \bmod 4a | s \in S_a\}$. This gives a set of conditions on the value of p_1 modulo $4a$ for each $a \in A$; typically a few candidates for $p_1 \bmod 4a$ remain for each value of a. By selecting one of these candidates z_a for each $a \in A$ and using the CRT, the conditions can be combined into a single condition on p_1 modulo $m = \text{lcm}(4, a_1, \ldots, a_t)$. The k_i values must be selected so that the sets on the right of (5) are non-empty; typically, they are set to small primes larger than the maximum of the $a \in A$ so that k_i^{-1} exists mod $4a$ for each a.

Arnault's method then brings into play other restrictions on $p_1 \bmod k_i$ for each $i = 2, \ldots, h$. These result from the requirement that n be a Carmichael number. We omit the full details, but, for example, when $h = 3$, the additional restrictions can be written as:

$$p_1 = k_3^{-1} \bmod k_2 \quad \text{and} \quad p_1 = k_2^{-1} \bmod k_3$$

Making the k_i co-prime to each other and to the $a \in A$ ensures that another application of the CRT can be made to incorporate these conditions. The end result is a single condition of the form:

$$p_1 = z \bmod \text{lcm}(4, a_1, \ldots, a_t, k_2, \ldots, k_h)$$

where z is a fixed value determined by the choice of the z_a values and the additional restrictions.

Finally, the method repeatedly generates candidates for p_1 satisfying the above constraint and uses the equations $p_i = k_i(p_1 - 1) + 1$ to determine the other p_i. The method is successful for a given p_1 if all of the resulting $p_1, \ldots, p_h$ are prime.

Evidently, the method is complex and not guaranteed to succeed on every attempt for a given set A. However, it can be iterated with different choices of the k_i until the sets on the right of (5) are non-empty; moreover a back-tracking approach can be used to select the z_a values to speed-up the entire process of constructing p_1. The density of all-prime solutions $(p_1, \ldots, p_h)$ amongst all possible candidates $(p_1, \ldots, p_h)$ satisfying $p_1 = z \bmod \text{lcm}(4, a_1, \ldots, a_t, k_2, \ldots, k_h)$ and $p_i = k_i(p_1 - 1) + 1$ for $i = 2, \ldots, h$ can be estimated using standard heuristics concerning the distribution of primes of size $L = \text{lcm}(4, a_1, \ldots, a_t, k_2, \ldots, k_h)$; it is roughly $1/(\log^h(L) \cdot \sum_{i=2}^{h} \log(k_i))$.

Notice that, the larger the set A, the larger the modulus L in the condition determining p_1 will be. Thus, if A contains many bases, then larger p_i and hence larger n will tend to result. Moreover, all-prime solutions will become less dense. As an example, when analysing the primality test in Maple V.2, Arnault [Arn95] considers $h = 3$ so $n = p_1p_2p_3$ and $A = \{2, 3, 5, 7, 11\}$ (so $t = 5$); he works with $k_2 = 13$ and $k_3 = 41$ and arrives finally at the condition:

$$p_1 = 827443 \bmod 4924920.$$

For $p_1 = 286472803$, this yields a 29-decimal digit composite passing Maple's fixed-base Miller-Rabin primality test.

We give a short example of the method described for an n of the form $n = p_1p_2p_3$ for which the first 10 primes are Miller-Rabin non-witnesses. That is, we target $A = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$.

We start by generating the set S_a of residues modulo $4a$ of primes p such that $\left(\frac{a}{p}\right) = -1$ for each base $a \in A$. Table 3 gives the sets S_a for our chosen set A.

We now set $k_2 = 41$ and $k_3 = 101$; these are coprime to all $a \in A$. We find subsets of the S_a that meet the requirement:

$$p_1 \pmod{4a} \in \bigcap_{i=1}^{h} k_i^{-1}(S_a + k_i - 1).$$

This gives us a set of residues modulo $4a$ for each $a \in A$ that p_1 must satisfy. We give an example of this for the first 10 primes in Table 4.

Table 3. Values a and subsets S_a of residues modulo $4a$ of primes p such that $\left(\frac{a}{p}\right) = -1$.

a	S_b
2	$\{3, 5\}$
3	$\{5, 7\}$
5	$\{3, 7, 13, 17\}$
7	$\{5, 11, 13, 15, 17, 23\}$
11	$\{3, 13, 15, 17, 21, 23, 27, 29, 31, 41\}$
13	$\{5, 7, 11, 15, 19, 21, 31, 33, 37, 41, 45, 47\}$
17	$\{3, 5, 7, 11, 23, 27, 29, 31, 37, 39, 41, 45, 57, 61, 63, 65\}$
19	$\{7, 11, 13, 21, 23, 29, 33, 35, 37, 39, 41, 43, 47, 53, 55, 63, 65, 69\}$
23	$\{3, 5, 17, 21, 27, 31, 33, 35, 37, 39, 45, 47, 53, 55, 57, 59, 61, 65, 71, 75, 87, 89\}$
29	$\{3, 11, 15, 17, 19, 21, 27, 31, 37, 39, 41, 43, 47, 55, 69, 73, 75, 77, 79, 85, 89, 95, 97, 99, 101, 105, 113\}$

a	$\bigcap_{i=1}^{h} k_i^{-1}(S_a + k_i - 1)$	Modulo
2	$\{\mathbf{3}, 5\}$	8
3	$\{\mathbf{7}\}$	12
5	$\{\mathbf{3}, 7, 13, 17\}$	20
7	$\{\mathbf{15}\}$	28
11	$\{21, \mathbf{23}\}$	44
13	$\{21, \mathbf{47}\}$	52
17	$\{5, 29, \mathbf{31}, 39, 63, 65\}$	68
19	$\{33, 37, 39, \mathbf{47}, 69\}$	76
23	$\{31, \mathbf{47}, 57, 87, 89\}$	92
29	$\{19, 37, 41, \mathbf{55}, 77, 95, 99, 113\}$	116

Table 4. Values a and the sets $\bigcap_{i=1}^{h} k_i^{-1}(S_a + k_i - 1)$ when $k_2 = 41$ and $k_3 = 101$.

We then need to make a choice of one residue z_a per set. This choice is arbitrary, but we note that not all combinations of choices will lead to a solution. We give an example of a good set of choices in Table 4 in bold.

We then have two additional conditions to add, based on our choice of the k_i values. These can be written as:

$$p_1 = k_3^{-1} \bmod k_2 \quad \text{and} \quad p_1 = k_2^{-1} \bmod k_3$$

In our example, we chose $k_1 = 41$ and $k_2 = 101$ which gives us:

$$p_1 \equiv 28 \pmod{41} \qquad \text{and} \qquad p_1 \equiv 32 \pmod{101}.$$

We can then use the Chinese Remainder Theorem to simultaneously solve for the 10 conditions implied by the bold entries in Table 4 and the two conditions above. In this case, we have the solution:

$$p_1 \equiv 36253030834483 \bmod 107163998661720.$$

The prime

$$p_1 = 1424453871614154824048263654181759622666891330061 63$$

satisfies this condition, and yields primes

$$p_2 = 5840260873618034778597880982145214452934254453252643$$
$$p_3 = 1438698410330296372288746290723577218893560243362 3363$$

such that the product $n = p_1 p_2 p_3$ is a 512-bit number that is a Miller-Rabin pseudoprime to the bases $2, 3, 5, 7, 11, 13, 17, 19, 23$ and 29.

B A Large strong Lucas Pseudoprime

Using our SAGE implementation of the method as described in Section 3.2.1, we construct an n of the form $n = p_1p_2p_3$, where $p_i = k_i(p_1 + 1) - 1$ with $(k_2, k_3) = (31, 43)$ and

$$\begin{aligned} p_1 = 2^{576} &\cdot \texttt{0x000000000000000000000000bc508ae6dacc43b138c0e9f22d} \\ + 2^{384} &\cdot \texttt{0xfb99b146bedd0ac93f84e8cfe2780a881fdbad85918a6b75} \\ + 2^{192} &\cdot \texttt{0xbd3af841123bad7438fe08c5433ec8b5fa7b0a1b149876bf} \\ + 2^{0} &\cdot \texttt{0x5af73cd9a608317066029e0cff4171ce336ff0b666344757}. \end{aligned}$$

Then $n = p_1p_2p_3$ is a 2050-bit strong Lucas pseudoprime for Selfridge's Method A of parameter selection.

C Constructing GMP Pseudoprimes

Recall that we work with candidates x of the form $x = kM + 189$, and then consider $n = (2x + 1)(4x + 1)$; we select x so that $2x + 1$ and $4x + 1$ are both prime, and we select M as a product of the first ℓ primes from the set $\mathcal{P} = \{2, 3, \ldots, 373\}$. We justify this construction here.

First, note that $2x + 1 = 2kM + 379$ while $4x + 1 = 4kM + 757$, where both 379 and 757 are prime. Considering $2x + 1$ modulo each of the ℓ prime factors p in M, we see that $2x+1 = 379 \bmod p \neq 0 \bmod p$ because $p < 379$; similarly, we obtain $4x+1 = 757 \bmod p \neq 0 \bmod p$. Hence no such p divides either $2x + 1$ or $4x + 1$, so these numbers are not divisible by any of the primes in the product M (i.e. the first ℓ primes). For this reason, with random choices of k and with $x = kM + 189$, it follows that $2x + 1$ and $4x + 1$ are more likely to be prime than they would be for random choices of x. An analysis of the effect involves an application of the inclusion-exclusion principle to determine how many numbers are "sieved out" by the process. We omit the full analysis here, but note that, for numbers of cryptographically interesting size and with $\ell = 69$ that we use in the construction of our 1024-bit example for n, the effect is to increase the probability of primality for each number from $1/\ln x$ to roughly $5/\ln x$. Since we have two numbers $2x + 1$, $4x + 1$ whose primality behaves largely independently over the choice of x, this yields a 25-fold improvement in the performance of our approach over the direct approach of trying random x values.

Next, we consider the Fermat test on n with base $a = 210$, assuming the factors $2x + 1$ and $4x + 1$ are prime. This test computes the value of $a^{n-1} \bmod n$ and compares it to 1. Now $n - 1 = (2x + 1)(4x + 1) = 8x^2 + 6x = 2x(4x + 3)$, so we obtain:

$$a^{n-1} = (a^{4x+3})^{2x} = 1 \bmod 2x + 1$$

and

$$a^{n-1} = a^{8x^2+6x} = (a^{2x+1})^{4x} \cdot a^{2x} = 1 \cdot a^{2x} = a^{2x} \bmod 4x + 1.$$

Here, we have made repeated use of Fermat's Little Theorem (which states that $a^{p-1} = 1 \bmod p$ for prime p and $a \neq 0 \bmod p$).

It follows that $a^{n-1} = 1 \bmod n$ if and only if a is a quadratic residue modulo $4x + 1$. It follows that n passes a Fermat test to base a for roughly half of the possible bases a (since roughly half of the values $a \bmod n$ are quadratic residues $\bmod 4x + 1$).

Now we use the fact that $a = 210 = 2 \cdot 3 \cdot 5 \cdot 7$ to write:

$$\left(\frac{210}{4x+1}\right) = \left(\frac{2}{4x+1}\right)\left(\frac{3}{4x+1}\right)\left(\frac{5}{4x+1}\right)\left(\frac{7}{4x+1}\right).$$

Since M is even, we can write $4x + 1 = 8k(M/2) + 757 = 5 \bmod 8$, hence $(\frac{2}{4x+1}) = -1$. Also $(\frac{3}{4x+1}) = (\frac{4kM+757}{3}) = (\frac{757}{3}) = (\frac{1}{3}) = 1$, where we use Gauss's Law of Quadratic Reciprocity and

$3|M$. Similarly, we obtain $(\frac{5}{4x+1}) = -1$ and $(\frac{7}{4x+1}) = 1$. Combining everything, we finally get

$$\left(\frac{210}{4x+1}\right) = (-1) \cdot 1 \cdot (-1) \cdot 1 = 1.$$

We conclude that the Fermat test for n of the given form with base $a = 210$ always passes.

D A Pseudoprime for Mini-GMP

Using our SAGE implementation of the composite fixed base technique as described in Section 3.1.4, we construct an n of the form $n = p_1 p_2 p_3$, where $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (10937, 11257)$ and

$$\begin{aligned} p_1 = 2^{960} &\cdot \texttt{0x0000000000000000000000000000000000000000000002e394} \\ + 2^{768} &\cdot \texttt{0x1a2fe4aa9e66358347f63732494d08635ccc9ae0a3c17764} \\ + 2^{576} &\cdot \texttt{0xa8e266f4d26758ab804a702c235f63b1e109a81fc007f94b} \\ + 2^{384} &\cdot \texttt{0xec5158f231a30b1cbf96a7fc444c09be62f5a809f049cc5d} \\ + 2^{192} &\cdot \texttt{0xe94b84275c38885c9b61a6bdc44111501527722a8ac87ea2} \\ + 2^{0} &\cdot \texttt{0xa5d4498caa2d9d07b34001a508fa53063991206268c547d7}. \end{aligned}$$

This yields a 2960-bit composite n that is guaranteed to pass any number up to and including $t = 101$ rounds of Mini-GMP's primality test.

E Java Code Listing

We include the source code of the function `primeToCertainty` from the class `java.math.BigInteger`.

Listing 1.1. `OpenJDK10 java.math.BigInteger` function `primeToCertainty`

```
boolean primeToCertainty(int certainty, Random random) {
    int rounds = 0;
    int n = (Math.min(certainty, Integer.MAX_VALUE-1)+1)/2;

    // The relationship between the certainty and the number of rounds
    // we perform is given in the draft standard ANSI X9.80, "PRIME
    // NUMBER GENERATION, PRIMALITY TESTING, AND PRIMALITY CERTIFICATES".
    int sizeInBits = this.bitLength();
    if (sizeInBits < 100) {
        rounds = 50;
        rounds = n < rounds ? n : rounds;
        return passesMillerRabin(rounds, random);
    }

    if (sizeInBits < 256) {
        rounds = 27;
    } else if (sizeInBits < 512) {
        rounds = 15;
    } else if (sizeInBits < 768) {
        rounds = 8;
    } else if (sizeInBits < 1024) {
        rounds = 4;
    } else {
        rounds = 2;
    }
    rounds = n < rounds ? n : rounds;

    return passesMillerRabin(rounds, random) && passesLucasLehmer();
}
```

F An Example Pseudoprime for JSBN

Using our SAGE implementation of the method as described in Section 3.1.2 with A containing the first 1000 primes, we construct a 4279-bit n of the form $n = p_1 p_2 p_3$, where $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (1013, 2053)$ and

$$\begin{aligned} p_1 = {} & 2^{1344} \cdot \texttt{0x000000000000000000000000000083dda18eb04a7597ca3} \\ & + 2^{1152} \cdot \texttt{0xc6bc877df8a08eec6725fa0832cba270c42adc358bc0cf50} \\ & + 2^{960} \cdot \texttt{0xc82cb10f2733c3fb8875231fc1498a7b14cb675fac1bf3c5} \\ & + 2^{768} \cdot \texttt{0x127a76fc11e5d20e27940c95ceba671fe1c4232250b74cbd} \\ & + 2^{576} \cdot \texttt{0xf8448c90321513324c0681afb4ba003353b1afb0f1e8b91c} \\ & + 2^{384} \cdot \texttt{0x60af672a5a6f4d06dd0070a4bc74e425f3eae90379e57754} \\ & + 2^{192} \cdot \texttt{0x82d26e80e247464a4bb817dfcf7572f89f8b9cacd059b584} \\ & + 2^{0} \cdot \texttt{0x0e4389c8af84f6a6ea15a3ea5d62cb994b082731ba4cde73}. \end{aligned}$$

This produces an n that is *guaranteed* to be declared prime by JSBN's primality test for *any* certainty parameter t.

G An Example Pseudoprime for Cryptlib

Using our SAGE implementation of the method as described in Section 3.1.2 with A containing the first 100 primes, we construct a 2315-bit n of the form $n = p_1 p_2 p_3$, where $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (641, 677)$ and

$$\begin{aligned} p_1 = {} & 2^{576} \cdot \texttt{0x24a027808260908b96d740bef8355ded63f6edb7f70de9a9} \\ & + 2^{384} \cdot \texttt{0xb99c408f131cef3855b4b0aea6b17a4469ed5a7ec8b2be62} \\ & + 2^{192} \cdot \texttt{0x66c3a9eae83a6769e175cb2598256da977b9e191b9b847a7} \\ & + 2^{0} \cdot \texttt{0xe2cf4750d9bc2d64ccd3406f5db662c22c3fc65e3c56eff3}. \end{aligned}$$

This n is declared prime for any valid number of rounds t of testing performed by Cryptlib's primality test.

H Example Pseudoprimes for LibTomMath, LibTomCrypt and WolfSSL

Using our SAGE implementation of the method as described in Section 3.1.2 with A containing the first 256 primes, we construct a 7023-bit n of the form $n = p_1 p_2 p_3$, where $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (2633, 5881)$ and

$$\begin{aligned} p_1 = {} & 2^{2304} \cdot \texttt{0x0000000000000000000000000000000000000001e46d6a8} \\ & + 2^{2112} \cdot \texttt{0x4d42d684ddb3415e871b661303b1c60f0388dfb9e525f8bc} \\ & + 2^{1920} \cdot \texttt{0x51c9de3c9f45627608de2f75dee580d9d4d97cab6fa86dad} \\ & + 2^{1728} \cdot \texttt{0x9e6bbfd721f297472480a9bed9508aa884bda9dc56833752} \\ & + 2^{1536} \cdot \texttt{0xfac8e89f413a9517d14731277148789987806654a8723593} \\ & + 2^{1344} \cdot \texttt{0xa452f960facc9b65f6962cb26131b42650c29c8735083c7e} \\ & + 2^{1152} \cdot \texttt{0x6c3a220d77d1cbe7f9628885a7b79465287d4b02ad546007} \\ & + 2^{960} \cdot \texttt{0x1d43306a8813836de5ccd162fbeca4f117552dba01975451} \\ & + 2^{768} \cdot \texttt{0x2f7684e32b0377e76f87b96906f8fa276381db612f76c2c7} \\ & + 2^{576} \cdot \texttt{0xdd97ab4380042c991a4719884377c70065a3614237a41289} \\ & + 2^{384} \cdot \texttt{0x24a1017fbb529443b0ad43c5424753db5b518cee5a1fcd87} \\ & + 2^{192} \cdot \texttt{0xea038ffcad33380db1d89cd4e0b15b480cf0c62e8999924d} \\ & + 2^{0} \cdot \texttt{0x0284af806081ea106f35f85a664456166b864650ef034cf3}. \end{aligned}$$

This n is declared prime for any valid number of rounds t for LibTomMath, LibTomCrypt and WolfSSL libraries.

Also using the method as described in Section 3.1.2 but now with A containing the first 40 primes, we can construct a 1024-bit n of the form $n = p_1p_2p_3$, where $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (233, 241)$ and

$$\begin{aligned} p_1 = 2^{192} &\cdot \texttt{0x000000000000e17516504450e648b6aedb0c0784e17dda33} \\ + 2^{0} &\cdot \texttt{0x63e1956a843076a9e5b6d15a819cf0907a96154d47662d0b}. \end{aligned}$$

This n is guaranteed to be declared prime by `mp_prime_is_prime` with $t \leq 40$, and therefore also guaranteed to be declared prime by `mp_prime_is_prime` as in LibTomCrypt 1.18.1 and WolfSSL 3.13.0 for the same values of t.

I An Example Pseudoprime for GoLang pre-1.8

Using the method described in Section 4.2, we construct a 1024-bit composite n that is declared prime by GoLang's primality test in versions prior to 1.8 with 100% success for $t \leq 13$. We take

$$\begin{aligned} n = 2^{960} &\cdot \texttt{0x00000000000000000000000000000000ff7d428a8a9f9ffc} \\ + 2^{768} &\cdot \texttt{0x2ea178501115ec855f1154c054f5f67e15967a139a92fe15} \\ + 2^{576} &\cdot \texttt{0xddf2c49b044820ea8c58551b74f81b45b116da4e1f11b926} \\ + 2^{384} &\cdot \texttt{0x93e0cdc58006bc2052eb9b2fc32c71dd041d1907225e2814} \\ + 2^{192} &\cdot \texttt{0xebe18736f626fea57c965b67b296a6461455226b39aba263} \\ + 2^{0} &\cdot \texttt{0x3faeb483847a715c6a01d8d0e401a4aaf8f3d22121fd142f}. \end{aligned}$$

J An Example of a Malicious DH Parameter Set for Botan

Using our SAGE implementation of the method in Section 3.1.3, we construct a 160-bit q of the form $q = q_1q_2q_3$, where $q_i = k_i(q_1 - 1) + 1$ with $(k_2, k_3) = (61, 101)$ and $q_1 = 537242417098003$.

This q is declared prime with probability 1/4096 by Botan's `verify_group` function. By setting $k = 2^{864} + 134$ in $p = kq + 1$ we obtain a prime p, and thus by setting the generator g as:

$$\begin{aligned} g = 2^{960} &\cdot \texttt{0x00000000000000000000000000000000075ead4f9fa60a06e} \\ + 2^{768} &\cdot \texttt{0x0787a1e0708f5e2055b2899691f7dd73303d5643e57b1636} \\ + 2^{576} &\cdot \texttt{0x66ce328086bd6a0df756175c35549ba7a5ffe517036c0ef1} \\ + 2^{384} &\cdot \texttt{0x44a9542f698255efb66cda28b0b8a5ebebf2c0892f8147d3} \\ + 2^{192} &\cdot \texttt{0x72083822a36098addcd30a1767ccefaae65d1dcd6b45de92} \\ + 2^{0} &\cdot \texttt{0x09047326d40b622af6a76218664ba3df13eb0fead02d772a} \end{aligned}$$

we obtain a parameter set (p, q, g) such that g generates the subgroup of order q. The probability that this set is accepted by Botan's `verify_group` function is 1/4096. The DLP in the subgroup generated by g can be solved using the Pohlig-Hellman algorithm over each of the 49-bit, 55-bit and 56-bit factors q_1, q_2 and q_3 of q. The cost is dominated by the largest prime factor, and is approximately 2^{28} operations.

K Details of Mathematics Software Packages

K.1 Magma

Magma V2.23-9 [BCP97] is a mathematical software package designed for computations in algebra, number theory, algebraic geometry and algebraic combinatorics.

Analysis. Magma provides a primality testing function that can either invoke a primality proving algorithm, or what they call a probable-primality test, depending on the arguments given when called. The main function call for primality testing is `IsPrime(n:- Proof)`. The more rigorous method of primality proving is based on an implementation of the ECPP (Elliptic Curve Primality Proving) method [AM93] is used by default, unless the number tested is greater than 34×10^{13} or the parameter `Proof = False`. In this case, the probable-primality test `IsProbablePrime` is instead called. By default, this consists of 20 rounds of Miller-Rabin with random bases. By setting the optional parameter `Bases` to some value B, the number of bases used is B instead of 20.

Pseudoprimes. The pseudoprimes generated in Section 3 attempt only to overcome probabilistic primality testing and are not designed to overcome primality proving methods such as ECPP.

However, if the parameters are set to invoke the probable-primality test with default parameters, then composites generated by the methods in Section 3.1 have a probability of 2^{-40} of being falsely declared prime. This probability is correctly alluded to as being *worst-case* by the documentation given for this function.

K.2 Maple

Maple 2017 [Wat17] is a computer algebra system developed by Maplesoft, that provides a general purpose software tool for mathematics, data analysis, visualisation, and programming.

Analysis. The primality test in Maple is called as `isprime(n)` on a candidate n to be tested. Documentation states that "*It returns false if n is shown to be composite within one strong pseudo-primality test and one Lucas test. It returns true otherwise*". The function begins with some trial division on a series of small primes before calling `gmp_isprime(n)`. If the result of `gmp_isprime(n)` is 1 (i.e. the number is "probably prime") and the candidate n being tested is greater than $5 \times 10^9 \approx 2^{33}$, then `isprime` will go on to perform a Lucas test on n. In all other cases, the Lucas test is omitted.

Although we cannot directly inspect the code of `gmp_isprime(n)` (since Maple is proprietary software) we are able to reverse-engineer this function by calling it on our own input n and assessing how it performs. Maple's documentation states that it performs a Miller-Rabin test and uses GMP for this function, yet since there is no other code indicative of a Miller-Rabin test in `gmp_isprime(n)`, we deduce that Maple is calling GMP's function `mpz_probab_prime_p(n, reps)`. Since `gmp_isprime(n)` takes only a single argument, we inferred that Maple passes a hardcoded value of `reps` to GMP. We were able to verify that the value of `reps` is actually 5. We did this by using the methods described in Section 4.2 to generate composite numbers of various bit-sizes that are declared prime by `mpz_probab_prime_p (n, reps)` for `reps = 1,2,3,4,5`. For composites that can only pass at most `reps = 4`, Maple's `gmp_isprime` correctly identifies these as composite. But for composites that pass `reps = 5`, the function falsely declares them to be prime.

Pseudoprimes. When testing numbers $n \leq 5 \times 10^9$, `isprime` acts as a deterministic version of the Miller-Rabin test. We have verified this by calling `mpz_probab_prime_p(n,5)` for all $n \leq 5 \times 10^9$ and comparing the results to a list of primes below 5×10^9. The different sets of bases that GMP chooses for each n are such that there are no composites below this threshold that are declared prime by `mpz_probab_prime_p` with `reps > 3`. However, any change made to the (flawed) way GMP currently chooses its bases for testing could actually make Maple's `isprime` function *less* accurate (and no longer deterministic) for $n \leq 5 \times 10^9$!

To fool Maple's primality testing for numbers larger than 5×10^9, we would need a composite n passing a Lucas test and 5 rounds of Miller-Rabin testing. We do not currently know any such n.

K.3 Maxima

Maxima 5.41.0 [Mac18] is a free, open source computer algebra system developed by the Macsyma group. Maxima is a general-purpose system including tools for a variety of mathematical functions and the manipulation of symbolic and numerical expressions.

Analysis. The primality test supplied by Maxima is the function `primep(n)`. When testing an n less than 341550071728321 ($\approx 2^{49}$) a deterministic version of Miller-Rabin's test is used. This is achieved by calling repeated rounds of Miller-Rabin tests with a set of bases for which it has been verified that no composites are falsely declared prime. These are as defined in [Jae93,McC97], and therefore can in general be used to create a deterministic test for numbers less than 2^{64}.

When testing an n bigger than 341550071728321, `primep(n)` performs 25 random base Miller-Rabin tests, then conducts one Lucas test. The source Maxima uses for base selection is then provided by the Maxima random number generator, which is an implementation of Mersenne twister MT 19937 [Mac17].

Maxima's documentation correctly states that "The probability that a non-prime n will pass one Miller-Rabin test is less than 1/4. Using the default value 25 for `primep_number_of_tests`, the probability of n being composite is much smaller that 10^{-15}."

Pseudoprimes. When testing numbers $n < 341550071728321$ ($\approx 2^{49}$) the function `primep(n)` is deterministic, so no pseudoprimes can arise. If $n > 341550071728321$, then the combination of Miller-Rabin testing and a Lucas test mean that no pseudoprimes for the test are known.

K.4 SageMath

SageMath 8.2 (or simply Sage) is a free Python-based open source mathematics software system originally created by William Stein [S$^+$17] but now developed by many volunteers. Sage provides a toolkit of mathematical functions in areas such as algebra, combinatorics, numerical mathematics, number theory, and calculus.

Analysis Although there are many methods one could use to test the primality of a number in Sage, the flagship function is `is_prime(n, proof)` found in `/src/sage/rings/integer.pyx`. If called with the value of `proof` set as `True` (default when starting Sage), the function will perform use a provable primality test. If set to `False` it uses a strong pseudo-primality test and instead calls `is_pseudoprime(n)`.

The "provable primality test" called when `proof = True` is the PARI [The18b] `isprime` function. This then uses a combination of the Baillie-PSW test, Selfridge "$p-1$", and Adleman-Pomerance-Rumely-Cohen-Lenstra (APRCL). It is indicated in documentation that this test can be "very slow" when testing a prime that "has more 1000 digits".

The "strong pseudo-primality test" called when `proof = False` is less accurate, but much quicker, and is therefore a likely choice when testing large candidates. The candidates are then tested by PARI's `is_pseudoprime(n)`, which consists of a Baillie-PSW test.

Pseudoprimes. Since a Baillie-PSW test is being performed, we know of no composites that are incorrectly declared prime by SageMath for either boolean value of `proof`.

K.5 SymPy

SymPy [Sym17b] is a free, open source and widely used symbolic computation Python library that provides a computer algebra system like functionality.

Analysis. SymPy provides the primality test `isprime(n)`, which like Maxima, uses select bases to perform a deterministic version of Miller-Rabin when testing candidates $n < 2^{64}$. We shall consider the latest couple of releases of SymPy (SymPy 1.0 and SymPy 1.1), since significant changes to the function `isprime` have been made recently.

SymPy 1.0 Prior to release 1.1 in July 2017, SymPy 1.0 conducted the primality test found in `isprime` in the same manner. After some initial trial divisions, if no factor is found, the function would call upon a deterministic version of the Miller-Rabin test, using bases described in [Jae93,McC97]. For numbers larger than $\approx 2^{53}$, the test would call additional rounds of Miller-Rabin. In all releases up to and including 0.6.6 of 2009, this would simply perform 8 rounds of Miller-Rabin on the bases $\{2, 3, 5, 7, 11, 13, 17, 19\}$. In version 0.6.7 [Sym17a], this was increased to 46 rounds of Miller-Rabin, using the first 46 primes as bases. The test then remained fundamentally unchanged until version 1.1 in 2017.

SymPy 1.1 onwards In July 2017 the function `isprime` was revised to remove the final Miller-Rabin test on 46 bases and replace it with a Baillie-PSW test as described in Section 2.4.

Pseudoprimes. SymPy 1.0 and all previous versions are vulnerable to composite numbers n generated by the methods in Section 3.1.2. These numbers are trivial to construct when the final Miller-Rabin test is based on the first 8 primes, but even after the changes made in 0.6.7, all versions prior to 1.1 would wrongly declare composites generated in this manner to be prime. For example, using the method of Section 3.1.2, we are able to construct a 1024-bit n of the form $n = p_1p_2p_3$ that is pseudoprime to all bases selected by SymPy in all versions prior to 1.1. Here $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (241, 257)$ and

$$\begin{aligned} p_1 = 2^{192} &\cdot \texttt{0x000000000000f8ae31e07964373e4997647e75fa186dd5e7} \\ + 2^{0} &\cdot \texttt{0xe42ada869da0b3a333813f8102b1fb5f20623d6543e78a3b}. \end{aligned}$$

Since SymPy 1.1 introduced a Baillie-PSW test, we can no longer generate composites that would be declared prime by SymPy.

K.6 Wolfram Mathematica

Wolfram Mathematica is a computational software package developed by Wolfram Research that covers scientific, engineering, mathematical, and computing fields. The current release, Mathematica 11.3 [RI18], features built-in integration with Wolfram Alpha.

Analysis. Mathematica provides the inbuilt primality test `PrimeQ` that is said to perform two Miller-Rabin tests using bases 2 and 3, combined with a "Lucas pseudoprime" test. Since the source code is not open source, we are unable to verify the parameters used in the Lucas test. We note that the documentation references Baillie and Wagstaff [BW80], from which Selfridge's parameters originate. Documentation of the function also indicates that this procedure is only known to be correct for $n < 10^{16}$ and that "*it is conceivable that for larger n it could claim a composite number to be prime*".

Pseudoprimes. Since a Baillie-PSW test is being performed, we know of no composites that are incorrectly declared prime.