A Performant, Misuse-Resistant API for Primality Testing

Jake Massimo
jake.massimo.2015@rhul.ac.uk
Royal Holloway, University of London

Kenneth G. Paterson
kenny.paterson@inf.ethz.ch
Department of Computer Science, ETH Zurich

ABSTRACT

Primality testing is a basic cryptographic task. But developers today are faced with complex APIs for primality testing, along with documentation that fails to clearly state the reliability of the tests being performed. This leads to the APIs being incorrectly used in practice, with potentially disastrous consequences. In an effort to overcome this, we present a primality test having a simplest-possible API: the test accepts a number to be tested and returns a Boolean indicating whether the input was composite or probably prime. For all inputs, the output is guaranteed to be correct with probability at least $1 - 2^{-128}$. The test is performant: on random, odd, 1024-bit inputs, it is faster than the default test used in OpenSSL by 17%. We investigate the impact of our new test on the cost of random prime generation, a key use case for primality testing. The OpenSSL developers have adopted our suggestions in full; our new API and primality test are scheduled for release in OpenSSL 3.0.

CCS CONCEPTS

• **Mathematics of computing** → **Random number generation**; • **Security and privacy** → **Mathematical foundations of cryptography**; • **Software and its engineering** → **Software libraries and repositories**.

KEYWORDS

Primality testing; Prime generation; Miller-Rabin test; Lucas test; Baillie-PSW test; API design

ACM Reference Format:
Jake Massimo and Kenneth G. Paterson. 2020. A Performant, Misuse-Resistant API for Primality Testing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3372297.3417264

1 INTRODUCTION

Primality testing, and closely related tasks like random prime generation and testing of Diffie-Hellman parameters, are core cryptographic tasks. Primality testing is by now very well understood mathematically; there is a clear distinction between accuracy and running time of different tests in settings that are malicious (i.e. where the input may be adversarially-selected) and non-malicious (e.g. where the input is random, as is common in prime generation).

Yet recent research by Albrecht *et al.* [3] on how primality testing is actually done in practice has highlighted the failure of popular cryptographic libraries to provide primality testing APIs that are "misuse-resistant", that is, which provide reliable results in all use cases even when the developer is crypto-naive. Extending [3], Galbraith *et. al.* [18] showed how failure to perform robust primality testing in the popular OpenSSL library has serious security consequences in the face of maliciously generated Diffie-Hellman parameter sets (see also Bleichenbacher [9] for an earlier example involving the GNU Crypto library).

The main underlying issue identified in [3] is that, while all libraries examined performed well on random inputs, some failed miserably on maliciously crafted ones in their default settings. Meanwhile code documentation was generally poor and did not distinguish clearly between the different use cases. And developers were faced with complex APIs requiring them to understand the distinctions between use cases and choose parameters to the APIs accordingly. An illustrative example is provided by the OpenSSL primality testing code that existed prior to our work. This required the developer using the function `BN_is_prime_fasttest_ex`[1] to pass multiple parameters, including `checks`, the number of rounds of Miller-Rabin testing to be carried out; and `do_trial_division`, a flag indicating whether or not trial division should be performed. Setting `checks` to 0 makes the test default to using a number of rounds that depends only on the size of the number being tested;[2] then the number of rounds *decreases* as the size increases, this being motivated by average-case error estimates for the Miller-Rabin primality test operating on random numbers [14, 29]. This makes the default setting performant for random prime generation, but dangerous in potentially hostile settings, e.g. Diffie-Hellman parameter testing.

As an illustration of how this can go wrong in practice, Galbraith *et. al.* [18] pointed out that OpenSSL (pre-1.1.1c May 2019) itself makes the wrong choice in using the default setting when testing finite field Diffie-Hellman parameters. Galbraith *et. al.* exploited this choice to construct Diffie-Hellman parameter sets (p, q, g) of cryptographic size that fool OpenSSL's parameter validation with a non-trivial success rate. OpenSSL's Diffie-Hellman parameter validation was subsequently changed to remedy this issue (though without changing the underlying primality test).[3] This example provides *prima facie* evidence that even very experienced developers can misunderstand how to correctly use complex primality testing APIs.

One may argue that developers who are not cryptography experts should not be using such security-sensitive APIs. However,

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417264

[1] See https://github.com/openssl/openssl/blob/3e3dcf9ab8a2fc0214502dad56d94fd95bcbbfd5/crypto/bn/bn_prime.c#L186.

[2] Strictly, the default is invoked by setting `checks` to `BN_prime_checks`, an environmental variable that is set to 0.

[3] See https://github.com/openssl/openssl/pull/8593.

they inevitably will, and, as our OpenSSL example shows, even expert developers can get it wrong. This motivates the search for APIs that are "misuse-resistant" or "robust", and that do not sacrifice too much performance. This search accords with a long line of work that identifies the problem of API design as being critical for making it possible for developers to write secure cryptographic software (see [20, 22, 43] amongst others).

1.1 Our Contributions

Given this background, we set out to design a performant primality test that provides strong security guarantees across all use cases and that has the simplest possible API: it takes just one input, the number being tested for primality, and returns just one integer (or Boolean) indicating that the tested number is highly likely to be prime (1) or is definitely composite (0). We note that none of the many crypto libraries examined in [3] provide such an API.

We examine different options for the core of our test – whether to use many rounds of Miller-Rabin (MR) testing (up to 64 or 128, to achieve false positive rates of 2^{-128} or 2^{-256}, respectively), or to rely on a more complex primality test, such as the Baillie-PSW test [37] which combines MR testing with a Lucas test. Based on a combination of code simplicity, performance and guaranteed security, we opt for 64 rounds of MR as the core of our test.

We also study the performance impact of doing trial division prior to more expensive testing. This is common practice in primality testing code, with the idea being that one can trade fast but inaccurate trial division for much slower but more accurate number theoretic tests such as Miller-Rabin. For example, OpenSSL tests for divisibility using a fixed list of the first 2047 odd primes. We show that this is a sub-optimal choice when testing random inputs of common cryptographic sizes, and that the running time can be reduced substantially by doing trial division with fewer primes. The optimal amount of trial division to use depends on the size of the input being tested, though is not a new observation – see for example [23, 28, 29]. What is more surprising is that OpenSSL chooses so conservatively and with a fixed list of primes (independent of the input size). For example, with 1024-bit random, odd inputs, trial division using the first 128 odd primes already removes about 83% of candidates, while extending the list to 2047 primes, as OpenSSL does, only removes a further 5.5%. On average, it turns out to be faster to incur the cost of an MR test on that additional 5.5% than it is to do the full set of trial divisions.

The outcome of our analysis is a primality test whose performance on random, odd, 1024-bit inputs is on average 17% *faster* than the current OpenSSL test, but which guarantees that composites are identified with overwhelming probability ($1 - 2^{-128}$), no matter the input distribution. The downside is that, for inputs that are actually prime rather than random, our test is significantly slower than with OpenSSL's default settings (since we do 64 MR tests compared to the handful of tests used by OpenSSL). This is the price to be paid for a misuse-resistant API.

We then examine how our choice of primality test affects the performance of a crucial use case for primality testing, namely *generation* of random k-bit primes. OpenSSL already includes code for this. It makes use of a sieving step to perform trial division at reduced cost across many candidates, obviating the need to perform per-candidate trial division internally to the primality test. OpenSSL avoids the internal trial division via the above-mentioned `do_trial_division` input to the primality test in OpenSSL. Since we do not allow such an input in our simplified primality testing API, a developer using our API would be (implicitly) forced to do trial division on a per candidate basis, potentially increasing the cost of prime generation. Moreover, our primality test may use many more rounds of MR testing than OpenSSL selects in this case, since our API does not permit the user to vary the number of rounds according to the use case. However, for random prime generation, most candidates are rejected after just one MR test, and so the full cost of our test (trial division plus 64 rounds of MR testing) is only incurred once, when a prime is actually encountered. So we seek to understand the performance impact of plugging our new API and primality test into the existing OpenSSL prime generation code. We find that, for generation of random 1024-bit primes OpenSSL's prime generation code is 35-45% slower when using our primality test internally. For this cost, we gain an API for primality testing that is as simple as possible and where the test has strong security guarantees across *all* use cases.

We communicated our findings to the OpenSSL developers, and they have adopted our suggestions with only minor modifications: the forthcoming OpenSSL 3.0 (scheduled for release in Q4 of 2020) will include our simplified API for primality testing, and the OpenSSL codebase has been updated to use it almost everywhere (the exception is prime generation, which uses the old API in order to avoid redundant trial division). Moreover, OpenSSL will now always use our suggested primality test (64 rounds of MR) on all inputs up to 2048 bits, and 128 rounds of MR on larger inputs. This represents the first major reform of the primality testing code in OpenSSL for more than 20 years.

1.2 Related Work

The topic of API design for cryptography has a long history and connections to related fields such as usable security and API design for security more generally.

As early as 2002, Gutmann [22] identified the need to carefully define cryptographic APIs, recommending to "[p]rovide crypto functionality at the highest level possible in order to prevent users from injuring themselves and others through misuse of low-level crypto functions with properties they aren't aware of." This is precisely what we aim to do for primality testing in this paper.

Later, Wurster and van Oorschot [43] (in the broader context of security) argued that attention should be focussed on those developers who produce core functionality used by other developers, e.g. producers of APIs. They identified the need to design APIs which can be easily used in a secure fashion.

Green and Smith [20] extensively discuss the need for usable security APIs, and focus on cryptographic ones. They give an extensive list of requirements for good APIs, including: APIs should be easy to learn, even without cryptographic expertise; defaults should be safe and never ambiguous; APIs should be easy to use, even without documentation; APIs should be hard to misuse and incorrect use should lead to visible errors. These precepts have influenced our API design for primality testing.

Acar *et al.* [2] advocate for a research agenda for usable security and privacy research that focusses on developers rather than end users. This encompasses cryptography. Recent research related to this agenda and having a cryptographic focus includes [1, 15, 16, 19, 25, 33, 34].

Nonce-based Authenticated Encryption (AE), a primitive introduced by Rogaway [39], can be seen as an attempt to simplify the symmetric encryption API for developers, replacing the need to understand various requirements on IVs with the arguably simpler need to be able to supply unique (per key) inputs to an encryption algorithm. It has become the standard target for algorithm designers. However, as [10] showed, developers can accidentally misuse even this simplified API, with disastrous results for nonce-sensitive modes like AES-GCM. This motivated the development of misuse-resistant AE schemes, which attempt to preserve as much security as possible even when nonces are repeated. Prominent examples include SIV [40], Deoxys-II (part of the CAESAR competition final portfolio), and AES-GCM-SIV [21] (see also RFC 8452). Later authors identified the fact that developers may want an even higher-level API, for example a secure streaming channel like that provided by TLS [17, 35] or channels that tolerate some forms of reordering and repetition [11]; the mismatch between what developers want and what nonce-based AE can provide can lead to attacks, cf. [8].

Bernstein's design for DH key exchange on Curve25519 [6] deliberately presents a simple API for developers: public and private keys are represented by 32-byte strings, and the need for public key validation is avoided.

The NaCl crypto library [7] has provision of a simple API to developers as one of its primary aims. It gives the user a `crypto_box` function that encrypts and authenticates messages, with a simple API of the form: `c = crypto_box(m,n,pk,sk)`, where `m` is a message, `n` is a nonce, `pk` is the public key of the recipient and `sk` is the private key of the sender. Its security does rely on developers correctly handling nonces; we are unaware of reports of any misuse of this type. Some criticism of NaCl's approach, especially the way in which it breaks the developer's expected paradigm, can be found in [20].

There is an extensive literature on primality testing and generation, nicely summarised in [29, Chapter 4]. The state-of-the-art has not changed significantly since the publication of that book in 1996. On the other hand, as Albrecht *et al.* [3] showed, primality testing and generation as it is done in practice has many shortcomings. Our work can be seen as an effort to narrow the gap between the literature and its practical application.

1.3 Paper Organisation

The remainder of this paper is organised as follows. In Section 2 we give further background on primality testing and detail the approach used in OpenSSL. In Section 3 we describe four different candidate primality tests and analyse them theoretically and experimentally. Our chosen primality test (64 rounds of Miller-Rabin with trial division on the first 128 odd primes) emerges from this analysis as our preferred test. We then evaluate the performance of this chosen test in the use case of prime generation in Section 4. Section 5 briefly discusses how our test is being adopted in OpenSSL, while Section 6 contains our conclusions and avenues for future work.

2 FURTHER BACKGROUND

2.1 Primality Testing

We begin by giving further details on the core primality tests that we will consider in this work.

2.1.1 Miller-Rabin. The Miller-Rabin (MR) [31, 38] primality test is a widely-used and efficient algorithm.

A single round of the test proceeds as follows. Suppose $n > 1$ is an odd integer to be tested for primality. We first write $n = 2^e d + 1$ where d is odd. If n is prime, we know that there are are no non-trivial roots of unity modulo n, thus for any integer a with $1 \leq a < n$, we have:

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

The test then consists of choosing a value a (often referred to as a base), and then checking the above conditions on n. We declare a number to be (probably) prime if either of the two conditions hold and to be composite if both conditions fail. If n is composite and at least 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). It is evident that computational cost of the test is that of a full-size exponentiation modulo n.

In practice, the test is iterated t times, using a different, random choice of base a in each round (though as observed in [3], fixed bases are often used in crypto libraries, which makes it possible to construct composites that are always declared prime by the test). The test is probabilistic, in that a t-round MR test using uniformly random bases declares any composite number to be composite with probability at least $1 - 4^{-t}$. Moreover, this bound is tight: there are composites which are not identified as being such over t rounds of testing with probability 4^{-t}. Such numbers, then, are worst-case adversarial inputs for the test. They are treated extensively in [3]. On the other hand, the test *never* declares a prime to be composite.

The above discussion holds for any input n, no matter how it is chosen. When n is a uniformly random odd k-bit integer, much better performance can be assured. For example, a result of [14] assures that the probability $p_{k,1}$ that a composite n chosen in this way passes one round of random-base MR testing is bounded by $k^2 4^{2-\sqrt{k}}$. Thus, for $k = 1024$, we have $p_{k,1} \leq 2^{-40}$. Using more precise bounds from [14], this can be improved to $p_{k,1} \leq 2^{-42.35}$. These bounds are what motivates the rather small numbers of rounds of MR testing in the default setting in OpenSSL's primality test, for example.

2.1.2 Lucas. The Lucas primality test [5] makes use of Lucas sequences, defined as follows:

Definition 2.1 (Lucas sequence [5]). 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:

$$\begin{aligned} U_k &= PU_{k-1} - QU_{k-2} \quad \text{where,} \quad && U_0 = 0, U_1 = 1, \\ V_k &= PV_{k-1} - QV_{k-2} && V_0 = 2, V_1 = P. \end{aligned}$$

Since we are concerned with primality testing cryptographic sized numbers, we can use efficient techniques for computing large Lucas sequences such as binary Lucas chains as described in [32]. 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 2.2 ([13]). *Let P and Q be integers, $D = P^2 - 4Q$, and let the Lucas sequences (U_k), (V_k) be defined as above. If p is a prime with $\gcd(p, 2QD) = 1$, then*

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

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 2.3 (Lucas pseudoprime). Let P and Q be integers and $D = P^2 - 4Q$. Let n be a composite number such that $\gcd(n, 2QD) = 1$. If $U_{n-\left(\frac{D}{n}\right)} \equiv 0 \pmod{n}$, then n is called a *Lucas pseudoprime* with respect to parameters (P, Q).

Similar results to those for the MR primality test can be established for the Lucas test: a single Lucas test will declare a given composite number as being composite with probability at least $1 - (4/15)$ and as being prime with probability at most $(4/15)$, with these bounds being tight [4].

2.1.3 Baillie-PSW.

The Baillie-PSW test [37] is a deterministic primality test consisting of a single Miller-Rabin test with base 2 followed by a single Lucas test. A slight variant of the test in which the Lucas test is replaced with a more stringent version, known as a *strong* Lucas test is mentioned in [5]. Generally, the consensus that has emerged over time is that the Lucas test should be used with the parameters (P, Q) set as defined by Selfridge's method A:

Definition 2.4 (Selfridge's Method A [5]). 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 $Q = (1 - D)/4$.

If no such D can be found, then n must be a square and hence composite. In practice, one might attempt to find such a D up to some bound $D_{\max}$, then perform a test for squareness using Newton's method for square roots (see Appendix C.4 of [24]), before reverting to a search for a suitable D if needed. This is generally more efficient than doing a test of squareness first.

The idea of the Baillie-PSW test is that its two components are in some sense "orthogonal" and should between them catch all composites. Extensive computations have never produced a pseudo-prime for the Baillie-PSW test, that is, a composite number that passes it. Indeed there are (moderate) cash prizes available for providing one. However, none of these computations extend to numbers of cryptographic size. Moreover, Pomerance [36] has given a heuristic argument for the existence of infinitely many Baillie-PSW pseudo-primes. There do not appear to exist any bounds demonstrating the test's strength on uniformly random k-bit inputs, in contrast to the results of [14] for the MR test. In summary, while the Baillie-PSW test appears to be very strong, there are no proven guarantees concerning its accuracy. One positive feature is that, being deterministic, it does not consume any randomness (whereas a properly implemented MR test does).

2.1.4 Supplementary and Preliminary Tests.

It is often more efficient to perform some supplementary or preliminary testing on an input n before executing the main work of the primality test. A common strategy is to first perform trial division on n using a list of r small primes. This can be done directly, or by equivalently checking if $\gcd(\prod_i^r p_i, n) \neq 1$ where $\{p_1, \ldots, p_r\}$ is the list of primes used. The list of primes can be partitioned and multiple gcds computed, so as to match the partial products of primes with the machine word-size. This is a very cheap test to perform, and can be quite powerful when testing random inputs. The question arises of how r, the number of primes to use in trial division, should be set. We shall return to this question later.

2.1.5 Primality Testing in OpenSSL.

Since we will extensively compare our primality test and its API with those of OpenSSL, we give a detailed description of the approach found in OpenSSL (1.1.1c May 2019). We note that the specific parts of the code studied remain almost completely unchanged in subsequent versions up to 1.1.1e (March 2020), and across other long term support (LTS) versions of OpenSSL such as 1.1.0 and 1.0.2.

OpenSSL provides two functions for primality testing: `BN_is_prime_ex` and `BN_is_prime_fasttest_ex`, both in file `bn_prime.c`. The core part of the code is in the second of these, while the first simply acts as a wrapper to this function that forces omission of trial division. The second function call has the form:

```
int BN_is_prime_fasttest_ex(const BIGNUM *w, int
checks, BN_CTX *ctx_passed, int do_trial_division,
BN_GENCB *cb)
```

Here, `w` is the number being tested. The option to do trial division is defined via the `do_trial_division` flag. When set, the function will perform trial division using the first 2047 odd primes (excluding 2), with no gcd optimisations (the code also separately tests whether the number being tested is equal to 2 or 3, and whether it is odd). After this, the function calls `bn_miller_rabin_is_prime` to invoke the MR testing with pseudo-random bases. The number of MR rounds is set using the argument `checks`. When `checks` is set to `BN_prime_checks`, a value that defaults to zero, then the number of MR rounds is chosen such that the probability of the test declaring a random composite number n with k bits as being prime is at most $2^{-\lambda}$, where λ is the security level that a $2k$-bit RSA modulus should provide. Thus, the number of MR rounds performed is based on the bit-size k, as per Table 1. The entries here are based on average case error estimates taken from [29], which in turn references [14].

2.2 Prime Generation

A critical use case for primality testing is prime generation (e.g. for use in RSA keys). The exact details of the algorithms used vary across implementations, but the majority follow a simple technique based on first generating a random initial candidate n of the desired bit size k, possibly setting some of its bits, then doing trial division against a list of small primes, before performing multiple rounds of primality testing using a standard probabilistic primality test such as the MR test. If the trial division reveals a factor or the MR test fails, then another candidate is generated. This can be a fresh random value, but more commonly, implementations add 2 to the previous candidate n. This allows an important optimisation: if a

k	t	λ (bits)
$k \geq 3747$	3	192
$k \geq 1345$	4	128
$k \geq 476$	5	80
$k \geq 400$	6	80
$k \geq 347$	7	80
$k \geq 308$	8	80
$k \geq 55$	27	64
$k \geq 6$	34	64

Table 1: The default number of rounds t of Miller-Rabin performed by OpenSSL 1.1.1c when testing k-bit integers determined by the function BN_prime_checks_for_size and the associated bits of security λ.

table of remainders for the trial divisions of n is created in the first step, then this table of remainders can be quickly updated for the new candidate $n + 2$. Fresh divisions can then be avoided – one just needs to inspect the updated table of remainders. We refer to this procedure as *trial division by sieving* or just *sieving*. It is, of course, much more efficient than performing trial divisions anew for each candidate. Note that this approach leads to a slightly non-uniform distribution on primes: primes that are preceded by a long run of composites are more likely to result from it than primes that are close to their preceding primes. However, it is known that the deviation from the uniform distribution is small [12].

2.2.1 OpenSSL. OpenSSL adopts the above high-level procedure, with one important difference. The code is found in BN_generate_prime_ex in file bn_prime.c. The function call has the following form:

```
int BN_generate_prime_ex(BIGNUM *ret, int bits,
int safe, const BIGNUM *add, const BIGNUM *rem,
BN_GENCB *cb)
```

Here bits is the desired bit-size, safe is a flag that, when set, asks the function to produce a safe prime $p = 2q + 1$, and add and rem allow the callee to set additional conditions on the returned prime. We will ignore safe, add and rem in our further work; an analysis of how they affect prime generation when using our primality test is left to future work.

The initial steps are performed together in a separate function called probable_prime. A cryptographically strong pseudorandom number is first generated by BN_priv_rand. The two most significant bits and the least significant bit are then set to ensure the resulting candidate n is odd and of the desired bit-size. This number is then sieved using a hard-coded list of the first 2047 odd primes $p_2, \ldots, p_{2048}$, so $p_1 = 2, p_2 = 3, \ldots, p_{2048} = 17863$. If a candidate passes the sieving stage, it is tested for primality by BN_is_prime_fasttest_ex. This function carries out the default number of Miller-Rabin rounds, as per Table 1. Trial division is omitted by setting the do_trial_division flag in the function call. This is because trial division has already been carried out externally via sieving. This exploits the complexity of the OpenSSL API for primality testing to gain performance, an option not available if a simplified API is desired (as we do). Importantly, if the MR tests fail, then instead of going to the next candidate that passes sieving, a fresh, random starting point is selected and the procedure begins again from the start.

3 CONSTRUCTION AND ANALYSIS OF A PRIMALITY TEST WITH A MISUSE-RESISTANT API

We now propose how to construct a performant primality test with a misuse-resistant API. Our design goal is to ensure good performance in the most important use cases (malicious input testing, prime generation) while still maintaining strong security. At the same time, we want the simplest possible API for developers: a single input n (the number being tested) and single a 1-bit output (0 for composite, 1 for probably prime).

We propose four different primality testing functions, all built from the algorithms described in Section 2.1. The first of these follows OpenSSL with its default settings, and we name this Miller-Rabin Average Case (MRAC). It provides a baseline for analysis and comparison. The second and third use 64 and 128 rounds of MR testing, respectively. We name them MR64 and MR128. The fourth uses the Baillie-PSW test, and we name it BPSW for short. For each of these four options, we provide an assessment (both by analysis and by simulation) of its security and performance when considering random composite, random prime, and adversarially generated composite inputs. We also consider the influence of trial division on each test's performance. For concreteness, throughout we focus on the case of 1024-bit inputs, but of course the results are easily extended to other bit-sizes.

3.1 Miller-Rabin Average Case (MRAC)

The first test we introduce, MRAC, is a reference implementation of OpenSSL's primality test, as per the function BN_is_prime_fasttest_ex described in Section 2.1.5 with input checks set to BN_prime_checks, so that the number of MR rounds performed is based on the bit-size k, as per Table 1. Recall that this function either does no trial division or does trial division with the first 2047 odd primes. Of course, this test is quite unsuitable for use in general, because it performs badly on adversarial inputs: [3] showed that it has a worst case false positive rate of $1/2^{2t}$ where for example $t = 5$ for 1024-bit inputs. On the other hand, it is designed to perform well on random inputs.

3.1.1 MRAC on Random Input. We now consider the expected number of MR rounds performed when receiving a random 1024 bit odd input. For now, we ignore the effect of trial division. The probability that a randomly chosen odd k-bit integer is prime is $q_k := 2/\ln(2^k)$ by standard estimates for the density of primes [42] (for $k = 1024, q_k \approx 1/355$). In this case MRAC will do t MR rounds, as per Table 1. Otherwise, for composite input, up to t rounds of MR testing will be done. One could use the bounds from [14] to obtain bounds on the expected number of MR rounds that would be carried out on composite input. However, for numbers of cryptographic size (e.g. $k = 1024$ bits), to a very good approximation, the number needed is just 1, since with very high probability, a single MR test is sufficient to identify a composite (recall that the probability that a single round of MR testing fails to identify a 1024-bit composite is less than 2^{-40}). From this, one can compute the expected number of rounds needed for a random, odd input: it is approximately the weighted sum $t \cdot q_k + 1 \cdot (1 - q_k) = 1 + (t-1)q_k$. For $k = 1024$, we have $t = 5$ and $q_k = 0.0028$, and this expression evaluates to 1.026.

3.1.2 MRAC on Random Input with Trial Division. Now we bring trial division into the picture. Its overall effectiveness will be determined by the collection of small primes in the list $P = \{p_1, p_2, \ldots, p_r\}$ used in the process (where we assume all the p_i are odd) and the relative costs of MR testing and trial division (about 800:1 in our experiments).

For random odd inputs, the fraction $\sigma(P)$ of non-prime candidates that are removed by the trial division by the primes in P can be computed using the formula:

$$\sigma(P) = 1 - \prod_{i=1}^{r} \left(1 - \frac{1}{p_i}\right). \tag{2}$$

This follows easily by noting that a fraction $1 - \frac{1}{p_i}$ of integers are not divisible by p_i, so the probability that a randomly sampled integer is *not* divisible by any of the p_i is $\prod_{i=1}^{r} \left(1 - \frac{1}{p_i}\right)$, and hence the probability that a randomly sampled odd integer is divisible by at least one p_i is $\sigma(P)$. In turn, this means that any candidate that passes the trial division stage is $1/(1 - \sigma(P))$ times more likely to be a prime than an odd candidate of equivalent bit-size chosen at random (this is because a fraction $1 - \sigma(P)$ of integers remain after sieving, and all primes survive sieving).

But simply adding more primes to the list P is not necessarily effective: fewer additional composites are removed at a fixed cost (one additional trial division per prime), and eventually it is better to move on to a more heavyweight test (such as rounds of MR testing). Moreover, from inspecting the formula for $\sigma(P)$, it is evident that, for a given size r of set P (and hence a given cost for trial division), it is better to set P as containing the r smallest odd primes (including 2 is not useful as the input n is already assumed to be odd). Henceforth, we assume that when P is of size r, then it consists of the first r odd primes. We write σ_r in place of $\sigma(P)$ in this case. Using Mertens' theorem [30], we can approximate σ_r as follows:

$$\sigma_r \approx 1 - 2e^{-\gamma} / \ln(p_r). \tag{3}$$

where $\gamma = 0.5772\ldots$ is the Euler-Mascheroni constant.

As an example, `BN_is_prime_fasttest_ex` in OpenSSL performs trial division on the first 2047 odd primes (ending at $p_{2047} = 17863$). As shown in Figure 1, using the first $r = 2047$ primes gives a value of $\sigma_{2047} = 0.885$. This is only a little larger than using, say, the $r = 128$ smallest odd primes yielding $\sigma_{128} = 0.831$.

Now we build a cost model for MRAC including trial division. This will also be applicable (with small modifications) for our other tests.

Let C_i denote the cost of a trial division for prime p_i and let C_{MR} denote the cost of a single MR test.[4] Then the total cost of MRAC on random *prime* k-bit inputs is:

$$\sum_{i=1}^{r} C_i + t \cdot C_{MR} \tag{4}$$

since the test then always performs all r trial divisions (assuming k is large enough) and all t MR tests. For random, odd *composite* inputs, the average cost is approximately:

$$\sigma_1 \cdot C_1 + (\sigma_2 - \sigma_1) \cdot (C_1 + C_2) + \ldots + (\sigma_r - \sigma_{r-1}) \cdot (C_1 + \cdots + C_r)$$
$$+ (1 - \sigma_r) \cdot (\sum_{i=1}^{r} C_i + C_{MR}). \tag{5}$$

This is because a fraction σ_1 of the composites are identified by the first trial division, a further fraction $\sigma_2 - \sigma_1$ are identified after 2 trial divisions, etc, while a fraction $(1 - \sigma_r)$ require all r trial divisions plus (roughly) 1 round of MR. Here we assume that the MR test performs in the same way on numbers after trial division as it does before. After some manipulation, this last expression can be simplified to:

$$\sum_{i=1}^{r} (1 - \sigma_{i-1}) \cdot C_i + (1 - \sigma_r) \cdot C_{MR} \tag{6}$$

where we set $\sigma_0 = 0$. This expression can be simplified further if we assume that the C_i are all equal to some C_{TD} (a good approximation in practice), and apply Mertens' theorem again. For details, see the equivalent analysis in [28].

From expressions (4) and (6), the expected cost for random, odd, k-bit input can be easily computed via a weighted sum with weights q_k and $1 - q_k$. However, the cost is dominated by expression (6) for the composite case. From (6), the futility of trial division with many primes is revealed: adding a prime by going from r to $r + 1$ on average adds a term $(1 - \sigma_r) \cdot C_{r+1}$, but only decreases by a fraction $\sigma_{r+1} - \sigma_r$ the term in front of C_{MR}. As can be seen from Figure 1, when r is large, $1 - \sigma_r$ is around 0.1, while $\sigma_{r+1} - \sigma_r$ becomes very small. So each increment in r only serves to increase the average cost by a fraction of a trial division (and with the cost of trial division increasing with r).

Figure 2 shows a sample (theoretical) plot of the average cost of MRAC as a function of r for $k = 1024$. This uses as costs $C_{TD} = 0.000371$ms and $C_{MR} = 0.298$ms obtained from our experiments (reported below) for $k = 1024$ and the weighted sum of expressions (4), (6). This curve broadly confirms the analysis of [28] which suggests setting $p_r = C_{MR}/C_{TD}$ to minimise the running time of primality testing with trial division; here we obtain $C_{MR}/C_{TD} \approx 800$, corresponding to $r \approx 140$.[5]

3.1.3 MRAC on Adversarial Input. Recall from [3] that worst-case adversarial inputs can fool random-base MR testing with probability 1/4 per round. The expected number of rounds needed to identify such inputs as composite is then 1.33. However, with t rounds of testing, MRAC will fail to identify such composites as being so with probability $1/2^{2t}$ (and will indicate that the input was prime). Note that this analysis is unaffected by trial division, since the adversarial inputs used have no small primes factors – the trial division just increases the running time of the test.

[4] In practice, we could set C_i to be a constant C_{TD} for the range of i we are interested in, but using a more refined approach is not mathematically much more complex.

[5] The analysis of [28] technically applies to prime generation, but ignores certain terms in such a way as to actually analyse the cost of primality testing of composite numbers. In this sense, it is only valid when the cost of primality testing for *prime* inputs can be ignored compared to the case of composite inputs; this is not the case in general, but is a reasonable approximation for MRAC.

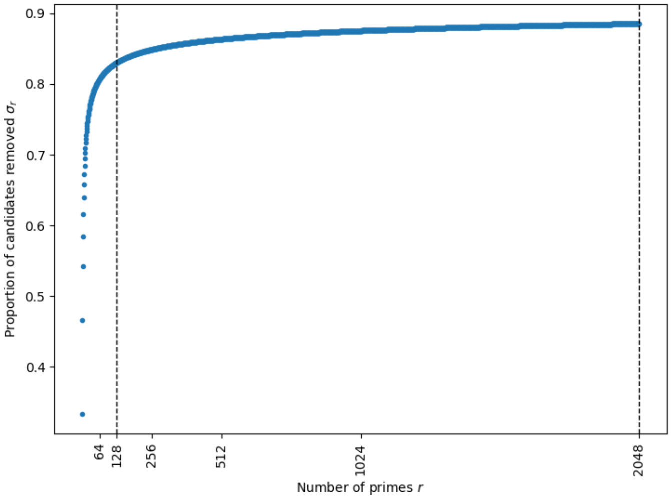

Figure 1: Proportion of candidates removed by trial division, σ_r, as a function of r, the number of primes used.

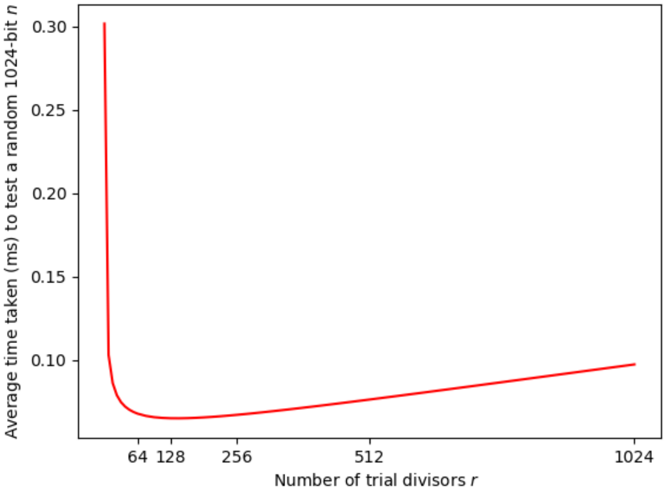

Figure 2: A plot of the theoretical running time of MRAC as a function of r, the number of primes r used in trial division for $k = 1024$, using $C_{TD} = 0.000371$ms and $C_{MR} = 0.298$ms obtained from our experiments.

3.2 Miller-Rabin 64 (MR64)

Next we consider trial division followed by up to 64 rounds of MR testing with random bases (the test will exit early if a base that is a witness to compositeness of the input n is found). We refer to this test as MR64. By design, this test guarantees a failure probability of at most 2^{-128}, no matter the input distribution, so it offers robust security guarantees without the user needing to understand the context of the test (i.e. whether the test is being done with adversarial inputs or not).

3.2.1 MR64 on Random Input. As for MRAC, for a random, odd composite, k-bit input, the expected number of rounds of MR testing (without trial division) is very close to 1. On the other hand, for prime, k-bit input, the number of rounds is exactly 64. This enables the average cost without trial division on random, odd, k-bit input to be computed: it is approximately given by the weighted sum

$$(64 \cdot q_k + 1 \cdot (1 - q_k)) \cdot C_{MR} = (1 + 63q_k) \cdot C_{MR} \qquad (7)$$

For $k = 1024$, we again have $q_k = 2/\ln(2^k) = 0.0028$, and this sum evaluates to $1.18C_{MR}$, about 17% higher than MRAC for the same input distribution.

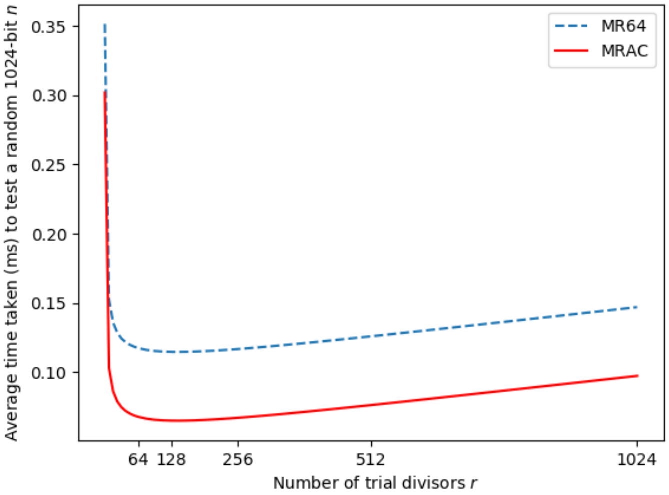

Figure 3: Comparing the theoretical running time of MR64 and MRAC as a function of r (the number of primes r used in trial division) for $k = 1024$, using $C_{TD} = 0.000371$ms and $C_{MR} = 0.298$ms obtained from our experiments.

3.2.2 MR64 on Random Input with Trial Division. Following the analysis for MRAC, we can compute the cost of MR64 on random, *prime*, k-bit input as:

$$\sum_{i=1}^{r} C_i + 64 \cdot C_{MR} \qquad (8)$$

since here all trial divisions are performed, together with 64 rounds of MR testing. For random, odd, *composite* input with r-prime trial division, the expected cost is very close to that of MRAC with the same r, since whenever MR testing is invoked, almost always one round suffices. As for the case of MR64 without trial division, it is the prime inputs that make the cost difference here: they involve 64 rounds of MR testing instead of the (close to) 1 needed for composite inputs. Again, a theoretical prediction for random, odd input can be made by combining the expressions for odd, composite and prime input using a weighted sum. We omit the details, but Figure 3 shows the theoretical curve for MR64 as compared to MRAC (using costs $C_{TD} = 0.000371$ms and $C_{MR} = 0.298$ms for $k = 1024$ as before).

3.2.3 MR64 on Adversarial Input. By design, the MR64 test will fail to identify a worst-case adversarial input as a composite with probability at most 2^{-128}, this after 64 rounds of MR testing. The *expected* number of rounds needed to successfully classify such inputs is again 1.33.

3.3 Miller-Rabin 128 (MR128)

This test is identical to MR64, but up to 128 rounds of MR testing are invoked. The intention is to reduce the false positive rate from 2^{-128} to 2^{-256}. The analysis is almost identical to that for MR64, replacing 64 by 128 where it appears in the relevant formulae. We include it for comparison purposes and because the OpenSSL documentation does target 256 bits of security when testing very large numbers (larger than 6394 bits in size[6]). The headline figure for this test is its expected cost (without trial division) of $(1 + 127q_k) \cdot C_{MR}$, equating to $1.36 \cdot C_{MR}$ on random, odd, 1024-bit inputs, roughly 35% higher than MRAC at the same input size.

3.4 Baillie-PSW (BPSW)

The final test we consider is the Baillie-PSW test. Recall that this is the combination of a single Miller-Rabin test to base 2, with a Lucas test using Selfridge's Method A to select D. If the input n we are testing is a perfect square, then there does not exist a valid choice of D (see Section 2.1.2). So we must decide upon a point to test for this. Baillie and Wagstaff [5] show that, when n is not square, the average number of D values that need to be tried until a suitable one is found is 1.78. We choose to run a test to check if n is a perfect square only after 7 unsuccessful attempts to select D. This choice is inspired by other implementations [27] and provides a balance between the relatively cheap process of testing a choice of D with the more expensive test for n being a perfect square. We perform the Miller-Rabin part of the test first, since it is the more efficient of the two techniques, omitting the Lucas test early if this indicates compositeness. We then search for D using Selfridge's Method A, using it to carry out a Lucas test if found. We abort the search for D after 7 attempts and then test n for being a perfect square. If this test fails, we revert to searching for a suitable D and then perform the Lucas test when one is eventually found.

3.4.1 BPSW on Random Input. The analysis without trial division is much like that of MRAC, assuming that MR with a fixed base 2 performs as well as MR with a random base when the number being tested is uniformly random. For prime inputs, the average cost is $C_{MR} + C_L$, where C_L is average the cost of doing the Lucas part of the test (and any tests of squareness); for composite inputs, the cost is roughly C_{MR} since the MR test catches the vast majority of composites. The performance on random inputs is the weighted sum of these, as usual. In our implementation, the average for C_L for 1024-bit inputs is equal to $17.04 \cdot C_{MR}$ (5.078ms compared to 0.298ms on average for 1024-bit inputs, based on 2^{20} trials). Overall, then, this test has an expected cost (without trial division) of $1.05 \cdot C_{MR}$ on random, odd, 1024-bit inputs, roughly 4% more than MRAC.

The analysis with trial division is again similar to that for MRAC: when the input is prime, the average cost is $\sum_{i=1}^{r} C_i + C_{MR} + C_L$, while when the input is composite, it is of the same form as in (6) (where we are able to omit a term C_L under the assumption that the base 2 MR test is effective in detecting composites). We omit further detail.

3.4.2 BPSW on Adversarial Input. It is relatively easy to construct composites passing a base 2 MR test. For example, integers of the form $(2x + 1)(4x + 1)$ with each factor a prime have a roughly 1 in 4 chance of doing so (see [3] for further discussion). Such inputs are highly likely to be detected by the Lucas part of the BPSW test, so the cost of BPSW on such inputs would be $\sum_{i=1}^{r} C_i + C_{MR} + C_L$. However, we do not know if such numbers are worst-case adversarial inputs for BPSW, and indeed, we cannot rule out the existence of BPSW pseudo-primes, that is, composites which are declared probably prime by the test. Recall that Pomerance [36] has given heuristic evidence that there are infinitely many such pseudo-primes. Perhaps the smallest is beyond the bit-size we care about in cryptographic applications, but we cannot be sure. Note also that such a pseudo-prime, if it can be found, would always fool the BPSW test (because the test is deterministic). This is in sharp contrast to MR64 and MR128, where we can give precise bounds on the false positive rate of the tests. We consider this, along with the relative complexity of implementing the BPSW test, to be a major drawback.

3.5 Experimental Results

Having described our four chosen primality tests and given a theoretical evaluation of them, we now turn to experimental analysis. This analysis gives us a direct comparison with the current approach of OpenSSL (MRAC with trial division either off or based on 2047 primes). It also allows us to study how the Baillie-PSW test performs against Miller-Rabin testing in practice, something that does not appear to have been explored before.

3.5.1 Random Input. Our results for random, odd, 512-bit, 1024-bit and 2048-bit inputs to the tests are shown in Table 2. We worked with 2^{25} inputs at each bit size, produced using OpenSSL's internal random number generator. All timings are in milliseconds, and are broken down into results for composite inputs, inputs that were declared prime, and overall results. We also report results for different amounts of trial division — none, $r \in \{64, 128, 384\}$ (which, from our theoretical analysis above, we consider to be a sensible amount of trial division for the differently-sized inputs) and $r = 2047$ (as in OpenSSL). All results were obtained using a single core of a Intel(R) Xeon(R) CPU E5-2690 v4 @ 3.20GHz processor, with code written in C using OpenSSL `1.1.1b` (26-Feb-2019) for big-number arithmetic and basic Miller-Rabin functionality. We also computed standard deviations to accompany each timing, but omit the details due to lack of space.

Of the 2^{25} random, odd, 1024-bit numbers that we generated, 94947 were prime. This is closely in line with the estimated $q_{1024} \times 2^{25} \approx 94548$ given by the usual density estimate.

The results in Table 2 are broadly in-line with our earlier theoretical analysis. Some highlights, focussing on 1024-bit inputs:

- MRAC is fast overall, but with $r = 2047$, OpenSSL is doing far too much trial division on 1024-bit inputs. Much better performance could be achieved for this input size in OpenSSL by setting $r = 128$ (more than 2x speed-up overall can be gained).
- MR64 is 8-9 times slower than MRAC on prime input, reflecting the many more rounds of MR testing being done in MR64.

[6] See the man page https://www.openssl.org/docs/man1.1.0/man3/BN_is_prime_fasttest_ex.html and code documentation https://github.com/openssl/openssl/blob/fa4d419c25c07b49789df96b32c4a1a85a984fa1/include/openssl/bn.h#L159.

r	Declared Composite			
	MRAC	MR64	MR128	BPSW
0	0.085	0.085	0.085	0.079
64	0.020	0.020	0.020	0.020
2047	0.067	0.067	0.067	0.067

r	Declared Prime			
	MRAC	MR64	MR128	BPSW
0	0.375	4.65	9.29	2.11
64	0.389	4.67	9.31	2.12
2047	0.818	5.10	9.73	2.55

r	Overall			
	MRAC	MR64	MR128	BPSW
0	0.086	0.111	0.137	0.091
64	0.022	0.046	0.073	0.031
2047	0.071	0.095	0.121	0.081

(a) 512-bit

r	Declared Composite			
	MRAC	MR64	MR128	BPSW
0	0.312	0.313	0.312	0.302
128	0.063	0.063	0.063	0.061
2047	0.135	0.134	0.134	0.133

r	Declared Prime			
	MRAC	MR64	MR128	BPSW
0	1.50	19.1	38.1	5.39
128	1.55	19.1	38.2	5.44
2047	2.26	19.8	38.9	6.15

r	Overall			
	MRAC	MR64	MR128	BPSW
0	0.315	0.366	0.419	0.316
128	0.067	0.117	0.170	0.077
2047	0.141	0.190	0.244	0.150

(b) 1024-bit

r	Declared Composite			
	MRAC	MR64	MR128	BPSW
0	2.40	2.40	2.40	2.39
384	0.401	0.401	0.402	0.401
2047	0.523	0.523	0.523	0.522

r	Declared Prime			
	MRAC	MR64	MR128	BPSW
0	9.55	152.6	305.2	41.6
384	9.87	152.2	304.0	41.9
2047	11.4	153.3	304.8	43.5

r	Overall			
	MRAC	MR64	MR128	BPSW
0	2.41	2.61	2.83	2.45
384	0.414	0.614	0.827	0.459
2047	0.538	0.737	0.948	0.582

(c) 2048-bit

Table 2: The mean running time (in ms) for each test when testing MRAC, MR64, MR128 and BPSW for random (a) 512-bit, (b) 1024-bit, and (c) 2048-bit odd inputs and various amounts of trial division (r). We show the breakdown of means for inputs declared as either prime or composite, as well as the overall averages. Results based on 2^{25} trials.

- MR128 is roughly twice as slow as MR64 on prime input (reflecting the doubling of rounds of MR testing). On random input, the gap between MR64 and MR128 is not so large (because most composites are identified by trial division or after just one round of MR testing).
- BPSW is quite competitive with MRAC overall and only 2-3 times slower for prime input. This is because the Lucas test part of BPSW is expensive but rarely invoked for random input, but always done for prime input.
- Based on overall figures, MR64 with $r = 128$ outperforms MRAC with $r = 2047$ (as used in OpenSSL) by 17% on 1024-bit input (and by 54% on 512-bit input with $r = 64$). This indicates that, by tuning parameters carefully, it is possible to obtain improved performance over the current approach used in OpenSSL whilst enjoying strong security across all use cases (i.e. a guaranteed false positive rate of 2^{-128}).

Further improvements in running time can be obtained by fine-tuning the value of r on a per test basis, and according to input size. Importantly, the latter is feasible even with a simple API (and indeed seems to be the only general, input-dependent optimisation possible). To illustrate this, we show in Figure 4 the average running times for MRAC and MR64 on random, odd, 1024-bit input for varying r. The figure also shows the theoretical curves obtained previously. There is excellent agreement between the experimental data and the curves obtained from the model. In both cases, the curve is quite flat around its minimum, but we see that using $r = 128$ gives close to optimal performance for this value of $k = 1024$. The figure also illustrates that using large amounts of trial division (as per OpenSSL) harms performance for this input size, as was also explained theoretically in Section 3.1. Specifically, OpenSSL uses $r = 2047$, putting its performance with default settings (MRAC) well above the minimum obtainable with MR64 with a carefully tuned choice of r.

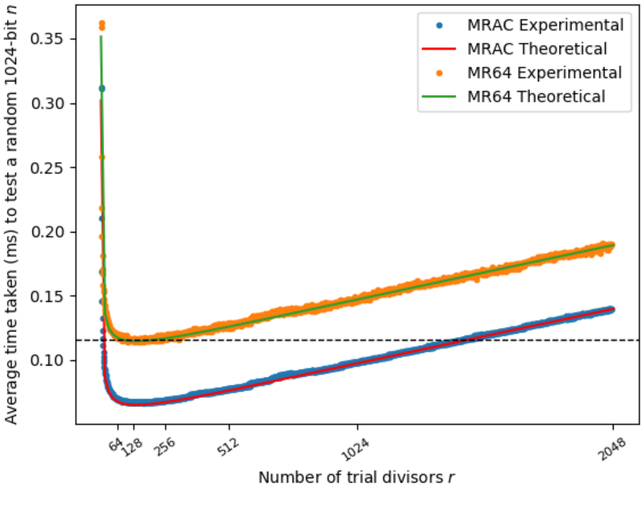

Figure 4: Experimental and theoretical performance of MRAC and MR64 on random, odd, 1024-bit input for varying amounts of trial division, r. The horizontal dashed line represents the minimum of the average running time of MR64 across all choices of r. This gives a visual representation of the comparison between MR64 with $r = 128$ and MRAC with $r = 2047$.

Rounds	MRAC	MR64
1	787054	786765
2	196110	196268
3	49167	49305
4	12157	12103
5	4088	3129
6	–	776
7	–	169
8	–	44
9	–	13
10	–	4

Table 3: Number of rounds of MR testing needed to identify as composite 1024-bit numbers of the form $n = (2x+1)(4x+1)$ with $2x + 1$, $4x + 1$ prime from an initial set of 2^{20} candidates. MRAC only performs 5 rounds of MR testing for this bit-size and failed to identify exactly 1000 candidates.

3.5.2 Adversarial Input. To bring into sharp relief the failings of MRAC as a general-purpose primality test, we generated a set of 2^{20} 1024-bit composites of the form $n = (2x + 1)(4x + 1)$ in which the factors $2x + 1$, $4x + 1$ are both prime. Numbers of this special form are known to pass random-base MR tests with probability 1/4. We then put these n through our MRAC and MR64 tests without trial division,[7] tracking how many rounds of MR were used on each input by each test. Table 3 shows the results. MR64 needed a maximum of 10 rounds of MR testing to correctly classify all the inputs, while MRAC, using only 5 rounds of MR for inputs of this size, incorrectly classified exactly 1000 of the inputs. This performance is in-line with expectations, as the expected number of misclassifications is $2^{20} \times (1/4)^5 = 2^{10}$.

3.6 Other Bit Sizes

So far in our experimental evaluation, we have focussed on $k = 1024$, i.e. testing of 1024-bit inputs. We have carried out similar testing also for $k = 512, 2048, 3072$. Figures 5, 6 and 7 show these additional results for the MRAC and MR64 tests, focussing on the effect of varying r on running time. Notice the characteristic "hockey-stick" shape of the curves in all the figures.

In each figure, the dashed horizontal time highlights the minimum running time for MR64. Notably, for $k = 512$, this is significantly lower than MRAC with $r = 2047$ (as in OpenSSL). We saw the same effect for $k = 1024$ in Figure 4. For $k = 2048$, MR64 with the best choice of r is slightly slower than MRAC with $r = 2047$ (but still competitive). For $k = 3072$, the influence of r on running time is quite small, and MRAC consistently comes out ahead of MR64 (but recall that MRAC is unsafe for maliciously chosen inputs).

These experiments confirm our earlier observation: the choice of r, the amount of trial division, can have a significant effect on running time of primality tests, and should be taken into account when selecting a test.

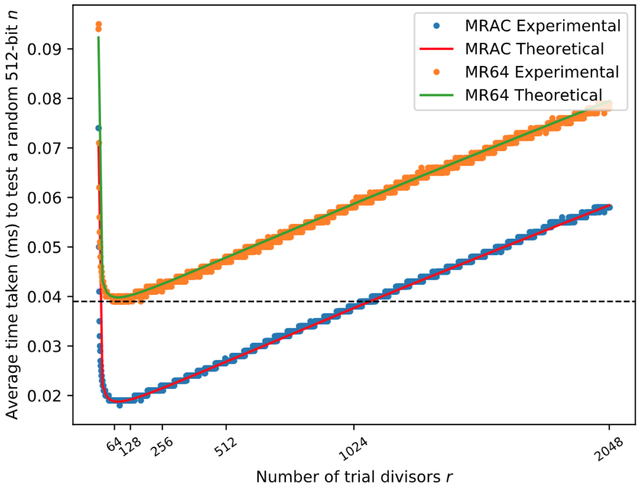

Figure 5: Experimental and theoretical performance of MRAC and MR64 on random, odd, 512-bit input for varying amounts of trial division, r.

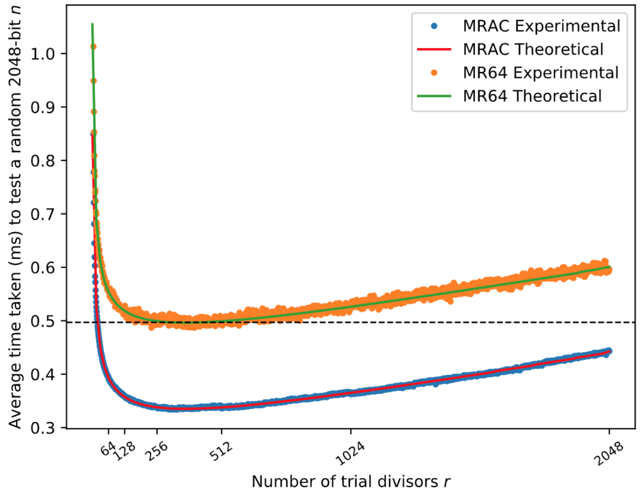

Figure 6: Experimental and theoretical performance of MRAC and MR64 on random, odd, 2048-bit input for varying amounts of trial division, r.

3.7 Selecting a Primality Test

We select MR64 with the amount of trial division, r, depending on the input size as our preferred primality test. Our reasons are as follows:

- MR64 has strong security guarantees across all use cases (unlike MRAC and BPSW). These guarantees can be improved by switching to MR128, but we consider the guarantees of MR64 to be sufficient for perhaps all but the most stringent requirements.
- MR64 is easy to implement, while a test like BPSW requires significant additional code (see Appendix B).

[7] Including trial division would not change the results.

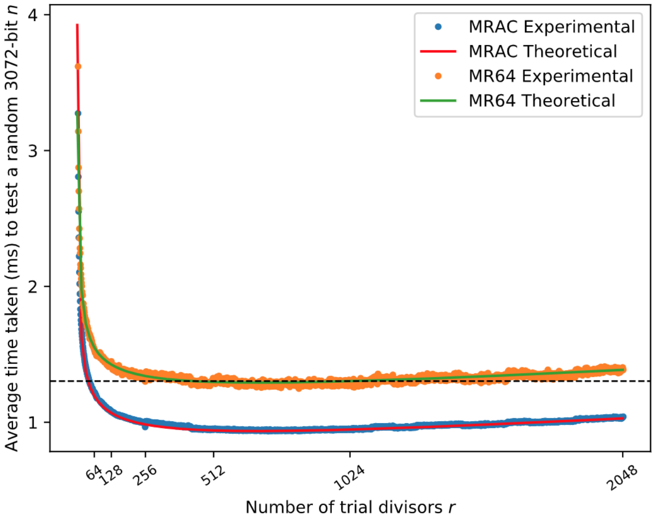

Figure 7: Experimental and theoretical performance of MRAC and MR64 on random, odd, 3072-bit input for varying amounts of trial division, r.

k	r
$k \in [1, 512]$	64
$k \in [513, 1024]$	128
$k \in [1025, 2048]$	384
$k \in [2049, 3072]$	768
$k \in [3073, \infty)$	1024

Table 4: Recommended values of r for use with the MR64 primality test.

k	r used	MR64	MRAC	Overhead
512	64	12.37	8.859	40%
1024	128	60.83	45.20	35%
2048	384	385.2	268.5	43%
3072	768	1379	946.7	46%

Table 5: Running time (in ms) for prime generation using our proposed primality test (MR64 with input-length-dependent trial division) and current OpenSSL primality test (MRAC with no trial division). Each timing is based on 2^{20} trials.

- MR64 with an input-size-dependent choice of r outperforms the current approach used in OpenSSL (MRAC with fixed $r = 2047$) up to $k = 1024$ and remains competitive with MRAC even for larger inputs. (Obviously OpenSSL could also be made faster by tuning r, but this would not improve security for malicious inputs).
- MR64 permits a very simple API, with a single input (the number being tested) and a single output (whether the input was composite or probably prime), whilst still allowing input-size-dependent tuning of r.

Table 4 shows our recommended values of r to use with MR64, based on the experimental results obtained above. Further small improvements in performance could be obtained by being more precise in setting r values and by further partitioning the set of k values, but the gains would be marginal.

We further validate this selection of MR64 in the next section, where we examine the performance of different tests when used as part of prime *generation* (as opposed to testing).

4 PRIME GENERATION

In this section, we want to assess the impact of our choice of primality test on a key use case, prime generation. We focus on the scenario where our primality test is used as a drop-in replacement for the existing primality test in OpenSSL, without making any modifications to the prime generation code. We are not suggesting this should be done in practice, but merely evaluating the impact of switching to our proposed test in a strawman application.

4.1 Experimental Approach

In order to establish a benchmark, we first use OpenSSL's prime number generating function `BN_generate_prime_ex` as it appears in the standard library. As discussed in detail in Section 2.2, this involves sieving with $s = 2047$ primes and using the OpenSSL primality test that consumes t rounds of MR testing on a sequence of candidates $n, n + 2, \ldots$, restarting the procedure from scratch whenever an MR test fails. Here t is determined as in Table 1 (i.e. the test is what we call MRAC). Importantly, OpenSSL exploits the rich API of its primality test to switch off trial division in the primality tests, since that trial division is already taken care of by the cheaper sieving step.

Next, we change the underlying primality test to use our selected test: MR64 with input-length-dependent trial division (as per Table 4), keeping all other aspects of OpenSSL's prime generation procedure the same. All the trial division done in our underlying primality test is of course redundant, because of the sieving step carried out in OpenSSL's prime generation code. However, with our deliberately simplified API for primality testing, that extra work would be unavoidable. Similarly, our underlying primality test performs more rounds of MR testing (64 instead of the 3-5 used in MRAC) when a prime is finally encountered. It is the amount of this extra work that we seek to quantify here.

Our experimental results are shown in Table 5. It can be seen that the overhead of switching to our primality test in this use case ranges between 35% and 46%. This is a significant cost for this use case, but recall that the gain is a primality test that has strong security guarantees across all use cases, along with a simple and developer-friendly API.

We can build simple cost models which illustrate the performance differences we have observed; see also [28] for a similar model. Details are deferred to Appendix A

5 IMPLEMENTATION AND INTEGRATION IN OPENSSL

We communicated our findings to the OpenSSL development team, specifically to Kurt Roeckx, one of the OpenSSL core developers. He did his own performance testing, and concluded that our new API and primality test should be deployed in OpenSSL. In personal communication with Roeckx, we were informed that these changes are slated for inclusion in OpenSSL 3.0, which is scheduled for release in Q4 of 2020.

In more detail, the following changes were made:

- Our proposed API is included via a new, external facing function (see https://github.com/openssl/openssl/blob/master/crypto/bn/bn_prime.c#L253):

  ```
  int BN_check_prime(const BIGNUM *p, BN_CTX *ctx,
    BN_GENCB *cb)
  {
      return bn_check_prime_int(p, 0, ctx, 1, cb);
  }
  ```

  This code wraps the existing "internal" primality testing function
  bn_check_prime_int. Note that the API has 3 parameters, instead of our desired 1: OpenSSL still needs to pass pointers to context and callback objects for programmatic reasons.
- The "internal" primality testing function bn_check_prime_int has been updated to do a minimum of 64 rounds of MR testing (and 128 rounds for 2048+ bit inputs). This deviates slightly from our recommendation to always do 64 rounds of testing – it is more conservative. Note that the average case analysis of [14] is no longer used to set the number of rounds of MR testing in the default case. This function also uses a small table to determine how many primes to use in trial division; the numbers are aligned with our recommendations in Table 4. Details are in the new function calc_trial_divisions.[8]
- The rest of the OpenSSL codebase has been updated to use the new API, except for the prime generation code. That code has also been updated (see https://github.com/openssl/openssl/blob/master/crypto/bn/bn_prime.c#L123). It now uses yet a third internal function for its primality testing (see bn_prime.c#L170):

  ```
          bn_is_prime_int(ret, checks, ctx, 0, cb);
  ```

  Here, checks determines the number of rounds of MR testing done, and is set to either 64 or 128 according to the input size. In the call, "0" indicates that trial division is no longer done. The number of MR rounds here could have been set based on average case performance, as was formerly the case, rather than worst case, but it seems the OpenSSL developers have opted for simplicity over performance. Not doing trial division inside the primality test is appropriate here because the inputs have already been sieved to remove numbers with small prime factors by this point.
- The "old" and complex external-facing APIs in the functions BN_is_prime_ex and BN_is_prime_fasttest_ex have been marked for deprecation in OpenSSL 3.0: they will only be included in a build of the library in case the environmental variable OPENSSL_NO_DEPRECATED_3_0 is set.[9]

5.1 Reference Implementation of Baillie-PSW

For completeness, in Appendix B, we give a reference implementation of the Baillie-PSW test as it could be implemented in OpenSSL. This also helps to provide an understanding of the increase in code complexity involved in using this test.

6 CONCLUSIONS AND FUTURE WORK

We have proposed a primality test that is both performant and misuse-resistant, in the sense of presenting a simplest-possible interface for developers. The test balances code simplicity, performance, and security guarantees across all use cases. We have not seen a detailed treatment of this fundamental problem in the literature before, despite the by-now classical nature of primality testing as a cryptographic task. Our recommendations – both for the API and for the underlying primality test – have been adopted in full by OpenSSL and are scheduled for inclusion in OpenSSL 3.0, which is expected to be released in Q4 2020.[10]

We have focussed in this work on regular prime generation. Our work could be extended to consider efficiency of safe-prime generation. Special sieving procedures can be used in this case: if one creates a table of values $n \bmod p_i$, then one can also test $2n + 1$ for divisibility by each of the p_i very cheaply; techniques like this were used in [18] in a slightly different context. Further work is also needed to fully assess the impact of the amount of sieving (s) on the performance of prime generation at different input lengths (k). Our work could also be extended to make a systematic study of prime generation code in different cryptographic libraries. For example, we have already noted that the OpenSSL code aborts and restarts whenever a Miller-Rabin test fails; this behaviour leads to sub-optimal performance, and it would be interesting to see how much the code in OpenSSL and in other leading libraries could be improved.

One can view our work as addressing a specific instance of the problem of how to design simple, performant, misuse-resistant APIs for cryptography. In our discussion of related work, we highlighted other work where this problem has also been considered, in symmetric encryption, key exchange, and secure channels. A broader research effort in this direction seems likely to yield significant rewards for the security of cryptographic software. As here, it may occasionally also yield improved performance.

ACKNOWLEDGEMENTS

We thank Yehuda Lindell for posing the question that led to this research. We also thank Kurt Roeckx for valuable discussions.

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 in part by a gift from VMware.

[8] See https://github.com/openssl/openssl/blob/master/crypto/bn/bn_prime.c#L74.

[9] See https://www.openssl.org/docs/manmaster/man3/BN_is_prime_fasttest_ex.html for details.

[10] See https://www.openssl.org/blog/blog/2019/11/07/3.0-update/.

REFERENCES

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Simson L. Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. 2017. Comparing the Usability of Cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 154–171. https://doi.org/10.1109/SP.2017.52

[2] Yasemin Acar, Sascha Fahl, and Michelle L. Mazurek. 2016. You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In *IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016*. IEEE Computer Society, 3–8. https://doi.org/10.1109/SecDev.2016.013

[3] Martin R. Albrecht, Jake Massimo, Kenneth G. Paterson, and Juraj Somorovsky. 2018. Prime and Prejudice: Primality Testing Under Adversarial Conditions, See [26], 281–298. https://doi.org/10.1145/3243734.3243787

[4] François Arnault. 1997. The Rabin-Monier Theorem for Lucas Pseudoprimes. *Mathematics of Computation of the American Mathematical Society* 66, 218 (1997), 869–881.

[5] Robert Baillie and Samuel S Wagstaff. 1980. Lucas Pseudoprimes. *Math. Comp.* 35, 152 (1980), 1391–1417.

[6] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *PKC 2006 (LNCS, Vol. 3958)*, Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin (Eds.). Springer, Heidelberg, 207–228. https://doi.org/10.1007/11745853_14

[7] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The Security Impact of a New Cryptographic Library. In *LATINCRYPT 2012 (LNCS, Vol. 7533)*, Alejandro Hevia and Gregory Neven (Eds.). Springer, Heidelberg, 159–176.

[8] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. 2014. Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 98–113. https://doi.org/10.1109/SP.2014.14

[9] Daniel Bleichenbacher. 2005. Breaking a Cryptographic Protocol with Pseudoprimes. In *PKC 2005 (LNCS, Vol. 3386)*, Serge Vaudenay (Ed.). Springer, Heidelberg, 9–15. https://doi.org/10.1007/978-3-540-30580-4_2

[10] Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. 2016. Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS. In *10th USENIX Workshop on Offensive Technologies, WOOT 16, Austin, TX, USA, August 8-9, 2016.*, Natalie Silvanovich and Patrick Traynor (Eds.). USENIX Association. https://www.usenix.org/conference/woot16/workshop-program/presentation/bock

[11] Colin Boyd, Britta Hale, Stig Frode Mjølsnes, and Douglas Stebila. 2016. From Stateless to Stateful: Generic Authentication and Authenticated Encryption Constructions with Application to TLS. In *CT-RSA 2016 (LNCS, Vol. 9610)*, Kazue Sako (Ed.). Springer, Heidelberg, 55–71. https://doi.org/10.1007/978-3-319-29485-8_4

[12] Jørgen Brandt and Ivan Damgård. 1993. On Generation of Probable Primes By Incremental Search. In *CRYPTO'92 (LNCS, Vol. 740)*, Ernest F. Brickell (Ed.). Springer, Heidelberg, 358–370. https://doi.org/10.1007/3-540-48071-4_26

[13] Richard Crandall and Carl Pomerance. 2006. *Prime Numbers: A Computational Perspective*. Vol. 182. Springer Science & Business Media. pp.136-140.

[14] Ivan Damgård, Peter Landrock, and Carl Pomerance. 1993. Average Case Error Estimates for the Strong Probable Prime Test. *Math. Comp.* 61, 203 (1993), 177–194.

[15] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in Android applications, See [41], 73–84. https://doi.org/10.1145/2508859.2516693

[16] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL development in an appified world, See [41], 49–60. https://doi.org/10.1145/2508859.2516655

[17] Marc Fischlin, Felix Günther, Giorgia Azzurra Marson, and Kenneth G. Paterson. 2015. Data Is a Stream: Security of Stream-Based Channels. In *CRYPTO 2015, Part II (LNCS, Vol. 9216)*, Rosario Gennaro and Matthew J. B. Robshaw (Eds.). Springer, Heidelberg, 545–564. https://doi.org/10.1007/978-3-662-48000-7_27

[18] Steven D. Galbraith, Jake Massimo, and Kenneth G. Paterson. 2019. Safety in Numbers: On the Need for Robust Diffie-Hellman Parameter Validation. In *PKC 2019, Part II (LNCS, Vol. 11443)*, Dongdai Lin and Kazue Sako (Eds.). Springer, Heidelberg, 379–407. https://doi.org/10.1007/978-3-030-17259-6_13

[19] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. 2018. Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018.*, Mary Ellen Zurko and Heather Richter Lipford (Eds.). USENIX Association, 265–281. https://www.usenix.org/conference/soups2018/presentation/gorski

[20] Matthew Green and Matthew Smith. 2016. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security & Privacy* 14, 5 (2016), 40–46. https://doi.org/10.1109/MSP.2016.111

[21] Shay Gueron, Adam Langley, and Yehuda Lindell. 2017. AES-GCM-SIV: Specification and Analysis. *IACR Cryptology ePrint Archive* 2017 (2017), 168. http://eprint.iacr.org/2017/168

[22] Peter Gutmann. 2002. Lessons Learned in Implementing and Deploying Crypto Software. In *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, Dan Boneh (Ed.). USENIX, 315–325. http://www.usenix.org/publications/library/proceedings/sec02/gutmann.html

[23] Achim Jung. 1987. Implementing the RSA cryptosystem. *Computers & Security* 6, 4 (1987), 342–350. https://doi.org/10.1016/0167-4048(87)90070-8

[24] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. 2013. FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS).

[25] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. Why Does Cryptographic Software Fail?: A Case Study and Open Problems. In *Asia-Pacific Workshop on Systems, APSys'14, Beijing, China, June 25-26, 2014*. ACM, 7:1–7:7. https://doi.org/10.1145/2637166.2637237

[26] David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). 2018. *ACM CCS 2018*. ACM Press.

[27] Jack Lloyd. 2020. Botan Github Repository. https://github.com/randombit/botan/blob/5d74496ee51b8a2d1c418b0a66bddac6f0263749/src/lib/math/numbertheory/primality.cpp#L51.

[28] Ueli M. Maurer. 1995. Fast Generation of Prime Numbers and Secure Public-Key Cryptographic Parameters. *Journal of Cryptology* 8, 3 (Sept. 1995), 123–155. https://doi.org/10.1007/BF00202269

[29] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. 1996. *Handbook of Applied Cryptography*. CRC press.

[30] Franz Mertens. 1874. Ein Beitrag zur analytischen Zahlentheorie. *Journal für die reine und angewandte Mathematik* 1874, 78 (1874), 46–62.

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

[32] Peter L. Montgomery. 1992. Evaluating Recurrences of Form $X_{m+n} = f(X_m, X_n, X_{m-n})$ via Lucas Chains. Unpublished manuscript.

[33] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2017. "Jumping Through Hoops": Why do Java Developers Struggle With Cryptography APIs?. In *Software Engineering 2017, Fachtagung des GI-Fachbereichs Softwaretechnik, 21.-24. Februar 2017, Hannover, Deutschland (LNI, Vol. P-267)*, Jan Jürjens and Kurt Schneider (Eds.). GI, 57. https://dl.gi.de/20.500.12116/1268

[34] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. 2017. Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 311–328. https://doi.org/10.1145/3133956.3134082

[35] Christopher Patton and Thomas Shrimpton. 2018. Partially Specified Channels: The TLS 1.3 Record Layer without Elision, See [26], 1415–1428. https://doi.org/10.1145/3243734.3243789

[36] Carl Pomerance. 1984. Are There Counter-Examples to the Baillie-PSW Primality Test? Dopo Le Parole aangeboden aan Dr. A. K. Lenstra.. pseudoprime.com/dopo.pdf

[37] Carl Pomerance, John L Selfridge, and Samuel S Wagstaff. 1980. The Pseudoprimes to $25 \cdot 10^9$. *Math. Comp.* 35, 151 (1980), 1003–1026.

[38] Michael O Rabin. 1980. Probabilistic Algorithm for Testing Primality. *Journal of number theory* 12, 1 (1980), 128–138.

[39] Phillip Rogaway. 2004. Nonce-Based Symmetric Encryption. In *FSE 2004 (LNCS, Vol. 3017)*, Bimal K. Roy and Willi Meier (Eds.). Springer, Heidelberg, 348–359. https://doi.org/10.1007/978-3-540-25937-4_22

[40] Phillip Rogaway and Thomas Shrimpton. 2006. A Provable-Security Treatment of the Key-Wrap Problem. In *EUROCRYPT 2006 (LNCS, Vol. 4004)*, Serge Vaudenay (Ed.). Springer, Heidelberg, 373–390. https://doi.org/10.1007/11761679_23

[41] Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). 2013. *ACM CCS 2013*. ACM Press.

[42] Atle Selberg. 1949. An Elementary Proof of the Prime-Number Theorem. *Annals of Mathematics* (1949), 305–313.

[43] Glenn Wurster and Paul C. van Oorschot. 2008. The Developer is the Enemy. In *Proceedings of the 2008 Workshop on New Security Paradigms, Lake Tahoe, CA, USA, September 22-25, 2008*, Matt Bishop, Christian W. Probst, Angelos D. Keromytis, and Anil Somayaji (Eds.). ACM, 89–97. https://doi.org/10.1145/1595676.1595691

A COST MODEL FOR PRIME GENERATION

Sieving can be recast as a one-time trial division of the first candidate n with the first s odd primes (OpenSSL uses $s = 2047$), followed by per candidate updating of a table of remainders. We assume the latter can be done essentially for free compared to other operations and ignore its cost henceforth. Then the average cost of prime generation when the underlying primality test uses up to t rounds of MR testing but no trial division, is given by:

$$\left(\sum_{i=1}^{s} C_i\right) + \left(\ln(2^k) \cdot (1 - \sigma_s)/2\right) \cdot C_{MR} + (t-1) \cdot C_{MR}. \quad (9)$$

Here the first term comes from sieving. The second term comes from, on average, inspecting $\ln(2^k) \cdot (1 - \sigma_s)/2$ odd, composite candidates in the sieved version of the list $n, n+2, n+4, \ldots$ before encountering a prime, and doing 1 MR test to reject each composite (recall that, because of sieving, the density of primes in the list $n, n+2, n+4, \ldots$ is boosted by a factor $1/(1 - \sigma_s)$; recall also that almost every random composite is rejected with just 1 MR test). The third term comes from doing a further $t - 1$ MR tests when a prime is finally found. To model OpenSSL's performance, we would set t according to Table 1.

It should be evident from expression (9) that, as with trial division, working with large s in the initial sieve is not profitable: eventually, the gains made from decreasing the term $1 - \sigma_s$ are outweighed by the cost of initial sieving by trial division. Moreover, this model neglects the true cost of updating the table of remainders between candidates. This cost is linear in s (albeit with a small constant) and so heightens the effect. A more detailed model including this cost could of course be developed.

If we now assume that (redundant) trial division with $r \leq s$ primes is also carried out in the underlying primality test, and that the test uses up to t' rounds of MR testing, then the average cost becomes:

$$\left(\sum_{i=1}^{s} C_i\right) + \left(\ln(2^k) \cdot (1 - \sigma_s)/2\right) \cdot \left(\left(\sum_{i=1}^{r} C_i\right) + C_{MR}\right) + (t' - 1) \cdot C_{MR} \quad (10)$$

Here, the additional cost compared to (9) is precisely that of doing a full set of r trial divisions for each candidate – this cost is always incurred because when $r \leq s$, all the candidates which might fail trial division at some early stage have already failed on sieving. To model the performance of OpenSSL with our chosen primality test, MR64, t' must be set to 64 rather than the values in Table 1; the difference means that, when a prime is finally encountered, the cost of testing it will be higher.

The difference in the costs as expressed in (9) and (10) is given by:

$$\left(\ln(2^k) \cdot (1 - \sigma_s)/2\right) \cdot \left(\sum_{i=1}^{r} C_i\right) + \delta_t \cdot C_{MR} \quad (11)$$

where $\delta_t = t' - t$, depending on k, is the difference in the maximum number of rounds of MR testing carried out in the two cases.

For MR64 and MRAC, and for k of cryptographic size, δ_t ranges between 59 and 61. For our selected primality test, MR64 with input-length-dependent trial division, r in the above expression is also k-dependent, and is set by Table 4. The first term in (11) accounts for the cost of redundant trial division over the first r primes for $N := \ln(2^k) \cdot (1 - \sigma_s)/2$ different candidates. Here both r and N are in the range of a few hundred. For example, when $k = 1024$ we set $r = 128$, and when $s = 2047$, we have $N \approx 41$. Hence, when $k = 1024$, we do about 5200 redundant trial divisions, compared to an extra $\delta_t = 59$ MR tests. For this k, the extra MR tests are about 8 times more expensive than the redundant trial divisions (roughly 17.5ms versus 2ms based on our experimental timings). This indicates that the redundant trial division contributes much less to the overhead of prime generation than do the extra MR tests that are necessary to make our primality test secure in all use cases.

Note that this analysis ignores the fact that OpenSSL aborts and restarts with a fresh, random value whenever an MR test fails; this effect may be significant in practice and we leave a detailed evaluation to future work. Note also that this modelling deficiency does not affect our experimental results reported in the main body, since they were obtained by measuring the running time of the actual OpenSSL code.

B REFERENCE IMPLEMENTATION OF THE BAILLIE-PSW TEST

For completeness, we include here our code that implements a Baillie-PSW primality test in the context of OpenSSL's `bn_prime.c`. Functions from the existing OpenSSL code-base have been omitted.

bn_prime_bpsw.c

```
int BN_is_prime_BPSW_ex(BIGNUM *a, BN_CTX *ctx_passed,
                             int do_trial_division, BN_GENCB *cb)
{
    int i, j, l, ret = -1;
    int k;
    BN_CTX *ctx = NULL;
    BIGNUM *A1, *A1_odd, *check = BN_new(); /* taken from ctx */
    BN_MONT_CTX *mont = NULL;
    TRIAL_DIVISION_PRIMES = 129;

    BN_set_word(check, 2); //only testing MR to base 2

    /* Take care of the really small primes 2 & 3 */
    if (BN_is_word(a, 2) || BN_is_word(a, 3))
        return 1;

    /* Check odd and bigger than 1 */
    if (!BN_is_odd(a) || BN_cmp(a, BN_value_one()) <= 0)
        return 0;

    /* first look for small factors */
    if (do_trial_division) {
        for (i = 1; i < TRIAL_DIVISION_PRIMES; i++) {
            BN_ULONG mod = BN_mod_word(a, primes[i]);
            if (mod == (BN_ULONG)-1)
                goto err;
            if (mod == 0)
                return BN_is_word(a, primes[i]);
        }
        if (!BN_GENCB_call(cb, 1, -1))
            goto err;
    }

    if (ctx_passed != NULL)
        ctx = ctx_passed;
    else if ((ctx = BN_CTX_new()) == NULL)
        goto err;
    BN_CTX_start(ctx);

    A1 = BN_CTX_get(ctx);
    A1_odd = BN_CTX_get(ctx);

    if (check == NULL)
        goto err;

    /* compute A1 := a - 1 */
    if (!BN_copy(A1, a) || !BN_sub_word(A1, 1))
        goto err;

    /* write  A1  as  A1_odd * 2^k */
    k = 1;
    while (!BN_is_bit_set(A1, k))
        k++;
    if (!BN_rshift(A1_odd, A1, k))
        goto err;

    /* Montgomery setup for computations mod a */
    mont = BN_MONT_CTX_new();
    if (mont == NULL)
        goto err;
    if (!BN_MONT_CTX_set(mont, a, ctx))
        goto err;

    j = witness(check, a, A1, A1_odd, k, ctx, mont);
    if (j == -1)
        goto err;
    if (j) {
        ret = 0;
        goto err;
    }
    if (!BN_GENCB_call(cb, 1, i))
        goto err;

    ret = 1;

    l = BN_lucas_test_ex(a);
    if (!l) {
        ret = 0;
        goto err;
      }
 err:
    if (ctx != NULL) {
        BN_CTX_end(ctx);
        if (ctx_passed == NULL)
            BN_CTX_free(ctx);
    }
    BN_MONT_CTX_free(mont);

    return ret;
}

int BN_lucas_test_ex(BIGNUM * n){
  //performs a Lucas test (with Selfridge's paramters) on n
  BIGNUM *two = BN_new();
  BN_set_word(two, 2);

  // sanity check input, n odd and > 2
  if (BN_cmp(two,n)==1) { // 1 if a > b i.e b < a
    BN_free(two);
    return 0;
  }
  if (BN_cmp(n,two)==0) {
    BN_free(two);
    return 1;
  }
  if (!BN_is_odd(n)) {
    BN_free(two);
    return 0;
  }
  BN_CTX *ctx = BN_CTX_new();
  BIGNUM *result = BN_new();
  BIGNUM *zero= BN_new();
  BIGNUM *np1 = BN_new();
  BIGNUM *minusone = BN_new();
  BIGNUM *u = BN_new();
  BIGNUM *d = BN_new();
  BIGNUM *minusnineteen = BN_new();

  int32_t J;
  int32_t res;

  const char *m1 = "-1";
  const char *m19 = "-19";
  BN_add(np1,n,BN_value_one());
  BN_zero(zero);
  BN_dec2bn(&minusone, m1);
  BN_dec2bn(&minusnineteen, m19);
  BN_set_word(d, 5);

 // while jacobi(d,n) != -1
  while ((J = BN_jacobi(d,n))!= -1) {
    if (J==0) { // if jacobi(d,n) == 0 then d | n, i.e n is composite
      res = 0;
      goto free;
    }
    if (BN_cmp(zero,d)==1) { // 0>d
      BN_mul(d,d,minusone,ctx);
      BN_add(d,d,two);
    }
    else{
      BN_add(d,d,two);
      BN_mul(d,d,minusone,ctx);
    }

    if (BN_cmp(d,minusnineteen)==0
        &&!(BN_cmp(BN_is_perfect_square(n),zero)==0)){
      res = 0;
      goto free;
    }
  }
  u = BN_lucas_sequence(d,np1,n);
  BN_mod(result,u,n,ctx);
  if (BN_cmp(result,zero)==0) {
    res = 1;
    goto free;
  }
  else{
    res = 0;
    goto free;
  }
  free:
    BN_CTX_free(ctx);
    BN_free(result);
    BN_free(zero);
    BN_free(np1);
    BN_free(minusone);
    BN_free(two);
    BN_free(u);
    BN_free(d);
    return res;
}

int BN_jacobi(BIGNUM *a, BIGNUM *n){
  // computes jacobi symbol of (a/n),
  //currently returns 2 if a,n are invalid input
  BIGNUM *x = BN_new();
  BIGNUM *y = BN_new();
  BIGNUM *halfy = BN_new();
  BIGNUM *r = BN_new();
  BIGNUM *s = BN_new();

  BN_CTX *ctx = BN_CTX_new();
  BN_nnmod(x,a,n,ctx);
  BN_copy(y,n);
  int J  = 1;
  int k = 0;

  BIGNUM *three = BN_new();
  BN_set_word(three, 3);
  BIGNUM *four = BN_new();
  BN_set_word(four, 4);
  BIGNUM *five = BN_new();
  BN_set_word(five, 5);
  BIGNUM *eight = BN_new();
  BN_set_word(eight, 8);

  if (!BN_is_odd(n)||BN_cmp(n,BN_value_one()) <= 0) {
    J = 2;
    goto free;
  }

  while (BN_cmp(y,BN_value_one()) == 1) { // while y > 1
    BN_mod(x,x,y,ctx);
    BN_rshift1(halfy,y);
    if (BN_cmp(x,halfy)==1) {
      BN_sub(x,y,x);
      BN_mod(r,y,four,ctx);
      if (BN_cmp(r,three)==0) {
        J = -J;
      }
    }
    if (BN_is_zero(x)) {
      //gcd(a,n)!=1 so we return 0
      J = 0;
      goto free;
    }
    //count the zero bits in x,
    //i.e the largest value of n s.t 2^n divides x evenly.
    k = 0;
    while (!BN_is_bit_set(x, k)) {
      k++;
    }
    BN_rshift(x,x,k);

    if (k%2) {
      BN_mod(s,y,eight,ctx);
      if (BN_cmp(s,three)==0 || BN_cmp(s,five)==0) {
        J = -J;
      }
    }
    BN_mod(r,x,four,ctx);
    BN_mod(s,y,four,ctx);
    if (BN_cmp(r,three)==0 && BN_cmp(s,three)==0) {
      J = -J;
    }
    BN_swap(x,y);
  }
  free:
     BN_CTX_free(ctx);
     BN_free(x);
     BN_free(y);
     BN_free(halfy);
     BN_free(r);
     BN_free(s);
     BN_free(three);
     BN_free(four);
     BN_free(five);
     BN_free(eight);
     return J;
}

void BN_rshift1_round(BIGNUM *r, BIGNUM *a){
  // temporary fix as part of code demo, but the rounding in BN_rshift1
  // is not consistant with python/java across positive and negative numbers.
  // This function adds one before the shift if a is negative and performs
  // BN_rshift1 normally otherwise. e.g this function rounds -127/2 = -63.5
  // to -64 (toward -infinity), where as BN_rshift1 would round to -63 (toward 0)
  // This is needed in my implementation of jacobi symbol calculation.
  //Can't simply negate result, as we still want 127/2 = 63.

  BIGNUM *zero= BN_new();
  BIGNUM *one= BN_new();
  BN_zero(zero);
  BN_one(one);

  if (BN_cmp(zero,a)==1) { //a < 0
    BN_sub(r,a,one);
    BN_rshift1(r,r);
  }
  else{
    BN_rshift1(r,a);
  }
  BN_free(zero);
  BN_free(one);
}

BIGNUM * BN_lucas_sequence(BIGNUM *d, BIGNUM *k, BIGNUM *n){
   //computes the Lucas sequence U_k modulo n, where d = p^2 -4q
   BN_CTX *ctx = BN_CTX_new();
   BIGNUM *kp1 = BN_new();
   BIGNUM *u = BN_new();
   BIGNUM *v = BN_new();
   BIGNUM *u2 = BN_new();
   BIGNUM *v2 = BN_new();
   BIGNUM *r= BN_new();
   BIGNUM *zero= BN_new();
   BIGNUM *one= BN_new();

   BN_add(kp1,k,BN_value_one());
   BN_zero(zero);
   BN_one(one);
   BN_one(u);
   BN_one(v);

   size_t k_bits = BN_num_bits(kp1) -1;

   for (size_t i = k_bits-1; i != (size_t) -1; --i) {
     BN_mod_mul(u2,u,v,n,ctx);
     BN_mod_sqr(r,u,n,ctx); //r = u^2 mod n
     BN_mod_mul(r,r,d,n,ctx); // r = r *d  = u^2 *d (mod n)
     BN_mod_sqr(v2,v,n,ctx); //v2 = v^2 mod n
     BN_mod_add(v2,v2,r,n,ctx); // v2 = v2 + r = v^2 + (u^2*d) (mod n)

     if (BN_is_odd(v2)) {
       BN_sub(v2,v2,n);// v2 = v2 - n
     }

     BN_rshift1_round(v2,v2);
     BN_copy(u,u2);
     BN_copy(v,v2);

     if (BN_is_bit_set(k,i)) {
       BN_nnmod(r,v,n,ctx); //r= v mod
       BN_add(u2,u,r); // u2 = u + v mod n

       if (BN_is_odd(u2)) {
         BN_sub(u2,u2,n);
       }
       BN_rshift1_round(u2,u2);
       BN_mod_mul(r,d,u,n,ctx); // r = d*u mod n
       BN_add(v2,v,r); // v2 = r + v = v + d*u mod n

       if (BN_is_odd(v2)) {
         BN_sub(v2,v2,n);
       }
       BN_rshift1_round(v2,v2);
       BN_copy(u,u2);
       BN_copy(v,v2);
       }
   }
BN_CTX_free(ctx);
BN_free(kp1);
BN_free(v);
BN_free(u2);
BN_free(v2);
BN_free(r);
BN_free(zero);
BN_free(one);
return u;
}

BIGNUM * BN_is_perfect_square(BIGNUM * C){
  //https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf sec C.4
  // checks if C is a perfect square.
  //If so, function returns X where C = X^2 else function returns 0
  BIGNUM *one= BN_new();
  BIGNUM *zero= BN_new();
  BIGNUM *ret= BN_new();
  BN_one(one);
  BN_zero(zero);

  if (BN_cmp(one,C)==1) {
    printf("is_perfect_square requires C >=1 \n");
    BN_free(one);
    return zero;
  }
  if (BN_cmp(one,C)==0) {
    BN_free(zero);
    return one;
  }
  BN_CTX *ctx = BN_CTX_new();
  BIGNUM *B = BN_new();
  BIGNUM *X = BN_new();
  BIGNUM *r = BN_new();
  BIGNUM *s = BN_new();
  BIGNUM *X2 = BN_new();
  BIGNUM *two= BN_new();
  size_t c_bits = BN_num_bits(C);
  size_t m = (c_bits+1)/2;

  BN_set_word(two, 2);
  BN_set_bit(B,m);
  BN_add(B,B,C);
  BN_set_bit(X,m);
  BN_sub(X,X,one);
  BN_mul(X2,X,X,ctx);

  for (;;) {
    BN_add(r,X2,C);
    BN_mul(s,X,two,ctx);
    BN_div(X,NULL,r,s,ctx);
    BN_mul(X2,X,X,ctx);
    if (BN_cmp(B,X2)==1) {
      break;
    }
  }
  if (BN_cmp(X2,C)==0) {
    ret = X;
    goto free;
  }
  else {
    ret = zero;
    goto free;
  }
  free:
    BN_CTX_free(ctx);
    BN_free(B);
    BN_free(r);
    BN_free(s);
    BN_free(X2);
    BN_free(one);
    BN_free(two);
    BN_free(zero);
    return ret;
}
```