Return of the Hidden Number Problem

A Widespread and Novel Key Extraction Attack on ECDSA and DSA

Keegan Ryan
NCC Group
What is ROHNP?

• Key extraction attack on DSA and ECDSA
• Uses an old technique to target a new part of the algorithm
• Common (11/20 tested implementations were vulnerable)
• Easy attack to understand and apply
Prior Attacks on (EC)DSA

\[ r = f(k \times G) \]
\[ s = k^{-1}(m + rx) \]

• The attacker knows \( r, s, m, \) and \( G. \)
• Recover information about nonce \( k. \)
• Derive information about private key \( x. \)
NonceLeaksandtheHiddenNumberProblem

\[ r = f(k \times G) \]
\[ s = k^{-1}(m + rx) \]

- Observe multiplication \( k \times G \) happens quickly
- Infer \( k \) is “small”
- Rewrite DSA equations [HGS01]

\[ k = s^{-1}m + (s^{-1}r)x < q/2^l \]

- Solve system of inequalities [BV96]
- Fix nonce leaks with constant time multiplication
Return of the Hidden Number Problem
Return of the Hidden Number Problem

\[ r = f(k \ast G) \]
\[ s = k^{-1}(m + rx) \]

- The attacker knows \( r, s, m, \) and \( G \).
- Target the addition in the calculation of \( s \).
def AddMod(a, b, q):
    # Assuming a and b are reduced modulo q,
    # return \((a + b) \mod q\)
    c = a + b
    if c >= q:
        c = c - q
    return c
Return of the Hidden Number Problem

- Observe the calculation of $m + rx$
- Use a side channel to see if the addition wraps around
- If not,
  \[ m + rx < q \Rightarrow 0 + rx < q - m \]
- If so,
  \[ m + rx \geq q \Rightarrow q - rx < m + 1 \]

- Result is a system of HNP inequalities
Benefits of the ROHNP attack

• Information can leak through many side channels
• Attacker can choose $m$ to tune the bits leaked per HNP inequality
• Can detect the presence of this vulnerability in a black box
  • Signatures with large $m$ are more likely to include the extra subtraction
  • Run statistical analysis to see if this case takes longer
  • Exploit with a side channel that detects subtraction in an individual sample
• Avoids prior countermeasures
• Common
Affected Implementations
Cryptographic Libraries

- LibreSSL
- Mozilla NSS
- OpenSSL
- WolfCrypt
- Botan
- Libgcrypt
- Libtomcrypt
- matrixSSL
- OpenJDK libsunec
- CryptLib
- Golang crypto/tls
- BouncyCastle
- mbedTLS
- C# Mono
- Trezor Crypto
- BoringSSL
- Nettle
- Crypto++
- BearSSL
- Libsecp256k1
- NaCl
- Netflix MSL
- ZeroMQ
- Pyca/cryptography
- Amazon s2n
- GnuTLS
- Cloudflare CFSSL
- NanoSSL
- Microsoft Schannel
- Apple Secure Transport
- RSA BSAFE
- SharkSSL
- Microsoft CryptoAPI/CNG
- JCA
- CryptoComply
- Oracle JSSE
# Cryptographic Libraries

- LibreSSL
- Mozilla NSS
- OpenSSL
- WolfCrypt
- Botan
- Libgcrypt
- Libtomcrypt
- matrixSSL
- OpenJDK libsunec
- CryptLib
- Golang crypto/tls
- BouncyCastle
- mbedTLS
- C#/Mono
- Trezor Crypto
- BoringSSL
- Nettle
- Crypto++
- BearSSL
- Libsecp256k1
- NaCl
- Netflix MSL
- ZeroMQ
- Pyca/cryptography
- Amazon s2n
- GnuTLS
- Cloudflare CFSSL
- NanoSSL
- Microsoft Schannel
- Apple Secure Transport
- RSA BSAFE
- SharkSSL
- Microsoft CryptoAPI/CNG
- JCA
- CryptoComply
- Oracle JSSE
Cryptographic Libraries

- LibreSSL
- Mozilla NSS
- OpenSSL
- WolfCrypt
- Botan
- Libgcrypt
- Libtomcrypt
- matrixSSL
- OpenJDK libsunec
- CryptLib
- Golang crypto/tls
- BouncyCastle

- mbedTLS
- C#/Mono
- Trezor Crypto
- BoringSSL
- Nettle
- Crypto++
- BearSSL
- Libsecp256k1
- NaCl
- Netflix MSL
- ZeroMQ
- Pyca/cryptography

- Amazon s2n
- GnuTLS
- Cloudflare CFSSL
- NanoSSL
- Microsoft Schannel
- Apple Secure Transport
- RSA BSAFE
- SharkSSL
- Microsoft CryptoAPI/CNG
- JCA
- CryptoComply
- Oracle JSSE

Closed Source Wraps (EC)DSA
Cryptographic Libraries

- LibreSSL
- Mozilla NSS
- OpenSSL
- WolfCrypt
- Botan
- Libgcrypt
- Libtomcrypt
- matrixSSL
- OpenJDK libsunec
- CryptLib
- Golang crypto/tls
- BouncyCastle
- mbedTLS
- C# Mono
- Trezor Crypto
- BoringSSL
- Nettle
- Crypto++
- BearSSL
- Libsecp256k1
- NaCl
- Netflix MSL
- ZeroMQ
- Pyca/cryptography
- Amazon s2n
- GnuTLS
- Cloudflare CFSSL
- NanoSSL
- Microsoft Schannel
- Apple Secure Transport
- RSA BSAFE
- SharkSSL
- Microsoft CryptoAPI/CNG
- JCA
- CryptoComply
- Oracle JSSE
Open Source Implementations

- LibreSSL
- Mozilla NSS
- OpenSSL
- WolfCrypt
- Botan
- Libgcrypt
- Libtomcrypt
- matrixSSL
- OpenJDK libsunec
- CryptLib
- Golang crypto/tls
- BouncyCastle
- mbedtlsTLS
- C#/Mono
- Trezor Crypto
- BoringSSL
- Nettle
- Crypto++
- BearSSL
- Libsecp256k1
Open Source Implementations

- LibreSSL
- Mozilla NSS
- OpenSSL
- WolfCrypt
- Botan
- Libgcrypt
- Libtomcrypt
- matrixSSL
- OpenJDK libsunec
- CryptLib

- Golang crypto/tls
- BouncyCastle
- mbedTLS
- C#/Mono
- Trezor Crypto
- BoringSSL (ECDSA)
- Nettle (ECDSA)
- Crypto++
- BearSSL
- Libsecp256k1

Constant Time
Open Source Implementations

- LibreSSL
- Mozilla NSS
- OpenSSL (DSA)
- WolfCrypt (DSA)
- Botan (DSA)
- Libgcrypt (DSA)
- Libtomcrypt (DSA)
- matrixSSL
- OpenJDK libsunec
- CryptLib

- Golang crypto/tls
- BouncyCastle
- mbedTLS
- C#/Mono
- Trezor Crypto
- BoringSSL (ECDSA)
- Nettle (ECDSA)
- Crypto++
- BearSSL
- Libsecp256k1

Constant Time
Wrong Operation Order
Open Source Implementations

- LibreSSL
- Mozilla NSS
- OpenSSL (DSA)
- WolfCrypt (DSA)
- Botan (DSA)
- Libgcrypt (DSA)
- Libtomcrypt (DSA)
- matrixSSL
- OpenJDK libsunec
- CryptLib

- Golang crypto/tls
- BouncyCastle
- mbedTLS
- C#/Mono
- Trezor Crypto
- BoringSSL (ECDSA)
- Nettle (ECDSA)
- Crypto++
- BearSSL
- Libsecp256k1
Example:
Solo: open security key supporting FIDO2 & U2F over USB + NFC - https://solokeys.com/somu

**micro-ecc**

A small and fast ECDH and ECDSA implementation for 8-bit, 32-bit, and 64-bit processors.

The static version of micro-ecc (ie, where the curve was selected at compile-time) can be found in the "static" branch.

**Features**

- Resistant to known side-channel attacks.
- Written in C, with optional GCC inline

NEW! We launched a new tiny security key called Somu, it's live on Crowd Supply and you can pre-order it now!
/* Computes result = (left + right) % mod.
   Assumes that left < mod and right < mod, and that result does not overlap mod. */

uECC_VLI_API void uECC_vli_modAdd(uECC_word_t *result,
     const uECC_word_t *left,
     const uECC_word_t *right,
     const uECC_word_t *mod,
     wordcount_t num_words) {
    uECC_word_t carry = uECC_vli_add(result, left, right, num_words);
    if (carry || uECC_vli_cmp_unsafe(mod, result, num_words) != 1) {
        /* result > mod (result = mod + remainder),
           so subtract mod to get remainder. */
        uECC_vli_sub(result, result, mod, num_words);
    }
}
Conclusion

• ROHNP targets a different part of (EC)DSA signing
• It is widespread
• It is easy to understand and exploit
Thank You

Keegan Ryan
kryan@eng.ucsd.edu
@inf_0_