On the Design of Cryptographic APIs: Primality Testing as a Case Study

Kenny Paterson

ETH Zürich

https://appliedcrypto.ethz.ch/

@kennyog

Joint work with Martin Albrecht, Jake Massimo and Juraj Somorovsky


Prime Numbers
51.9K Tweets
Follow
Prime Numbers
@_primes_
Every prime number, eventually. (Or the heat death of the universe; whichever happens first.)
github.com/fardog/primebot
Joined September 2013
1 Following
13.7K Followers
Not followed by anyone you're following
Tweets
Tweets & replies
Media
Likes
Prime Numbers @_primes_ · 19m
637291
2
Prime Numbers @_primes_ · 1h
637283
2
Prime Numbers @_primes_ · 2h
637277
3

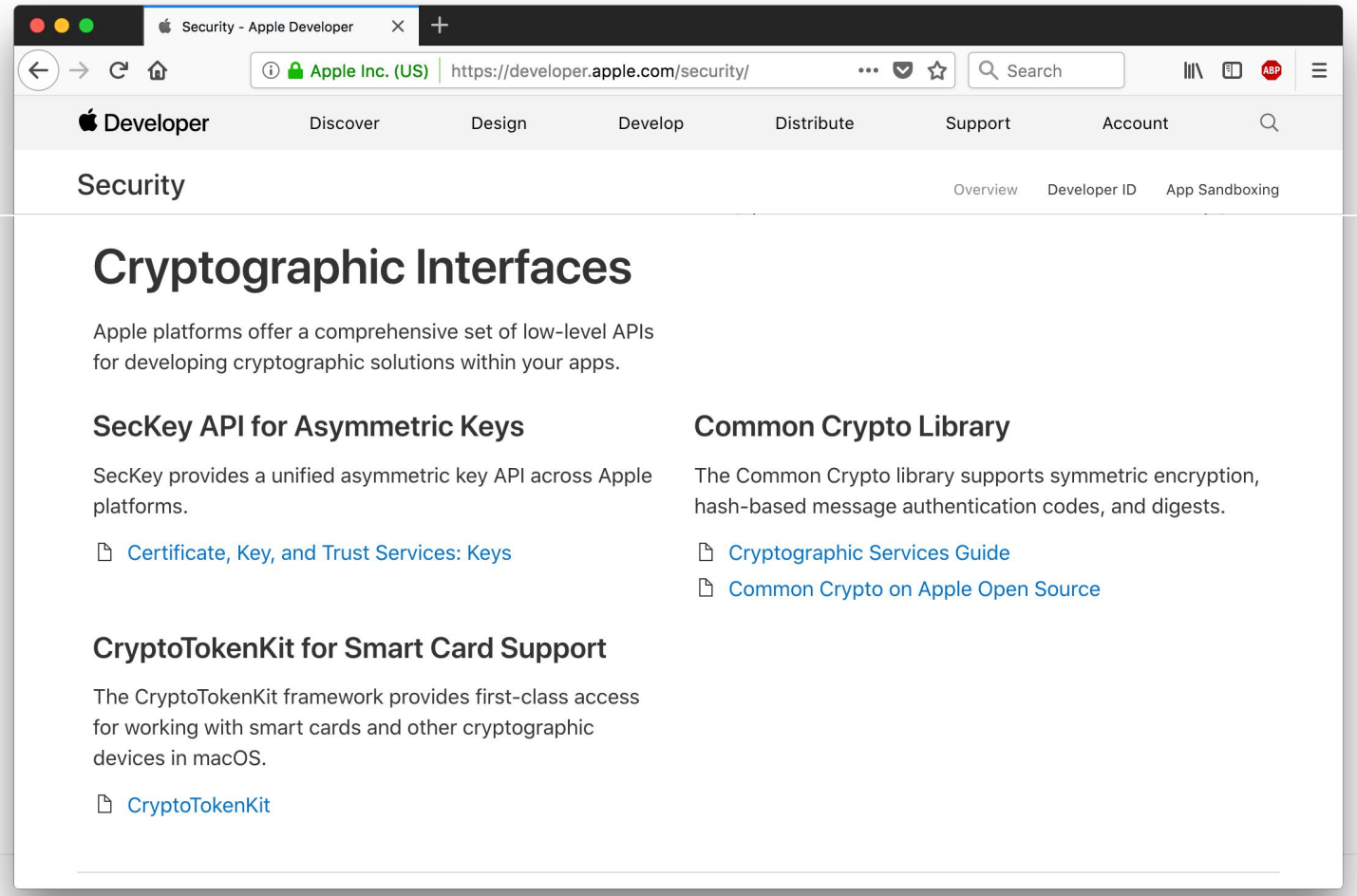
Apple

CommonCrypto

Primality test:
CCBigNumIsPrime

Arguments:
&status – result
n – number to test

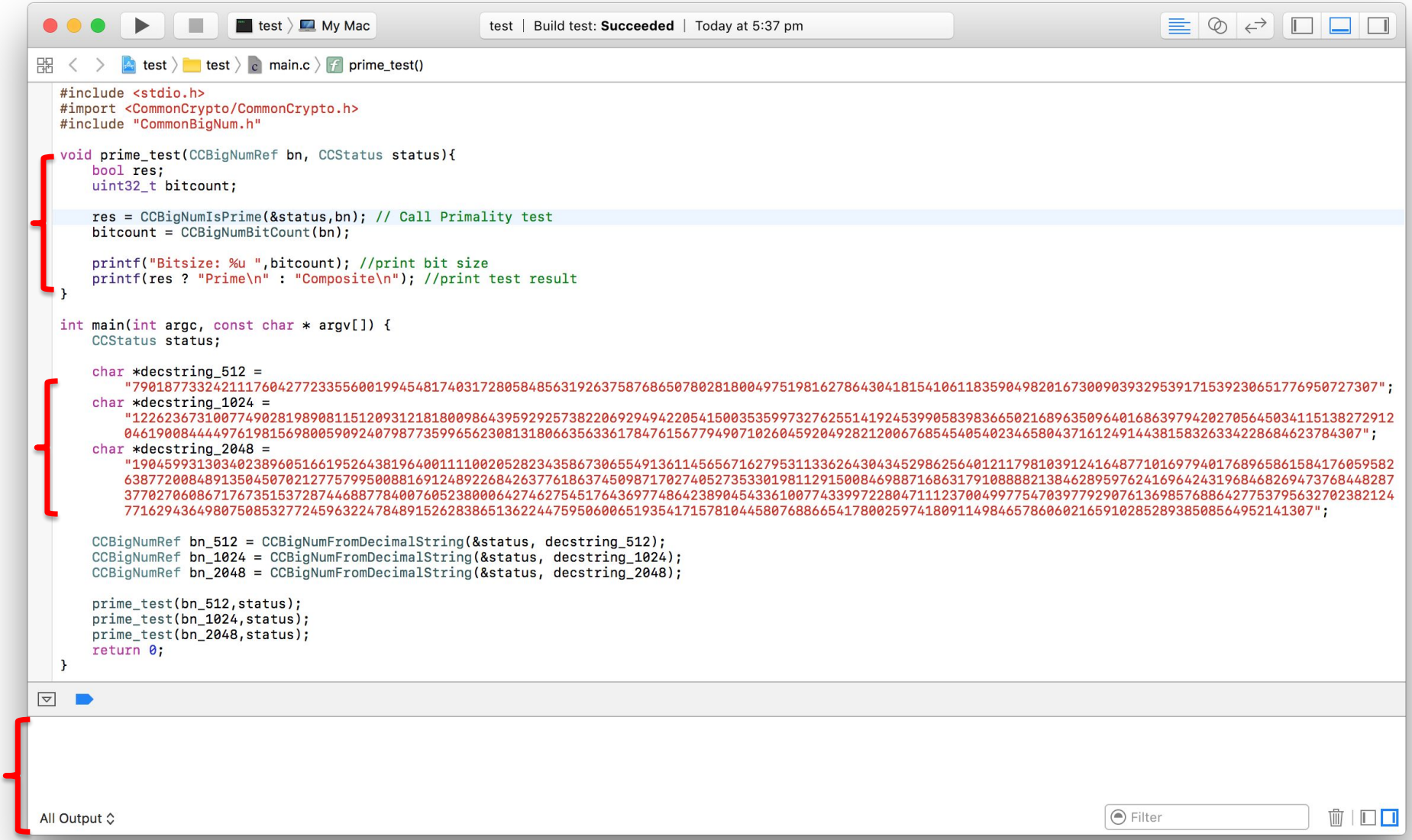
Method:
16 rounds of Miller-Rabin

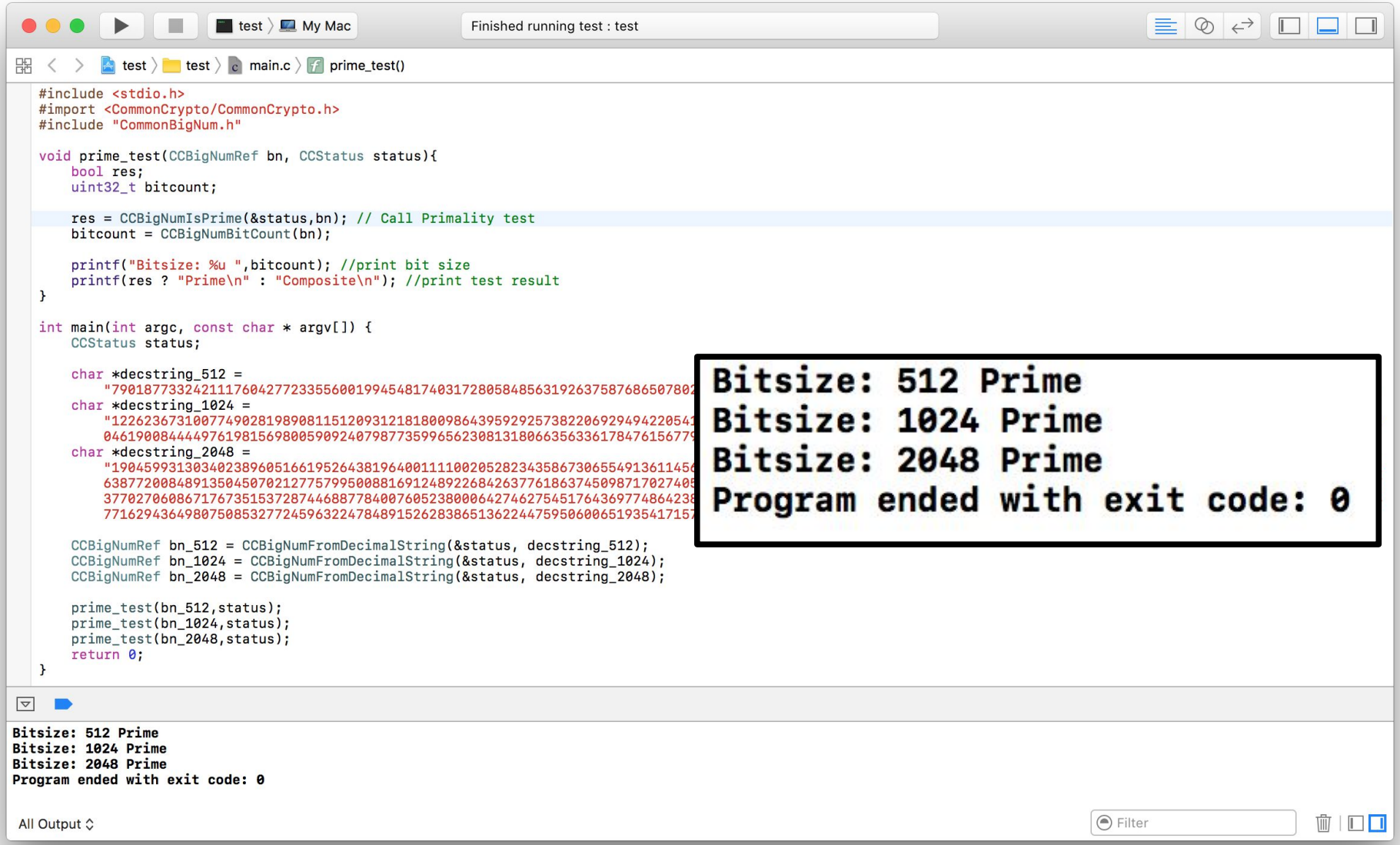


Test

Input

Output




test
My Mac
Finished running test : test
test
test
main.c
prime_test()
#include <stdio.h>
#import <CommonCrypto/CommonCrypto.h>
#include "CommonBigNum.h"
void prime_test(CCBigNumRef bn, CCStatus status){
bool res;
uint32_t bitcount;
res = CCBigNumIsPrime(&status,bn); // Call Primality test
bitcount = CCBigNumBitCount(bn);
printf("Bitsize: %u ",bitcount); //print bit size
printf(res ? "Prime\n" : "Composite\n"); //print test result
}
int main(int argc, const char * argv[]) {
CCStatus status;
char *decstring_512 =
"7901877332421117604277233556001994548174031728058485631926375876865078
char *decstring_1024 =
"1226236731007749028198908115120931218180098643959292573822069294942205
0461900844449761981569800590924079877359965623081318066356336178476156779
char *decstring_2048 =
"1904599313034023896051661952643819640011110020528234358673065549136114
6387720084891350450702127757995008816912489226842637761863745098717027405
3770270608671767351537287446887784007605238000642746275451764369774864238
7716294364980750853277245963224784891526283865136224475950600651935417157
CCBigNumRef bn_512 = CCBigNumFromDecimalString(&status, decstring_512);
CCBigNumRef bn_1024 = CCBigNumFromDecimalString(&status, decstring_1024);
CCBigNumRef bn_2048 = CCBigNumFromDecimalString(&status, decstring_2048);
prime_test(bn_512,status);
prime_test(bn_1024,status);
prime_test(bn_2048,status);
return 0;
}
Bitsize: 512 Prime
Bitsize: 1024 Prime
Bitsize: 2048 Prime
Program ended with exit code: 0
Bitsize: 512 Prime
Bitsize: 1024 Prime
Bitsize: 2048 Prime
Program ended with exit code: 0
All Output
Filter

A closer look at n (1024 bit)

n = 12262367310077490281989081151209312181800986439592925738220692949422054150035359973276255141924539905839836650216896350964016863979420270564503411513827291204619008444497619815698005909240798773599656230813180663563361784761567794907102604592049282120067685454054023465804371612491443815832633422868462378430 7

A closer look at n (1024 bit)

n = 18442364082124525409951026398972618527555526435477996345973043746687208641458040368164832160949954320320839871412800713713244659830839058936137744872090135870949539433756545764847585616026260341873448381827 *

3190528986207542895921527567022263005267106073337693367853336568176887094972240983692515963844342097394 7

$n = p_1 p_2 p_3$ where $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (113,173)$

- This n was carefully constructed to **always** be declared prime by the 16 rounds of **fixed-base** Miller-Rabin testing used by Apple's CommonCrypto.
- 512 and 2048 bit examples are also composites of this form.

Disclosure

CVE-ID	
CVE-2018-4398	Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information
Description	
An issue existed in the method for determining prime numbers. This issue was addressed by using pseudorandom bases for testing of primes. This issue affected versions prior to iOS 12.1, macOS Mojave 10.14.1, tvOS 12.1, watchOS 5.1, iTunes 12.9.1, iCloud for Windows 7.8.	
References	
Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.	

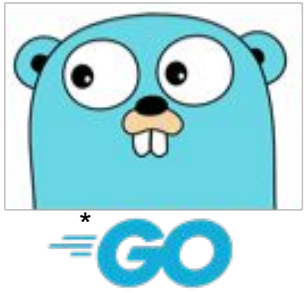
- MISC:https://support.apple.com/kb/HT209192
- MISC:https://support.apple.com/kb/HT209193
- MISC:https://support.apple.com/kb/HT209194
- MISC:https://support.apple.com/kb/HT209195
- MISC:https://support.apple.com/kb/HT209197
- MISC:https://support.apple.com/kb/HT209198

Library analysis

LibTomCrypt

Bouncy Castle

Libgcrypt



SymPy

Library analysis

For each library we studied:

- Which primality tests are being performed?
- How are the tests implemented?
- How do these tests perform against maliciously generated composite numbers?

Bouncy Castle

SymPy

Library analysis

Prime and Prejudice: Primality Testing Under Adversarial Conditions

Martin R. Albrecht
martin.albrecht@rhul.ac
Royal Holloway, University of

Kenneth G. Paterso
kenny.paterson@rhul.ac
Royal Holloway, University of

Current OpenSSL API for primality testing

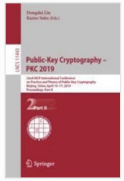
```
int BN_is_prime_fasttest_ex(const BIGNUM *w, int
checks, BN_CTX *ctx_passed, int do_trial_division,
BN_GENCB *cb)
```

- `checks` = 0: defaults to setting number of MR rounds based on size of input.
  - Typically 2 or 3 MR rounds for inputs of cryptographic size.
  - Designed for random input testing, not appropriate when testing maliciously-generated inputs.

Bad Diffie-Hellman in OpenSSL

- Set $q = q_1 q_2 \ldots q_9$.
- Then q fools a single random-base MR test with probability 1/256.
- $p = 2q+1$ is prime and has 1024 bits.
- We can solve DLP in subgroup of order q in "only" $O(2^{64})$ operations.
- OpenSSL prior to 1.1.0i validates these parameters as being good for DH with probability 2^{-24}.

$$
\begin{aligned}
q_1 &= 219186431519361672882122216610071 \\
q_2 &= 407060515678814535352512687990131 \\
q_3 &= 202277763945010915259787074185864 7 \\
q_4 &= 585540895630294754699383635801 1871 \\
q_5 &= 141594434761507640681850951930 10523 \\
q_6 &= 148733649959566849455725789842 54751 \\
q_7 &= 146385223907573688674845908950296751 \\
q_8 &= 109702808975440517277502169413340 0351 \\
q_9 &= 135347621463205833004710468756718 2251.
\end{aligned}
$$



IACR International Workshop on Public Key Cryptography

PKC 2019: Public-Key Cryptography – PKC 2019 pp 379-407 | Cite as

Safety in Numbers: On the Need for Robust Diffie-Hellman Parameter Validation

Authors | Authors and affiliations

Steven Galbraith, Jake Massimo, Kenneth G. Paterson

Analysis: Bad Diffie-Hellman in OpenSSL

- The OpenSSL developers misused their primality testing API in their own library!
- They used `checks` = 0 defaulting to “random case” instead of “malicious case” test parameters.
- Can lead to acceptance of weak DH parameters (with low prob.)
- Attack scenario: PAKE protocol in which attacker impersonates server, offers bad DH parameters, leading to client password breach.

Designing a better API

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

Research Question:
Is it possible to design a **robust** and **performant** primality test whose API has a **single** input: *n*, the number being tested?

Designing a better API

- Having a simple API means the primality test **cannot** be fine-tuned to different use cases by a well-informed developer.
  - OpenSSL's `checks` and `do_trial_division` options are no longer available.

- The test must always use pessimistic settings because inputs **may** be malicious.
  - What is the impact on performance?

Candidate for underlying primality test

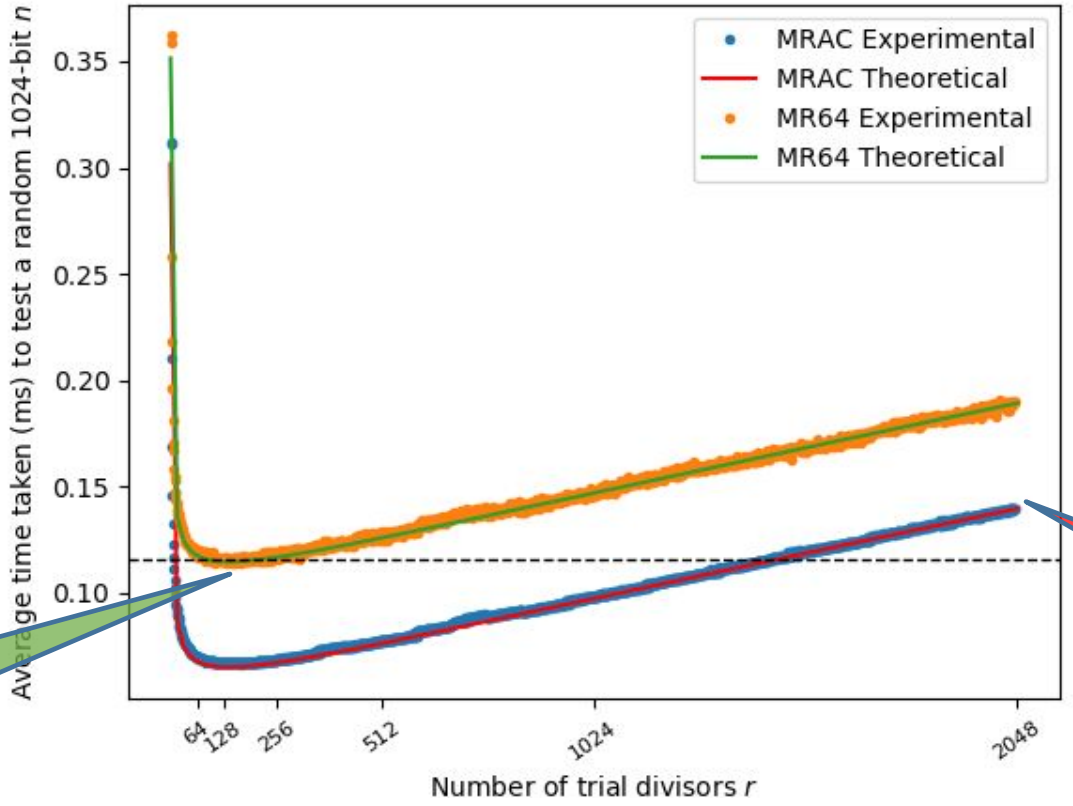
- **MR64** test has worst-case error rate 2^{-128} on composite input, and error rate 0 on prime input.
  - Simple to implement, widely supported in existing libraries.

- For comparison: **MRAC (Miller-Rabin average case):** what OpenSSL currently does by default
  - Has worst-case error rate 2^{-4} on adversarial input.
  - **But it's fast!**

Trial division

- Primality tests typically use trial division with small divisors to speed up performance.
- For example, OpenSSL offers the option to do trial division on the first **2047** odd primes (selected via the `do_trial_division` flag).
- How does the amount of trial division affect test performance?

Tuning trial division



More sensible

Overkill!

Running time of OpenSSL default (MRAC) and MR64

Designing a better API

- MR64 using trial division with *r* = 128 primes is **17% faster** than current OpenSSL for random, odd, 1024-bit inputs!

- More generally, we can tune *r* to the input size *k*.

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

Prime *Generation* Use Case

- OpenSSL uses **sieving** and primality testing (with `checks=0`, `do_trial_division=0`) to generate random primes.
- MR64 does up to 64 rounds of MR and redundant trial division.
- So what is the perf impact of using MR64 as a drop-in replacement in OpenSSL?

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%

- **We can even beat OpenSSL by 5% if we allow ourselves to tune sieving as well!**

Deploying the API

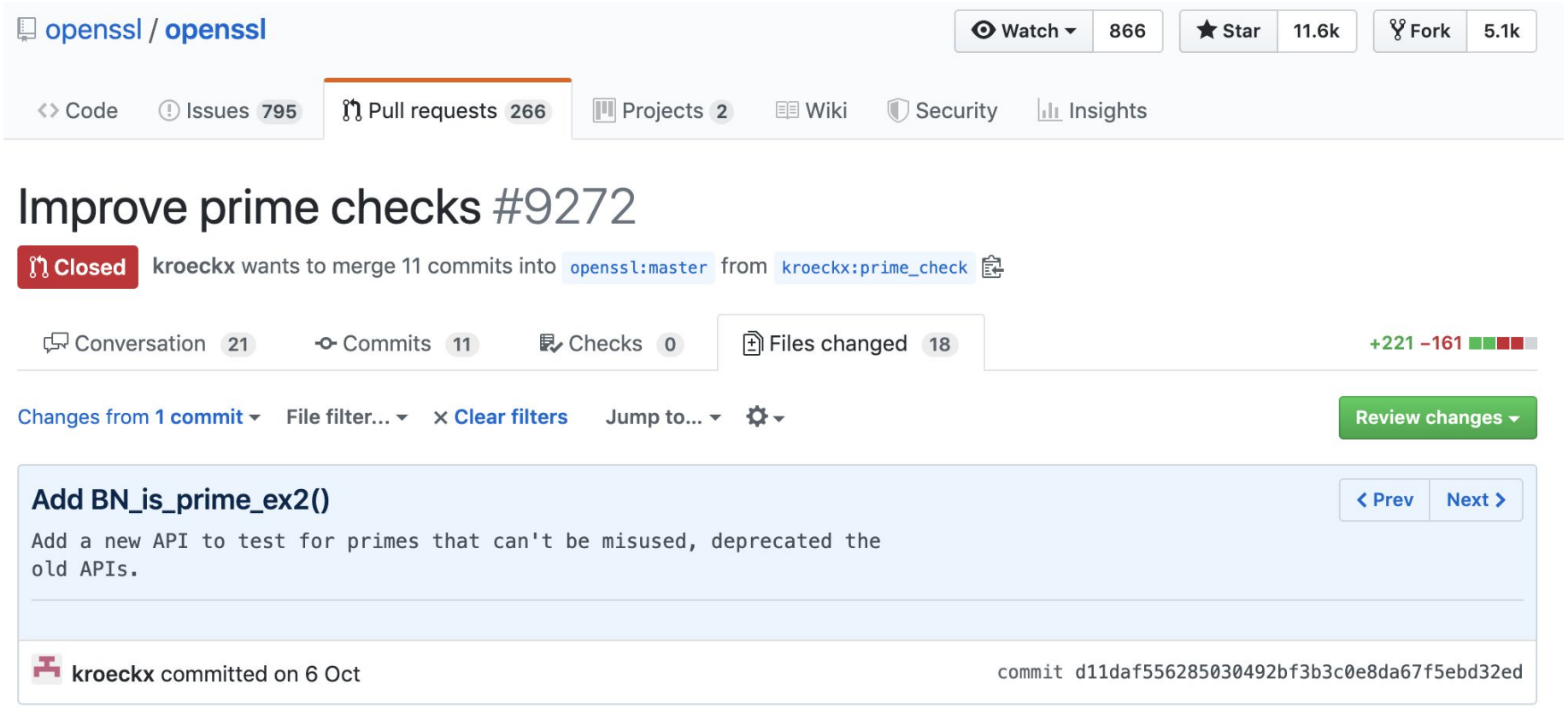
- Our proposed API for MR64 wraps the existing OpenSSL function.

```
int BN_is_prime_robust_ex(const BIGNUM *a) {
   return BN_is_prime_fasttest_ex(a, 64, NULL, 1,
NULL); }
```

- But we also need to modify function internals to tune trial division.

- **What about legacy code using the existing API?**
  - We can modify the existing OpenSSL primality test to **force** the use of MR64, no matter what the calling code asks for!

Deploying the API – OpenSSL



Deploying the API – OpenSSL

```
566 +
567 + int BN_is_prime_ex2(const BIGNUM *p, BN_CTX *ctx, BN_GENCB *cb)
568 + {
569 +     /*
570 +      * By default use 64 rounds of Miller-Rabin, which should give a false
571 +      * positive rate of 2^-128. If the size of the prime is larger than 2048
572 +      * the user probably wants a higher security level than 128, so switch
573 +      * to 128 rounds giving a false positive rate of 2^-256.
574 +      */
575 +     int checks = 64;
576 +     if (BN_num_bits(p) > 2048)
577 +         checks = 128;
578 +
579 +     return bn_is_prime_int(p, checks, ctx, 1, cb);
580 + }
```

https://github.com/openssl/openssl/pull/9272/commits/d11daf556285030492bf3b3c0e8da67f5ebd32ed

Deploying the API – OpenSSL

https://github.com/openssl/openssl/pull/9272/commits/d11daf556285030492bf3b3c0e8da67f5ebd32ed

Concluding Remarks

- Elevate the study of crypto APIs to a first-class research concern.
  - cf. Nonce-based AEAD, Curve25519, NaCl crypto library,…
  - [LCWZ14,GS16, NKMB17,…] on crypto API design/testing.

- Simplicity and security **versus** flexibility and performance.

- If we continue to get it wrong for a classical task like primality testing, what hope do we have for more advanced cryptographic functions?

- Interested? We are hiring at ETH Applied Cryptography!

Further reading

- M.R. Albrecht, J. Massimo, K.G. Paterson and J. Somorovsky. *Prime and Prejudice: Primality Testing Under Adversarial Conditions*. In D. Lie, M. Mannan, M. Backes and X. Wang (eds.), ACM CCS, 281-298, 2018.

  Full Paper: https://eprint.iacr.org/2018/749

- S.D. Galbraith, J. Massimo and K.G. Paterson. *Safety in Numbers: On the Need for Robust Diffie-Hellman Parameter Validation*. PKC 2019(2), LNCS 11443, 379-407, 2019.

  Full Paper: https://eprint.iacr.org/2019/032

- J. Massimo and K.G. Paterson. *A Performant, Misuse-Resistant API for Primality Testing.* In submission.